Secure Terraform Delivery Pipeline

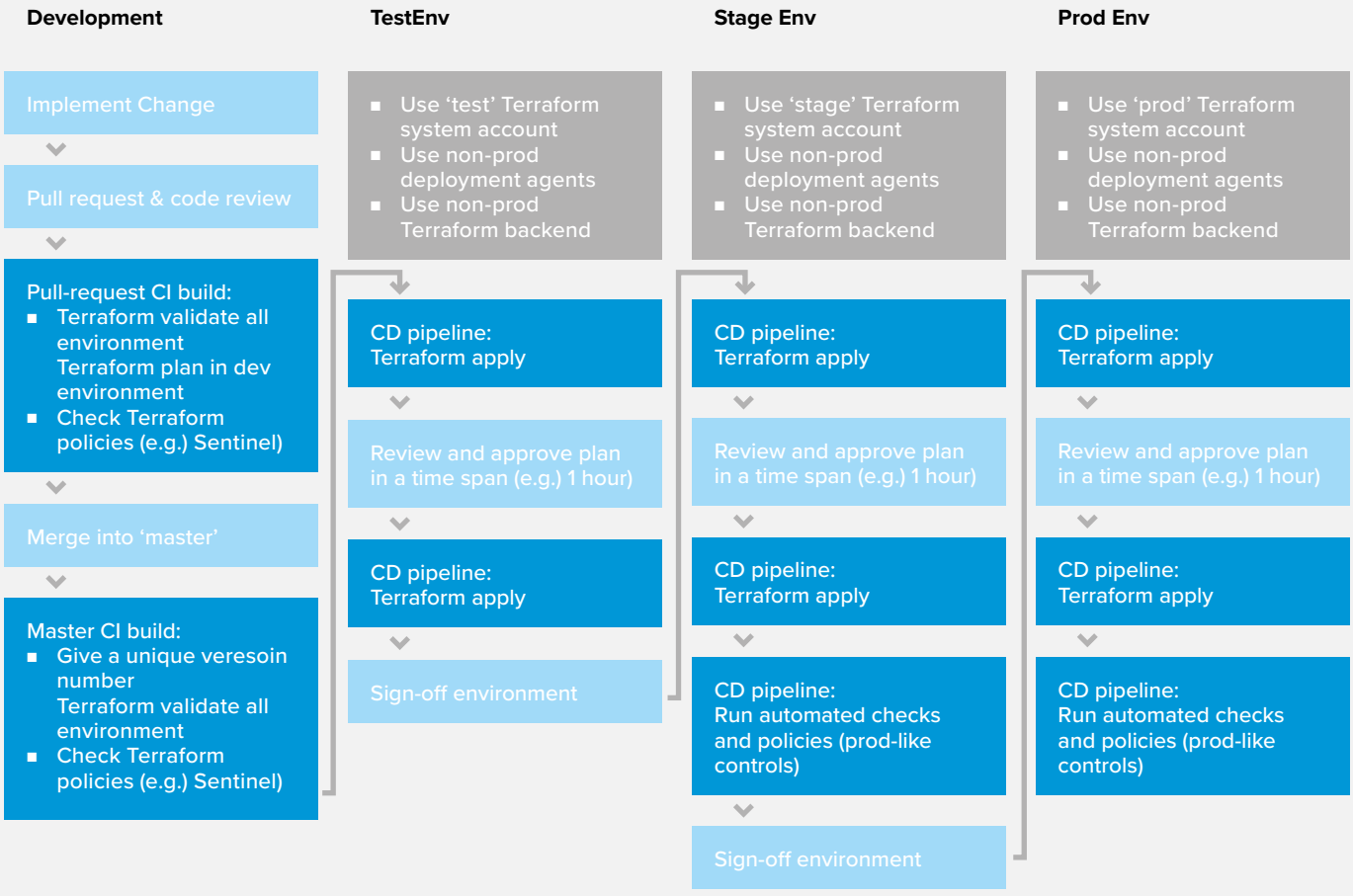
With the beginning of the cloud era, the need for automation of cloud infrastructure has become essential. Although still very young (version 0.12), Terraform has already become the leading solution in the field of Infrastructure as Code. A completely new tool in an emerging area, working in a new programming model – this brings a lot of questions and doubts, especially when handling business-essential cloud

01. Why a secure Terraform pipeline is needed?

At GFT, we face challenges of delivering Terraform deployments at scale: on top of all major cloud providers, supporting large organizations in a highly regulated environment of financial services, with multiple teams working in environments in multiple regions around the world. Automation of Terraform delivery whilst ensuring proper security and mitigation of common risks and errors is one of the main topics across our DevOps teams.

The goal is to create a process that allows a user to introduce changes into a cloud environment without having explicit permissions for manual actions. The process is as follows:

- A change is reviewed and merged with a pull request after a review of the required reviewers. There is no other way to introduce the change.
- The change is deployed to a test environment. Before that, the Terraform plan is reviewed manually and approved.
- The change needs to be tested/ approved in a test environment.
- The Terraform plan is approved for the staging environment. The change is **exactly** the same as in the test environment (e.g. the same revision).
- Terraform changes are applied to staging using a designated Terraform system account. There is no other way to use this Terraform account as in this step of the process.
- Follow the same procedure to promote changes from staging to the production environment.



01.1 Non-functional requirements ┓

Environments
Environments (dev/uat/stage/prod) have a proper level of separation ensured:

- Different system accounts are used for Terraform in these environments. Each Terraform system account has permissions only for its own environment.
- Network connectivity is limited between resources across different environment
- (Optional) Only a designated agent or set of agents configured in a special virtual network is permitted to modify the infrastructure (i.e. run Terraform) and access sensitive resources (e.g. Terraform backend, key vaults etc). It is not possible to release to e.g. prod using a non-prod build agent.

There is a way to ensure that Terraform configuration is as similar as possible between environments. (I.e. I cannot forget about the whole module in PROD as compared to UAT)

Terraform backend in higher environments (e.g. UAT) is not accessible from local machines (network + RBAC limitation). It can be accessed only from build machines and optionally from designated bastion hosts.

System accounts for Terraform
Terraform runs with a system account rather than a user account when possible. Different system accounts are used for:

- Terraform (a system user that modifies the infrastructure),
- Kubernetes (a system user that is used by Kubernetes to create required resources e.g. load balancers or to download docker images from the repo
- Runtime application components (as compared to build-time or release-time)

System accounts that are permitted to Terraform changes can be used only in designated CD pipelines. I.e. it is not possible that I can use e.g. a production Terraform system account in a newly created pipeline without a special permission. (Optional) Access to use the Terraform system account is granted “just-in-time” for the release. Alternatively, the system account is granted permissions only for the time of deployment.

System accounts in higher environments have permissions limited to only what is required in order to perform actions. Limit permissions to only the types of resources that are used.

Remove permissions for deleting critical resources (e.g. databases, storage) to avoid automated re- creation of these resources and losing data. On such occasions, special permissions should be granted “just-in-time”.

Process
A change to a higher environment (e.g. STAGE) can be deployed only if it was previously tested in a lower environment. There is a method to ensure that this is exactly the same Git revision tested. The change can only be introduced with a pull request with a required review process.

An option to apply Terraform changes can be only allowed after manual terraform plan review and approval on each environment.

02. Implementing a secure Terraform pipeline

TIP! Take a look at this external documentation to start setting up Terraform in CI/CD pipelines:

- Running Terraform in Automation
- Terraform Cloud
- Terraform Enterprise
- How to Move from Semi-Automation to Infrastructure as Code
- How to Move from Infrastructure as Code to Collaborative Infrastructure as Code

02.1 Terraform backends

Having a shared Terraform backend is the first step to build the pipeline. A Terraform backend is a key component that handles shared state storage, management, as well as locking, in order to prevent infrastructure modification by multiple Terraform processes.

Some initial documentation:

- Terraform Backend Configuration
- backend providers list
- AWS s3
- GCP cloud storage
- Azure storage account
- Remote backend for Terraform Cloud/ Enterprise

Make sure that the backend infrastructure has enough protection. **State files will contain all sensitive information that goes through Terraform (keys, secrets, generated passwords etc.).**

- This will most likely be AWS S3+DynamoDB, Google Cloud Storage or Azure Storage Account.
- Separate infrastructure (network + RBAC) of production and non-prod backends.
- Plan to disable access to state files (network access and RBAC) from outside of a designated network (e.g. deployment agent pool).
- Do not keep Terraform backend infrastructure with the run-time environment. Use separate account/project/subscription etc.

Enable object versioning/soft delete options on your Terraform backends to avoid losing changes and, state-files, and in order to maintain Terraform state history. In some special cases manual access to Terraform state files will be required. Things like refactoring, breaking changes or fixing defects will require running Terraform state operations by operations personnel. For such occasions plan extraordinary, controlled access to the Terraform state using bastion host, VPN etc.

When using Terraform Cloud/Enterprise with remote backend the tool will handle the requirements for state storage.

02.2
Divide into multiple projects

Naturally, Terraform allows you to divide the structure into modules. However, you should consider dividing your entire infrastructure into separate projects. A "Terraform project" in this description is a single piece of infrastructure that can be introduced in many environments, usually with a single pipeline. Terraform projects will usually match cloud architectural patterns like Shared VPC, Landing zone (Azure and AWS), hub-and-spoke network topology. There are many patterns in AWS Well-Architected Framework, Azure Cloud Adoption Framework, Architecture Center or Google Cloud Solutions.

Here are some samples:

Terraform Bootstrap
This is needed when Terraform remote state-files are stored in the cloud. This is going to be a simple project that will create the infrastructure required for the backends of other projects. In general, avoid stateless projects. But this will be one of them (the old chicken and egg problem).

Landing Zone
Have a separate project (or projects) to set-up the presence in the cloud - a network or a VPN connection, core resources, security baseline. Building a landing zone is a separate topic. See for example https://www.tranquilitybase.io/

Shared build-time infrastructure
A piece of company infrastructure, usually global, that handles the build-time operations. For example:

- Build agent pools
- container registry (or registries)
- core key vaults storing company certificates
- global DNS configurations etc.

Host runtime infrastructure
Usually, runtime environments have some prerequisites and pieces of infrastructure that might be shared between prod and non-prod environments, such as bastion hosts, DNS, key vaults. This is also a good place to configure deployment agent pools separate for production and non-prod environments (you may not need separate for dev1, dev2, uat1, uat2 etc.)

Runtime environments
Naturally, this is the infrastructure under the applications and services serving the business. Be sure that there is an environment to test Terraform scripts, not necessarily the same that the application is tested, in order to avoid interrupting the QA team's work when applying potentially imperfect Terraform configurations.
Moreover, be prepared to divide runtime environments across teams, services, departments. It might be impossible to have a single project with the whole "company production" environment.

Some general situations that suggest dividing infrastructure into projects:

Use different system accounts for different security levels
Have a separate project when you need to use a different Terraform system account for pieces of infrastructure. Otherwise, you'll have to give very wide permissions for a single system account. Examples:

- pieces of infrastructure across multiple projects or organizational units,
- build-time infrastructure vs runtime infrastructure,
- separate systems,
- shared infrastructure vs single system/service,
- serving different regions.

Have a different set of environments
When you identify that for one piece of infrastructure you only need "prod" and "non-prod" and for the other part you will have "dev", "uat", "stage" or "prod", then this is a sign that these pieces of infrastructure should be separate.

Build layers and overlays
If the Terraform configuration in one project grows too big it might become challenging to handle it. Constructing Terraform plans will become slower and refactoring might be very risky.
It might be a good idea to divide the entire infrastructure into layers. Overlays may initially look like optional modules required only in certain environments.
Here are some theoretical examples:

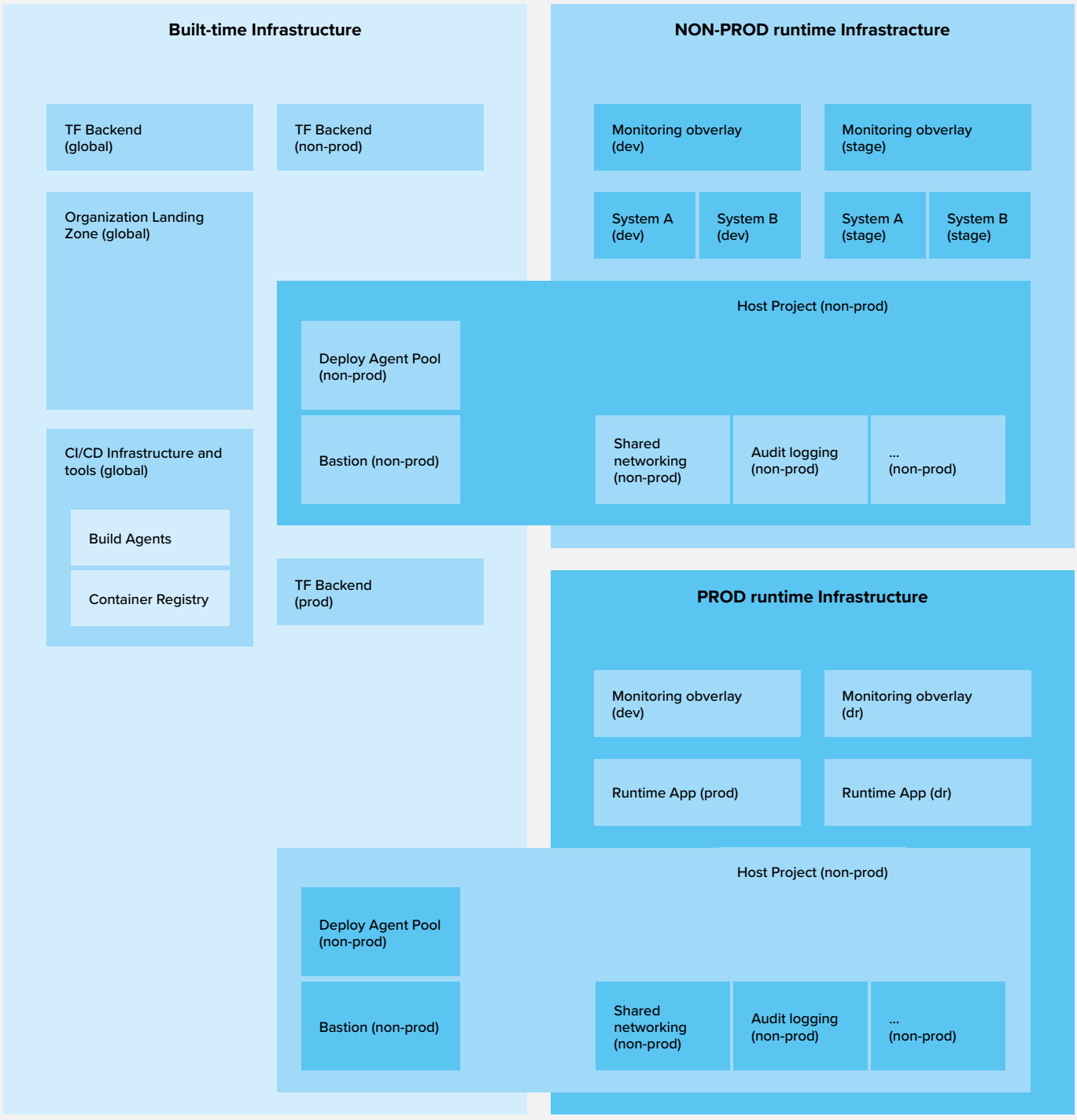
- Shared networking layer - virtual networks, firewalls, VPNs,
- Core infrastructure - compute, storage, Kubernetes,
- Application layer - messaging services, databases, key vaults, log aggregation,
- Monitoring overlay - custom metrics, health checks, alerting rules.

Serve different departments/systems
Pieces of infrastructure that have different change sources or serve different business areas or departments, usually have different security levels – in such cases, access control may be treated separately with different Terraform pipelines, release cycles etc. Try to avoid huge monolithic configurations.

TIP! When resources of multiple projects need to interact with each other and rely on each other you can use Terraform data sources to "reach" resources from different projects. Terraform allows to have multiple providers of the same type that, for example, access different projects/accounts/subscriptions.

Example: use separate provider and data source to "find" the company's global VPN gateway subnet for setting up network connectivity for the runtime environment.

An example of how to divide infrastructure into Terraform projects:



02.3 Handle environments separately

Prepare to handle multiple environments on day 1. This will be a very complicated change later and may require heavy refactoring.

- Make sure that each environment of each project has its own state-file. Don't keep multiple environments (dev/stage/prod) in a single Terraform state-file.
- Use different Terraform system accounts for environments. Make sure early on that the system accounts have limited permissions and cannot access each other's infrastructure.
- Lock-down access to e.g. staging state file early. It will force you to think about building an automated and secure pipeline quickly.
- Prepare non-prod and prod deployment agent pools early. Lockdown network access to storage, key vaults, Kubernetes API etc. from specific virtual networks of deployment agent pools.

Keep in mind that Terraform does not allow using variables in the provider and backend sections. A simple approach with multiple '.tfvars' files may be challenging in the long run.

Some options for handling environments

- Use Terraform Workspaces and terraform init --backend-config option to switch backends and environments.
- Use the module and directory layout (e.g. Terraform main-module approach) that allows handling multiple environments by making use of module structure

When building multiple environments, make sure that you handle differences properly. Different environments will have different pricing tiers, VM sizes, or even some resources totally disabled (WAF, DDoS protection).

- Use parameters and "feature flags" to enable/disable optional features combined with "count" and "for_each" construct in Terraform,
- If you use defaults make sure that default is the production setting and you override non-prod environment settings,
- You can easily point the differences per environment and you know how feature changes across environments,
- Have an explicit prod-like environment to test with 1:1 production settings. Have an explicit test for that in the release procedure.

TIP! In the pipeline's build step always run "terraform validate" for all environments. Make sure that this step fails if you forget to set a property. Make sure that it is not possible to forget about an entire module in your environment.

02.4 Organize into modules

Terraform Modularization is a wide topic with a primary purpose of building a catalogue of reviewed, maintained and reusable infrastructure components. Besides a global, public Terraform Registry, companies build their own module libraries. This can be done either with the use of Terraform Cloud/Enterprise or with use of Git repositories.

Nevertheless, even if a piece of a Terraform project does not look like a shared module, it might be worth to encapsulate it into a submodule. There are major benefits:

- large infrastructure code decomposition,
- focusing on single responsibility per module,
- the code readability improvement,
- following the same
- tracking the dependencies (variables and outputs) between logical pieces of infrastructure.

Like with any other programming language, modularization brings value even if modules are an integral part of a single repository and are used just once.

03. Build a Terraform Pipeline with CI/CD

NOTE: Terraform Cloud/Terraform Enterprise is an opinionated solution addressing some of the aspects of implementing a secure Terraform pipeline such as:

- Remote Backend
- Module registry
- Terraform plan review and approval
- Remote Terraform execution
- Secure system accounts credentials handling

Since it can be used through its UI as well as API and CLI interfaces it can be used together with a CI/CD tool, however, Terraform Cloud/Enterprise is not a CI/CD tool itself.

This will depend a lot on the tool that is available. Each CI/CD tool will have different features and approaches, more or less terraform specific. There are two general options:

Use a built-in mechanism of the CD tool for release control

- Build-in package versioning
- Secured pipeline/build artifacts
- Release pipelines with the environment promotion process
- Manual checks and approvals by entitled people
- Release plans, rollback plans
- Handling system accounts and secrets

Follow a GitOps approach

- Rely on Git repositories, branches and tags to control the process
- Rely on Git access control, permissions and pull request enforcement
- Use Git webhooks
- Have a separate "code" repository and "delivery" repository for GitOps control

Besides Terraform Cloud/Enterprise a popular free tool that supports GitOps Terraform control is
› https://www.runatlantis.io/

General steps to implement a secure Terraform pipeline

01. Protect the "master" branch
Configure your Git in such a way that "push" to the "master" branch is forbidden and allowed only with a pull request (or merge request). Set the required steps (e.g. Terraform "build" must pass) and required number of approvers from a list.
› **Ask:** Can I put an arbitrary change into a branch that is the source of a release?

02. Build a multi-stage pipeline
If possible, build a pipeline that will visualize that a certain Terraform configuration version is promoted from one environment to another.
› **Ask:** Can I put something in production without testing in staging? Can I deploy something from an arbitrary branch other than "master"?

03. Rely on versions rather than branches
When you are building your pipeline make sure that you promote an immutable snapshot of your Terraform configuration. Avoid using branches for this. Promote a build artifact or a Git tag instead.
› **Ask:** Am I 100% sure that I'm deploying exactly the same version as tested before?

04. Have a Terraform "build" step
It is not obvious that the Terraform code can be built. However, at least validating Terraform against configurations of all environments and running a plan on a designated environment will allow catching obvious errors. Terraform validate action can be executed without connecting to a remote backend. It is worth doing it on the master branch as well as on a pull request. This can also allow setting a unique version number or a tag on the master branch version.
› **Ask:**: How do I ensure that the Terraform code has at least proper syntax and complete configuration?

05. Have a manual approval step of Terraform plan
This will be one of the hardest requirements to implement in most the tools. Start from here. This is to ensure that the plan is reviewed and approved by a person and exactly this plan is applied.
› **Ask:** How do I review what is going to change in each environment?

06. Ensure that there is only one pending plan per environment
When a plan is waiting for approval and some other plan is applied, the first plan is not true anymore. Prevent concurrent plans pending on a single environment.
› **Ask:** How do I ensure that no other changes are applied between "plan" and "apply" steps?

07. Protect access to separate system accounts per environment
Multiple systems accounts need to be prepared for different environments. The goal of the pipeline is to ensure that the access to the system account is provided securely (i.e. credentials/keys are hidden/secret, write-only, are not logged in the console). Only an approved pipeline should be able to use this system account.
› **Ask:** Am I able to use the production Terraform system account in a non-prod pipeline or non-prod stage?

08. Protect access to separate agent pools per environment
Agent pools for non-prod and prod deployments should be separate. Network access to services like Kubernetes API, key vaults, sensitive storage etc. should be limited, including the deployment build agents. However, non-prod build agents must not have access to the production runtime environment. Only approved pipelines should be able to execute on production deployment agent pool.
› **Ask:** Is it possible to run an arbitrary pipeline on production deployment agent pools?

09. Have a rollback plan
Rolling changes back usually means running a release for a previous version of the Terraform configuration.
› **Ask:** How do I run a previous version?

10. Control user permissions to environments
Make sure that only certain people can deploy changes to certain environments. Implement a four-eye-check (approval by at least 2 people) for production releases. Have control over initializing a release and Terraform plan approval.
› **Ask:** Can I approve the Terraform plan in production if I am not permitted?

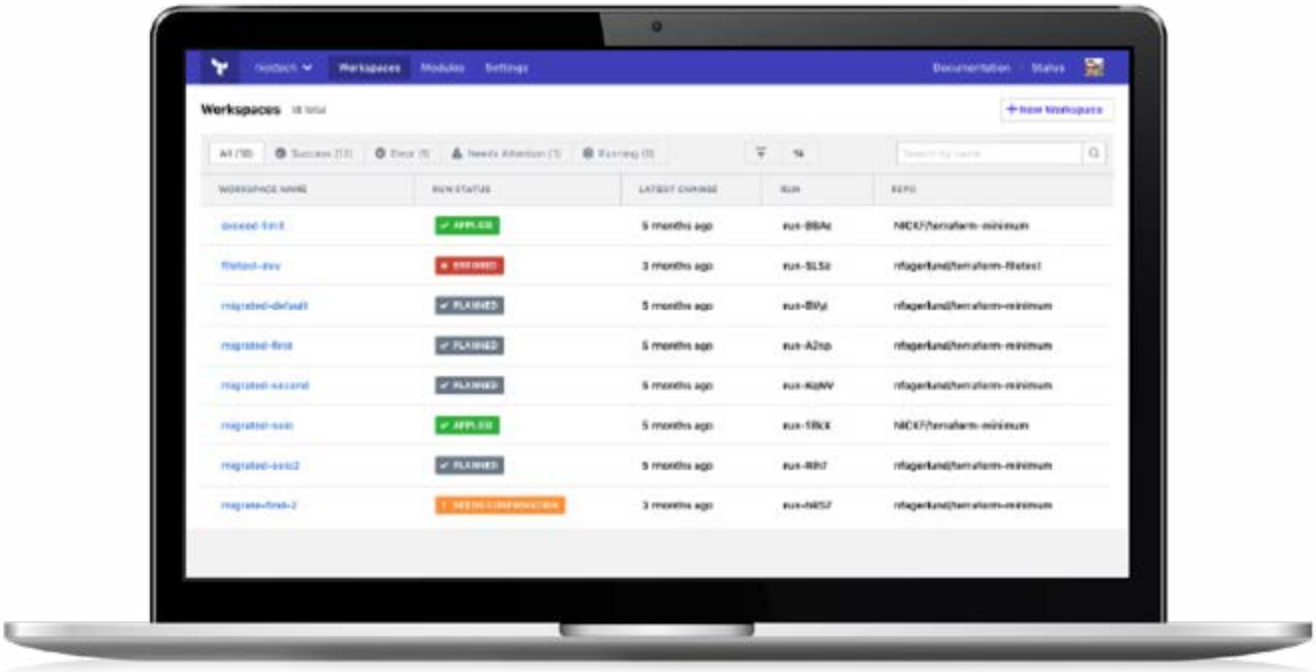
11. Have just-in-time access control for Terraform
Introduce checks into the process to ensure that the production Terraform system account will be available only during the time of a planned release. Alternatively, it can be granted production-level RBAC permissions only for the time of the release. This is to make sure that proper personnel have access to both the CI/CD pipeline as well as to the cloud provider during the release. This is another method of a four-eye-check. Think: multi-factor authentication for CI/CD.
› **Ask:** Can I perform production release with access to CI/CD pipeline only?

12. Run tests as part of the pipeline
Like any other piece of software, virtual infrastructure can and should be tested. Below is the Continuous Compliance section with some testing guidelines. Make sure that after deployment to at least prod-like and prod environments, there is a step to verify compliance, security policies and run some tests, even smoke tests. The purpose is to have quick feedback.
› **Ask:** Will I immediately know if my changed infrastructure is not compliant with non-functional requirements and policies?

Example of a release pipeline with the use of Azure DevOps

- Code version is stored as an artifact and promoted from environment to environment
- All “Plan” and “Apply” stages are controlled by pre-stage and post-sage approvals.
- The tool maintains the system credentials per environment and ensures access to deployment agents.

Example GitOps process with Terraform Cloud:



from terraform.io

- There is a “delivery” repository (or repositories) to control the actual deployment to environments
- The “delivery” repository will import modules from “code” repositories or module registry
- The release procedure starts with a pull request with a change to a given repository
- An automated tool will prepare a Terraform Plan
- The approval of the plan is implemented as pull request approval, the merge will trigger actual release

04. Continuous compliance

With the ease and speed of introducing changes in resource configuration of cloud resources, comes a great risk of introducing issues as well as breaking compliance rules or company standards.

The goal of infrastructure testing and continuous compliance is to ensure automated verification of infrastructure rules. For example:

- Limit allowed resource types and locations,
- Verify machine types and sizes,
- Verify resource configuration (e.g. parameters, naming, tags, tiers, encryption configuration),
- Check software versions and extensions installed on VMs,
- Check audit configuration applied to resources,
- Verify IAM roles and assignments (e.g. min/max count of administrators),
- Verify the configuration of Kubernetes deployments like allowed images, ports, limits, naming conventions.

Terraform can introduce changes very quickly and have a huge impact on the infrastructure. This is why automated testing and policy verification is just as important as in any other programming platform. It can mean:

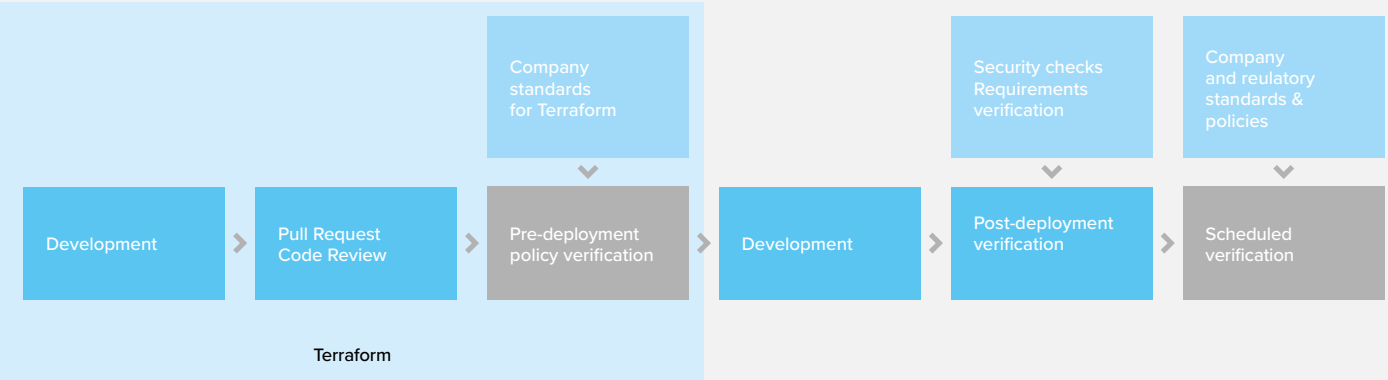
- **Requirements verification**
  Automated verification of some non-functional requirements and assumptions for the project that need to be verified continuously before the sign-off. This is similar to unit/integration testing in software development. Usually, the team that creates Terraform scripts provides the tests as well.

- **Compliance as Code**
  Automated verification of company, organization or regulatory compliance policies using a set of rules. Rules can be related to resource type (e.g. forbidden services), resource location, machine types, OS type and version, replication options, pricing tier/SLA, tagging or naming conventions etc. required across a whole organization. A separate team might provide company-wide policies and compliance standards.

- **Security as Code**
  Automated verification of security policies of introduced infrastructure. Rules can be related to RBAC control, network, firewalls, cloud access control, key vaults, keys, secrets and certificates, encryption etc. required across the entire organization. Security rules may be built with the IT Security team as well as with third-party tooling.

In general, there are two layers of Infrastructure as Code verification:

- Pre-deployment - Verification of the Terraform code or Terraform plan, more akin to static code analysis,
- Post-deployment - Verification of the resources created in the cloud environment after Terraform configuration is applied.



04.1 Pre-deployment verification (build-time)

Terraform validate is a built-in tool. It will check the correctness of syntax, variables etc. A good idea is to also run at least a Terraform plan as a validation step to check if it doesn't fail with an error. Keep in mind that running the plan against "empty" infrastructure may have different results than a plan against the previous version of infrastructure.

There are certain tools that allow verifying Terraform code or plan before it gets applied with more rules than just syntax and correctness

- Terraform Sentinel can be used with Terraform Cloud/Terraform Enterprise. This tool allows creating a set of company-wide policies and applying them to all Terraform projects across multiple teams to ensure, that each project adheres to the rules. This tool will verify the actual plan before the deployment to live infrastructure and look for not allowed resource types, configuration options etc. Think about it as static code analysis for Terraform, like Sonarqube.
- An open source Terraform Compliance using Python BDD framework
- A simple tool tflint will also check if configuration parameters are correct in a given cloud (for example inexistent VM instance type). Currently available for AWS only.
- Forseti Terraform Validator can run Forseti rules verification against Terraform plan file. This one is for Google Cloud only.
- Another example of checking Terraform files with Python (HCL can also be just parsed and verified with a programming language)

04.2 Post-deployment verification (runtime)

Verification of Terraform code before it is applied brings just partial value. This will not verify items applied with custom scripts (when Terraform does not support some options) or will not find changes introduced manually or due to an error. Therefore, it is also valuable to verify the running infrastructure.

NOTE: To address continuous compliance run verification on production-like environments and in the production always after deployment but not only then (e.g. daily). Always use a system account with read-only permissions.

Testing infrastructure
Each cloud provider exposes the whole infrastructure as plain REST/JSON API (or gRPC) as well as SDKs for common programming languages.

- AWS API Reference and AWS SDKs
- Azure API Reference and SDKs and resource explorer
- Google Cloud API Reference and API SDKs

It is very easy to use a language of choice and a favourite testing framework to easily create tests with the use of language SDK or even pure REST API and tools like REST Assured or JSON Assert.

Use a testing framework that will provide a nice and readable test report that can be a document (BDD rather than pure JUnit).
The tests can be executed:

- in a live environment (including production) to apply all requirements checks as well as security or compliance policies
- In a deploy → test → undeploy flow to verify the correctness of the whole Terraform configuration

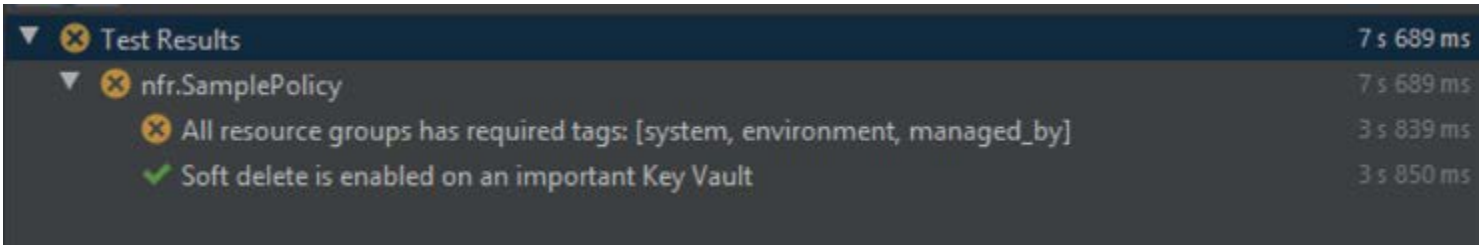
Here is an example using Kotlin and Azure SDK:

Here is an example using Kotlin and Azure SDK:

```
class SamplePolicy: FunSpec({

    val requiredTags = listOf("system", "environment", "managed_by")

    /**
     *This is using Azure SDK
     */
    test("All resource groups has required tags: $requiredTags") {

        val azure = Azure.authenticate(tokenCredentials)
            .withSubscription("SUBSCRIPTION-ID-GOES-HERE")

        azure.resourceGroups().list()
            .forAll { rg ->
                rg.tags().keys.shouldContainAll(requiredTags)
            }
    }

    /**
     * This is using pure API and RestAssured
     */
     test("Soft delete is enabled on an important Key Vault") {
        val path = "https://management.azure.com/subscriptions/SUBSCRIPTION-ID-GOES-HERE" +
            "/resourceGroups/resource-group-x" +
            "/providers/Microsoft.KeyVault/vaults/imporrant-key-vault?api-version=2018-02-14"
    RestAssured.given()
        .header("Authorization", "TOKEN-GOES-HERE")
        .get(path)
        .then()
        .log().body(true)
        .statusCode(200)
        .body("properties.enableSoftDelete", Matchers.equalTo(true))
    }

})
```

Using regular programming skills, it is very easy to build a shared, parameterized set of tests with some effort.
Bash scripting with CLI might not be a best choice because the tests will become complex. Using a scripting language (Python, PowerShell) should be good. Strongly-typed languages (like Kotlin) helps a lot when using cloud provider SDK due to IDE support.

NOTE: In this approach, a native API or SDK is used which is usually the "source of truth". This is important when new cloud features are added to 3rd-party tools (like Terraform or InSpec) with a delay and potential bugs. Relying on 3rd party tools for testing may cause problems. Sometimes even cloud CLI (bash or PowerShell) is delayed. API is always implemented first.



04.3 Built-in cloud policy tools

Each cloud provider has a native tool to address company-wide governance policies. These are:

- AWS Config
- Azure Policy + Azure Security Center
- Forseti Config Validator for GCP

Cloud compliance services are sometimes provided with a set of rules mapped to industry standards such as HIPAA, ISO 27001 or CSA Benchmarks. Creating custom rules is not always easy. These tools can scope policy verification over a set of company projects/accounts/subscriptions not always “on-demand” during the Terraform pipeline run. One of the approaches observed in large organizations is that there are separate teams maintaining company-wide compliance rules and infrastructure as code. This means that the infrastructure team needs to adhere to standards and policies but is not always the author of new rules. The continuous policy tools should be used in addition to infrastructure testing.

AWS Config and Control Tower

AWS Config is a service that continuously monitors and records AWS resource configurations and allows verifying overall compliance against the configurations specified in the internal company guidelines. It comes with a set of around 150 pre-built managed rules as well with the SDK for creating and testing custom AWS Config rules.
Since AWS Config rules are in fact AWS Lambda functions defined in NodeJS or Python, there is a large library of rules available in GitHub. A sample fragment of code in Python:

```
if not configuration_item['configuration']['distributionConfig']['logging']['enabled']:
    return build_evaluation_from_config_item(configuration_item,
        'NON_COMPLIANT', annotation='Distribution is not configured to store logs.')
```

AWS Config verification can be woven into the Terraform Deployment pipeline as a post-release check. The results are however not immediate, and some coding will be required to gather an end-to-end compliance report. Therefore, this is to ensure that the implemented change still adheres to company standards rather than to use it as a testing step.

AWS Config allows grouping the rules together with remediation actions into Conformance Packs (also “as a code” using YAML templates) to be easily deployable into multiple accounts and regions. Sample conformance packs:

- Operational Best Practices for Amazon S3
- Operational Best Practices for Amazon DynamoDB
- Operational Best Practices for PCI-DSS
- Operational Best Practices for AWS Identity and Access Management

With the use of AWS Control Tower, it is possible to:

- integrate AWS Config rules into an end-to-end compliance and conformance overview dashboard over a multi-account organization,
- have an Account Factory for creating new AWS Accounts with predefined rules and settings.

In general, AWS Config is a versatile solution to handle company-wide standard compliance as a code and security as a code. It might not be the fastest way to implement individual solution requirements verification, where simple tests may be easier to maintain. Its use in Terraform pipeline is possible as an addition to built-in AWS continuous compliance solution, but not necessary.

+ Pros:

- Flexibility, and extensibility since the rules are actual code in Python or JavaScript,
- SDK for rules development and a wide set of open-source rules in addition to built-in ones,
- Open-source tools for the whole multi-account Compliance Engine available as well as integration with Control Tower.

— Cons:

- The rules code can become complex and constitute a whole programming project,
- The rule cannot prevent creating a non-compliant resource (only detective mode),
- Including asynchronous rules verification into Terraform CD pipeline requires a complex solution.

Azure Policy and Security Center

Azure Policy is a system built with declaratively defined rules applied to all resources in the scope of the assigned policy. Azure Policies can be assigned on Azure Subscription level as well as on Management Group level (a group of Subscriptions, e.g. whole organization, all production subscriptions etc.).
Azure provides over 1000 predefined, parameterized policies. Custom policies are defined in JSON code and each policy consists of 3 parts:

- Parameters - they are defined during the assignment
- Policy Rule - the “if” part of the rule
- Effect - policy can either raise an alert (audit) or prevent creating a resource (deny)

A sample fragment of code of a policy:

```
“policyRule”: {
    “if”: {
    “field”: “[concat(‘tags[‘, parameters(‘tagName’), ‘]’)]”,
        “exists”: “false”
    },
    “then”: {
        “effect”:
        “modify”,
        “details”: {
            “roleDefinitionIds”: [
                “/providers/microsoft.authorization/roleDefinitions/b24988ac-6180-42a0-ab88-20f7382dd24c”
            ],
            “operations”: [
                {
                    “operation”: “add”,
                “field”: “[concat(‘tags[‘, parameters(‘tagName’),
                ‘]’)]”, “value”: “[parameters(‘tagValue’)]”
                }
            ]
        }
    }
}
```

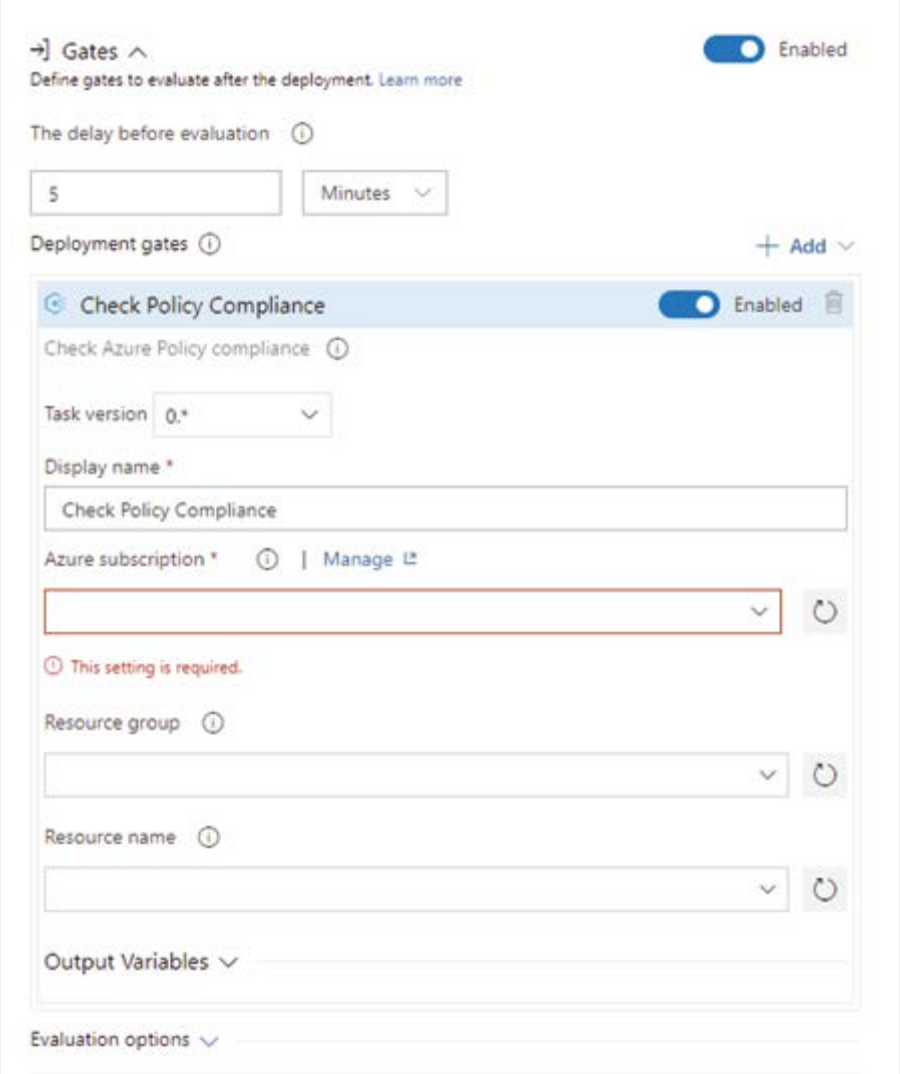
A sample fragment of code of a policy:

```
“policyRule”: {
    “if”: {
    “field”: “[concat(‘tags[‘, parameters(‘tagName’), ‘]’)]”,
        “exists”: “false”
    },
    “then”: {
        “effect”:
        “modify”,
        “details”: {
            “roleDefinitionIds”: [
                “/providers/microsoft.authorization/roleDefinitions/b24988ac-6180-42a0-ab88-20f7382dd24c”
            ],
            “operations”: [
                {
                    “operation”: “add”,
                “field”: “[concat(‘tags[‘, parameters(‘tagName’),
                ‘]’)]”, “value”: “[parameters(‘tagValue’)]”
                }
            ]
        }
    }
}
```

Policies in “deny” mode work like additional validation rules, which means that a resource that is not passing the verification will not be created.

In addition to that, policies can remediate some threats - e.g. automatically install required VM extensions or modify the configuration.

Azure Policy check can be included as a post-deployment correctness check in the Azure DevOps release pipeline.



Besides individual policies, there are several predefined Policy Initiatives in Azure, for example:

- Audit ISO 27001:2013 controls and deploy specific VM Extensions to support audit requirements (56 policy checks)
- Audit PCI v3.2.1:2018 controls and deploy specific VM Extensions to support audit requirements (37 policy checks)
- Audit CIS Microsoft Azure Foundations Benchmark 1.1.0 recommendations and deploy specific supporting VM Extensions (83 policy checks)

Policy Initiatives are parameterized groups of policies to be assigned on Subscription or Management Group level and can be custom created for company standards. There is a default Security Center initiative, containing over 90 configurable policies and is assigned by default to every Azure subscription.

Azure Security Center is a single place to govern results of all policy checks across the organization as well as group results of different threat detection systems (Network, Active Directory, VMs etc.). Since policies are verified periodically, the Security Center can address continuous compliance in Azure providing the alerting mechanism and verification history. To simplify the process of managing corporate-wide compliance, companies can also maintain Azure Blueprints. A blueprint is a combination of policies and initiatives together with default resource groups and IAM access configuration.
Azure Policies are very powerful, but the tool does not provide a developer-friendly interface for creating custom rules, especially, when JSON is used as a language. Even without custom policies, the set of predefined policies is impressive and can address a wide range of compliance requirements. A complete Compliance as Code solution may combine infrastructure tests and Azure policies.

+ Pros:

- A large set predefined policies and initiatives for industry-standard compliance requirements,
- Built-in integration with Azure DevOps and Azure Security Center,
- The policy can work in “deny” mode,
- Policy management with initiatives and blueprints.

— Cons:

- Developing custom policies in JSON is hard,
- Executing policies “on-demand” is not possible.

Forseti and Google Cloud Security Command Center

Google has open-sourced the Forseti Security project to address security rules validation and policy enforcement in Google Cloud. This is a policy-as-code system that consists of multiple modules and works together with:

- Forseti Security service that runs in Google Cloud and takes configuration snapshots for policy monitoring.
- Forseti Config Validator that evaluates GCP resources against Forseti rules.
- Forseti Terraform Validator that verifies terraform plan against Forseti rules.

The Forseti Rule Library is Open Source and uses Rego (by Open Policy Agent framework) files to define policy rule templates.
Here is a sample policy that forbids public IPs for Cloud SQL databases:

Here is a sample policy that forbids public IPs for Cloud SQL databases:

```
deny      [ {
     “msg”: message,
     “details”: metadata,
} ]  {
     asset := input.asset
     asset.asset_type == “sqladmin.googleapis.com/Instance”

     ip_config := lib.get_default(asset.resource.data.settings, “ipConfiguration”, {})
     ipv4 := lib.get_default(ip_config, “ipv4Enabled”, true)
     ipv4 == true
     message := sprintf(“%v is not allowed to have a Public IP.”, [asset.name])
     metadata := {“resource”: asset.name}
}
```

Using policy templates only requires defining constraints in YAML code.
If a policy template for the actual use case is missing, Forseti provides tools and guidelines for authoring and testing custom policies. Setting up Forseti Security requires running dedicated infrastructure in GCP that consists of Cloud SQL, compute and Cloud Storage. Google recommends creating a separate project to serve as a policy monitoring environment. Forseti provides a Terraform configuration for installation. The server picks policy configuration deployed to Google Cloud Storage and then:

- constantly monitors policies,
- can enforce security rules,
- stores Cloud Configuration snapshots in Cloud SQL.

To increase the overall policy and security status visibility, Google offers the Security Command Center dashboard.
Besides Forseti, GCSCC can use other threat detection systems as a source of vulnerabilities alerts like:

- Cloud Data Loss Prevention Discovery,
- Anomaly Detection,
- Event Threat Detection,
- 3rd party cloud security tools from Acalvio, Capsule8, Cavirin, Chef, Check Point CloudGuard Dome9, Cloudflare, CloudQuest, McAfee, Qualys, Redblaze, Redlock, StackRox, Tenable.io, and Twistlock.

Forseti Security offers complete compliance-as-code tooling that can be used as part of Terraform CD pipeline in the pre-deployment step (with Forseti Terraform Validator) as well as post-deployment (with Forseti Config Validator). Continuous compliance support with Cloud Command Center and other plug-in systems adds up to a complete solution. Being a security-oriented solution, this may require significant effort to implement additional custom non-functional requirements checks, thus, it might be a good idea to combine it with infrastructure testing approach.

+ Pros:

- Support for GCP config validation as well as for Terraform validation,
- Declarative rules language with tooling support,
- Integration with Security Command Center.

− Cons:

- Requires installation and maintenance of infrastructure and setup of open-source components,
- A small library of predefined rules (around 70 templates) and lack of industry-standard policy sets (e.g. CIS Benchmarks, HIPAA etc.),
- Lack of preventive-mode, only reactive/detection mode.

04.4
3rd party Policy as Code tools

NOTE: Continuous Compliance and security management for public cloud solutions is an emerging market for enterprise-grade solutions. Several products became available like Check Point CloudGuard Dome9, CloudQuest Guardian, Carvin or Qualys.

Here are some examples of emerging open-source tools addressing the policy-as-code topic.

InSpec
Chef InSpec is an open Compliance as Code tool that can run a set of rules on top of running infrastructure. Policies are defined in a DSL that is quite descriptive and readable. Many resource definitions are available for AWS, Azure and Google Cloud. The drawback of this solution is that resource definitions are added to InSpec with some delay compared to cloud provider API (just like to Terraform or ev en to cloud provider's CLI and SDK) and some of the resources may be very hard to test.

Pulumi CrossGuard
Another solution that allows policy as code is Pulumi CrossGuard. It allows for a more programmatic approach (JavaScript/ TypeScript) over an SDK supporting AWS, Azure, GCP as well as Kubernetes. Here is an example: This is currently in beta and has the same dependency on 3rd party provider for resources.

Azure Secure DevOps Kit
Azure Secure DevOps Kit is an open-source set of policies and rules implemented in a PowerShell-based framework and ready to be executed automatically in a pipeline or using e.g. Azure Automation. It supports Azure only, implemented in Microsoft, but it is not an official Microsoft product. Some of the policies overlap with the built-in Azure Security Center.

Summary

The last few years is the time of "-as-code" approaches to infrastructure, compliance, security, configuration management etc. Using software to address hardware or process problems is the most effective approach, and becomes possible with hardware virtualization and the cloud.

When infrastructure configuration and scale changes are introduced in minutes rather than hours or days, and data or workload can be placed on the wrong continent just "by accident", a totally new approach for tooling and practices is required. This will bring a lot more changes in the nearest future and hopefully, the industry will start to standardize around well-known tools.

Stay in touch

Piotr Gwiazda
Senior Solutions Architect & Cloud Specialist
GFT Poland
E.: piotr.gwiazda@gft.com

Technical Lead and Cloud Solution Architect, with 12 years of experience. Certified on professional levels with Google Cloud and MS Azure. Piotr is leading GFT PL Cloud Practice. Highly experienced with design of multitier solutions for the financial sector including global investment banks. Often acts as an Agile mentor, helping organizations to build their Agile mindset.

GFT Technologies SE Schelmenwasenstr. 34, 70567 Stuttgart, Germany T. +49 711 620 420 www.gft.com info@gft.com