Aim:

The main aim of the task is to integrate new functionalities to the existing template (that servers for creating new repositories with default settings) that allows both Platform Engineers and Data Engineers to generate repositories specific to their workloads.

Proposed Solution:

I am modifying the script to accept parameters or flags indicating the type of repository being created for example I am adding a "—engineer-type" flags with options "platform" and "data".

- Based on the engineer-type specified execute relevant functionalities within the script. For example, for platform engineers.
  * Setup Terraform configurations.
  * Add Terraform specific **tf files.**
  ***Configure CI/CD pipelines.

  For Data engineers
  *Setup python configurations.
  *Add python specific files such as **main.py, requirements.txt**
  ***** Configure CI/CD pipelines.

Best Practices:

- Implement functionality to enforce best practices specific to each engineer type. This cloud involves below.
- Providing predefined template for terraform configuration or python.
- Enforcing code linting and formatting rules.
- Implementing pre commits hooks for quality checks.
- Include Readme file to explain the repository usage.
- Add permissions where platform engineers can be able to access to terraform configuration. Data engineers can be able to access data scripts.

Example:

- Below example explains clearly how platform engineer's repository works.
- When a platform engineer executes the template, it will create a repository and the repository by default contains below files as template.
  1. **.git ignore :-** The purpose of gitignore files is to ensure that certain files not tracked by Git remain untracked. By default it contains **.terraform, .terraform.tfstate, *.tfstate*.**
  2. **Provider.tf :-** The purpose of this file is to specify the provider type and its version along with terraform version.
  3. **Main.tf:-** The purpose of this file is to specify the actual terraform code.
  4. **Backend.tf:-** It stores the backend configuration based on provider.
  5. **Variable.tf :-** Place where we declare variables .
  6. **Workflow.yml :-** It contains CI/CD configuration for repository.

Example code snippet in yaml:-

```yaml
name: Terraform CI/CD

on:
  pull_request:
    types: [opened, synchronize, reopened, ready_for_review]

jobs:
  tflint:
    name: TFLint
    runs-on: ubuntu-latest
    steps:
      # Steps to run TFLint

  tfplan:
    name: TFPlan
    runs-on: ubuntu-latest
    needs: tflint
    steps:
      # Steps to generate Terraform plan

  code_scanning:
    name: Code Scanning
    runs-on: ubuntu-latest
    needs: [tflint, tfplan]
    permissions:
      actions: write
      contents: read
    strategy:
      matrix:
        language: [ 'terraform' ]
    steps:
      # Steps to perform CodeQL analysis

  tfapply:
    name: TFApply
    runs-on: ubuntu-latest
    needs: tfplan
    if: |
      github.event_name == 'pull_request' &&
      github.event.action == 'closed' &&
      github.event.pull_request.merged == true &&
      contains(github.actor, 'team/') &&
      github.actor == 'team/your-team-name'
    steps:
```

 # Steps to apply Terraform changes

Workflow explanation:

- Pull Request is opened.
- TFLint runs to check the Terraform code.
- Terraform plan is generated.
- Code scanning (e.g., CodeQL analysis) is performed.
- If the pull request is closed and merged by an approved user, Terraform changes are manually applied.

Achievements:

1. Code Quality and Efficiency:
   TFLint performs static analysis of Terraform configuration files (.tf and. tfvars files) without executing them. This allows it to catch issues early in the development process, even before applying changes to infrastructure. These rules cover various aspects such as syntax errors, deprecated features, security vulnerabilities, naming conventions, and best practices recommended by the Terraform community.
2. Best Practices:
   * Lint tool is used to perform static code analysis.
   * The code_scanning job is added to perform code scanning using CodeQL analysis.
   * Implemented pre-configured hooks to identify bugs in early stage.
   *Implemented workflows to perform CI/CD automation.
   * Limit the permissions granted to the GitHub Actions workflow. Only grant the necessary permissions required for the workflow to function properly.
   * Utilize GitHub Secrets to securely store and access sensitive information. These secrets are encrypted and only accessible to authorized users with the proper permissions.
   * Implement security scanning tools to identify vulnerabilities in your codebase.

NOTE: The same example is applicable for data engineer as well only tools will vary (for terraform we use tflint tool python we use different tool)