NYCPS TMS: Infrastructure Provisioning with Terraform

Introduction & Prerequisites

This document provides detailed, step-by-step instructions for provisioning the AWS GovCloud infrastructure required for the NYCPS Transportation Management System, using Terraform. It follows modern Infrastructure as Code (IaC) best practices, emphasizing modularity, environment separation, security, and automation.

> **Important:** The Terraform code provided here is **representative and requires customization**. It includes placeholders (marked with `TODO:` or using variable names like `var.specific_cidr`) that MUST be replaced with values specific to the NYCPS environment and requirements. **Do not deploy this code directly to production without thorough review, customization, and testing by experienced cloud and security engineers familiar with AWS GovCloud and NYCPS policies.** This guide assumes a foundational understanding of Terraform concepts (providers, resources, variables, modules, state).

Prerequisites:

- **Terraform CLI:** Installed locally or on a build server (latest stable version recommended).

- **AWS CLI:** Installed and configured with credentials possessing sufficient permissions in the target AWS GovCloud accounts (preferably via STS AssumeRole from a central management account).
- **Git:** Installed for version control.
- **Code Editor:** VS Code with Terraform extension, or similar.
- **AWS GovCloud Access:** Confirmed access and necessary base configuration (e.g., organization structure if used).
- **NYCPS Specific Info:** Required CIDR ranges, domain names, specific integration endpoints, compliance requirements details, KMS key policies, etc.

Step 1: Establish Project Structure

We will organize the Terraform code into reusable modules and environment-specific configurations. This promotes consistency and simplifies management.

Recommended Directory Structure:

```
# Root of your Git repository
infra-tf/
├── environments/
|   ├── _common/            # (Optional) Common variables/locals
|   |   └── common.tfvars
|   ├── dev/
|   |   ├── main.tf         # Instantiates modules for dev
|   |   ├── variables.tf    # Environment-specific variable declarations
|   |   ├── outputs.tf      # Environment-level outputs
|   |   └── dev.tfvars      # Values for variables specific to dev
|   ├── qa/
```

```
|   |   ├── main.tf
|   |   ├── variables.tf
|   |   ├── outputs.tf
|   |   └── qa.tfvars
|   ├── uat/
|   |   ├── ...
|   ├── staging/
|   |   ├── ...
|   └── prod/
|       ├── main.tf
|       ├── variables.tf
|       ├── outputs.tf
|       └── prod.tfvars
├── modules/
|   ├── networking/          # VPC, Subnets, Route Tables, GWs, Endpoints,
|   |   ├── main.tf
|   |   ├── variables.tf
|   |   └── outputs.tf
|   ├── iam/                 # IAM Roles, Policies, Instance Profiles
|   |   ├── main.tf
|   |   ├── variables.tf
|   |   └── outputs.tf
|   ├── security/            # Security Groups (alternative), WAF, GuardDu
|   |   ├── ...
|   ├── rds_postgres/        # RDS PostgreSQL + PostGIS specific module
|   |   ├── ...
|   ├── dynamodb_table/      # Reusable DynamoDB table module
|   |   ├── ...
|   ├── kinesis_stream/      # Kinesis Data Stream module
|   |   ├── ...
|   ├── s3_bucket/           # Secure S3 bucket module (logging, encryptic
|   |   ├── ...
|   ├── ecs_fargate_service/ # Module for a standard Fargate service
|   |   ├── ...
|   ├── lambda_function/     # Module for a standard Lambda function
|   |   ├── ...
|   ├── api_gateway/         # Module for API Gateway setup
```

```
|   |   ├── ...
|   ├── sns_topic/           # SNS Topic module
|   |   └── ... # etc. for other reusable components (ElastiCache, MSK, Re
|
├── main.tf                 # Root configuration (optional, mainly for ba
├── variables.tf            # Global variable declarations (region, etc.)
├── outputs.tf              # Global outputs
└── backend.tf              # Defines S3 remote state backend
```

Explanation:

- **`environments/`**: Contains directories for each deployment environment (dev, qa, prod, etc.). Each environment directory has its own `main.tf` that calls the reusable modules, and a `.tfvars` file containing the specific input values for that environment.
- **`modules/`**: Contains reusable Terraform modules for logical infrastructure components (e.g., a VPC setup, an RDS instance, an ECS service). This promotes Don't Repeat Yourself (DRY) principles.
- **Root Files (`main.tf`, `variables.tf`, `backend.tf`):** Define the S3 backend for remote state management and potentially global variables like the AWS region.

Step 2: Configure S3 Backend for Remote State

Terraform needs to store its state file, which maps resources to your configuration. Using an S3 backend is crucial for team collaboration and state locking, preventing conflicts.

> **Manual Step Required:** The S3 bucket and DynamoDB table for state locking must exist *before* you can initialize Terraform with this backend. Create these manually in your primary AWS GovCloud region (or via a separate, simple Terraform config applied once). Ensure the bucket has versioning and encryption enabled. The DynamoDB table needs a primary key named `LockID` (Type: String).

File: `infra-tf/backend.tf`

```
terraform {
  backend "s3" {
    # TODO: Replace with your globally unique S3 bucket name in GovCloud
    bucket         = "nycps-tms-tfstate-bucket-unique-name"
    # Key will be overridden in environment-specific backend configs
    key            = "tfstate/global.tfstate"
    # TODO: Ensure this matches the GovCloud region of your S3 bucket
    region         = "us-gov-west-1"
    # TODO: Replace with the name of your DynamoDB lock table
    dynamodb_table = "nycps-tms-tfstate-lock"
    encrypt        = true
  }
}
```

File: `infra-tf/environments/dev/main.tf` (Backend Re-configuration)

Inside each environment's `main.tf`, you re-declare the backend block but change the `key` to store state separately per environment.

```
 terraform {
  # Define required providers within the environment config too
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0" # Or latest appropriate version
```

```
    }
  }

  backend "s3" {
    # NOTE: Bucket, Region, Table MUST match the root backend.tf
    bucket         = "nycps-tms-tfstate-bucket-unique-name" # Same bucket
    key            = "tfstate/dev/terraform.tfstate"     # Environment-spe
    region         = "us-gov-west-1"             # Same region as root
    dynamodb_table = "nycps-tms-tfstate-lock"      # Same table as root
    encrypt        = true
  }
}
```

Note: Repeat the backend re-configuration block in `main.tf` for *each* environment directory (qa, prod, etc.), changing only the `key` value (e.g., `"tfstate/qa/terraform.tfstate"`).

Step 3: Develop Reusable Infrastructure Modules

Create modules for common infrastructure patterns. This involves defining resources, input variables, and outputs for each module.

Example: Networking Module (`modules/networking/`)

This module creates the VPC, subnets, routing, gateways, and baseline security groups.

File: `modules/networking/variables.tf`

```
variable "environment_name" {
  description = "Name of the environment (e.g., dev, qa, prod)"
  type        = string
}

variable "vpc_cidr" {
  description = "CIDR block for the VPC"
  type        = string
}

variable "public_subnet_cidrs" {
  description = "List of CIDR blocks for public subnets (one per AZ)"
  type        = list(string)
}

variable "private_subnet_cidrs" {
  description = "List of CIDR blocks for private subnets (one per AZ)"
  type        = list(string)
}

variable "availability_zones" {
  description = "List of Availability Zones to use"
  type        = list(string)
  # TODO: Ensure these are valid AZs in your GovCloud region
}

# TODO: Add variables for tags, specific endpoint services, etc.
```

File: `modules/networking/main.tf`

```
# --- VPC ---
resource "aws_vpc" "main" {
  cidr_block           = var.vpc_cidr
  enable_dns_support   = true
  enable_dns_hostnames = true
```

```
  tags = {
    "Name"        = "vpc-${var.environment_name}"
    "Environment" = var.environment_name
  }
}

# --- Subnets ---
resource "aws_subnet" "public" {
  count             = length(var.public_subnet_cidrs)
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.public_subnet_cidrs[count.index]
  availability_zone = var.availability_zones[count.index]
  map_public_ip_on_launch = true

  tags = {
    "Name" = "public-subnet-${var.environment_name}-${count.index}"
    # ... other tags
  }
}

resource "aws_subnet" "private" {
  count             = length(var.private_subnet_cidrs)
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.private_subnet_cidrs[count.index]
  availability_zone = var.availability_zones[count.index]

  tags = {
    "Name" = "private-subnet-${var.environment_name}-${count.index}"
     # ... other tags
  }
}

# --- Gateways (IGW, NAT) ---
resource "aws_internet_gateway" "gw" {
  vpc_id = aws_vpc.main.id
  # ... tags
}
```

```
# Allocate Elastic IPs for NAT Gateways (one per AZ for HA)
resource "aws_eip" "nat" {
  count = length(var.public_subnet_cidrs)
  vpc   = true
}

resource "aws_nat_gateway" "nat" {
  count         = length(var.public_subnet_cidrs)
  allocation_id = aws_eip.nat[count.index].id
  subnet_id     = aws_subnet.public[count.index].id

  tags = {
    "Name" = "nat-gw-${var.environment_name}-${count.index}"
  }
  # Ensure IGW is created first
  depends_on = [aws_internet_gateway.gw]
}

# --- Route Tables ---
# Public Route Table (points to IGW)
resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.gw.id
  }
  # ... tags
}
# Associate public subnets
resource "aws_route_table_association" "public" {
  count          = length(var.public_subnet_cidrs)
  subnet_id      = aws_subnet.public[count.index].id
  route_table_id = aws_route_table.public.id
}
```

```
# Private Route Tables (one per AZ, pointing to NAT GW in that AZ)
resource "aws_route_table" "private" {
  count  = length(var.private_subnet_cidrs)
  vpc_id = aws_vpc.main.id

  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat[count.index].id
  }
  # ... tags
}
# Associate private subnets
resource "aws_route_table_association" "private" {
  count          = length(var.private_subnet_cidrs)
  subnet_id      = aws_subnet.private[count.index].id
  route_table_id = aws_route_table.private[count.index].id
}

# --- Security Groups (Baseline Example) ---
resource "aws_security_group" "allow_internal" {
  name        = "allow-internal-${var.environment_name}"
  description = "Allow all internal traffic within the VPC"
  vpc_id      = aws_vpc.main.id

  ingress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1" # All protocols
    cidr_blocks = [var.vpc_cidr] # Only allow from within the VPC
  }

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

```
  # TODO: Implement more granular security groups per component
  # e.g., Web SG allowing 443 from LB, App SG allowing traffic from Web SG
}

# --- VPC Endpoints (Example: S3 Gateway Endpoint) ---
resource "aws_vpc_endpoint" "s3" {
  vpc_id            = aws_vpc.main.id
  service_name      = "com.amazonaws.${data.aws_region.current.name}.s3" #
  route_table_ids   = concat(aws_route_table.public[*].id, aws_route_table
}

data "aws_region" "current" {} # Helper to get current region

# TODO: Add Interface Endpoints for other services (Kinesis, SQS, KMS, Se
```

File: `modules/networking/outputs.tf`

```
output "vpc_id" {
  description = "The ID of the VPC"
  value       = aws_vpc.main.id
}

output "public_subnet_ids" {
  description = "List of IDs of public subnets"
  value       = aws_subnet.public[*].id
}

output "private_subnet_ids" {
  description = "List of IDs of private subnets"
  value       = aws_subnet.private[*].id
}

output "baseline_internal_sg_id" {
  description = "ID of the baseline security group allowing internal VPC t
  value       = aws_security_group.allow_internal.id
}
```

```
# TODO: Add outputs for specific Security Group IDs (Web, App, DB) once c
```

Example: RDS PostgreSQL Module (`modules/rds_postgres/`)

This module provisions an RDS PostgreSQL instance with PostGIS, configured for HA.

File: `modules/rds_postgres/variables.tf`

```
variable "environment_name" { type = string }
variable "allocated_storage" { type = number; default = 100 } # GB
variable "instance_class" { type = string; default = "db.m5.large" } # TOD
variable "db_name" { type = string; default = "nycps_tms_db" }
variable "db_subnet_group_name" { type = string } # Will be created based
variable "vpc_security_group_ids" { type = list(string) } # Pass the speci
variable "kms_key_id" { type = string } # ARN of the KMS key for encryptic
variable "multi_az" { type = bool; default = true } # Enable Multi-AZ for
variable "backup_retention_period" { type = number; default = 7 } # Days.
variable "engine_version" { type = string; default = "14" } # Specify desi

# TODO: Variables for username/password - USE SECRETS MANAGER!
variable "db_master_username_secret_arn" { type = string }
variable "db_master_password_secret_arn" { type = string }
```

File: `modules/rds_postgres/main.tf`

```
# Data source to retrieve secrets from Secrets Manager
data "aws_secretsmanager_secret_version" "db_username" {
  secret_id = var.db_master_username_secret_arn
}
data "aws_secretsmanager_secret_version" "db_password" {
  secret_id = var.db_master_password_secret_arn
}
```

```
resource "aws_db_instance" "postgres_gis" {
  identifier           = "nycps-tms-rds-${var.environment_name}"
  allocated_storage    = var.allocated_storage
  engine               = "postgres"
  engine_version       = var.engine_version
  instance_class       = var.instance_class
  db_name              = var.db_name
  username             = data.aws_secretsmanager_secret_version.db_usernam
  password             = data.aws_secretsmanager_secret_version.db_passwor
  db_subnet_group_name = var.db_subnet_group_name
  vpc_security_group_ids = var.vpc_security_group_ids
  parameter_group_name = aws_db_parameter_group.postgres_gis.name # Refere

  storage_encrypted   = true
  kms_key_id          = var.kms_key_id
  multi_az            = var.multi_az
  publicly_accessible = false
  skip_final_snapshot = false # Typically false for prod, maybe true for d
  backup_retention_period = var.backup_retention_period
  apply_immediately   = false # Apply changes during maintenance window

  # Enable PostGIS extension requires specific parameter group settings
  # Ensure CloudWatch Logs exports are enabled
  enabled_cloudwatch_logs_exports = ["postgresql", "upgrade"]

  tags = {
    "Name"        = "rds-postgres-gis-${var.environment_name}"
    "Environment" = var.environment_name
  }
}

# Parameter group to enable PostGIS
resource "aws_db_parameter_group" "postgres_gis" {
  name_prefix = "rds-pg-gis-${var.environment_name}-"
  family      = "postgres${var.engine_version}" # e.g., postgres14

  parameter {
```

```
    name  = "shared_preload_libraries"
    value = "pg_stat_statements,postgis-3" # Check exact PostGIS version r
    apply_method = "pending-reboot"
  }
  # TODO: Add other necessary parameters (logging, performance tuning)
}

# TODO: Define aws_db_subnet_group resource based on private subnet IDs
# TODO: Define specific DB Security Group allowing access only from App ti
```

File: `modules/rds_postgres/outputs.tf`

```
output "db_instance_endpoint" {
  description = "The connection endpoint for the database instance"
  value       = aws_db_instance.postgres_gis.endpoint
  sensitive   = true
}
output "db_instance_id" {
  description = "The ID of the database instance"
  value       = aws_db_instance.postgres_gis.id
}
# ... other relevant outputs ...
```

Other Modules (Conceptual Examples):

- **IAM Module (`modules/iam/`):** Define reusable resources for creating IAM roles (e.g., Lambda execution role, EC2 instance profile role, ECS task role) and attaching managed or custom IAM policies. Use variables for naming and policy specifics. Output Role ARNs.
- **S3 Bucket Module (`modules/s3_bucket/`):** Create S3 buckets with standardized configurations: versioning enabled, server-side encryption (SSE-S3 or SSE-KMS), access logging enabled, lifecycle rules (for transitions/expiration), public access blocked, appropriate bucket policies.

Use variables for bucket name prefix, KMS key, lifecycle rules. Output bucket name/ARN.

- **ECS Fargate Service Module (`modules/ecs_fargate_service/`):** Define resources for an ECS cluster (if not shared), Task Definition (referencing ECR image URI, CPU/Memory, IAM roles, environment variables/secrets), Fargate Service (desired count, subnets, security groups, load balancer target group association), Auto Scaling configuration. Use variables for image URI, CPU/Memory, environment variables, scaling thresholds, LB details.
- **Kinesis Stream Module (`modules/kinesis_stream/`):** Provision Kinesis Data Stream with variables for shard count, retention period, encryption (KMS). Output stream ARN/name.
- **DynamoDB Table Module (`modules/dynamodb_table/`):** Create DynamoDB tables with variables for table name, primary key (hash/range), attributes, billing mode (provisioned/on-demand), GSIs/LSIs, stream specification, Point-in-Time Recovery (PITR) enabled, encryption (KMS). Output table ARN/name.

> **Best Practice:** Keep modules focused on a single logical component (e.g., an RDS database, a VPC, an ECS service). Use module outputs to pass necessary information (like VPC ID, subnet IDs, security group IDs) as inputs to other modules.

Step 4: Configure Environment Variables (`.tfvars`)

Create `.tfvars` files for each environment to define the specific values for the variables declared in your modules and environment-level `variables.tf` files.

File: `infra-tf/environments/dev/variables.tf` (Example Declarations)

```
variable "aws_region" { type = string; default = "us-gov-west-1" }
variable "environment_name" { type = string }
variable "vpc_cidr" { type = string }
variable "public_subnet_cidrs" { type = list(string) }
variable "private_subnet_cidrs" { type = list(string) }
variable "availability_zones" { type = list(string) }
variable "rds_instance_class" { type = string }
variable "rds_allocated_storage" { type = number }
variable "db_master_username_secret_arn" { type = string }
variable "db_master_password_secret_arn" { type = string }
variable "kms_key_id" { type = string }
# TODO: Add declarations for all variables used in environments/dev/main.t
```

File: `infra-tf/environments/dev/dev.tfvars` (Example Values)

```
# Networking
environment_name     = "dev"
vpc_cidr             = "10.10.0.0/16"
public_subnet_cidrs  = ["10.10.1.0/24", "10.10.2.0/24", "10.10.3.0/24"]
private_subnet_cidrs = ["10.10.101.0/24", "10.10.102.0/24", "10.10.103.0/2
availability_zones   = ["us-gov-west-1a", "us-gov-west-1b", "us-gov-west-1

# RDS
rds_instance_class    = "db.t3.medium"
rds_allocated_storage = 50
kms_key_id            = "arn:aws-us-gov:kms:us-gov-west-1:ACCOUNT_ID:key/Y

# Secrets
db_master_username_secret_arn = "arn:aws-us-gov:secretsmanager:us-gov-west
db_master_password_secret_arn = "arn:aws-us-gov:secretsmanager:us-gov-west
```

```
# TODO: Define values for all other variables needed by the modules called
# e.g., Kinesis shard counts, ECS task definitions, specific security grou
```

Security: NEVER commit `.tfvars` files containing sensitive information (like passwords or API keys) directly to version control. Use a secure method like AWS Secrets Manager, HashiCorp Vault, environment variables injected by the CI/CD system, or SOPS for managing secrets. The example above uses ARNs pointing to Secrets Manager secrets, which is a recommended approach.

Note: Create similar `.tfvars` files (e.g., `qa.tfvars`, `prod.tfvars`) in the respective environment directories, adjusting values like CIDR blocks, instance sizes, scaling parameters, KMS keys, and secret ARNs as needed for each environment.

Step 5: Instantiate Modules Per Environment

In each environment's `main.tf` file, call the reusable modules defined in Step 3, passing in the environment-specific configuration values via variables (which will be loaded from the corresponding `.tfvars` file).

File: `infra-tf/environments/dev/main.tf` (Example Instantiation)

```
# Backend re-configuration block (from Step 2) should be here
terraform { ... }

# Provider block ensures resources are created in the correct region
provider "aws" {
```

```
  region = var.aws_region # Using variable defined in environments/dev/var
}

# --- Networking ---
module "networking" {
  source             = "../../modules/networking" # Relative path to the m

  # Pass variables defined in dev.tfvars (or defaults)
  environment_name   = var.environment_name
  vpc_cidr           = var.vpc_cidr
  public_subnet_cidrs= var.public_subnet_cidrs
  private_subnet_cidrs= var.private_subnet_cidrs
  availability_zones = var.availability_zones
  # ... other networking variables ...
}

# --- RDS Database (depends on networking) ---
# First, create the DB subnet group using the private subnet IDs output by
resource "aws_db_subnet_group" "rds_subnet_group" {
  name       = "sng-rds-${var.environment_name}"
  subnet_ids = module.networking.private_subnet_ids
  # ... tags
}

# TODO: Define a specific Security Group for RDS access
resource "aws_security_group" "rds_sg" {
   name        = "sg-rds-${var.environment_name}"
   description = "Allow PostgreSQL access from App tier"
   vpc_id      = module.networking.vpc_id

   ingress {
     description       = "PostgreSQL from App Security Group"
     from_port         = 5432
     to_port           = 5432
     protocol          = "tcp"
     # TODO: Reference the Security Group ID of your Application tier
     security_groups   = ["sg-xxxxxxxxxxxxxxxxx"] # Replace with actual Ap
```

```
    }
    # Egress typically allows all outbound
     egress {
       from_port   = 0
       to_port     = 0
       protocol    = "-1"
       cidr_blocks = ["0.0.0.0/0"]
    }
     # ... tags
}

module "rds_postgres_main" {
  source                 = "../../modules/rds_postgres"

  environment_name       = var.environment_name
  instance_class       = var.rds_instance_class
  allocated_storage    = var.rds_allocated_storage
  db_subnet_group_name = aws_db_subnet_group.rds_subnet_group.name
  vpc_security_group_ids = [aws_security_group.rds_sg.id]
  kms_key_id             = var.kms_key_id
  db_master_username_secret_arn = var.db_master_username_secret_arn
  db_master_password_secret_arn = var.db_master_password_secret_arn
  # ... other RDS variables ...
}

# --- Other Modules ---
# Instantiate modules for S3 buckets, IAM roles, Kinesis, DynamoDB, ECS/Fa
# Lambda functions, API Gateway, SNS topics, etc., passing necessary varia
# and referencing outputs from other modules (like VPC ID, subnet IDs, SG

module "data_ingest_stream" {
  source      = "../../modules/kinesis_stream"
  stream_name = "gps-data-stream-${var.environment_name}"
  shard_count = 2 # TODO: Parameterize based on environment load
  # ...
}
```

```
module "location_table" {
  source     = "../../modules/dynamodb_table"
  table_name = "RealTimeBusLocation-${var.environment_name}"
  hash_key   = "DeviceId"
  # ... define attributes, keys, throughput/billing mode ...
}

# ... etc. for all components defined in the architecture
```

Step 6: Initialize, Plan, and Apply Terraform

Run Terraform commands within the specific environment directory to manage that environment's infrastructure.

1. **Navigate to Environment Directory:**

   ```
   cd infra-tf/environments/dev
   ```

2. **Initialize Terraform:** Downloads provider plugins and configures the backend. Run this once per environment directory initially, and if you add new modules or change backend configuration.

   ```
   terraform init
   ```

3. **(Optional) Select Workspace:** If using Terraform workspaces instead of separate directories per environment (less common with module structure shown but possible), select the workspace:

   ```
   terraform workspace select dev || terraform workspace new
   ```

4. **Plan Changes:** Terraform determines what actions are needed to achieve the desired state defined in your `.tf` files, using variables from the specified `.tfvars` file. Review the plan carefully.

```
terraform plan -var-file="dev.tfvars" -out="tfplan.out"
```

> **Note:** The `-out="tfplan.out"` saves the plan to a file. This is best practice, especially in CI/CD, to ensure that what you `apply` is exactly what you `plan`ned.

5. **Apply Changes:** Executes the actions proposed in the plan file.

```
terraform apply "tfplan.out"
```

   Terraform will prompt for confirmation before making changes unless the `-auto-approve` flag is used (use with extreme caution, mainly in automated pipelines after thorough review).

6. **Repeat for Other Environments:** Follow steps 1-5 for `qa`, `prod`, etc., using their respective directories and `.tfvars` files (e.g., `cd ../qa`, `terraform init`, `terraform plan -var-file="qa.tfvars" ...`).

Step 7: Integrating Auxiliary Scripts & Application Code Deployment

Terraform primarily manages infrastructure resources. Application code deployment and some configurations are typically handled *after* Terraform runs, often orchestrated by the CI/CD pipeline.

- **Application Code:** Use AWS CodeDeploy, ECS/EKS deployment mechanisms (triggered by CodePipeline), Lambda deployment packages (managed by CI/CD or Terraform's `aws_lambda_function` resource pointing to an S3 artifact), or manual deployment scripts to deploy your application code onto the provisioned infrastructure (EC2, Fargate, Lambda).
- **Database Migrations:** Use database migration tools (like Flyway, Liquibase, or framework-specific tools like Alembic/Django Migrations). These scripts should be version-controlled and executed as a step in your CI/CD pipeline *after* the database infrastructure is provisioned/updated by Terraform, but *before* the application requiring the new schema is fully deployed or receives traffic.
- **EC2 User Data / Configuration Management:** For EC2 instances (if used, e.g., for the routing engine or ArcGIS), use EC2 User Data scripts (defined within the Terraform `aws_instance` or `aws_launch_template` resource) to bootstrap instances (install agents, configure base settings). For more complex configuration, consider AWS Systems Manager State Manager or tools like Ansible/Chef/Puppet, potentially triggered after instance creation.
- **Terraform Provisioners (`local-exec`, `remote-exec`):** Use these sparingly. They run scripts locally or remotely during `terraform apply`. They can be brittle and make state management complex. Prefer integrating script execution into your CI/CD pipeline or using EC2 User Data/Cloud-Init where possible.

Step 8: Securely Managing Secrets

Never store secrets directly in Terraform code or `.tfvars` files committed to version control.

- **AWS Secrets Manager / Parameter Store (SecureString):**
    - Create secrets manually in the AWS console or via AWS CLI/SDK in a secure manner (e.g., from a secure workstation or CI/CD secret variable).
    - Store database passwords, API keys, third-party credentials, etc.
    - Configure fine-grained IAM permissions controlling who/what can access these secrets.
    - In Terraform, use `data` sources (like `aws_secretsmanager_secret_version`) to fetch secret values at *plan/apply time* and pass them to resource configurations (e.g., RDS password, Lambda environment variables referencing secrets). See RDS module example above.
    - Alternatively, and often preferred for applications, grant the application's IAM Role permission to read specific secrets directly from Secrets Manager/Parameter Store at *runtime* using the AWS SDK. This avoids exposing the secret even in the Terraform state file or plan output.
- **Environment Variables (CI/CD):** For secrets needed *only* during the CI/CD process itself (e.g., API tokens for interacting with external services during build), use the secure environment variable or secrets management features of your CI/CD platform (CodeBuild secrets, Jenkins Credentials, GitLab CI variables).

Step 9: Iteration and Maintenance

Infrastructure evolves. Use the same workflow to manage changes:

1. Modify your Terraform code (`.tf` files in modules or environment configurations) or variable values (`.tfvars` files).
2. Navigate to the relevant environment directory (`cd environments/dev`).
3. Run `terraform plan -var-file="dev.tfvars" -out="tfplan.out"` to see the planned changes. Carefully review the output.
4. Run `terraform apply "tfplan.out"` to apply the reviewed changes.
5. Commit all changes (code and `.tfvars` if appropriate) to version control.

> **State Locking:** The DynamoDB table configured in the backend ensures that only one person or CI/CD job can run `terraform apply` on a specific environment's state at a time, preventing corruption.

Important Considerations & Next Steps

- **GovCloud Region:** Ensure all `provider` blocks and service endpoint references consistently use the correct AWS GovCloud region (`us-gov-west-1` or `us-gov-east-1`).
- **Service Availability:** Double-check that all desired AWS services and specific features (e.g., certain instance types, specific Kinesis/MSK features, AWS Location Service) are available in your chosen GovCloud region.
- **Detailed IAM Policies:** The example IAM policies are placeholders. Define granular, least-privilege policies for every role based on the specific actions each component needs to perform.
- **Security Group Rules:** The example security groups are basic. Define specific, strict ingress and egress rules for each application tier and service

based on required communication paths. Deny all by default.

- **Cost Estimation:** Use the AWS Pricing Calculator (specifically for GovCloud regions) to estimate costs based on chosen instance types, storage amounts, data transfer, etc., defined in your `.tfvars` files.
- **Testing IaC:** Develop a strategy for testing Terraform modules and configurations (e.g., using Terratest, pre-commit hooks with `terraform validate` and `tflint`, deploying to temporary test environments).
- **CI/CD Integration:** Integrate the Terraform plan and apply steps into your CI/CD pipeline for automated environment provisioning and updates, including necessary approval stages for production deployments.
- **Senior Engineer Review:** ALL infrastructure code, especially IAM policies and security group rules, MUST be reviewed by senior cloud/security engineers before applying to any environment, especially production.