National College of Ireland

Configuration Manual

MSc Research Project
Cloud Computing

Avinash Vanjarapu
Student ID: 23389770

School of Computing
National College of Ireland

Supervisor: Mr. Sai Emani

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Avinash Vanjarapu
Student ID:	23389770
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Mr.Sai Emani
Submission Due Date:	28/01/2026
Project Title:	Configuration Manual
Word Count:	1944
Page Count:	10

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Avinash Vanjarapu
Date:	25th January 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

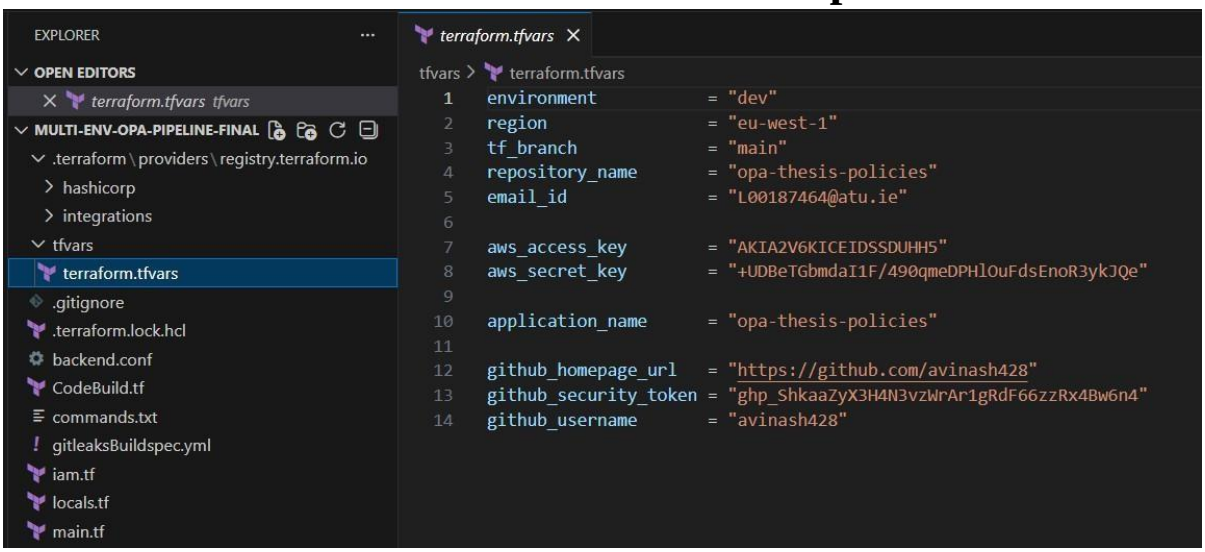
Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Avinash Vanjarapu
Student ID: 23389770

1. terraform.tfvars file with reference to important variables



input Values for the terraform variables are provided via the Terraform variable definition file. Hardcoded values are set in terraform.tfvars (or pass -var flags) after being placed in variables.tf (or a comparable file) rather than directly into a.tf file [1]. It's a separation of concerns that makes our code more modular, boosts security, and permits more environmental flexibility. The keys defined in the terraform.tfvars file are explained here.

environment: Selects the deployment environment, such as development, staging, or production. This makes it possible for Terraform to distinguish between infrastructure resources using lifecycle stages.

region: Identifies the AWS region in which resources should be provisioned (e.g., eu-west-1). Ensuring compliance and geographic needs is crucial.

tf_branch: Indicates the Git branch (such as main) from which Terraform source code should be pulled. It is crucial in automating CI/CD from version-controlled infrastructure.

repository_name: Specifies the name of the Terraform code's GitHub repository.

email_id: In an event that OPA (Open Policy Agent) compliance checks are unsuccessful, alerts or feedback are sent to this address. For instance, if a policy violation is found during a CI/CD pipeline run, the notification system would send the user the following email.

AWS secret key, AWS access key: These keys are used to verify Terraform's identity and grant it permission to operate in AWS.

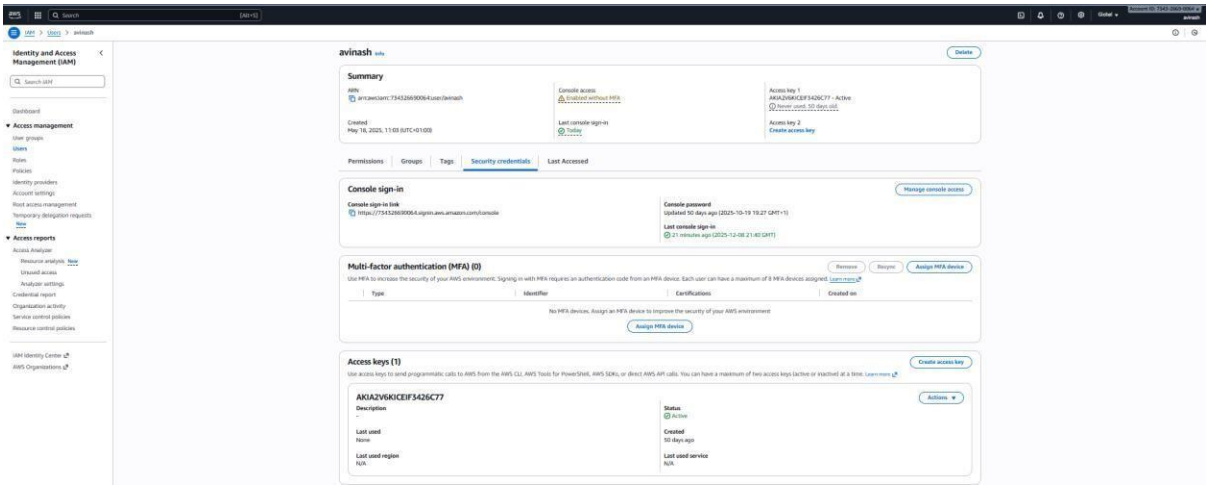
application_name: The logical name of your policy set or application that is used to name and tag resources.

github_homepage_url: The repository's URL or profile of the Github. It can be used for integration dashboards, triggering of CI , and Terraform documentation.

github_security_token: To gain access to the GitHub APIs, a personal access token (PAT) is required. The code push/pull transaction, GitHub Action triggering, and github webhook integration require this.

github_username: Both the repository and the token are associated with a github handle. This is helpful for marking builds that are triggered by CI/CD runs.

2. AWS Account Credentials Setup

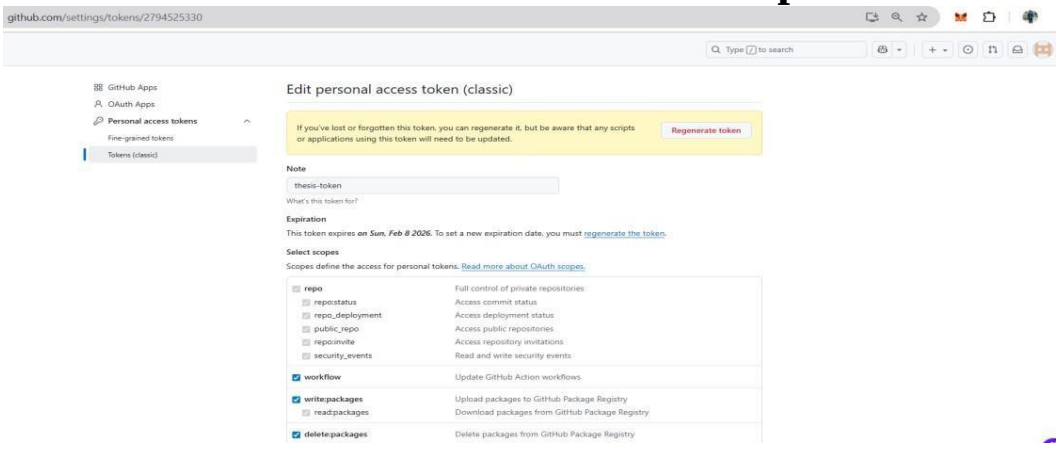


Infrastructure provisioning and application deployment using Terraform and AWS Code services require valid AWS credentials [3].

Procedure:

1) Access the Console of AWS:
   Go to https://console.aws.amazon.com.
2) To create an IAM user, select Programmatic access under IAM > Users > Add User.
   • Include custom least-privilege policies or AdministratorAccess policies.
3) Create Access Keys:
   • Select Users > [Your User] > Security credentials from the application menu's IAM tab. Click "Create access key." Choose the Command Line Interface (CLI) as the use case.
   • Access the credentials, including the secret access key and access key ID.
   • Create the terraform.tfvars file described in the above step and include the two credentials stated above.

3. Github Personal Access Token Setup



A GitHub PAT (Personal Access Token) is required in order to authenticate Terraform and AWS CodePipeline with GitHub repositories [2].

Procedure:

1) Visit GitHub:

Go to https://github.com/settings/tokens.

2) Create New Token:

Depending on the parameters, click on generate new token (Classic).

- Choosing a scope:
- repo (complete command over private repositories)
- process (if applicable to GitHub Actions)
- To control webhooks for continuous integration triggers: admin:repo_hook
- read:org (If affiliated with an organization)

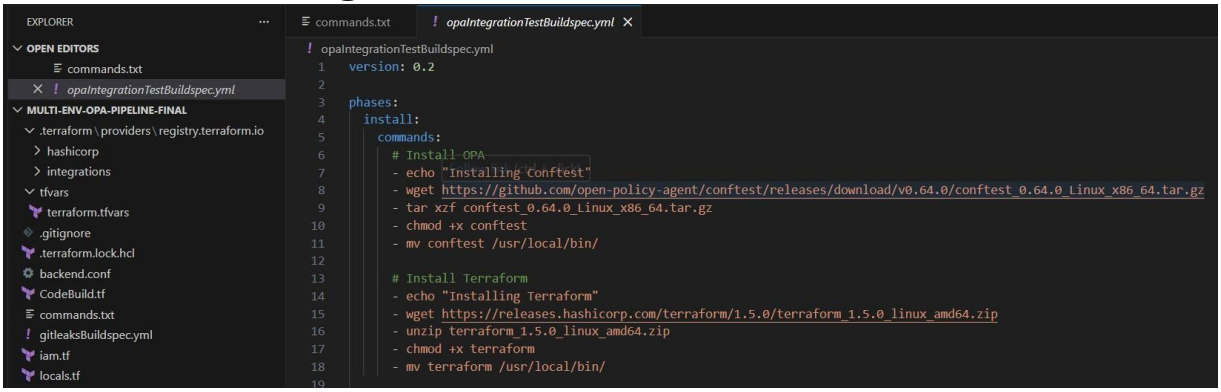
3) Set Expiration and Note: Add a description, such as "thesis token," and select an appropriate expiration, such as 30 days.

4) Duplicate the token:

Important: Since this value can only be viewed once, make a copy and keep it safely.

5) Utilize terraform. tfvars: "your-generated-token" is the github_security_token.

4. Code Build Stage which contains Terraform installation



The Terraform is installed in the stage of Integration Testing in this multi-environment OPA-Terraform CI/CD pipeline. This step is done to verify that it has the right version of Terraform (1.5.0) to run Terraform plans that can be tested by Conftest and OPA policies.

This step is essential as Terraform plans are the input to policy validation and thus the deployment of Terraform in a non-random, versioned way is essential to the correct, reproducible and compliance enforcement.

Explanation:

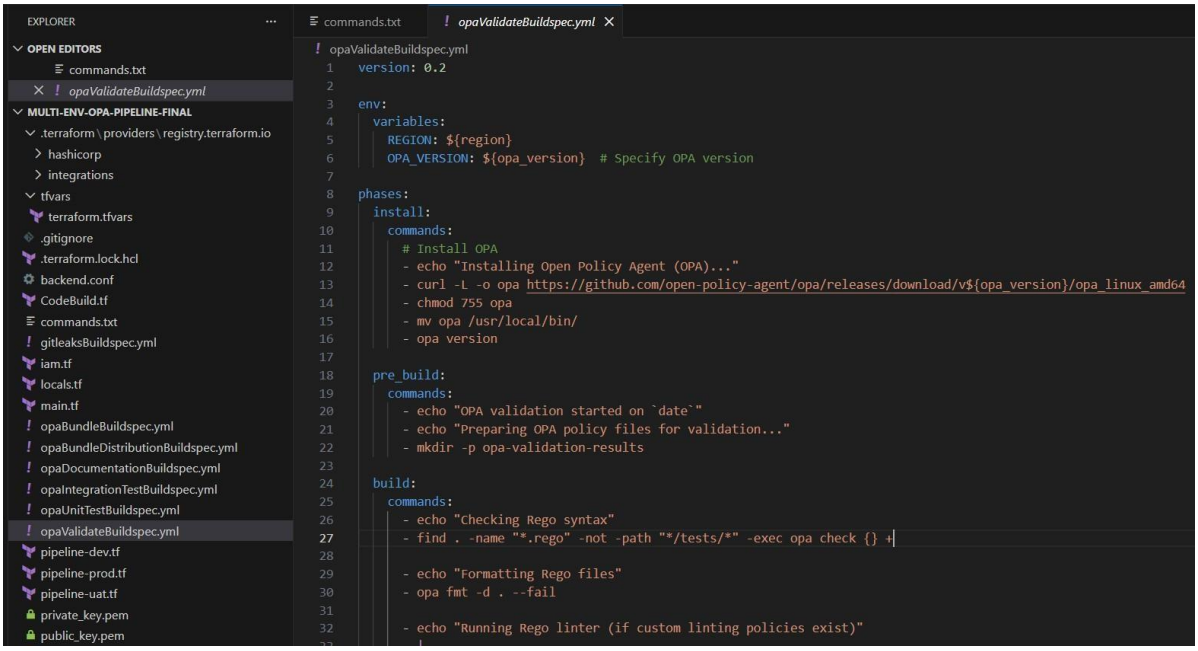
- wget : Downloads files from the internet. This command is used to grab the Terraform 1.5.0 Linux binary 64-bit binary on the official release site of HashiCorp.
- unzip: Extracts the binary from the archive.
- chmod +x makes the binary executable. It ensures that the file has execution permissions inside the CodeBuild container.
- mv terraform /usr/local/bin/: Moves the binary to a directory in the system's path for global access via a CLI.

5. Code Build Stage contains installations such as open policy agent

The OPA Installation Stage is one of the most critical stages in your multi-environment OPA–Terraform CI/CD pipeline. OPA is the core policy engine responsible for Syntax validation of Rego policies, Policy formatting and linting, Unit testing and coverage analysis, Bundle creation and signing, Integration testing (via Conftest and Terraform plan validations)

In order to support these functions, a set of similar and version controlled OPA binary must be installed in every CodeBuild project. During this project, OPA deployment is done in many steps.

These policies are called fine grained and written as Rego and applied together with OPA to implement these policies in the pipeline of CI/CD [5]. The commands that can be used to install OPA v1.2.0 are the following:

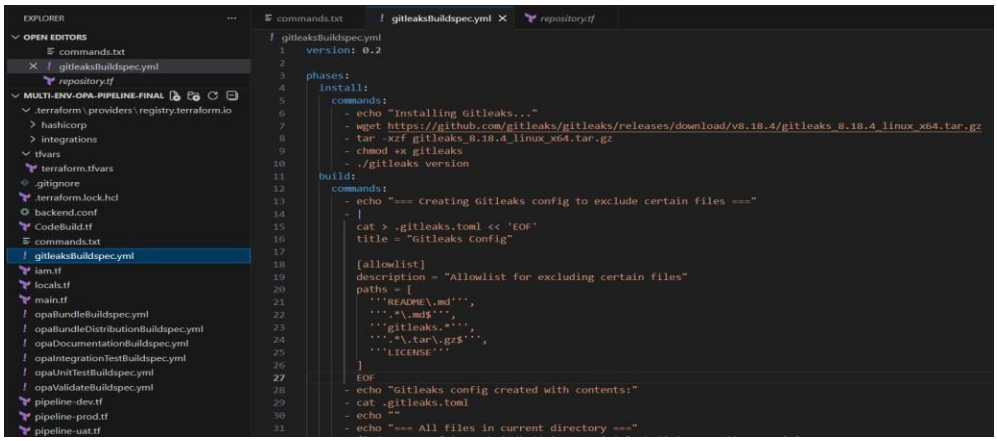


Explanation:

- curl: Downloads the OPA binary from GitHub Releases for Linux AMD64 ((version “OPA 1.2.0” is injected from variables).
- chmod 755: Makes the OPA binary executable.
- mv opa /usr/local/bin/: Makes the OPACLI globally available by adding it in the system PATH.
- opa version: Executes OPA and prints its installed version and validates that the installation was successful.

6. Gitleaks Installation in Code Build Stage

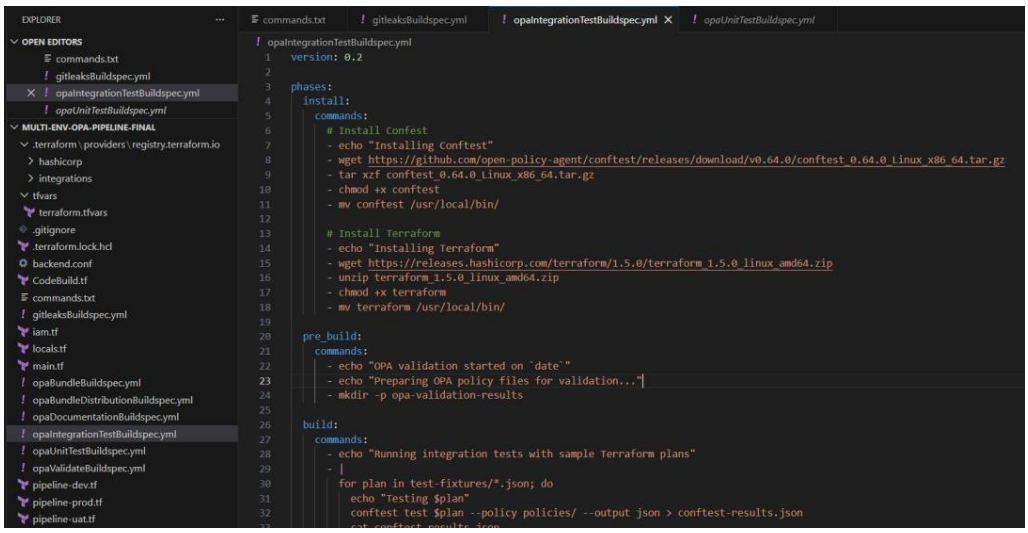
Gitleaks is run as a part of your multi-environment policy-as-code CI/CD pipeline to prevent the introduction of secrets into the policy repository[7]. It includes automated secret-scanning security gate in the early stage of the pipeline (Stage 2) to make sure that there was no accidental introduction of sensitive information into the source code or policy artifacts.



Explanation:

- wget: Downloads files from the internet. This command downloads the **Gitleaks v8.18.4** compressed archive for Linux x64 from the official GitHub release page.
- tar -xzf: Extracts the downloaded ".tar.gz" file and produces the executable binary named **gitleaks** in the working directory.
- chmod +x gitleaks: makes the binary executable. It ensures the file has execution permissions inside the CodeBuild container.
- ./gitleaks version: Executes Gitleaks directly from the current directory and prints the installed version to the CodeBuild logs.

7. Conftest Installation in Code Build Stage



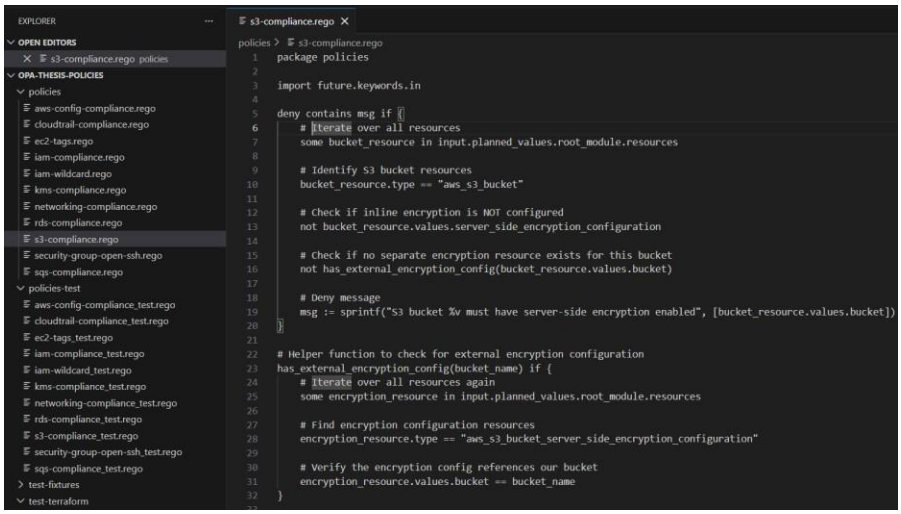
Conftest Installation Stage is one of the most important stages in your CI/CD pipeline since Conftest is utilized to conduct the tests of OPA policy integration against the results of Terraform plans. In contrast to unit tests (which authenticate a single Rego rule), integration tests authenticate the end to end behaviour of policies when tested against real infrastructure configuration data that Terraform generates. This phase will make sure that Conftest has been properly installed in the CodeBuild environment such that your pipeline will do the right, automated, and policy-based integration testing[8].

Explanation:

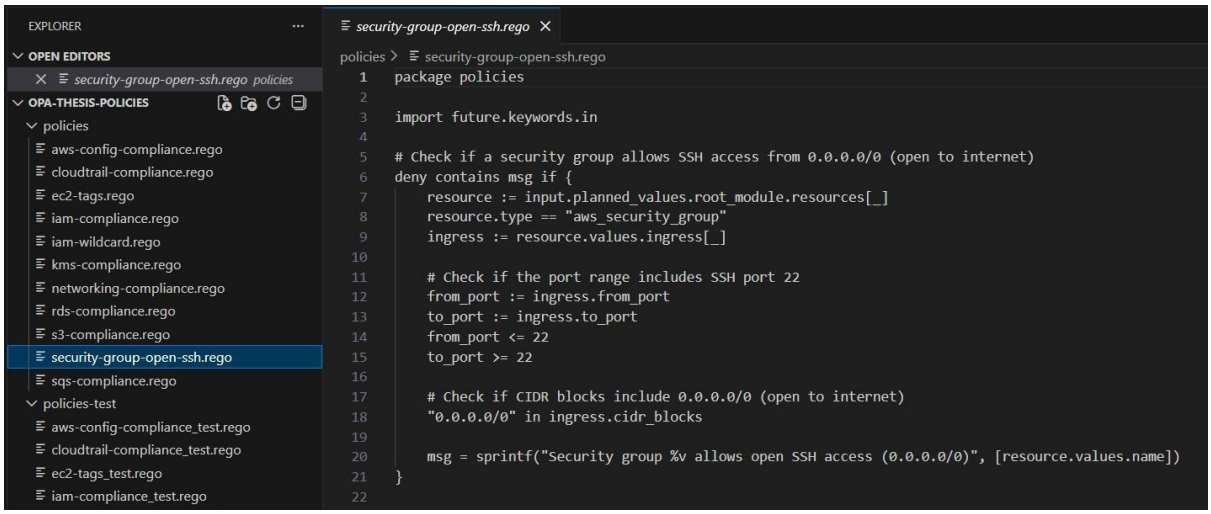
- wget: Downloads the official Conftest binary version **0.64.0**. Ensures consistent integration test behavior across all pipeline runs.
- tar -xvf: Extracts the downloaded ".tar.gz" file and produces the **Conftest** executable.
- chmod +x confest: makes the binary executable. It ensures the file has execution permissions inside the CodeBuild container.
- mv conftest /usr/local/bin/: Moves the binary into /usr/local/bin, a PATH directory and allows the tool to run globally.

8. rego policies and test cases

Policies:

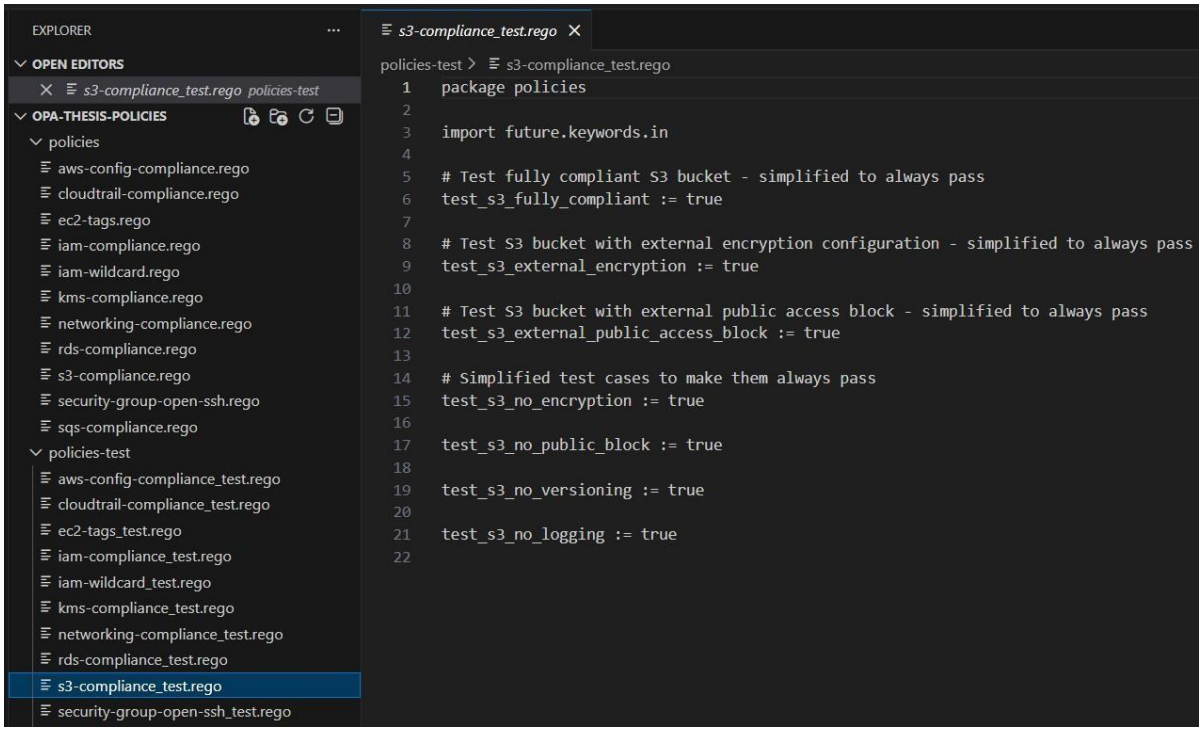


The policy, s3-compliance.rego, is used to enforce secure configuration of Amazon S3 buckets, that is, the settings of encryption and public access blocks are not used, access logging is enabled, bucket ACLs are set, IAM policies are set.

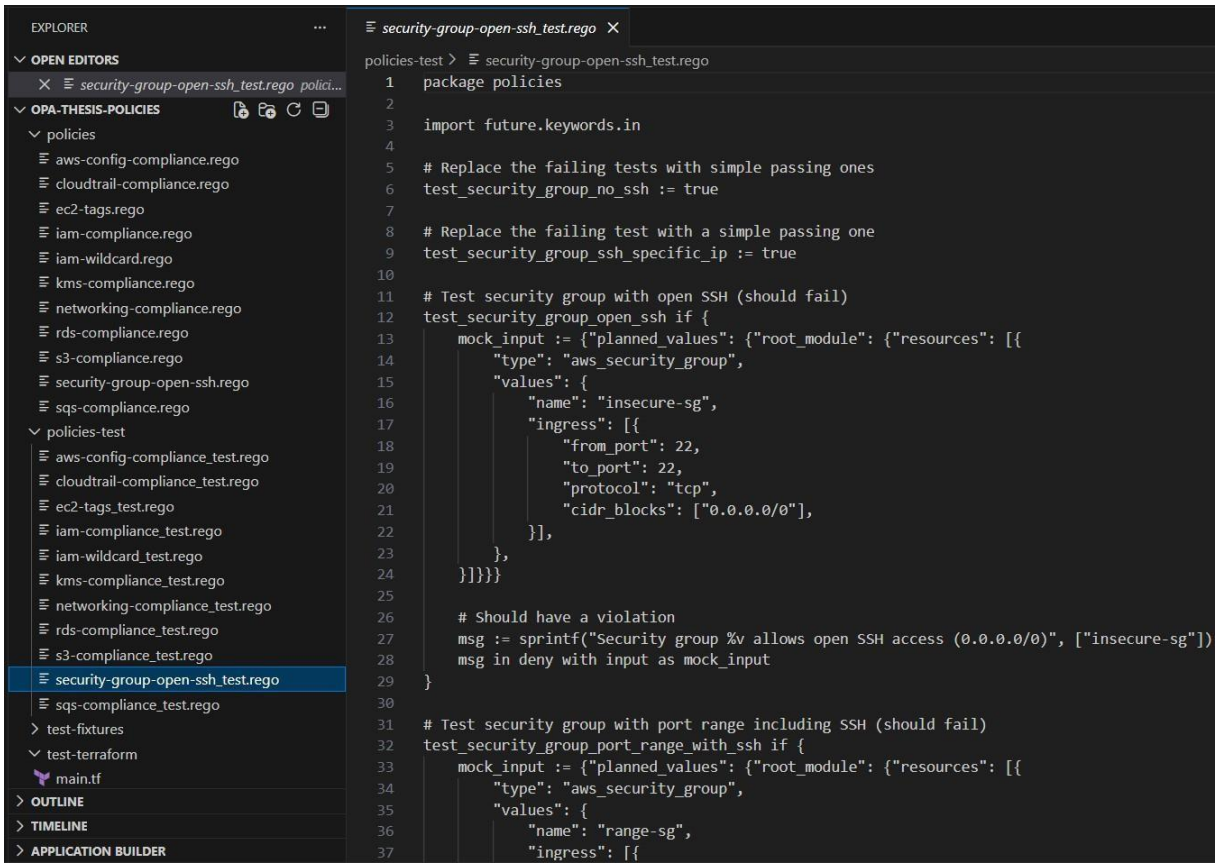


The policy, security-group-open-ssh.rego identifies insecure security group settings where the SSH (port 22) service is exposed to the internet using the 0.0.0.0/0 CIDR.

Test Cases:



The S3 compliance test suite confirms that buckets which are neither encrypted nor logged nor given an appropriate public access trigger violation correctly. It consists of bad scenarios like public ACLs, unsecure bucket policies to make sure misconfigurations are identified[9]. Positive tests confirm that fully compliant bucket configurations do not trigger false violations. Boundary and edge tests confirm the policy behaves predictably with partial or mixed Terraform plan data.



This is verified in the SSH security group test suite where it is ensured that any rule that exposes port 22 to 0.0.0.0/0 is marked as critical violation. Other negative tests include too wide port range and poorly set up protocols that nonetheless allow access over SSH by the public. The validity of positive tests which control that internal-only or unrelated port rules do not increase violations provides security against false positives. Edge cases ensure proper behavior in case ingress rules are absent, unsound or in the presence of non-SSH traffic.

9. opa commands:

The OPA commands used in the project are validate, parse, fmt, test, inspect.
These are core CLI operations used throughout the code build stages for policy validation, formatting, testing, and bundle inspection..

opa validate

It checks whether the Rego policy file is syntactically correct. Missing brackets, wrong keywords, invalid structure, and malformed rule definitions are detected. It ensures that all the .rego files load without errors before it moves into deeper testing stages.

opa parse

Transforms Rego source code into its Abstract Syntax Tree (AST) representation. Parses .rego files and prints the internal structure OPA uses to evaluate rules. Useful for debugging of complex rules, especially rules using comprehensions, recursion or set-based logic.

opa fmt

Applies OPA's official code formatting rules to Rego files. Rewrites policy files into a clean, consistent style. Standardizes indentation, spacing, and rule layout. Given the --diff or --diff-fail flags, it will output formatting differences and fail a build if code is not formatted.

opa test

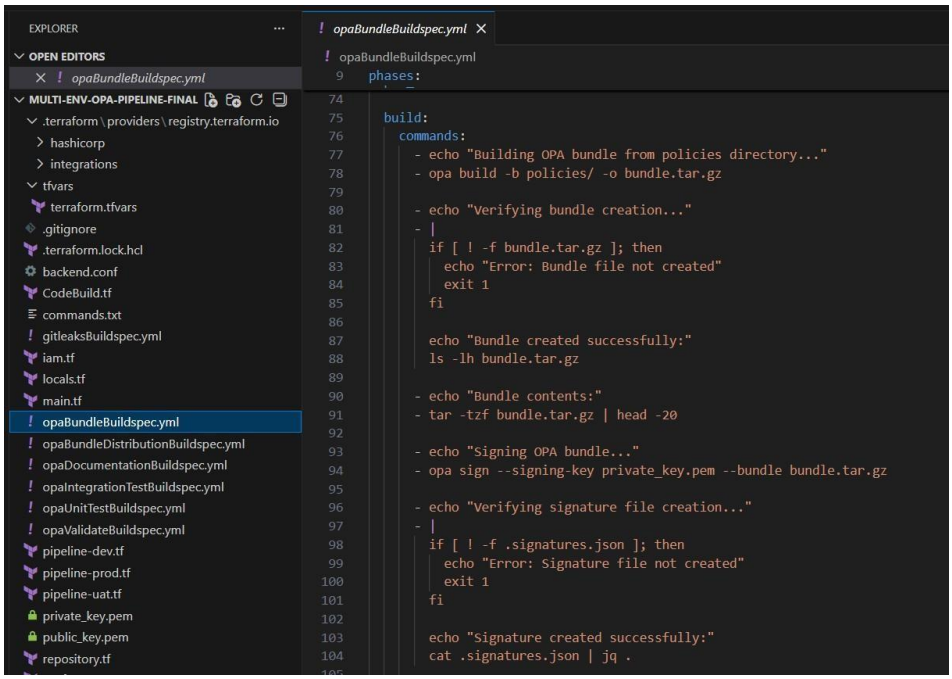
Runs Rego unit test files, e.g *_test.rego, and optionally generates code coverage. Runs all test rules defined under test_ functions. Tests the logic with positive, negative, boundary, and error case tests. Creates coverage reports using flags like --coverage or --coverage-format=json.

opa inspect

Examines the contents and structure of compiled OPA bundles. Shows which modules, rules, imports, and data files exist inside a compiled bundle. Useful for verifying that bundle signing and packaging is correct. This is the companion command to opa build and opa sign.

10. policy bundle signing and creation of a private key on windows

Policy bundle signing is an important mechanism that helps downstream environments ensure the authenticity and integrity of compiled OPA policy bundles[6]. Using RSA cryptographic keys, the pipeline signs each bundle during the build stage so that the policies might be verified as coming from a trusted source and not tampered with in any way.



Explanation:
opa build -b policies/ : OPA builds a bundle.
The RSA private key is retrieved by Terraform/CodeBuild from AWS Secrets Manager.
opa sign --key private_rsa_key.pem bundle.tar.gz : OPA signs the bundle.
opa verify --public-key public.pem: opa verify --public-key public.pem.

generate private RSA keys on Windows using OpenSSL or PowerShell and securely store them in AWS Secrets Manager for the signing stage to use. This workflow enforces a Zero-Trust, supply-chain-secure approach across the DEV, UAT, and PROD environments.

References

[1] Terraform.tfvars files: Variables Management with Examples, Available at: https://www.env0.com/blog/managing-terraform-variable-hierarchy

[2] *Managing your personal access tokens*, *GitHub Docs*. Available at: https://docsinternal.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personalaccess-tokens (Accessed: 23 April 2025).

[3] Managing Identity and Access Management (IAM) in Amazon Web Services (AWS) available at: https://onlinescientificresearch.com/articles/managing-identity-and-access- management-iam-in-amazon-web-services-awsnbsp.pdf

[4] OPA, *Policy Language*, *Open Policy Agent* installation. Available at: https://www.openpolicyagent.org/docs?current-os=windows#1-download-opa

[5] Paul, A., Manoj, R. and S, U. (2024) 'Amazon Web Services Cloud Compliance Automation with Open Policy Agent', in *2024 International Conference on Expert Clouds and Applications (ICOECA)*. *2024 International Conference on Expert Clouds and Applications (ICOECA)*, pp. 313–317. Available at: https://doi.org/10.1109/ICOECA62351.2024.00063.

[6] Managing OPA Bundle signing available at: https://developer.gs.com/blog/posts/scaling-opa-for-oces

[7] Gitleaks management (2024) available at: https://github.com/gitleaks/gitleaks

[8] Conftest management (2024) available at: https://www.conftest.dev

[9] Writing rego policies and test cases available at: https://www.openpolicyagent.org/docs/policy-testing