1. **Create the folders** manually.
2. **Copy these configs**.
3. **Write dummy code** (e.g., Python worker just sleeps for 5 seconds to simulate work).
4. Run `docker-compose up --build` and watch the orchestration magic happen.

Would you like the **dummy code** for the Python Worker and Node API so you can actually run this right now?

Terraform ...ourse (11)
PDF

we will do k8 later, for now we go terraform

This is a smart pivot. Terraform is the industry standard for **Infrastructure as Code (IaC)**. If Docker creates the "house" (container), Terraform buys the land, pours the concrete, and builds the walls (AWS/Azure/GCP infrastructure).

I have analyzed your **11th PDF (Terraform Course)**. It is incredibly detailed ("God-Level Notes"), covering everything from basic HCL syntax to advanced state manipulation.

However, just like with Docker, there is a difference between **"knowing the commands"** (Tutorial Mode) and **"managing infrastructure without getting fired"** (Professional Mode).

Let's harden your skills. We will start with the most dangerous part of Terraform: **The State File.**

🚀 Phase 1: The State File (The "Nuclear Codes")

In your notes (Source 1654), you learned that `terraform.tfstate` maps your real-world resources to your configuration.

🟢 Level 1: The Tutorial Way (Local State)

You run `terraform init` and `terraform apply`. Terraform creates a `terraform.tfstate` file on your laptop.

- **Why this is fine for learning:** It's fast and simple.
- **Why this gets you fired:**
  1. **The "Bus Factor":** If your laptop breaks (or you leave the company), the state is lost. The infrastructure becomes "orphaned" and unmanageable.
  2. **Race Conditions:** If you and I both run `terraform apply` at the same time, we corrupt the file.

🔴 Level 2: The "Professional" Architecture (Remote Backend + Locking)

Your notes mention **S3 Backends** (Source 2063), but let's clarify the **Production Standard**.

1. The Storage (S3) We store the state in a centralized AWS S3 bucket.

- **Pro Tip:** Enable **Versioning** on this bucket. If Terraform corrupts your state file (it happens), you can "undelete" or roll back to yesterday's state file via S3.

2. The Lock (DynamoDB) Storage isn't enough. We need a traffic cop. We use an AWS DynamoDB table to handle **State Locking**.

- **How it works:** When you run `terraform apply`, Terraform sends a "Lock ID" to DynamoDB. If your colleague runs `apply`, Terraform checks DynamoDB, sees the lock, and says: *"Error: State locked by User A."*

The Pro Configuration (`backend.tf`):

Terraform

```
terraform {
  backend "s3" {
    bucket         = "company-terraform-state-prod"
    key            = "payment-service/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true  # Always encrypt at rest!
    dynamodb_table = "terraform-locks" # The traffic cop
  }
}
```

🚀 Phase 2: Directory Structure (The "Blast Radius")

Your notes cover Modules (Source 3276), which is great. But how do you organize them?

🟢 Level 1: The Monolith

You put your VPC, EC2, Database, and S3 config all in one folder with one `main.tf` and one state file.

- **The Risk:** You want to change a Tag on an S3 bucket. You run `terraform apply`. Terraform checks *everything* (Database, VPC, etc.). If you made a typo in the VPC code, you might accidentally destroy the Production Network while trying to update a bucket.

🔴 Level 2: The "Professional" Isolation (Environments)

We separate environments to limit the **Blast Radius**. If you break "Dev", "Prod" should not even feel it.

The Structure:

Plaintext

```
/infrastructure
├── /modules          # Reusable blueprints (VPC, EC2)
│   ├── /vpc
│   └── /server
├── /env              # Live Environments
│   ├── /dev          # Uses modules, has its OWN state file
│   │   └── main.tf
│   └── /prod         # Uses modules, has its OWN state file
│       └── main.tf
```

The Rule: Dev and Prod **never** share a state file. They are totally separate universes.

🚀 Phase 3: Safety Guards (Lifecycle Rules)

In your notes (Source 2184), you touched on `prevent_destroy`. Let's operationalize this.

🟢 Level 1: Trusting the user

You run `terraform destroy` to clean up. You accidentally type "yes" too fast.

- **Result:** You just deleted the Production Database.

🔴 Level 2: The "Professional" Seatbelts

For stateful resources (Databases, S3 buckets with data), we explicitly forbid destruction in the code.

The Code:

Terraform

```
resource "aws_db_instance" "prod_db" {
  allocated_storage = 10
  engine            = "mysql"

  # The Seatbelt
  lifecycle {
    prevent_destroy = true
  }
}
```

The Effect: If you run `terraform destroy`, Terraform will error out and **refuse** to delete this resource. You physically have to go into the code, remove those 3 lines, apply, and *then* destroy. It forces you to think twice.

🧠 Quick Quiz (Terraform Edition)

Let's test your grasp of these professional concepts based on your notes.

1. **The Scenario:** You are using an S3 Backend. You run `terraform apply`, but your internet cuts out in the middle. You get connection back, try to run `apply` again, but Terraform says: **"Error: State locked."**
   - **Why is it locked?** (Who locked it?)
   - **How do you fix this?** (Do you delete the S3 file? Or do you touch DynamoDB?)
2. **The Scenario:** You are deploying an EC2 instance. You want to install Nginx *automatically* when the server turns on.
   - In the AWS Console, we use "User Data".
   - In Terraform, we can use the `user_data` argument.
   - **The Professional Question:** Instead of writing a massive shell script inside your `.tf` file (hard to read), how can you keep the script in a separate file (e.g., `install_nginx.sh`) and inject it into the Terraform resource? (Hint: There is a specific Terraform Function mentioned in your notes Source 2963).

Give me your answers!

1.the terraform code locks it, wait to fix?
2.idk

No worries! This is exactly why we test scenarios—to catch these nuances before they happen in a live production environment.

Let's break down the **Professional answers** so you can lock this knowledge in.

📝 Quiz Review

1. The Scenario: Internet cuts out during `apply`. Terraform says "State Locked."

- **Your Answer:** "terraform code locks it, wait to fix?"
- **The Professional Reality:**
  - **Who locked it?** Not the code. **DynamoDB** locked it. When Terraform started, it wrote a "Lock ID" entry into the DynamoDB table to warn others.
  - **Why waiting fails:** If your internet cut out, your local Terraform process "died." It never got the chance to send the "Unlock" signal to DynamoDB. That lock will sit there *forever* until you intervene.
  - **The Fix:** You must manually break the lock.

- Command: `terraform force-unlock <LOCK_ID>`.
- *Warning:* Only do this if you are 100% sure no other process is actually running.

2. The Scenario: Injecting a shell script into `user_data` without cluttering the `.tf` file.

- **Your Answer:** "idk"
- **The Professional Answer:** The `file()` Function.
  - Instead of pasting a 50-line bash script inside your HCL code (which looks messy), you put the script in `scripts/install_nginx.sh`.
  - In Terraform: `user_data = file("scripts/install_nginx.sh")`.
  - Terraform reads the text from that file and injects it during the apply. Clean and readable.

🏗️ The Major Project: "Terraform Titan" (AWS 3-Tier Architecture)

Since you wanted a "Major Project" for Docker, let's do the equivalent for Terraform. We are going to build a **Production-Grade 3-Tier Architecture** on AWS.

This project covers: **Modules, Remote State, VPC Networking, Security Groups, EC2, and RDS.**

📁 1. The Professional Folder Structure

We stop using one big `main.tf`. We split logic into **Modules** (reusable blueprints) and **Environments** (live deployments).

Plaintext

```
/terraform-titan
├── /modules                # The Blueprints (Reusable)
│   ├── /vpc                # Network Module
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   ├── /db                 # Database Module (RDS)
│   └── /web                # Web Server Module (EC2 + SG)
│
├── /env                    # The Live Infrastructure
│   └── /dev                # Development Environment
│       ├── main.tf         # Calls the modules above
│       ├── variables.tf
│       ├── outputs.tf
│       └── terraform.tfvars # Dev-specific settings (e.g., t2.micro)
└── .gitignore              # Ignore .terraform and .tfstate files
```

🛠️ 2. Step 1: The VPC Module (The Foundation)

We don't use the default VPC. We build a custom one.

File: `modules/vpc/main.tf`

Terraform

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

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

```
  }
}

# Public Subnet (Accessible to Internet)
resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.main.id
  cidr_block              = var.public_subnet_cidr
  map_public_ip_on_launch = true # Auto-assign IP
  availability_zone       = var.az
}

# Internet Gateway (The door to the internet)
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id
}

# Route Table (The map to the door)
resource "aws_route_table" "public_rt" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
}

# Associate Subnet with Route Table
resource "aws_route_table_association" "public_assoc" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public_rt.id
}
```

💻 3. Step 2: The Web Module (Compute)

This creates the server. Notice how we use `user_data` with the `file()` function we just learned.

File: `modules/web/main.tf`

Terraform

```
resource "aws_security_group" "web_sg" {
  name        = "${var.project_name}-web-sg"
  description = "Allow HTTP"
  vpc_id      = var.vpc_id # Passed in from the VPC module output!

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"] # Open to world
  }

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

resource "aws_instance" "web_server" {
  ami           = var.ami_id
  instance_type = var.instance_type
  subnet_id     = var.subnet_id # Passed from VPC module

  vpc_security_group_ids = [aws_security_group.web_sg.id]
```

```
  # PRO TIP: Inject script efficiently
  user_data = file("${path.module}/scripts/install_nginx.sh")

  tags = {
    Name = "${var.project_name}-web"
  }
}
```

🚀 4. Step 3: The "Dev" Environment (Putting it together)

This is where we actually build the infrastructure by "calling" our modules.

File: `env/dev/main.tf`

Terraform

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

# 1. Create the Network
module "my_vpc" {
  source             = "../../modules/vpc"
  vpc_cidr           = "10.0.0.0/16"
  public_subnet_cidr = "10.0.1.0/24"
  project_name       = "titan-dev"
  az                 = "us-east-1a"
}

# 2. Create the Server (Connecting it to the Network)
module "my_web" {
  source        = "../../modules/web"
  project_name  = "titan-dev"
  instance_type = "t2.micro"
  ami_id        = "ami-0c7217cdde317cfec" # Ubuntu 22.04 (US-East-1)

  # MAGIC: We pass the OUTPUT of the VPC module as the INPUT to the Web module
  vpc_id        = module.my_vpc.vpc_id_out
  subnet_id     = module.my_vpc.public_subnet_id_out
}
```

🚦 Your Next Steps

This project forces you to understand **Inputs**, **Outputs**, and **Module Composition** (how modules talk to each other).

1. **Create the folder structure.**
2. **Copy the code** for the VPC module first.
3. **Initialize (** `terraform init` **)** inside `env/dev` and try to create *just* the network.

Ready to start building the VPC module? Or do you want me to explain how `module.my_vpc.vpc_id_out` works first?

enxt

Awesome! Let's break ground on **Terraform Titan**. 🏗️

We are starting with the **VPC Module** (The Network). This is the "land" where your servers and databases will live.

In your notes (Lecture 9), you learned about **Object Variables**. We are going to use that immediately because it is the "Professional" way to group related configuration. Instead of passing `var.vpc_name`, `var.vpc_cidr`, etc., we will pass one single `var.vpc_config` object.

📂 Step 1: The Folder Structure

Make sure you have this folder ready:

Plaintext

```
/terraform-titan
└── /modules
    └── /vpc
        ├── main.tf
        ├── variables.tf
        └── outputs.tf
```

📝 Step 2: The Inputs (`modules/vpc/variables.tf`)

The Pro Move: We use `type = object(...)`. This creates a strict "Contract". If a user tries to use this module without providing a CIDR block, Terraform fails immediately (which is good!).

Terraform

```
# modules/vpc/variables.tf

variable "vpc_config" {
  description = "Configuration for the VPC"
  type = object({
    name       = string
    cidr_block = string
    az         = string # Availability Zone (e.g., "us-east-1a")
  })
}

variable "subnet_cidrs" {
  description = "CIDR blocks for subnets"
  type = object({
    public  = string
    private = string
  })
}
```

⚙️ Step 3: The Logic (`modules/vpc/main.tf`)

Here we build the Virtual Private Cloud.

The Pro Move: Notice the **Tags**. In a real company, untagged resources get deleted by "Janitor Bots." We tag *everything* with the Project Name.

Terraform

```
# modules/vpc/main.tf

# 1. The VPC (The walled garden)
resource "aws_vpc" "main" {
  cidr_block           = var.vpc_config.cidr_block
  enable_dns_support   = true
  enable_dns_hostnames = true
```

```
  tags = {
    Name    = "${var.vpc_config.name}-vpc"
    Project = "Terraform-Titan"
  }
}

# 2. The Internet Gateway (The front door)
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name = "${var.vpc_config.name}-igw"
  }
}

# 3. Public Subnet (Where the Web Server lives)
resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.main.id
  cidr_block              = var.subnet_cidrs.public
  availability_zone       = var.vpc_config.az
  map_public_ip_on_launch = true # Auto-assign Public IP

  tags = {
    Name = "${var.vpc_config.name}-public-subnet"
  }
}

# 4. Private Subnet (Where the DB lives - Hidden from internet)
resource "aws_subnet" "private" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.subnet_cidrs.private
  availability_zone = var.vpc_config.az

  tags = {
    Name = "${var.vpc_config.name}-private-subnet"
  }
}

# 5. Public Route Table (The map to the Internet Gateway)
resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0" # All traffic
    gateway_id = aws_internet_gateway.igw.id
  }

  tags = {
    Name = "${var.vpc_config.name}-public-rt"
  }
}

# 6. Association (Connecting the Subnet to the Route Table)
resource "aws_route_table_association" "public" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}
```

📤 Step 4: The Exports (`modules/vpc/outputs.tf`)

The Pro Move: A module is useless if it keeps secrets. We need to export the IDs so our **Web Server Module** (Step 2) knows where to launch.

Terraform

```
# modules/vpc/outputs.tf

output "vpc_id" {
  value = aws_vpc.main.id
}

output "public_subnet_id" {
  value = aws_subnet.public.id
}

output "private_subnet_id" {
  value = aws_subnet.private.id
}
```

🚦 Checkpoint: "Did I build it right?"

You cannot run `terraform apply` inside the `modules/` folder directly (modules are just blueprints). You need to call them from an **Environment**.

Action Item:

1. Create these 3 files.
2. Go to your root folder and run `terraform fmt -recursive`. (This formats your code to look professional).
3. **Tell me when you are done**, and we will create the **Dev Environment (** `env/dev/main.tf` **)** to actually turn this blueprint into a real AWS Network! 🚀

next

Great! Now that we have the blueprint (Module), let's build the actual "house" (The Environment).

This is where the magic of **Infrastructure as Code** happens. We are going to create a **Development Environment** that uses your VPC module.

📂 Step 1: Create the Environment Folder

Create this folder structure inside your project root:

Plaintext

```
/terraform-titan
└── /env
    └── /dev
        ├── main.tf
        └── outputs.tf
```

(Note: We are skipping `variables.tf` *for the environment right now to keep it simple, but in a real job, you'd have one here too).*

📝 Step 2: The Logic (`env/dev/main.tf`)

This is the "Consumer" code. It calls the module we just wrote.

The Pro Move: Notice how clean this file is. We don't see `aws_vpc` or `aws_subnet` resources here. We just see **Business Logic**. We are telling Terraform *what* we want ("A VPC named titan-dev"), not *how* to build it (that's hidden in the module).

Terraform

```
# env/dev/main.tf

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

module "titan_network" {
  # 1. Source: Where is the blueprint?
  source = "../../modules/vpc"

  # 2. vpc_config (The Object Variable we defined)
  vpc_config = {
    name       = "titan-dev"
    cidr_block = "10.0.0.0/16"
    az         = "us-east-1a"
  }

  # 3. subnet_cidrs (The other Object Variable)
  subnet_cidrs = {
    public  = "10.0.1.0/24"
    private = "10.0.2.0/24"
  }
}
```

📤 Step 3: The Verification (`env/dev/outputs.tf`)

We need to see what we built. Since the module exports the IDs, we need to "re-export" them here so we can see them in our terminal.

Terraform

```
# env/dev/outputs.tf

output "vpc_id" {
  description = "The ID of the Dev VPC"
  value       = module.titan_network.vpc_id
}

output "public_subnet_id" {
  description = "The ID of the Public Subnet (For Web Servers)"
  value       = module.titan_network.public_subnet_id
}
```

🚀 Step 4: Launch Time! (The "Apply")

Now, let's make it real. Open your terminal and navigate to the dev environment:

1. **Go to the folder:**

   Bash

   ```
   cd env/dev
   ```

2. **Initialize (Download the AWS Provider & Initialize Module):**

   Bash

```
terraform init
```

(You should see: "Initializing modules... module.titan_network")

3. **Plan (The Dry Run - Always do this!):**

```bash
terraform plan
```

(You should see "Plan: 6 to add, 0 to change, 0 to destroy")

4. **Apply (Build it):**

```bash
terraform apply
```

(Type `yes` when prompted)

🧠 Quick Quiz: The "State" Trap

While that is building, let's test your knowledge of **State Files** based on what you just did.

You just ran `terraform apply` inside `env/dev/`.

1. **Where is the `terraform.tfstate` file created?** (In `modules/vpc/` or `env/dev/`?)
2. **Scenario:** You want to create a **Production** environment (`env/prod/`). You copy `main.tf` to that folder and change the CIDR block.
   - Will running `apply` in `env/prod/` destroy your Dev VPC? **Why or why not?**

Give me your answers! (Once the apply finishes, you should see green outputs with your new VPC ID).

1.env/dev/
2.No, it will not.tf only manages resources in the state file

2/2 Correct! You nailed the logic. 🎯

- **Answer 1:** Correct. `env/dev/terraform.tfstate` is created in the working directory.
- **Answer 2:** Correct. Separate folders = Separate State Files = Separate Worlds. Terraform in `prod` has no idea that `dev` resources even exist because they aren't in its state file (the "ledger"). This is exactly how we limit the "Blast Radius."

🏗️ Phase 2: The Web Server Module (Compute)

Now we need to put something *inside* that network you just built. We are going to launch an EC2 instance running Nginx.

The Professional Challenge: This module needs to know *where* to launch (VPC ID & Subnet ID). It cannot guess. We have to **inject** those values from the VPC module's outputs.

📂 Step 1: Create the Web Module Structure

Plaintext

```
/terraform-titan
└── /modules
    └── /web
        ├── main.tf
        ├── variables.tf
        ├── outputs.tf
        └── /scripts            <-- New folder!
            └── install_nginx.sh
```

📜 Step 2: The Startup Script (`modules/web/scripts/install_nginx.sh`)

This is a standard Bash script. AWS runs this automatically when the server boots (User Data).

Bash

```bash
#!/bin/bash
sudo apt-get update
sudo apt-get install -y nginx
echo "<h1>Deployed via Terraform Titan</h1>" | sudo tee /var/www/html/index.html
sudo systemctl start nginx
sudo systemctl enable nginx
```

📝 Step 3: The Inputs (`modules/web/variables.tf`)

Notice we are asking for `vpc_id` and `subnet_id`. These are **mandatory**.

Terraform

```terraform
variable "project_name" { type = string }
variable "instance_type" { type = string }
variable "ami_id" { type = string }

# Dependency Injection: We need these from the Networking module
variable "vpc_id" { type = string }
variable "subnet_id" { type = string }
```

⚙️ Step 4: The Logic (`modules/web/main.tf`)

Here we create the **Security Group** (Firewall) and the **Instance**.

The Pro Move: Notice `file("${path.module}/scripts/...")`. This tells Terraform: *"Look inside the module folder, read the text from that script, and pass it to AWS."*

Terraform

```terraform
# 1. Security Group (Firewall)
resource "aws_security_group" "web_sg" {
  name        = "${var.project_name}-web-sg"
  description = "Allow HTTP and SSH"
  vpc_id      = var.vpc_id # Injected from input

  # Allow HTTP from anywhere
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

```
  # Allow outbound traffic (so it can run apt-get update)
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "${var.project_name}-web-sg"
  }
}

# 2. The Server
resource "aws_instance" "server" {
  ami                         = var.ami_id
  instance_type               = var.instance_type
  subnet_id                   = var.subnet_id # Injected from input
  vpc_security_group_ids      = [aws_security_group.web_sg.id]
  associate_public_ip_address = true

  # Read the script file and pass it to the server
  user_data = file("${path.module}/scripts/install_nginx.sh")

  tags = {
    Name = "${var.project_name}-server"
  }
}
```

📤 Step 5: The Exports (`modules/web/outputs.tf`)

We need the Public IP so we can visit our website!

Terraform

```
output "public_ip" {
  value = aws_instance.server.public_ip
}
```

🚀 Phase 3: Wiring it Together (`env/dev/main.tf`)

This is the most important part. Go back to `env/dev/main.tf` and **add the Web Module**.

We use the syntax `module.NAME.OUTPUT_NAME` to pass data from the Network to the Server.

Terraform

```
# ... (Your existing VPC module block is here) ...

# Add this BELOW the VPC module
module "titan_web" {
  source        = "../../modules/web"
  project_name  = "titan-dev"
  instance_type = "t2.micro"

  # AMI for Ubuntu 24.04 LTS (US-East-1)
  # Note: Verify this ID for your specific region if not using us-east-1
  ami_id        = "ami-04b70fa74e45c3917"

  # WIRING: Pass outputs from 'titan_network' to inputs of 'titan_web'
  vpc_id    = module.titan_network.vpc_id
```

```
  subnet_id = module.titan_network.public_subnet_id
}
```

And update `env/dev/outputs.tf` so you can see the IP:

Terraform

```
output "website_url" {
  value = "http://${module.titan_web.public_ip}"
}
```

🚦 Final Launch

1. **Run** `terraform init` again (because you added a new module).
2. **Run** `terraform apply`.
3. Wait for it to finish. You will get a `website_url`.
4. **Click that link.** (It might take 1-2 minutes for Nginx to install after the server starts).

Tell me when you see "Deployed via Terraform Titan" in your browser!

next'

This is the moment where all your hard work pays off. You spent time building "Blueprints" (Modules). Now, we are going to prove why that was worth it.

We are going to launch a **Production Environment** in less than 2 minutes. ⏱️

🏗️ Phase 4: The Production Environment (`env/prod`)

In the "Old Way," creating a Production environment meant manually clicking through the AWS console for hours or copy-pasting 500 lines of Terraform code.

The Titan Way: We just create a new folder and call the modules again with different settings.

📂 Step 1: The Prod Folder

Create this structure:

Plaintext

```
/terraform-titan
└── /env
    └── /prod          <-- New!
        ├── main.tf
        └── outputs.tf
```

📝 Step 2: The Logic (`env/prod/main.tf`)

Copy the code below. Notice how we change the **CIDR Block** (to avoid IP conflicts if we peer networks later) and the **Instance Type** (Prod usually gets bigger servers).

Terraform

```
# env/prod/main.tf

provider "aws" {
```

```
  region = "us-east-1"
}

# 1. Create the Prod Network (Different IP Range!)
module "titan_network" {
  source = "../../modules/vpc"

  vpc_config = {
    name       = "titan-prod"
    cidr_block = "10.1.0.0/16" # <--- Changed from 10.0.0.0/16
    az         = "us-east-1a"
  }

  subnet_cidrs = {
    public  = "10.1.1.0/24"    # <--- Changed IP Range
    private = "10.1.2.0/24"    # <--- Changed IP Range
  }
}

# 2. Create the Prod Server (Bigger & Better)
module "titan_web" {
  source = "../../modules/web"

  project_name  = "titan-prod"
  instance_type = "t3.medium"   # <--- Upgrade from t2.micro!
  ami_id        = "ami-04b70fa74e45c3917"

  # Injection
  vpc_id    = module.titan_network.vpc_id
  subnet_id = module.titan_network.public_subnet_id
}
```

(Copy `outputs.tf` from `env/dev/outputs.tf` to `env/prod/outputs.tf` as well so you can get the IP).

🚀 Step 3: Launch Prod

1. `cd env/prod`
2. `terraform init`
3. `terraform apply -auto-approve`

Boom. You just replicated your entire infrastructure architecture in a new environment with zero code duplication.

🔒 Phase 5: The "Professional" Upgrade (Remote Backend)

Right now, your `terraform.tfstate` file is sitting on your laptop.

- **If your laptop crashes:** You lose control of your cloud.
- **If you work in a team:** You overwrite each other's work.

We need to move that state file to **AWS S3** and lock it with **DynamoDB**.

🐣 The "Chicken and Egg" Problem

We need an S3 bucket to store our Terraform State. But... we should use Terraform to create that S3 bucket. But... where do we store the state for *that* Terraform code?

The Solution: We create a **Global Bootstrap** project first (Local State), and then the Environments use it (Remote State).

📁 Step 1: The Global Folder

Plaintext

```
/terraform-titan
└── /global
    └── /s3
        ├── main.tf
        └── outputs.tf
```

⚙️ Step 2: The Logic (`global/s3/main.tf`)

This creates the "Vault" for your state files.

Terraform

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

# 1. The Bucket (Where the files live)
resource "aws_s3_bucket" "terraform_state" {
  bucket = "titan-terraform-state-storage" # Must be globally unique! Change this name slightly.

  # Prevent accidental deletion (The Safety Guard)
  lifecycle {
    prevent_destroy = true
  }
}

# 2. Enable Versioning (The "Undo" Button)
resource "aws_s3_bucket_versioning" "enabled" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
    status = "Enabled"
  }
}

# 3. Server-Side Encryption (Security)
resource "aws_s3_bucket_server_side_encryption_configuration" "default" {
  bucket = aws_s3_bucket.terraform_state.id

  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}

# 4. DynamoDB Table (The Lock)
resource "aws_dynamodb_table" "terraform_locks" {
  name         = "titan-terraform-locks"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

🚀 Step 3: Deploy the Backend

1. `cd global/s3`
2. `terraform init`
3. `terraform apply`

4. **Copy the S3 Bucket Name** and **DynamoDB Table Name** from your outputs/console.

🔄 Phase 6: The Great Migration

Now we tell our **Dev Environment** to stop using your laptop and start using the Cloud.

File: `env/dev/backend.tf` (Create this file)

Terraform

```
terraform {
  backend "s3" {
    # Replace with YOUR bucket name from Phase 5
    bucket         = "titan-terraform-state-storage"

    # The path inside the bucket (Use a different key for prod!)
    key            = "env/dev/terraform.tfstate"

    region         = "us-east-1"

    # Replace with YOUR table name
    dynamodb_table = "titan-terraform-locks"
    encrypt        = true
  }
}
```

The Final Action:

1. `cd env/dev`
2. Run `terraform init`.
3. Terraform will detect the change and ask: **"Do you want to copy existing state to the new backend?"**
4. Type `yes`.

Congratulations. You now have a production-grade, modular, remote-state-backed infrastructure setup.

This is exactly what I would expect to see in a Senior DevOps Engineer's portfolio.

🧹 The Cleanup (Important!)

Since `t3.medium` costs money, don't forget to tear down when you are done showing off!

1. `cd env/prod` -> `terraform destroy`
2. `cd env/dev` -> `terraform destroy` *(Note: Do NOT destroy the `global/s3` folder yet if you plan to use this again later, or you'll lose your state history).*

How does that feel? You basically just built a mini-AWS architecture firm.

next

This is the final polish. You have a Production Environment, but it's still "at risk" because it’s using a local state file, and it has a "Hardcoded AMI" (which will break if you switch regions).

Let's fix both to make this **"Senior Engineer Approved."**

🔄 Phase 7: Secure the Production State

We already moved Dev to the cloud. Now, we must do the same for Prod. **Never leave Prod state on a laptop.**

File: `env/prod/backend.tf` (Create this file)

```terraform
terraform {
  backend "s3" {
    # Use the SAME bucket name you created in Global
    bucket         = "titan-terraform-state-storage"

    # CRITICAL: Use a DIFFERENT key. If you use "dev", you overwrite Dev with Prod!
    key            = "env/prod/terraform.tfstate"

    region         = "us-east-1"
    dynamodb_table = "titan-terraform-locks"
    encrypt        = true
  }
}
```

Action:

1. `cd env/prod`
2. `terraform init`
3. Type `yes` to migrate.

Now both environments are safe in the vault.

🧠 Phase 8: Dynamic Data Sources (No More Hardcoding)

The Problem: In `env/dev/main.tf`, we hardcoded `ami_id = "ami-04b..."`.

- **Why this gets you fired:** Amazon changes AMI IDs constantly. If you try to deploy this code in `us-west-2` (Oregon), that ID won't exist, and the build fails.
- **The Fix:** Ask AWS for the ID *dynamically*.

We need to update our `web` module to find its own AMI.

1. Update `modules/web/main.tf`

Add this `data` block at the top. This asks AWS: *"Give me the latest Ubuntu 24.04 AMI."*

```terraform
# modules/web/main.tf

# 1. Ask AWS for the latest Ubuntu Image
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical (The makers of Ubuntu)

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}

# 2. Update the Resource to use the DATA, not the VARIABLE
resource "aws_instance" "server" {
```

```
  # OLD: ami = var.ami_id
  # NEW:
  ami = data.aws_ami.ubuntu.id

  # ... keep the rest the same ...
}
```

2. Clean up `modules/web/variables.tf`

Remove the `ami_id` variable since the module now finds it automatically!

Terraform

```
# Remove this line:
# variable "ami_id" { type = string }
```

3. Clean up `env/dev/main.tf` and `env/prod/main.tf`

Remove the `ami_id = "..."` line from your module calls.

Why this is Pro: Your code is now **Region Agnostic**. You can change `region = "eu-west-1"` in your provider, and Terraform will automatically find the correct London AMI ID without you changing a single line of config.

🛠️ Phase 9: Day 2 Operations (Taint & Replace)

The infrastructure is built. Now comes "Maintenance."

Scenario: You updated the `install_nginx.sh` script to change the homepage text. You run `terraform apply`.

- **The Problem:** Terraform looks at the EC2 instance and says, *"The instance exists. User Data only runs on first boot. No changes needed."*
- **The Result:** Your new script is ignored. The server keeps the old homepage.

The Fix: We need to force Terraform to **destroy and recreate** the server so the script runs again.

Option A: The "Nuclear" Option (Taint)

This marks the resource as "corrupted." Terraform will destroy it on the next run.

Bash

```
# Syntax: terraform taint <resource_address>
terraform taint module.titan_web.aws_instance.server
```

Now run `terraform apply`. It will say: **"1 to destroy, 1 to add."**

Option B: The "Surgical" Option (-replace)

This does it all in one command (Better for automation).

Bash

```
terraform apply -replace="module.titan_web.aws_instance.server"
```

🎓 Project Graduation

You have completed **Terraform Titan**. Look at what you built:

1. **Modular Architecture:** Reusable `vpc` and `web` blueprints.
2. **Multi-Environment:** Isolated `dev` and `prod` worlds.
3. **Dependency Injection:** Passing VPC IDs into Web Modules.
4. **Remote Backend:** S3 state storage with DynamoDB locking.
5. **Dynamic Data:** Auto-discovering AMI IDs.
6. **Bootstrapping:** Auto-installing software with `user_data`.

🛑 Final Teardown (Save your money!)

Since we are done, let's practice the **Destroy Workflow** one last time.

1. **Destroy Prod:** `cd env/prod` -> `terraform destroy -auto-approve`
2. **Destroy Dev:** `cd env/dev` -> `terraform destroy -auto-approve`
3. **(Optional) Destroy the Backend:**
    - You have to empty the S3 bucket manually in the AWS Console first (Versioning makes it hard to delete via Terraform unless you force it).
    - `cd global/s3` -> `terraform destroy`.

You are now ready for Kubernetes. We have the Docker skills (Images/Containers). We have the Terraform skills (Infrastructure/Networking).

Shall we start the Kubernetes module now, or do you have questions on this Terraform project?