**Staging Workflow for Terraform Code**

**Introduction & Purpose**

* The lecture focuses on workflows in GitHub using GitHub Actions.
* Each GitHub repository has an Actions tab that allows you to build, test, and deploy code without needing external CI/CD tools like Jenkins.

**Workflows Explained**

* In GitHub Actions, automated processes are called **workflows** (the equivalent of pipelines in other tools).
* Workflows are not created manually via the UI; rather, you write code in your repository, and when you push that code, the workflow is automatically created.

**Setting Up a Workflow**

* Open VS Code and navigate to the **IAC-Vprofile** repository.
* At the repository level, create the folder structure:
  + .github/workflows This is the directory where all workflow files are stored.
* Within the .github/workflows folder, create a new file named (for example) Terraform.yml.
  + Any file with a .yml extension in this folder containing valid YAML code will be recognized as a workflow.

**Defining the Workflow File**

* Begin the file by specifying the workflow’s name using the name: key. For example:
  + name: "Vprofile infrastructure as a code"
* Next, determine when the workflow should trigger by defining events under the on: key.

**Configuring Triggers**

* **Push Event:**
  + Use push: to trigger the workflow when code is pushed.
  + Specify the branches that should trigger the workflow (e.g., branches: - main and - stage).
  + Optionally, restrict the trigger to changes in a specific folder by defining paths: and using a list format (e.g., - terraform/\*\*).
* **Pull Request Event:**
  + Use pull\_request: to trigger the workflow when a pull request is created.
  + Specify the target branch (usually a locked branch like main where only authorized personnel can commit).
  + Again, the same paths: (e.g., - terraform) is used to limit the trigger to changes within that folder.
* **Focusing on the Terraform Folder:**
  + The workflow is configured so that it only triggers when there is a change in the Terraform folder (using the pattern Terraform/\*\*).
* **Setting Environment Variables:**
  + Define the environment variables using env:.
  + **AWS Access Key and Secret Key:**
    - Set the variable AWS\_ACCESS\_KEY\_ID to the value retrieved from the corresponding GitHub secret (using ${{ secrets.AWS\_ACCESS\_KEY\_ID }}). This export makes the variable available for Terraform when executing the code.
  + **Bucket Variable:**
    - Set the variable bucket\_TF\_state. Its value is obtained from the stored secret for the bucket name.
  + **AWS Region:**
    - Set the variable AWS\_REGION to the region being used (in this case, US east two, the Ohio region).
  + **EKS Cluster:**
    - Set the variable EKS\_CLUSTER to match the value defined in the variables.tf file.
  + *(Note: The AWS region and EKS cluster variables are not mandatory but are used repeatedly in the code.)*
* **Defining the Job:**
  + Under jobs:, create a job (named, for example, "Terraform") with a description such as "apply Terraform code changes."
  + Specify the runner using runs-on: ubuntu-latest. GitHub Actions provides different runners (e.g., Ubuntu, macOS, Windows), and Ubuntu-latest is chosen for having many prebuilt packages.
* **Setting Defaults for the Job:**
  + Use defaults: to specify:
    - run:
      * shell: Bash
      * working-directory: ./Terraform
  + This configuration ensures that all steps run using a Bash shell in the Terraform directory.
* **Defining the Steps:**
  + Under the job's steps:, include tasks that execute commands.
  + **First Step – Checkout the Source Code:**
    - Name the step "Checkout the source code."
    - Use the predefined GitHub action actions/checkout (for example, actions/checkout@V4 or @V2) to pull the repository’s code into the runner container.
  + After checking out the code, the container (running an Ubuntu image) will change into the specified working directory (./Terraform), and subsequent commands will be executed there

**After the Checkout Step:**

* Once the source code is checked out, the next step is to execute the Terraform commands.

**Installing Terraform:**

* Terraform is not pre-installed, so a GitHub Action is used for setup.
* The step uses HashiCorp/setup-Terraform@v2.
* Optionally, you can specify the Terraform version using a variable (e.g., Terraform\_version), but here it is left commented so that the latest version is used.

**Terraform Init Step:**

* A step named "Terraform init" (with an assigned ID) is added to run the terraform init command.
* The command includes a backend configuration to specify the bucket for storing state information, using the bucket name from the environment variable (accessed as $bucket\_TF\_state).

**Terraform Format Check:**

* A step named "Terraform fmt" (with an ID "FMT") runs terraform fmt -check to verify that the code is properly formatted.
* If the format is incorrect, it returns a non-zero exit code and fails the workflow.

 **Terraform Validate Step:**

* A step named "Terraform validate" (with its own ID) executes terraform validate to ensure that the syntax and resource definitions in the Terraform code are correct.
* A non-zero exit code here will also fail the workflow.

**Terraform Plan Step:**

* A step named "Terraform plan" (with an ID) runs terraform plan using options:
  + -no-color (to disable colored output),
  + -input=false (to prevent interactive input), and
  + -out to store all output into a plan file.
* Storing the output in a file is used as a workaround for a parallel execution issue encountered on GitHub Actions.
* The command includes the option continue-on-error: true to allow the workflow to proceed even if there is an error.

**Terraform Plan Status Check:**

* A final step named "Terraform plan status" checks if the previous plan step failed (by evaluating if steps.plan.outcome equals "failure").
* If the condition is met, it forces the workflow to fail by using an exit command (like exit 1).

**Container Exit Behavior:**

* When the exit command is executed within a container, it kills the shell (PID 1), which stops the container and causes the workflow to fail if an error occurs—for example, during the Terraform plan.

**Reviewing and Committing the Code:**

* The instructor advises checking the code for correct spelling, formatting, and proper alignment (noting that a space error in the jobs section caused the workflow to initially fail). After correcting the whitespace, the code is saved, committed, and pushed.

**Triggering the Workflow:**

* A change is made in the Terraform folder (by adding a hash in the variables.tf file) to trigger the workflow. The commit is pushed, and a commit message (e.g., "test workflow") is noted in the Actions tab.

**Workflow Execution Overview:**

The "Terraform apply" job begins and executes all defined steps. The output in the Actions tab shows:

* **Checkout of Source Code:** The source code is successfully checked out.
* **Terraform Setup and Init:** Terraform is installed via the Action HashiCorp/setup-Terraform@v2, and terraform init is executed with the backend configured to use the S3 bucket (with the secret value hidden in logs).
* **Terraform Format Check:** terraform fmt -check runs to ensure proper formatting.

**Additional Notes**

* In pull request scenarios, users typically work on a staging branch and, after testing, create a pull request to merge into the main branch.
* When the pull request is raised, the workflow automatically triggers to test the code before merging.
* Further details on pull request handling will be discussed later in the lecture.

**Main Workflow for Terraform Code**

**Infrastructure Application Warning:**

The lecture begins by warning that the upcoming steps will create the entire infrastructure (VPC and EKS cluster) on AWS, which may incur billing costs. The instructor advises that if you do not wish to incur charges, you should stop at this point and only watch the lecture. However, the recommendation is to push forward and execute the code to gain hands-on experience.

**Conditional Terraform Apply:**

The workflow is set up so that the step executing terraform apply runs only when there is a push event on the main branch.

* The step (named “Terraform Apply” with an ID like apple) runs the command: terraform apply -auto-approve -input=false -parallelism=1 -planfile <planfile>.
* The -parallelism=1 flag is used to avoid bugs with parallel execution (specifically with EKS modules).
* This command will only execute if both conditions are met: the Git reference is the main branch, and the GitHub event is a push.

**Ingress Controller and Kubeconfig Setup:** Additional steps are added for application code deployment:

* An ingress controller (Nginx Ingress Controller) is to be deployed.
* A step is introduced to run kubectl apply to install the ingress controller.
* To execute kubectl commands, a kubeconfig file is needed. This file is generated by running:
* aws eks update-kubeconfig --region <region> --name <cluster>

Running the AWS CLI command requires AWS credentials (access key and secret key) to be configured.

* A condition is applied such that the kubeconfig generation (and subsequently the kubectl apply) only runs if both the terraform apply step and the kubeconfig step succeed.

**Testing on Staging Branch:** The workflow is designed so that these apply steps only run on the main branch. When changes are made in the Terraform folder (for example, by modifying the variables.tf file) and pushed on the staging branch:

* The workflow triggers the plan-related steps, and the plan output is created.
* However, the Terraform Apply step is skipped on the staging branch, as intended.

**Merging Changes to Main Branch:** To trigger the apply phase:

* The instructor demonstrates using Git Bash or a terminal to pull changes, then merge the staging branch into the main branch (using commands such as git pull, git merge stage, and git push origin main).
* There is also an explanation of branch protection—where the main branch can be locked to require a pull request for merging—but in this case, the merge is done directly since the main branch is not locked.
* Once the change is merged and pushed to the main branch, the workflow detects the push on main and triggers the Terraform Apply step.

**Execution Outcome:** On pushing the merged change to the main branch, the workflow starts running the Terraform Apply step. This apply process takes significant time (around 15–20 minutes), with the VPC being created first as part of the infrastructure setup

**Infrastructure Creation:** The VPC is created with a NAT gateway (including an Elastic IP) and the necessary subnets (both public and private). Once the VPC is fully set up, the EKS cluster is created along with its node groups. The entire provisioning process takes around 10–20 minutes.

**Verification and Resource Check:** After the infrastructure is applied, the instructor verifies it on the AWS console. In the EKS section, the cluster (named *vprofile-eks*) is visible with its two node groups. In the VPC console, the VPC is present along with its configured subnets, the NAT gateway, and the associated Elastic IP. The installation of the Nginx Ingress Controller is confirmed by the appearance of a load balancer in the EC2 console, while the Kubernetes nodes (managed by an EKS autoscaling group) are visible as instances.

**Workflow for Future Changes:** For any changes to the infrastructure, modifications should be made in the staging branch where only a Terraform plan is executed. Once validated, a pull request is created comparing the staging branch with the main branch. When the pull request is approved and merged, it triggers the GitHub Actions workflow that applies the changes from the main branch to the AWS Cloud.

**Next Steps:** The instructor concludes by stating that future lectures will cover workflows to deploy application-level code changes, and invites viewers to join the next session.

**Setting Up the Build and Publish Job**

After successfully testing the source code, the next step is to **build the Docker image** and **upload it to AWS ECR**. This requires a new job, separate from the Testing job.

**Defining the New Job**

* The job is named BUILD\_AND\_PUBLISH (in uppercase for clarity).
* It must be aligned in the same column as the previous job name, Testing.

**Creating a Dependency Between Jobs**

* The BUILD\_AND\_PUBLISH job should execute **only after** the Testing job completes.
* To achieve this, the dependency is set using:

yaml

needs: Testing

* If needs: Testing is not included, both jobs would run **in parallel**, which is not desired.

**Configuring the Runner and Steps**

**Specifying the Execution Environment**

* The job runs on ubuntu-latest:

yaml

runs-on: ubuntu-latest

This ensures that the steps execute in a Linux-based environment.

**Checking Out the Source Code**

* The first step is to retrieve the repository contents using GitHub Actions:

yaml

- name: Checkout source code

uses: actions/checkout@v3

**Using Docker ECR Action for Image Build and Upload**

**Selecting the Action**

* The workflow uses the docker ECR action provided by **apple boy**.
* This action simplifies the process of **building** and **pushing images** to ECR.

**Authenticating with AWS ECR**

* Authentication credentials are securely passed from GitHub Secrets:

yaml

- name: Build and Push to ECR

uses: appleboy/docker-ecr-action@v1

with:

access\_key: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

secret\_key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

**Setting Up Environment Variables**

**Defining the Required Variables**

* The following values are required for building and pushing the image:
  + **Registry URI** (stored as a GitHub Secret).
  + **ECR Repository Name** (ECR\_REPOSITORY).
  + **AWS Region** (AWS\_REGION).
* They are specified as:

yaml

registry: ${{ secrets.REGISTRY\_URI }}

repository: ${{ env.ECR\_REPOSITORY }}

region: ${{ env.AWS\_REGION }}

**Tagging the Docker Image**

**Using Dynamic Tags**

* The latest tag is used along with github.run\_number to ensure unique identifiers:

yaml

tags: latest,${{ github.run\_number }}

* This helps:
  + **Identify** builds using a unique run\_number assigned by GitHub.
  + **Ensure** each new image has a distinct identifier.

**Setting Docker Build Context and Path**

**Defining the Dockerfile Path**

* The location of the Dockerfile is explicitly provided:

yaml

dockerfile: ./Dockerfile

context: ./

* The context is set to the current directory (./).

**Committing and Pushing the Workflow**

**Saving and Syncing**

* The YAML file is:
  + **Saved** and **committed** with an appropriate message.
  + **Pushed** to the repository.

**Fixing YAML Syntax Issues**

**Indentation Error Detection**

* The workflow detected an **error at line 59** due to incorrect indentation.
* steps: must be properly aligned with runs-on:.
* Correction:

yaml

steps:

- name: Checkout source code

uses: actions/checkout@v3

* The fix is saved, committed, and pushed.

**Running the Workflow**

**Triggering Execution**

* The workflow is manually **triggered** to confirm execution.
* After a short wait, it **completes successfully**.

**Verifying Image Upload in ECR**

**Checking the Repository**

* The ECR repository is inspected for new images.
* The following tags appear:
  + latest
  + build ID 6
* Every code change now **automatically** results in a **new image build**.

**Outcome**

Now, the **CI/CD pipeline is fully operational**:

* **Testing precedes deployment**
* **Images are dynamically tagged**
* **Each new commit generates a fresh build**

Would you like to **expand** this setup by integrating additional security checks or Kubernetes deployment?

**Deploy to EKS**

You're now transitioning into **Helm-based Kubernetes deployments**—a great approach for dynamically managing images in your workflow! Here's a detailed breakdown of what you're doing:

**1. Addressing the Challenge**

* Your **app deployment file** contains a hardcoded image reference.
* You need a way to **pass dynamic image tags** during workflow execution.

**2. Introducing Helm**

* **Helm Charts** allow you to replace static values with **variables**.
* These variables will be passed at runtime when applying the Helm chart.

**3. Installing Helm**

**On macOS**

* Use Homebrew:

bash

brew install helm

**On Windows**

* Open **PowerShell as Administrator** and run:

powershell

choco install kubernetes-helm

**Post Installation**

* **Restart** the terminal (Git Bash for Windows, Terminal for Mac).
* Navigate to your repository folder.

**4. Creating Helm Charts**

* Execute:

bash

helm create vprofilecharts

* This generates a folder called **helm charts**.
* **Move it** into a separate helm directory:

bash

mkdir helm

mv vprofilecharts helm/

**5. Preparing the Templates**

* Navigate into the Helm structure:

bash

cd helm/vprofilecharts/templates

* **Remove Default Templates**:

bash

rm -rf \*

* **Copy Your Existing Kubernetes Files**:

bash

cp vpro-app/\* helm/vprofilecharts/templates/

* Verify:

bash

ls helm/vprofilecharts/templates

**6. Updating Deployment Files**

* Open vproapp.yml in **VS Code** and modify the image reference.

**Replace Hardcoded Image Name with Variables**

yaml

image: "{{ .Values.appimage }}:{{ .Values.apptag }}"

**7. Defining Variables in Helm**

* In **values.yaml**, define:

yaml

appimage: "your-app-image"

apptag: "latest"

* But instead of keeping static values, you'll **pass them dynamically** via Helm commands.

**8. Installing Helm Charts Manually**

bash

helm install vprofile-stack helm/vprofilecharts --set appimage=<IMAGE> --set apptag=<TAG>

* The --set flag **injects values dynamically** at runtime.

**9. Automating Deployment via GitHub Actions**

* Add the **final job** in the workflow: **Job Name: Deploy\_To\_EKS**
* Ensure it **depends on** BUILD\_AND\_PUBLISH:

yaml

needs: BUILD\_AND\_PUBLISH

* Steps:
  1. **Pull source code (Helm charts).**
  2. **Set AWS credentials using GitHub Actions**.
  3. **Generate the Kube config file using AWS EKS commands.**
  4. **Store Docker registry credentials in Kubernetes** for authentication.
  5. **Deploy using Helm.**

**Final Outcome**

* **Dynamic Image Referencing**
* **Efficient Kubernetes Deployment via Helm**
* **Fully Automated Pipeline with GitHub Actions**