Terraform Internship Revision Roadmap

ok so i have plenty devops but i dont think i know anything , coz it was mostly learning from udemy and ai , i am now revising , what i want you to do is fill what i miss, example if i say starting docker using dockerhub , you would give me knowledge of other methods as well , like i want professional level knowledge so i can start looking for good internships

This is a fantastic approach. This is exactly how you bridge the gap between **"Tutorial Hell"** (where everything works perfectly in isolation) and **"Production Engineering"** (where security, scale, and compliance break everything).

Given your background with **Argus** and your goal for a **100cr empire**, we need to treat your knowledge like a product: it needs to be robust, scalable, and audit-ready.

Here is how we will play this:

1. **You:** Give me a specific concept or workflow you are revising (e.g., "I know how to SSH into a server").
2. **Me:** I will act as the "Senior Architect." I will confirm your method, but then I will expose the **Enterprise/Production** layer—the tools, security practices, and automation standards that actual unicorns and enterprises use.

🧠 Example Round: Docker Images

To set the standard, let's take the example you just gave: **"Starting Docker using DockerHub."**

🟢 Level 1: The Tutorial Way (What you likely know)

- You write a `Dockerfile`.
- You run `docker build -t myapp .`.
- You login to DockerHub (`docker login`).
- You push (`docker push user/myapp`).
- **Result:** It works, but it's dangerous for business.

🔴 Level 2: The "Professional/Internship-Ready" Gap (What you missed)

1. Private Registries (Not Public DockerHub) Enterprises rarely use public DockerHub for proprietary code.

- **The Pro Way:** They use **Private Container Registries (PCR)** like **AWS ECR (Elastic Container Registry)**, **Azure ACR**, or self-hosted solutions like **Harbor** or **JFrog Artifactory**.
- **Why?** Security compliance, role-based access control (RBAC), and speed (images stay within the cloud network).
- **The Interview Question:** "How do you handle authentication for a private registry inside a Kubernetes cluster?" (Answer: ImagePullSecrets).

2. Supply Chain Security (Signing Images)

- **The Pro Way:** Anyone can tag an image as `v1`. How do you trust it? We use **Cosign (Sigstore)** or **Docker Content Trust** to digitally sign images.
- **Why?** To prevent "Man-in-the-Middle" attacks where a hacker swaps your image for a malicious one.

3. Image Scanning (DevSecOps)

- **The Pro Way:** You never push blindly. The CI/CD pipeline runs a vulnerability scanner like **Trivy**, **Clair**, or **Snyk** during the build process.
- **Why?** If the base image (e.g., `python:3.9`) has a critical vulnerability (CVE), the build **fails** automatically before it ever reaches the registry.

4. The "Rate Limit" Problem

- **The Pro Way:** DockerHub creates rate limits (pull limits). Enterprises use **Pull-Through Caches** or mirror registries to ensure production autoscaling doesn't fail because DockerHub blocked your IP request.

🚀 Your Turn

I am ready to harden your skills. Pick a topic you are currently revising or feel "shaky" on.

Examples to get you started:

- "I use `git push` to save my code."
- "I use `kubectl apply -f deployment.yaml` to deploy apps."
- "I use an `.env` file to store my API keys."
- "I manually create EC2 instances in the AWS Console."

What is the first topic on your revision list?

Docker & K...Course (4)
PDF

kuch samajh nahi aaya par achha laga kya kya bol gaye 😭 😭 😭,heres my notes from docker course transcripts, can we start from here pls,like i want professional docker knowledge and practice, this is just basics, so lets revise this and level up

Haha, koi baat nahi! (No problem at all!) It happens. You have consumed a **massive** amount of information (150 lines of transcripts!), so feeling overwhelmed is normal.

Here is the good news: **Your notes are actually excellent.** They cover the "Academind" course (likely Maximilian Schwarzmüller), which is gold. But watching a course and *doing* it professionally are two different things.

Let's switch modes. No more reading theory. Let's do **Professional Revision**.

I will pick a core concept from your notes, show you the "Basic/Tutorial" version you learned, and then teach you the **"Enterprise/Pro"** version that gets you hired.

🚀 Phase 1: The Dockerfile (Building the Foundation)

In your notes (around **Source 340**), you have a standard Node.js Dockerfile.

🟢 Level 1: The Basic Way (From your notes)

This works fine for a tutorial, but if you push this to a production server in a bank or a top tech company, security will reject it.

Dockerfile

```
FROM node:14
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
EXPOSE 3000
CMD ["node", "app.mjs"]
```

🔴 Level 2: The Professional Way (What you need to know)

Here are the 3 major issues a Senior Engineer would spot in the file above:

1. **Security Risk (Running as Root):** By default, Docker runs as `root`. If a hacker compromises your app, they have root access to the container.
2. **Image Bloat:** You are shipping the whole Node.js development environment. You don't need all that for production.
3. **No Version Pinning:** `node:14` might change tomorrow.

🛠 The Fix: The "Pro" Dockerfile

Here is how we rewrite this using **Multi-Stage Builds** and **Non-Root Users**.

Dockerfile

```dockerfile
# --- Stage 1: The Builder (Compiling/Installing) ---
FROM node:14-alpine AS builder
# 1. Use 'alpine' (smaller OS)
# 2. Name the stage 'AS builder'

WORKDIR /app
COPY package.json package-lock.json ./
# 3. Always copy lock file too!

RUN npm ci
# 4. Use 'npm ci' instead of 'npm install' (Strict version matching for pro envs)

COPY . .

# --- Stage 2: The Runner (Production Ready) ---
FROM node:14-alpine

WORKDIR /app

# 5. Create a non-root user for security
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

# 6. Copy ONLY what is needed from the 'builder' stage
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./package.json
COPY --from=builder /app/app.mjs ./app.mjs

# 7. Switch to the non-root user
USER appuser

EXPOSE 3000
CMD ["node", "app.mjs"]
```

🧠 Quick Quiz (Test your Pro Knowledge)

To verify you understand the "Pro" changes, answer these 2 questions (think regarding your interview):

1. **Why did I use `npm ci` instead of `npm install` in the Pro version?** (Hint: It relates to the `package-lock.json`).
2. **Why did I use two `FROM` commands?** What is the benefit of copying files `--from=builder`?

Give me your best guess, and I will correct/refine it. Then we move to **Volumes & Data.**

ans 1 - npm ci selectively installs only those parts of alpine which are necessary

ans 2 - two from commands ensure that work is separated in builder and runner , and stupid things do not get installed before the good things are installed ? something like that

Great effort! You got the **intuition** right for the second one, but the first one needs a little course correction. This is exactly why we practice—to fix these mental models before an interviewer asks.

📝 Quiz Review

1. Why `npm ci` instead of `npm install`?

- **Your Answer:** "selectively installs only those parts of alpine which are necessary."
- **The Professional Answer:** Not quite! `npm ci` stands for **Clean Install**. It has nothing to do with Alpine (the OS).
  - `npm install` creates a `package-lock.json` file and might upgrade minor versions automatically. This is bad for production because "it worked yesterday" might break today.
  - `npm ci` **strictly** installs the exact versions listed in `package-lock.json`. If the lockfile doesn't match `package.json`, it fails (which is good!). It ensures **Reproducibility**.

2. Why use two `FROM` commands (Multi-Stage Builds)?

- **Your Answer:** "work is separated... stupid things do not get installed before the good things"
- **The Professional Answer:** Spot on! You nailed the logic.
  - **Stage 1 (Builder):** Has all the "stupid things" (compilers, build tools, source code, secrets) needed to *make* the app.
  - **Stage 2 (Runner):** We copy **only** the final executable/files. We leave the heavy tools behind.
  - **Result:** A tiny, secure image (e.g., 50MB instead of 800MB).

🚀 Phase 2: Data & Storage (Volumes)

Moving on to **Section 3** of your notes. You learned about **Volumes**, **Bind Mounts**, and **Anonymous Volumes**.

🟢 Level 1: The Tutorial Way (What you know)

To develop locally and see your code changes instantly (Live Reload), you use a **Bind Mount**:

```bash
docker run -v $(pwd):/app -p 3000:3000 my-node-app
```

(This maps your current folder `.` to `/app` inside the container).

🔴 Level 2: The "Professional" Trap (The `node_modules` Disaster)

If you run the command above in a professional setting, **your app will likely crash immediately.**

Why? When you bind mount your local folder `$(pwd)` into `/app`, you **overwrite** everything in the container's `/app` folder with your local files.

- **The Problem:** Your local folder probably *doesn't* have `node_modules` (or has Windows/Mac versions), but the container *built* its own Linux versions in `/app/node_modules` during the build process.
- **Result:** The bind mount hides the container's `node_modules`, and the app says "Module not found".

🛠️ The Fix: The "Data Volume" Trick

To be a pro, you must tell Docker: *"Overwrite everything in `/app` with my local code, **EXCEPT** the `node_modules` folder. Keep the container's version of that."*

The Command:

Bash

```
docker run -v $(pwd):/app -v /app/node_modules ...
```

Why this works (The "Longest Path Wins" Rule):

1. `-v $(pwd):/app` tells Docker to mount your code to `/app`.
2. `-v /app/node_modules` (an **Anonymous Volume**) creates a specialized bucket just for that folder.
3. Docker sees a specific rule for `/app/node_modules` and preserves the files you built inside the image, ignoring the overwrite from the bind mount.

🧠 Quick Quiz: Permissions

You are running your container as a **non-root user** (like we learned in Phase 1). You use a bind mount to sync your code.

Scenario: You try to create a file from *inside* the container (e.g., `touch /app/test.txt`), but you get `Permission Denied`.

Why is this happening? (Hint: Who owns the folder on your host machine vs. who is trying to write to it inside the container?)

> both migght be the same person, but the difference is accessing the host dirrectly and through a secured container as named volumes live on host and are only mounted on containers

This is a tricky concept, and you are touching on the right *intuition*—that the container is "secured" or isolated—but the technical reason is much simpler and purely Linux-based.

📝 Quiz Review

The Problem: `Permission Denied` when writing to a Bind Mount.

- **Your Answer:** "difference is accessing the host directly and through a secured container... named volumes live on host."
- **The Professional Answer:** It is a **UID/GID (User ID) Mismatch**.
  - **On your Host:** Your folder is owned by you (e.g., User ID `1000`).
  - **Inside Container:** You created a secure user `appuser`. Docker assigns this user a random ID (e.g., `1001` or `5000`).
  - **The Clash:** Since Docker shares the **Linux Kernel** with the host, the Kernel sees: *"User 5000 is trying to write to a folder owned by User 1000."* -> **BLOCK.**

The Pro Fix: Ensure the UID inside the container matches the UID outside, or use "User Remapping" (advanced).

🚀 Phase 3: Networking (Connecting the Dots)

Moving to **Section 4** of your notes. You learned about:

1. **Container-to-World:** (Works automatically).
2. **Container-to-Host:** (Using `host.docker.internal`).
3. **Container-to-Container:** (Using Docker Networks and DNS names).

🟢 Level 1: The Tutorial Way (What you know)

You have a Node app and a MongoDB container. You create a network:

Bash

```bash
docker network create my-net
docker run --network my-net --name mongo-db mongo
docker run --network my-net --name my-app node-app
```

Inside your Node code, you connect using the **container name**: `mongoose.connect('mongodb://mongo-db:27017/...')`

🔴 Level 2: The "Professional" Gap (Service Discovery & Secrets)

1. No `host.docker.internal` in Production

- **The Trap:** You might leave `host.docker.internal` in your code because it worked on your laptop.
- **The Reality:** In a real production cluster (Kubernetes or AWS ECS), there is no "host" concept like on your laptop. Your database is likely a managed service (RDS/Atlas) with a totally different URL.
- **The Fix:** Always use **Environment Variables** (`ENV`) for connection strings. Never hardcode hostnames.

2. The "Exposed Ports" Security Risk

- **The Trap:** In tutorials, we often publish ports for databases so we can check them: `docker run -p 27017:27017 mongo`
- **The Reality:** If you do this on a cloud server, you just opened your database to the entire internet. Hackers scan for port 27017 constantly.
- **The Pro Way:**
  - **Internal Only:** Put the DB and App on the same Docker Network.
  - **No `-p` flag for the DB:** The App can still talk to the DB because they are on the same private network (using the container name). The outside world cannot touch the DB.

3. Waiting for Dependencies (The "Race Condition")

- **The Trap:** You start your App and DB at the same time (e.g., `docker-compose up`). The App starts faster, tries to connect to the DB (which is still booting), crashes, and dies.
- **The Pro Way:** Use **Healthchecks** or "Wait-for-it" scripts.
  - In `docker-compose`, you define a `healthcheck` for the DB.
  - The App container is configured with `depends_on: { condition: service_healthy }`. It physically waits until the DB says "I am ready" before starting.

🧠 Quick Quiz (Networking Edition)

You have a `docker-compose.yml` file. You want to run a **Python API** and a **Postgres DB**.

1. **The API needs to talk to the DB.** What "hostname" do you put in your Python code to connect? (Hint: It relates to the `services` names).
2. **Security Check:** You want the Python API to be accessible from your browser, but you want the Postgres DB to be **totally locked down** (accessible *only* by the API).
   - Which service gets a `ports:` section?
   - Which service does **NOT** get a `ports:` section?

Give me your answers, and we will see if you are ready for Docker Compose orchestration!

1.we give the service/network name
2.python gets port section but postgres doesnt

2/2 Correct! You have the logic down cold. 🎯

📝 Quiz Review (Fine-Tuning)

1. **Hostname:** You said "Service/Network name."
   - **Refinement:** It is specifically the **Service Name** (the key you write in YAML, e.g., `postgres-db`).
   - *Why:* Docker's internal DNS maps the Service Name `postgres-db` to the internal IP address. You don't use the network name as the host.
2. **Security:** You said "Python gets port section, Postgres doesn't."
   - **Refinement: 100% Accurate.** By omitting `ports:` on Postgres, you have created a **private subnet** where the DB exists but is invisible to the outside world (hackers).

🚀 Phase 4: Docker Compose (The Orchestrator)

Moving to **Section 6** of your notes. You learned that `docker-compose` replaces the headache of running 5 different `docker run` commands manually.

🟢 Level 1: The Tutorial Way (What you know)

You likely write a `docker-compose.yml` like this:

YAML

```yaml
version: '3.8'
services:
  backend:
    build: ./backend
    ports:
      - "80:80"
    depends_on:
      - mongodb # Waits for mongo container to start
  mongodb:
    image: mongo
    volumes:
      - data:/data/db
```

🔴 Level 2: The "Professional" Gap (Why this breaks in Prod)

1. The "Race Condition" Crash (`depends_on` is a lie)

- **The Trap:** You use `depends_on: - mongodb`. You think: *"Great, the backend will wait for the DB to be ready."*

- **The Reality:** Docker only waits for the DB container to **start** (turn on). It does **not** wait for the Database process to be ready to accept connections.
  - Your backend starts 100ms after the DB container -> The DB is still initializing -> Backend tries to connect -> **Connection Refused** -> Backend Crashes.
- **The Pro Fix:** You need a **Healthcheck**.

2. Hardcoded Secrets

- **The Trap:** `environment: - MONGO_PASSWORD=secret`.
- **The Reality:** You just committed your password to Git.
- **The Pro Fix:** Use `.env` files (which you `.gitignore`) or Docker Secrets.

3. Zombie Containers

- **The Trap:** If your backend crashes (bug/memory), it stays dead.
- **The Pro Fix:** `restart: always` or `restart: on-failure`.

🛠️ The "Pro" Docker Compose File

Here is how a Senior DevOps Engineer writes that same file. Look closely at the **Healthcheck** and **Condition**.

YAML

```yaml
version: '3.8'

services:
  backend:
    build: ./backend
    restart: on-failure  # 1. Auto-restart if it crashes
    ports:
      - "80:80"
    environment:
      # 2. Read from .env file automatically
      - DB_HOST=mongodb
    depends_on:
      mongodb:
        condition: service_healthy # 3. The Magic: Wait until DB is ACTUALLY ready

  mongodb:
    image: mongo
    restart: always
    environment:
      # 4. Secrets loaded from hidden .env file
      MONGO_INITDB_ROOT_USERNAME: ${MONGO_USER}
      MONGO_INITDB_ROOT_PASSWORD: ${MONGO_PASS}
    volumes:
      - mongo-data:/data/db

    # 5. The Heartbeat: Tells Docker how to check if DB is alive
    healthcheck:
      test: ["CMD", "mongo", "--eval", "db.adminCommand('ping')"]
      interval: 10s
      timeout: 5s
      retries: 5

volumes:
  mongo-data:
```

🧠 Quick Quiz (Compose Edition)

To solidify this, imagine you are reviewing a Junior Dev's Pull Request. They submitted this `docker-compose.yml`:

YAML

```yaml
services:
  web:
    image: nginx
    volumes:
      - ./src:/var/www/html
  api:
    image: my-python-api
    ports:
      - "5000:5000"
```

Find the 2 Critical Mistakes:

1. **Deployment Mistake:** They are using a **Bind Mount** (`./src:/var/www/html`). Why will this fail if I deploy this file to a remote AWS server using Github Actions?
2. **Networking Mistake:** They published port `5000` for the API. If the `web` (Nginx) is supposed to be the *only* entry point for users, why is publishing port 5000 on the API bad practice?

Spot the errors!

1.data needed to run the app gets overwritten
2.universal app access.hacking

You nailed the security question, and you are very close on the deployment one! Let's refine that mental model because it is the **#1 reason** deployments fail for beginners.

📝 Quiz Review

1. Deployment Mistake (Bind Mounts in Prod)

- **Your Answer:** "data needed to run the app gets overwritten"
- **The Professional Explanation:** You are technically correct—the bind mount *does* overwrite the directory—but here is the specific **Production Failure Scenario**:
  - On your laptop, `./src` contains your code.
  - On the AWS server, `./src` **does not exist.** You only deployed the *Docker Image*. You didn't Git clone your code onto the server (that's the old non-Docker way).
  - **The Crash:** Docker starts on AWS. It sees `- ./src:/var/www/html`. It looks for `./src` on the server, finds an empty folder (or creates one), and mounts that **Empty Folder** over your app code inside the container.
  - **Result:** Your container starts, finds 0 files, and crashes immediately.
  - **Pro Rule: Never** use bind mounts in production. Use `COPY . .` in your Dockerfile so the code lives *inside* the image.

2. Networking Mistake (Exposed Ports)

- **Your Answer:** "universal app access.hacking"
- **The Professional Explanation:** 100% Correct.
  - You created a "Front Door" (Nginx) for security/routing.
  - By adding `ports: 5000:5000` to the API, you punched a **giant hole in the wall** right next to the front door. Hackers can bypass Nginx and hit your API directly.

- **Pro Rule:** Only the **Load Balancer / Gateway** (Nginx) gets `ports:`. Everything else stays private.

🚀 Phase 5: Utility Containers (The "No Install" Philosophy)

Moving to **Section 7** of your notes. This is a superpower that distinguishes seniors from juniors.

The Philosophy: "I should be able to format my laptop, install *only* Docker, and start coding immediately."

🟢 Level 1: The "Old School" Way

You want to start a new Node.js project.

1. You download the Node.js installer for Windows/Mac.
2. You run the installer.
3. You restart your terminal.
4. You run `npm init`.
5. **Problem:** Your teammate has Node 18. You have Node 14. Your `package-lock.json` files fight each other.

🔴 Level 2: The "Professional Utility Container"

You don't install Node.js. You don't install Python. You don't install PHP. **You use Docker to run commands.**

The Command (One-Liner Magic):

Bash

```
docker run --rm -v $(pwd):/app -w /app node:18-alpine npm init
```

Let's break down this "Pro" command:

1. `--rm`: **Auto-Delete.** Once the command is done, delete the container. Don't leave trash behind.
2. `-v $(pwd):/app`: **Bind Mount.** Connect my current folder to `/app`.
3. `-w /app`: **Working Directory.** "Go inside `/app` before running the command."
4. `node:18-alpine`: The image.
5. `npm init`: The command to run.

The Result: A `package.json` file appears on your laptop, created by a Node version you never actually installed.

🧠 Quick Quiz (Utility Containers)

You are using a Utility Container to run a Python script that generates a report: `docker run --rm -v $(pwd):/data python:3.9 python gen_report.py`

Scenario: The script runs successfully and creates `report.pdf`. You try to open `report.pdf` on your Linux/Mac laptop, but the OS says **"Access Denied"** or "Read-only". You can't even delete the file!

Why is this file "locked"? (Hint: Think back to the "User ID" issue we discussed in Phase 2).

1. Who owns the file: You or Root?
2. How would you fix this in the `docker run` command? (Think broadly, don't worry about exact syntax).

1.host owns the file not the user in container
2.idk this one, we need to either give the user host permissions or bring the file in the container

That is a very common point of confusion! Let's clear up exactly who owns what, because this "file locking" issue drives developers crazy on Linux systems.

📝 Quiz Review

1. Who owns the file?

- **Your Answer:** "host owns the file not the user in container"
- **The Professional Reality:** It is actually the **opposite** (on Linux).
  - By default, a Docker container runs as **Root**.
  - When the container creates `report.pdf` on the bind mount, the file is stamped with `Owner: Root`.
  - **The Problem:** On your host machine, *you* are a standard user (e.g., `ubuntu` or `jdoe`). You try to open the file, and the OS says: *"Stop! This file belongs to Root. You are not Root. Access Denied."*

2. The Fix (`-u` Flag)

- **Your Answer:** "give the user host permissions"
- **The Professional Command:** You tell Docker to run *as you*.

Bash
```bash
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/data ...
```

- `-u $(id -u):$(id -g)`: This extracts your current User ID (e.g., 1000) and Group ID from your host machine and tells the container: *"Don't run as Root. Run as User 1000."*
- **Result:** The container creates the file as User 1000. You own it. You can open it.

🚀 Phase 6: Kubernetes (The "Big Boss")

We are moving to **Section 11 & 12** of your notes. You are graduating from "running containers" to "orchestrating them."

🟢 Level 1: The Tutorial Way (What you know)

You learned that a **Pod** is the smallest unit in Kubernetes. To run your app, you might be tempted to write a `pod.yaml`:

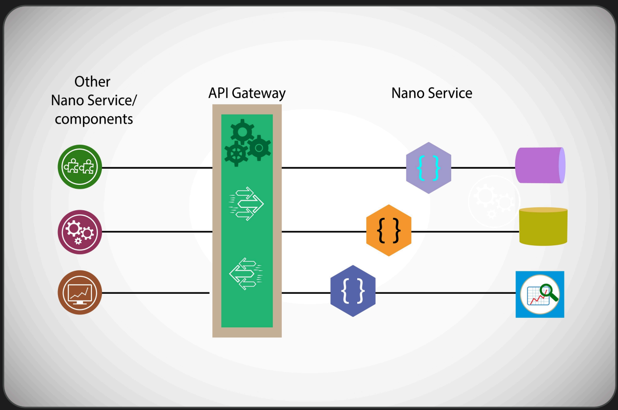
YAML
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-node-app
spec:
  containers:
    - name: app
      image: my-node-image
```

You apply it: `kubectl apply -f pod.yaml`. It works!

🔴 Level 2: The "Professional" Gap (Pods vs. Deployments)

1. The "Naked Pod" Problem

- **The Trap:** You deploy a naked Pod (just the YAML above) to production.
- **The Reality:** If that Pod crashes (out of memory), or if the worker node dies, **the Pod stays dead.** Kubernetes does *not* restart naked Pods automatically. Your app goes offline at 3 AM.
- **The Pro Way:** Never create Pods directly. Use a **Deployment**.
  - A Deployment creates a **ReplicaSet**.
  - The ReplicaSet ensures that if you asked for 3 pods and one dies, it instantly creates a new one to get back to 3. It provides **Self-Healing**.



Shutterstock

2. The "Updates" Nightmare

- **The Trap:** You want to update your app from `v1` to `v2`. With a naked Pod, you have to delete it (`kubectl delete pod ...`) and create a new one. This causes downtime.
- **The Pro Way:** Deployments support **Rolling Updates**. You change the image to `v2`, and K8s slowly replaces pods one by one (v1 -> v2) so users never experience downtime.

🧠 **Quick Quiz (Kubernetes Architecture)**

To test if you are ready for K8s production, answer these 2 questions based on **Service Discovery** (Section 12/14 of your notes):

1. **The Scenario:** You have a **Frontend Pod** and a **Backend Pod**. The Frontend needs to talk to the Backend.
   - **The Trap:** You check the Backend Pod's IP address (e.g., `10.244.0.5`) and hardcode it into your Frontend code.
   - **Why is this a guaranteed disaster?** (What happens to that IP if the Backend Pod crashes and K8s restarts it?)
2. **The Fix:** Kubernetes gives you a stable "Front Door" object that sits in front of your Pods and gives you a permanent IP/DNS name. **What is this object called?**

Give me your answers!

give me a major docker centered project that covers everything, dont include kubernetes rn , we can upgrade it later to use k8 , but not rn.

This is a **major, portfolio-grade project**. We are going to build a **Distributed Task Processing System** (like a mini-version of a video transcoding service or an email campaign manager).

This architecture separates the "heavy lifting" from the user interface, a pattern used by Uber, Netflix, and Amazon. It covers **multi-stage builds, custom networking, volume persistence, reverse proxies, message queues, and background workers.**

🏗️ Project Architecture: "The Enforcer"

We will build 6 interacting services.

1. **Client (React):** A frontend to submit tasks and view real-time status.
2. **API Gateway (Nginx):** A reverse proxy to route traffic and secure the backend.
3. **API Service (Node.js):** Receives tasks, saves them to DB, and pushes them to a Queue.
4. **Message Queue (RabbitMQ):** Decouples the API from the Worker (Async communication).
5. **Worker Service (Python):** Listens to the Queue, performs the "heavy" task (simulation), updates the DB + Cache.
6. **Database (PostgreSQL) & Cache (Redis):** Persistent storage and fast real-time status updates.

📂 1. The Professional Folder Structure

Don't dump everything in one folder. Organize like a microservices repo.

Plaintext

```
/the-enforcer
├── /client             # React App
│   ├── Dockerfile      # Multi-stage build
│   └── ...code
├── /server             # Node.js API
│   ├── Dockerfile
│   └── ...code
├── /worker             # Python Worker
│   ├── Dockerfile
│   └── ...code
├── /nginx              # Nginx Configs
│   └── default.conf
└── docker-compose.yml  # The Master Orchestrator
```

🛠️ 2. The "Pro" Dockerfiles (Multi-Stage & Secure)

A. Frontend (Client) - Multi-Stage Build

Concept: We build the React app in Node, but we *serve* it with Nginx. We don't ship Node.js to production.

Dockerfile

```
# /client/Dockerfile

# Stage 1: Build
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2: Serve
FROM nginx:alpine
# Copy ONLY the build artifacts from the builder stage
COPY --from=builder /app/dist /usr/share/nginx/html
```

```
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

B. Worker (Python) - Lightweight Alpine

Concept: A dedicated worker that only runs python scripts.

Dockerfile

```dockerfile
# /worker/Dockerfile
FROM python:3.9-alpine

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .

# Run unbuffered so logs show up immediately in Docker Dashboard
CMD ["python", "-u", "main.py"]
```

⚙️ 3. The Master `docker-compose.yml` (The "Everything" Config)

This config uses **Healthchecks**, **Dependency Chaining**, **Networks**, and **Volumes**.

YAML

```yaml
version: '3.8'

services:
  # --- 1. The Gateway (Reverse Proxy) ---
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"  # The only port exposed to the world
    volumes:
      - ./nginx/default.conf:/etc/nginx/conf.d/default.conf:ro
    depends_on:
      - api
      - client

  # --- 2. The Frontend ---
  client:
    build: ./client
    # No ports exposed! Only Nginx can talk to it.

  # --- 3. The API (Producer) ---
  api:
    build: ./server
    environment:
      - DB_HOST=postgres
      - REDIS_HOST=redis
      - RMQ_HOST=rabbitmq
    depends_on:
      postgres:
        condition: service_healthy # Wait for DB to be ready
      rabbitmq:
        condition: service_healthy

  # --- 4. The Worker (Consumer) ---
  worker:
    build: ./worker
    environment:
      - RMQ_HOST=rabbitmq
      - DB_HOST=postgres
```

```
    depends_on:
      rabbitmq:
        condition: service_healthy

  # --- 5. Infrastructure Services ---
  postgres:
    image: postgres:15-alpine
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: tasks_db
    volumes:
      - pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user -d tasks_db"]
      interval: 5s
      retries: 5

  redis:
    image: redis:alpine

  rabbitmq:
    image: rabbitmq:3-management-alpine
    ports:
      - "15672:15672" # Management UI (for debugging)
    healthcheck:
      test: ["CMD", "rabbitmq-diagnostics", "-q", "ping"]
      interval: 10s
      timeout: 5s
      retries: 5

# Define persistent storage
volumes:
  pgdata:
```

🧠 4. Why this is "Professional Level"

1. The "Healthcheck" & `depends_on` Pattern

- **The Rookie Mistake:** The API container starts *before* the Database is ready. It tries to connect, fails, crashes, and you get "Connection Refused."
- **The Pro Fix:** Look at the `postgres` service. We added a `healthcheck`. Then, in the `api` service, we used `depends_on: condition: service_healthy`. Docker will literally **pause** the API startup until Postgres says "I am ready."

2. The Reverse Proxy (Nginx)

- **The Rookie Mistake:** Opening ports `3000`, `5000`, `5432` all to the host.
- **The Pro Fix:** We only opened port `80`. Nginx acts as a traffic cop.
  - Requests to `/api/*` -> Route to Node API container.
  - Requests to `/*` -> Route to React Client container.
  - This eliminates CORS issues and secures your backend.

3. Separation of Concerns (Queues)

- **The Rookie Mistake:** The API does the heavy work (e.g., video processing) itself. If 100 users submit requests, the API freezes.
- **The Pro Fix:** The API simply drops a message into **RabbitMQ** (takes 5ms) and replies "Task Received." The **Worker** picks it up later. This allows the system to handle millions of requests without crashing.

📝 Next Steps for You

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

```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

```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?