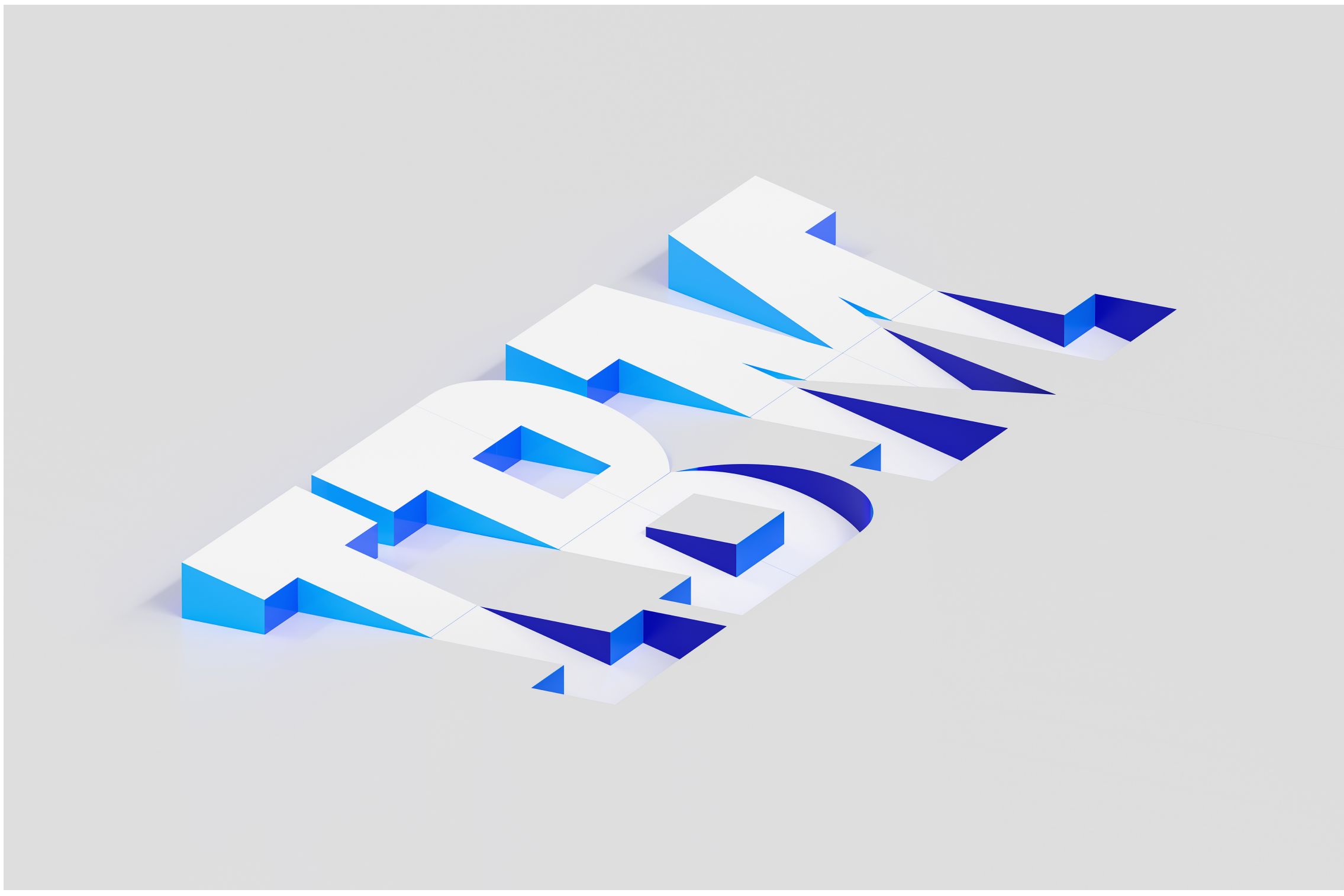
IBM Cloud

Best practices for Terraform on IBM Cloud

White paper



IBM

Edition notices

This PDF was created on 2025-04-02 as a supplement to *Best practices for Terraform on IBM Cloud* in the IBM Cloud docs. It might not be a complete set of information or the latest version. For the latest information, see the IBM Cloud documentation at https://cloud.ibm.com/docs/terraform-on-ibm-cloud.

Best practices for Terraform on IBM Cloud

Efficiently designing and managing Infrastructure as Code (IaC) is a critical success factor as organizations adopt cloud solutions. Terraform, combined with IBM Cloud, offers a powerful way to create scalable and reusable infrastructure. However, success depends on following best practices to help ensure that IaC solutions are modular, secure, and maintainable throughout their lifecycle.

What's in this guide?

This white paper opens with a general overview of Terraform and its role on IBM Cloud, serving as both an introduction for new users and a refresher for experienced professionals. It includes links to further resources and outlines key tools and services available in IBM Cloud, such as Schematics, the IBM Cloud Terraform provider, and prebuilt deployable architectures.

While IBM Cloud provides extensive documentation for configuring Terraform resources, the following guide focuses on best practices for both developing and operating Terraform-based solutions. Designed as a practical playbook, it addresses real-world challenges across the Terraform automation lifecycle, from creating reusable modules to deploying and maintaining secure, scalable infrastructure.

The best practices for designing, developing, and managing IaC solutions on IBM Cloud are organized in three key areas:

1. Architecting Terraform-based solutions: Tips for creating modular, reusable, and scalable IaC architecture.
2. Developing and managing Terraform code: Standards for coding, automated testing, and CI/CD integration.
3. Securing and governing Terraform solutions: Approaches to help ensure security, enforce compliance, and maintain control over IaC deployments.

From simple, reusable Terraform modules to advanced, production-grade deployable architectures in the IBM Cloud catalog, IBM® provides actionable recommendations to help developers write maintainable, module, secure code while providing operators with strategies for securely managing and monitoring infrastructure.

Target audience

This guide provides a comprehensive overview of both implementing Terraform-based automation and, deploying and operating infrastructure created from such automation. While different personas in a typical enterprise might complete these tasks, it is essential to have a single guide that covers both jobs:

- A developer of automation code needs to be familiar with the deployment and operations aspects to implement high-quality code automation.
- A platform engineering team, or administrator, that deploys Terraform-based infrastructure solutions benefit from understanding the Terraform ecosystem, automation code, and best-practices to do their job effectively.

This guide assumes that the target audience has a basic understanding of cloud computing, IBM Cloud, and Terraform:

- Basic familiarity with IBM Cloud services and features. What is the IBM Cloud platform is a good starting point.
- Basic knowledge of Terraform and its concepts. A good starting point is Terraform on IBM Cloud. However, this documentation quickly covers some of the basics as well.
- Some level of familiarity with programming languages.

Primary audience

- Infrastructure engineers: Responsible for designing, coding, and implementing infrastructure automation, and IaC solutions more broadly.
- Platform engineers: Responsible for managing and maintaining cloud infrastructure resources on IBM Cloud.
- Cloud architects: Responsible for designing and implementing cloud architectures on IBM Cloud. With this guide, you can transition from writing architecture documentation to implementing automation that deploys the prescribed architecture.
- DevOps engineers: Focused on ensuring the smooth operation of cloud-based applications and infrastructure on IBM Cloud.

Secondary audience

- Security engineers: Focused on ensuring the security and compliance of cloud-based applications and infrastructure on IBM Cloud.
- Application developers: Interested in understanding how to manage and provision cloud infrastructure resources on IBM Cloud by using Terraform.

Terraform overview

Cloud infrastructure management is a critical aspect of modern IT operations. As organizations increasingly adopt cloud computing, they need to manage their cloud resources efficiently and effectively.

IaC offers significant benefits such as improved consistency, reliability, and scalability in managing infrastructure, enabling faster deployments and reducing human error through automation. Current adoption trends indicate a growing IaC market, projected to expand from $800 million in 2022 to $2.3 billion by 2027. Increasing demand for automation and efficiency in cloud environments drives this market growth.

Terraform is the dominant tool for managing cloud IaC, with widespread adoption by companies globally and support from a large, active community. Its multi-cloud capabilities, ease of use, and declarative configuration language (HCL) enable efficient and scalable infrastructure management. IBM's acquisition of HashCorp for $6.4 billion highlights its commitment to Terraform, further solidifying its position as the industry standard.

What is Terraform?

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Versioning enables users to track changes to infrastructure configuration, revert to earlier versions, and collaborate effectively. Terraform allows users to define and manage their cloud resources by using human-readable configuration files, which make it easy to version, reuse, and share infrastructure configurations.

Terraform uses HashiCorp Configuration Language (HCL), a declarative language that lets you define what infrastructure you need, without worrying about how it's deployed. The key idea behind Terraform's declarative configuration is that you define the final state of your infrastructure, and Terraform handles the details of making it happen.

To illustrate this concept, consider the following HCL code snippet, which creates a IBM Cloud Object Storage bucket within a Object Storage instance by using Terraform. This example showcases the simplicity and clarity of Terraform's declarative syntax, allowing users to define infrastructure configurations without requiring extensive scripting knowledge. Even without prior experience with Terraform, most users can grasp the overall logic of the code.

```
$ # Create a Cloud Object Storage (COS) instance
resource "ibm_resource_instance" "cos_instance" {
  name = "my-storage-service"
  service = "cloud-object-storage"
  plan = "standard"
  location = "global"
}


# Create a COS bucket within the instance
resource "ibm_cos_bucket" "my_storage" {
  bucket_name = "my-family-photos-2024"
  # COS instance that owns the bucket
  resource_instance_id = ibm_resource_instance.cos_instance.id
  region_location = "us-south"
  storage_class = "standard"
}
```

Benefits of Terraform

Terraform provides several benefits for cloud infrastructure management:

Declarative configuration

A declarative approach is a simpler way to create infrastructure. It also reduces the cognitive burden on teams to understand and manage infrastructure changes, making it easier to collaborate, review, and approve changes, while also reducing the effort required to write and maintain complex infrastructure code.

Version control and auditing

Version control systems, such as Git, store the configuration files. These systems make it easier to track and audit infrastructure changes for compliance and regulatory requirements.

Reduced human error

Terraform automates the process of creating and managing infrastructure, reducing the risk of human error and increasing efficiency. By adopting an IaC approach, common software engineering practices, such as peer reviews, can be applied to the configuration code before it is used to provision or manage any environment.

For more information, see What are the benefits of using Terraform on IBM Cloud? .

Terraform and IBM Cloud

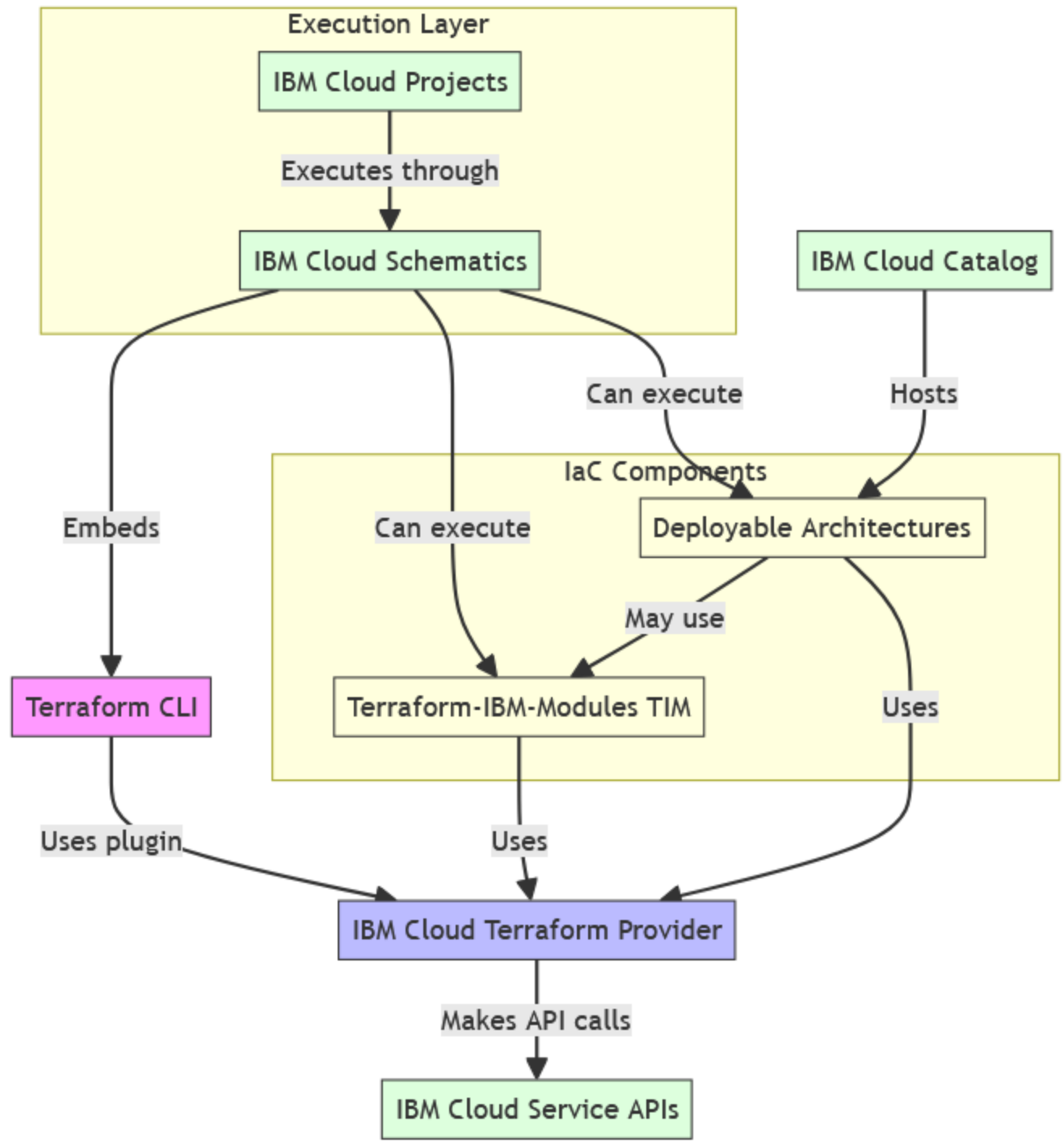
On IBM Cloud, you can use Terraform to create and manage your IBM Cloud resources. IBM Cloud provides a comprehensive set of services and tools for building, deploying, and managing cloud-based applications.

- **IBM Cloud Terraform plug-in**: The IBM Cloud Terraform Provider allows users to create and manage their IBM Cloud resources by using Terraform, providing a consistent and automated way to provision and manage cloud infrastructure. The IBM Cloud Terraform provider can be used to manage a

wide range of IBM Cloud services, including compute, storage, networking, and more.

- **Comprehensive set of Terraform modules**: Terraform-IBM-Modules (TIM) are a collection of curated Terraform modules that provide pre-built infrastructure configurations for common IBM Cloud use cases. The modules are built on the IBM Cloud Terraform provider. Many of these modules are actively maintained by the IBM Cloud development organization. These modules are also used by the IBM Cloud services to build their own infrastructure.
- **Full architecture patterns as code**: Deployable architectures are modular and reusable infrastructure configurations that you can easily publish or consume through the IBM Cloud catalog. They are based on Terraform, but go to the next level by providing further integration into the IBM Cloud ecosystem: documentation, compliance claims, support for public deployable architectures, and integration with IBM Cloud *projects* for facilitating deployment at scale.
- **Terraform as a Service**: IBM Cloud Schematics is an IBM Cloud SaaS offering for managing and running Terraform configurations.
- **IaC at scale**: IBM Cloud projects is an IBM Cloud service that allows users to organize and manage their cloud resources and infrastructure configurations in a single, unified view. IBM Cloud projects use IBM Cloud Schematics for the execution of the Terraform configurations.

Figure 1. shows the interactions between the IBM Cloud services and Terraform components.



Interactions between IBM Cloud services and Terraform components

Review the following sections to learn about the building blocks and services in the Terraform ecosystem.

IBM Cloud Terraform Provider

The IBM Cloud Terraform Provider is a plug-in that enables Terraform to interact with IBM Cloud resources and services. It enables users to manage and provision IBM Cloud resources by using Terraform. You can define and manage IBM Cloud resources, such as virtual machines, storage, networks, and more, by using Terraform configuration files.

You can manage and provision a wide range of IBM Cloud services with the IBM Cloud Terraform provider, like IBM Cloud Kubernetes Service, IBM Cloud Databases, and IBM Cloud Object Storage. Check out the IBM Cloud Provider plug-in reference for a full list of supported resources and services.

Note: As new features and services are added to IBM Cloud, the Terraform provider is regularly updated to include them. This helps ensure that you can incorporate the latest capabilities into your IaC workflows.

The following sections highlight essential Terraform concepts through creating and deploying IaC solutions specifically for IBM Cloud. For more information, see Getting started with Terraform on IBM Cloud

```
$ variable "ibmcloud_api_key" {
  type        = string
  description = "The IBM Cloud API key used to authenticate and authorize Terraform to provision and manage IBM Cloud resources."
  sensitive   = true
}

terraform {
  required_providers {
    ibm = {
      source = "IBM-Cloud/ibm"
    }
  }
}

provider "ibm" {
  ibmcloud_api_key = var.ibmcloud_api_key
}

# Create a Cloud Object Storage (COS) instance
resource "ibm_resource_instance" "cos_instance" {
  name     = "my-storage-service"
  service  = "cloud-object-storage"
  plan     = "standard"
  location = "global"
}

# Create a COS bucket within the instance
resource "ibm_cos_bucket" "my_storage" {
  bucket_name          = "my-family-photos-2024"
  resource_instance_id = ibm_resource_instance.cos_instance.id
  region_location      = "us-south"
  storage_class        = "standard"
}
```

The example uses the IBM Cloud Terraform provider to create a Object Storage (COS) instance and a bucket within that instance on the IBM Cloud platform.

1. **Variable declaration**: Declare a variable ibmcloud_api_key, which is a named value that the user can pass as an input to Terraform when they run `terraform plan` or `terraform apply` to create the infrastructure.
2. **Terraform configuration and required provider** : Specify the provider configuration by using the Terraform block. This block tells Terraform to use the IBM Cloud provider from the IBM-Cloud/ibm source in the Hashicorp provider registry. This information is used by the `terraform init` command to know from where to download the IBM Cloud Terraform provider binary.
3. **IBM Cloud provider configuration**: Configure the IBM Cloud Terraform provider with the API key. An API key is associated with a specific IBM Cloud user or Service ID. In either case, the API key is used to authorize the provider to create, read, update, and delete resources with the identity of the API key owner. An API key is scoped to a specific IBM Cloud account, and all actions that the provider runs are within the context of that account. More details on API Key at Understanding API Keys
4. **COS instance creation:** Define an ibm_resource_instance resource, which is a multi-purpose Terraform resource in the IBM Cloud provider that can be used to create any service instance on IBM Cloud. In this case, use it to create a COS instance by specifying the service parameter as "cloud-object-storage”. Also specifying the instance's name, plan, and location.
5. **COS bucket creation**: Define an ibm_cos_bucket resource, which represents the bucket that you want to create within the COS instance. Specify the bucket's name, the ID of the COS instance that owns the bucket, the region location, and the storage class.

This example demonstrates how to use the IBM Cloud Terraform provider to create a COS instance and bucket on the IBM Cloud platform.

Example: Provisioning the infrastructure with Terraform

Now that the Terraform configuration is in place, provision the declared infrastructure on the IBM Cloud platform.

In this example, you get a general sense of how Terraform uses the configuration in the previous example to provision the infrastructure. For more information about provisioning resources, see Managing IBM Cloud resources with Terraform.

Step 1: Terraform Init

The `terraform init` command is used to initialize a Terraform working directory. This command is used to download and install the required providers,

including the IBM Cloud provider, and prepare the working directory for use. When you run `terraform init`, Terraform does the following actions:

- Downloads the IBM Cloud provider (ibm-cloud/ibm) binary from the Hashicorp provider registry
- Installs the provider in the Terraform working directory

Example output:

```
$ $ terraform init
Initializing the backend...
Initializing provider plugins...
- Finding latest version of ibm-cloud/ibm...
- Installing ibm-cloud/ibm v1.71.3...
- Installed ibm-cloud/ibm v1.71.3 (self-signed, key ID AAD3B791C49CC253)
```

Step 2: Terraform Plan

The `terraform plan` command is used to generate an execution plan for the Terraform configuration. The execution plan is the exact set of actions (creation, delete and update infrastructure) that Terraform completes when you apply the configuration with `terraform apply`. The plan is typically used to view what actions Terraform takes to create the declared infrastructure. It is an opportunity to review and validate the set of actions that Terraform intends to take before they are actually run by `terraform apply`.

Example output:

```
$ $ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following
symbols:

  + create

Terraform will perform the following actions:

# ibm_cos_bucket.my_storage will be created
  + resource "ibm_cos_bucket" "my_storage" {
      + bucket_name          = "my-family-photos-2024"
      + crn                  = (known after apply)
      + endpoint_type        = "public"
      + force_delete         = true
      + id                   = (known after apply)
      + region_location      = "us-south"
      + resource_instance_id = (known after apply)
      + s3_endpoint_direct   = (known after apply)
      + s3_endpoint_private  = (known after apply)
      + s3_endpoint_public   = (known after apply)
      + storage_class        = "standard"
    }

# ibm_resource_instance.cos_instance will be created
  + resource "ibm_resource_instance" "cos_instance" {
      + account_id               = (known after apply)
      + allow_cleanup            = (known after apply)
      + created_at               = (known after apply)
      + created_by               = (known after apply)
      + crn                      = (known after apply)
      + dashboard_url            = (known after apply)
      + deleted_at               = (known after apply)
      + deleted_by               = (known after apply)
      + extensions               = (known after apply)
      + guid                     = (known after apply)
      + id                       = (known after apply)
      + last_operation           = (known after apply)
      + location                 = "global"
      + locked                   = (known after apply)
      + name                     = "my-storage-service"
      + onetime_credentials      = (known after apply)
      + plan                     = "standard"
      + plan_history             = (known after apply)
      + resource_aliases_url     = (known after apply)
      + resource_bindings_url    = (known after apply)
      + resource_controller_url  = (known after apply)
```

```
      + resource_crn             = (known after apply)
      + resource_group_crn       = (known after apply)
      + resource_group_id        = (known after apply)
      + resource_group_name      = (known after apply)
      + resource_id              = (known after apply)
      + resource_keys_url        = (known after apply)
      + resource_name            = (known after apply)
      + resource_plan_id         = (known after apply)
      + resource_status          = (known after apply)
      + restored_at              = (known after apply)
      + restored_by              = (known after apply)
      + scheduled_reclaim_at     = (known after apply)
      + scheduled_reclaim_by     = (known after apply)
      + service                  = "cloud-object-storage"
      + service_endpoints        = (known after apply)
      + state                    = (known after apply)
      + status                   = (known after apply)
      + sub_type                 = (known after apply)
      + tags                     = (known after apply)
      + target_crn               = (known after apply)
      + type                     = (known after apply)
      + update_at                = (known after apply)
      + update_by                = (known after apply)
    }

Plan: 2 to add, 0 to change, 0 to destroy.
```

Step 3: Terraform Apply

The `terraform apply` command applies the execution plan that `terraform plan` generates. The command runs the creation, update, or delete actions in the plan against the IBM Cloud environment. This command creates the declared infrastructure on the IBM Cloud platform.

When you run `terraform apply`, Terraform does the following actions:

- Uses the execution plan that is generated by `terraform plan` to create the COS instance and bucket
- Creates the COS instance and bucket with the specified configuration (name, plan, location, ...)
- Updates the `terraform.tfstate` file to reflect the state of the created infrastructure

Example output:

```
$ $ terraform apply --auto-approve

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following
symbols:

+ create

Terraform will perform the following actions:

( Plan details omitted for brevity - identical to the plan generated by terraform plan )

Plan: 2 to add, 0 to change, 0 to destroy.

ibm_resource_instance.cos_instance: Creating...

ibm_resource_instance.cos_instance: Still creating... [10s elapsed]

ibm_resource_instance.cos_instance: Creation complete after 19s [id=crn:v1:bluemix:public:cloud-object-
storage:global:a/a965a7fbb5f7459496adf5930b670b1a:ac2e40b7-b5b1-4cd4-84f4-57be8a8a1487::]

ibm_cos_bucket.my_storage: Creating...

ibm_cos_bucket.my_storage: Creation complete after 6s [id=crn:v1:bluemix:public:cloud-object-
storage:global:a/a965a7fbb5f7459496adf5930b670b1a:ac2e40b7-b5b1-4cd4-84f4-57be8a8a1487:bucket:my-family-photos-2025:meta:rl:us-
south:public]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

IBM Cloud Schematics

IBM Cloud Schematics is a managed cloud service that runs Terraform configurations and automates infrastructure deployment. It is a secure environment where teams and users can import and run their existing Terraform code to deploy and control resources on IBM Cloud. IBM Cloud Schematics manages the Terraform runtime environment, state management, and access control.

IBM Cloud recommends that you use Schematics over the Terraform CLI for several reasons:

Managed runtime environment

Schematics provides a fully managed Terraform execution environment, which eliminates the need to maintain your own Terraform infrastructure. This way, no machine provisioning, CLI installation, or backend installation is necessary and you can simply provide your Terraform code and run it. For advanced scenarios, you can deploy a private agent on your own infrastructure, like a Kubernetes Service cluster, to maintain control over the execution environment.

Simplified state management

Schematics simplifies state management by securely storing the Terraform-generated state files in encrypted IBM Cloud Object Storage buckets. Schematics also helps ensure that only one execution can run at a time in a workspace, preventing accidental overrides or corruption of the state file.

Team collaboration

Schematics provides built-in collaboration features through its SaaS model. Teams can share workspaces, manage access through fine-grained permissions, and track all activities through comprehensive audit logs. This way, it's easier to work together on infrastructure deployments while you maintain security and visibility.

IBM Cloud integration

Schematics integrates seamlessly with IBM Cloud services, including logging, monitoring, key management service, and identity and access management. IBM Cloud integration provides a foundation for enterprise-grade Terraform automation with proper security controls and operational visibility.

You can use Schematics at no cost. You pay for only the resources that you provision through your Terraform configuration. To create your first Schematics workspace, see Getting started with IBM Cloud Schematics.

Terraform IBM® Modules (TIM)

The Terraform IBM® Module (TIM) community is an open-source initiative that brings together developers, engineers, and operators to create and share Terraform modules for IBM Cloud resources. The IBM Cloud development team maintains and governs the TIM repositories, which includes testing and validating modules to align with IBM Cloud's best practices.

Each module has comprehensive documentation and example code, demonstrating how to use the module in a broader IaC solution. This documentation and example code provide a clear understanding of how to integrate the module into your existing IaC solution, and how to customize it to meet your specific needs.

Take advantage of IBM®'s community expertise with the vast library of pre-built, reusable modules that simplify deploying and managing IBM Cloud resources. These modules act as building blocks for your infrastructure, so you can focus on architecture and applications instead of low-level configurations.

These modules typically encapsulate a set of related resources and provide a simplified interface for configuring and deploying them. For example, the IBM® Secure Landing Zone VPC module creates a comprehensive VPC infrastructure. It includes the VPC itself, public gateways, subnets, network ACLs, VPN gateways, and VPN gateway connections. This module also supports advanced options, like the creation of a hub and spoke DNS-sharing model, custom resolver, DNS resolution binding, a service-to-service authorization policy to name a few, and more.

The real power of Terraform IBM® modules lies in their ability to be used as building blocks in broader IaC solutions, such as deployable architectures. For instance, the IBM®Secure Landing Zone VPC module is one of the several components that constitute the IBM Cloud Landing Zone deployable architecture.

Tip: When you implement an IaC solution, start by exploring the TIM reusable module library to see whether a pre-built module meets your needs. Use these modules as the foundation, and rely on the lower-level IBM Cloud Terraform provider only for specific resources or configurations that are not covered by the module. This approach saves time, reduces errors, and helps ensure that you're using tested, reliable infrastructure code.

Curated and opinionated development tools and templates

Beyond providing a comprehensive library of Terraform modules, the open source community behind TIM develops a suite of opinionated development tools and templates. These tools and templates are designed to streamline the creation of high-quality Terraform modules and deployable architectures.

While these tools are primarily used to support the development and maintenance of the TIM modules, they can also serve as a valuable resource for the

broader Terraform community. As a developer that creates your own Terraform modules or deployable architectures, you can benefit from borrowing from these tools and templates to accelerate and facilitate your own development workflows.

Repository template and development setup tools : The TIM community offers a range of resources to simplify Terraform module development:

- **Local development setup**: Documentation and scripts that install all necessary command line tools on your local workstation to implement Terraform modules and deployable architecture.
- **Deployable architecture repository template**: The GitHub repository template is pre-configured with the best practices for Terraform module and deployable architecture development. This template provides:
  - A pre-defined directory structure for new Terraform module.
  - Configuration files for CI pipeline and quality tools.

Pre-commit hooks: Pre-commit hooks are scripts that run automatically before code is committed to a version control system, like Git. Enforcing coding standards, validating the configuration, and detecting potential errors occurs directly on the developer's machine, before changes are pushed to shared repositories. This approach aligns with the shift-left philosophy, which reduces downstream costs by addressing problems at the earliest stage. The TIM community uses an opinionated Git pre-commit hooks configuration that enforces these best practices, helping to ensure consistency and quality across all modules. This configuration can also serve as a reference or starting point for configuring custom pre-commit hooks for your own Terraform modules or deployable architectures:

- **General checks** for code formatting, licensing, and security vulnerabilities are implemented. These checks ensure that code is properly formatted, licensed, and secure, and that no known security vulnerabilities are introduced into the codebase.
- **Terraform-specific checks** validate Terraform configurations and ensure that they are secure and follow best practices. Tools such as Terraform `fmt` and Terraform `validate` are used to enforce these checks.
- **Code quality checks** for multiple programming languages are implemented. These checks ensure that code is well-structured, readable, and maintainable, and that best practices for coding are followed. Tools such as Flake8 and Golangci-lint are used to enforce coding standards and catch errors in code.
- **Automated tasks** for maintaining the consistency and organization of the codebase. These tasks help ensure that the codebase is well-organized and easy to navigate, and that no inconsistencies or errors are introduced into documentation or configuration files. Automated documentation tools such as terraform-docs are used to maintain the accuracy and consistency of documentation, and Git hooks are used to automate these tasks.

Figure 2. is an example of pre-commit run on a local development machine for the terraform-ibm-resource-group module.

```
                    :~/git/terraform-ibm-resource-group$ pre-commit run --all-files
Check git submodule up to date.....................................Passed
License Checker....................................................Passed
check yaml.........................................................Passed
check json.........................................................Passed
fix end of files...................................................Passed
trim trailing whitespace...........................................Passed
check for merge conflicts..........................................Passed
detect private key.................................................Passed
mixed line ending..................................................Passed
Lint Dockerfiles.....................................(no files to check)Skipped
Terraform fmt......................................................Passed
Terraform validate.................................................Passed
Terraform validate with tflint.....................................Passed
Terraform validate with trivy......................................Passed
Checkov............................................................Passed
Forbid binaries......................................(no files to check)Skipped
Test shell scripts with shellcheck...................(no files to check)Skipped
go fmt.............................................................Passed
Detect secrets.....................................................Passed
flake8...............................................(no files to check)Skipped
isort................................................(no files to check)Skipped
black................................................(no files to check)Skipped
golangci-lint......................................................Passed
Add examples section to README.....................................Passed
Add terraform docs section to README...............................Passed
Add overview section to README.....................................Passed
Validate catalogValidationValues.json.template file...............Passed
Add module repository to go.mod....................................Passed
helmlint...........................................................Passed
```

Pre-commit validation results for a Terraform project, showing all checks passed or skipped, including Terraform formatting, validation, and code linting.

Testing framework: A comprehensive automated testing framework is also included in TIM at ibmcloud-terratest-wrapper. This framework is built on the Terratest library, is written in Go, and includes the following features:

- Incorporates IBM Cloud features, including dynamic selection of a deployment region based on factors like IBM Cloud account quotas and service limits.

- Covers a range of tests, including idempotency, version upgrades, and compatibility with an earlier version

For more information on testing, see the TIM testing documentation.

Deployable architectures

A *deployable architecture* is a way to package and publish, or consume, an IaC solution through the IBM Cloud catalog, open source community registry, or your own private catalogs. Terraform authors can transform their automation code into discoverable and consumable solutions, which includes supporting content that users need to understand and evaluate them.

IBM uses the deployable architecture framework to author a set of IBM validated solutions in the IBM Cloud catalog. These IBM-authored solutions focus on common and foundational infrastructure patterns, undergo rigorous testing, and include comprehensive documentation, architecture diagrams, and validated compliance controls.

The deployable architecture framework itself is open to all authors, so your team can package and share their infrastructure solutions with the same capabilities and built-in best practices that IBM teams use. For more information, see Creating a deployable architecture.

Users can browse available infrastructure patterns and deploy them in a few clicks. No Terraform knowledge is required to deploy. Discover and publish deployable architectures through different types of catalogs to meet various needs:

- **IBM Cloud catalog**: Contains deployable architectures that are validated and supported by IBM.
- **Community registry**: Hosts open source solutions from the IBM® community. These solutions are't supported by the IBM Cloud Support Center.
- **Private catalogs**: Enable sharing your deployable architectures with specific IBM Cloud accounts. For example, a platform engineering team can create and curate a set of deployable architectures for their organization to control access and maintain infrastructure standards as code. This way, deployments are easy for consuming teams. For more information, see Onboarding a deployable architecture to a private catalog.

Terraform code as a catalog solution

Deployable architectures provide a framework that presents infrastructure solutions in a way that's easy to consume by experts and nonexperts alike. Through content provided by the Terraform authors, users can:

- Find and deploy infrastructure patterns through a centralized catalog interface.
- Understand what they're deploying through architecture diagrams and deployment guides.
- View estimated infrastructure cost and deployment duration.
- View access permissions required for deployment.
- Verify compliance controls that the infrastructure creates from the automation comply with regulation by default.
- Deploy complex infrastructure with just a few clicks, without needing to understand the underlying Terraform code.

Figure 3. is an example of a deployable architecture in the IBM Cloud catalog. It depicts the Red Hat OpenShift Container Platform solution and shows how you can easily discover and consume complex infrastructure automation on IBM Cloud:

- Architecture diagrams and clear overview
- Estimated costs and deployment time
- Multiple deployment variations (QuickStart, Standard)
- Complete infrastructure details, which include permissions, security and compliance details, and how to get help.

Catalog /

Red Hat OpenShift Container Platform on VPC landing zone

Creates Red Hat OpenShift workload clusters on a secure VPC network

Related links
Docs
Get help
Readme file
Release notes

View details

Overview

The Red Hat OpenShift Container Platform on VPC landing zone provides the tools to deploy a Red Hat OpenShift Container Platform cluster in a single Virtual Private Cloud (VPC) network. The VPC is a multi-zoned,
See more

Architecture

Product version

v6.6.0

Version last updated: 12/12/2024

Variation

QuickStart

Starting at **$655.38*/mo**

Est. deployment time: 1 h 10 min

See more

Standard

Starting at **$4182.75*/mo**

Est. deployment time: 55 min

See more

*Starting cost is an estimate based on available data as of December 11th, 2024. It does not include all resources, usage, licenses, fees, discounts, or taxes.

Architecture overview | Permissions | Security & compliance | Help

Red Hat OpenShift Container Platform on VPC landing zone - Standard variation

The Standard variation of the Red Hat OpenShift Container Platform on VPC landing zone is based on the IBM Cloud for Financial Services reference architecture. The architecture creates secure and compliant Red Hat OpenShift Container Platform workload clusters on a Virtual Private Cloud (VPC) network.

Summary

Red Hat OpenShift Container Platform on VPC landing zone

11 resources

Red Hat OpenShift Container Platform on VPC landing zone 11 resources	*$4,182.75/mo
Red Hat OpenShift on IBM Cloud Cluster (2)	$4,117.90/mo
Object Storage Bucket	*
Object Storage Bucket	*
VPN for VPC	$64.84/mo
Cloud Object Storage	provided
Key Protect	*
is.flow-log-collector	*
Flow Logs for VPC	*
Virtual Private Cloud (2)	*$0.00/mo
Flow Logs for VPC	*

Total estimated cost ***$4,182.75/mo**

**Estimate does not include usage-based costs and is based on resource prices as of December 11th, 2024.*

Add to project

Review deployment options

Details of the Red Hat OpenShift Container Platform on VPC landing zone deployable architecture, showcasing pricing, architecture options, and deployment details in IBM Cloud

Using IBM Cloud projects for IaC deployments

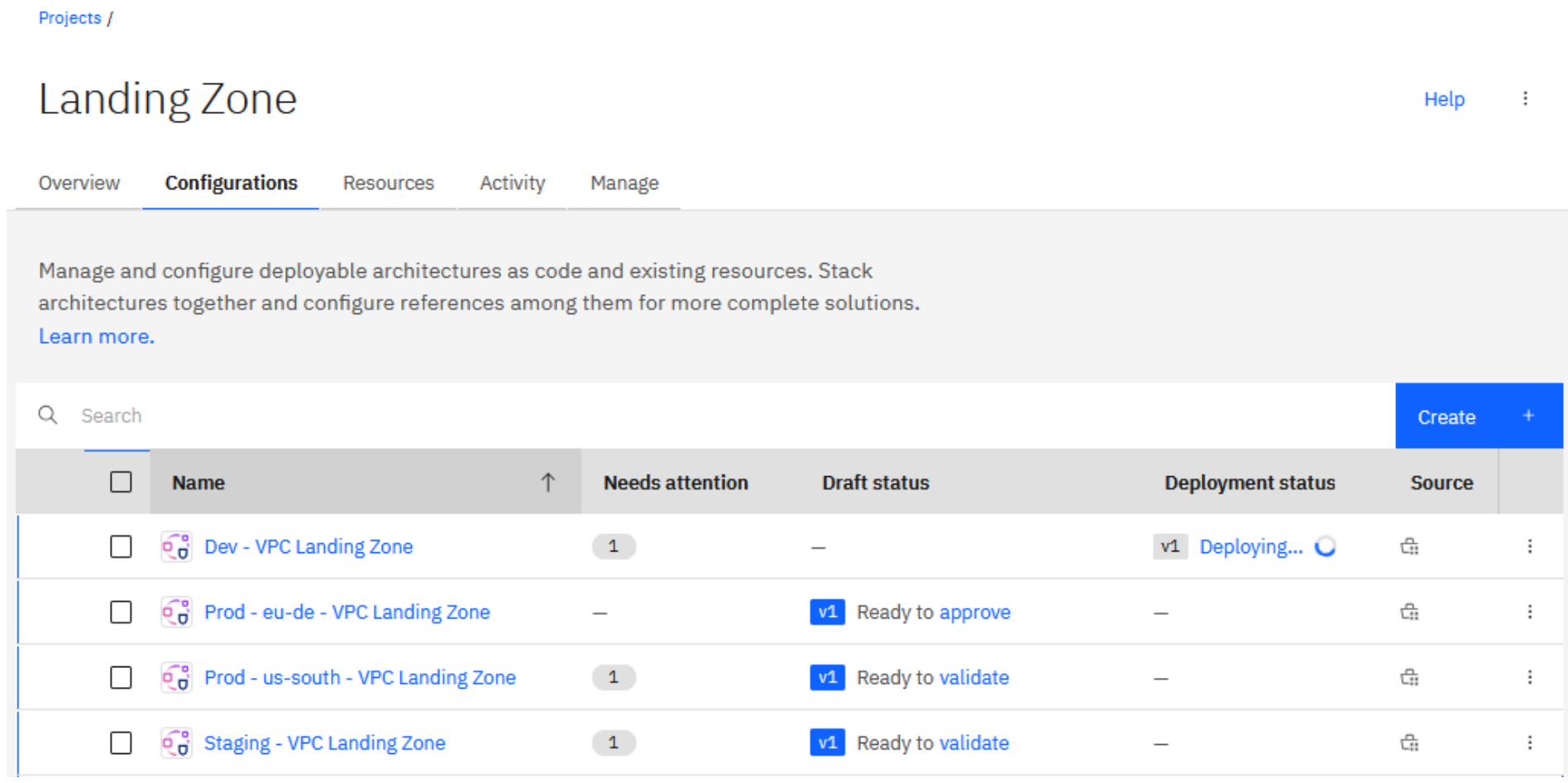
IBM Cloud projects specifically enable users to deploy, update, and operate deployable architectures. When you discover a deployable architecture in the catalog, you use a project to deploy and manage it. Projects support both single deployment and more complex scenarios across multiple environments and teams.

Note: While Schematics is a general-purpose service for Terraform automation, projects use it to run Terraform and integrate features designed for deployable architectures.

Key features of a project include the following:

Configuration management: When you add a deployable architecture to a project from a catalog, you create a project "configuration". This configuration is a deployment profile that specifies which version of the deployable architecture to use, along with input parameters, security settings, and compliance requirements. The ability to create different configurations of the same architecture, each with its own input parameters, enables scenarios like managing

development, test, and production environments or deploying across regions.

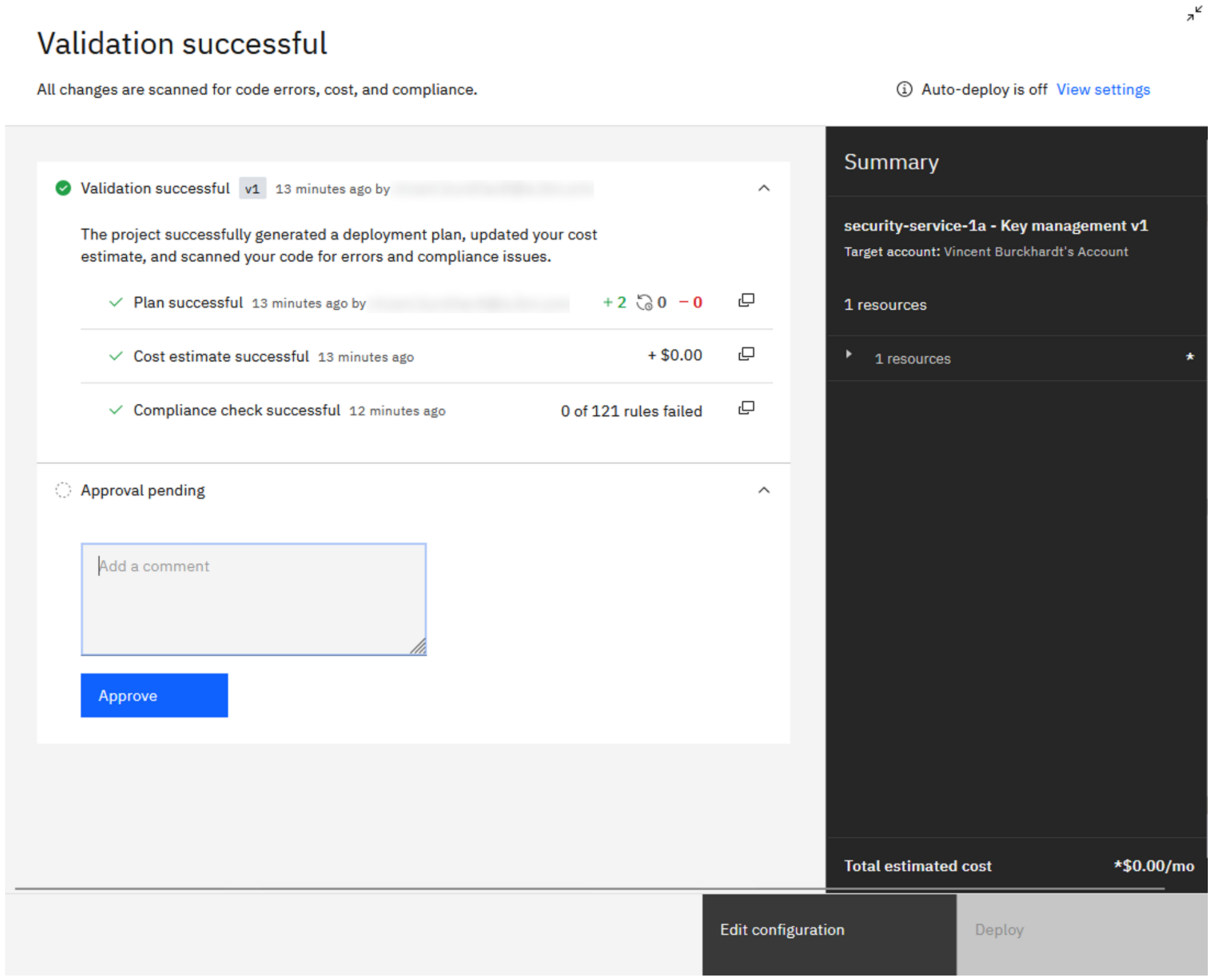


Multiple configurations of the same deployable architecture in an IBM Cloud Project named Landing Zone

Configurations tab of a project named "Landing Zone". In this example, the project has 4 different configurations, all using the same deployable architecture: VPC Landing Zone.

Version control: The project automatically creates a new version of each configuration, providing a traceable record of input change over time. This helps track exactly what parameters were used for each deployment and enables auditing of configuration changes.

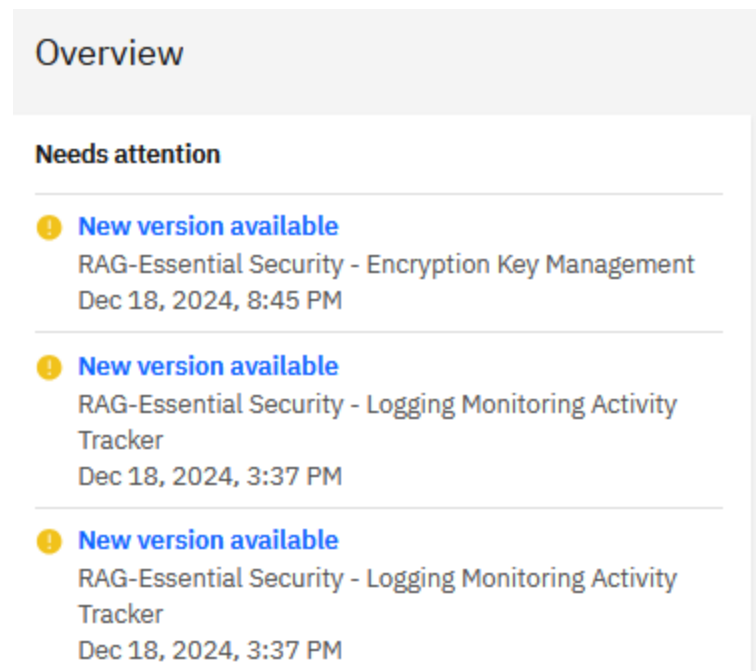
Governance controls: Projects enforce deployment policies through automated validation of compliance requirements and cost estimates. Your project administrators can review and approve changes before deployment.



Validation status and approval screen for a configuration in IBM Cloud Project

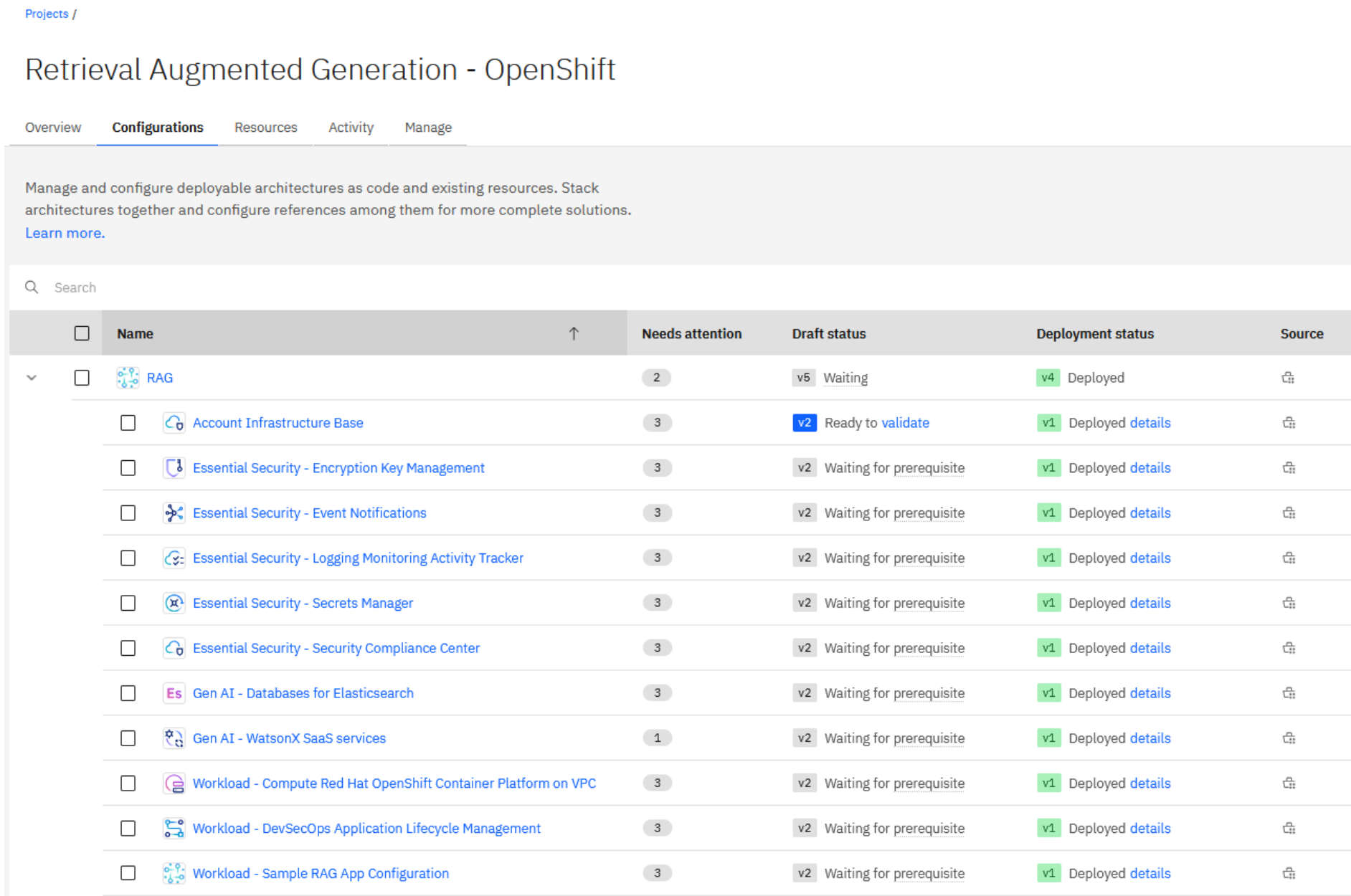
Infrastructure drift detection: Projects automatically detect differences between deployed infrastructure and the deployable architecture code, which helps maintain infrastructure consistency by detecting unexpected manual changes to the deployed infrastructure configuration and the option to correct them.

Update management: Projects provide notifications when deployments are outdated, such as when a new version of a deployable architecture is released. This helps maintain infrastructure with the latest features and security updates.



IBM Cloud project notifications

Architecture stacks: Combine multiple deployable architectures into a single managed unit that can then be shared through the catalog like any other deployable architecture. With stacks, you can create sophisticated systems that meet your business needs and maintain the flexibility and scalability of individual deployable architectures. More details can be found at Breaking down IaC solutions with IBM Cloud Project . For example, you can create a stack that combines a web server deployable architecture with a database deployable architecture, and then manage and share the entire stack as one cohesive system.



A project with the Retrieval Augmented Generation (RAG) stack. The RAG stack groups configurations for 11 different deployable architectures to create a holistic end-to-end environment.

Best practices for architecting Terraform-based solutions

These actionable best practices for architecting Terraform-based solutions focus on scalability, maintainability, and organization. Key factors like implementation approach, code structure, configuration scope, and large-scale management support secure and efficient Terraform development.

Select the right implementation for your solution

When you implement IaC on IBM Cloud, you can structure your solution as a Terraform module, deployable architecture, or stack. The right choice depends on your solution's scope, reusability, and your target users. These implementations are not mutually exclusive, and you can combine them based on your infrastructure complexity and how you want others to use your solution. Technical considerations affect how these approaches can be combined. For example, deployable architectures can use Terraform modules from a Git repository, while stacks can be created only by combining deployable architectures from a catalog. Before you can use a module in a stack, create a deployable architecture wrapper for the module and import it into a private catalog.

Create Terraform modules for single service solutions

When you need to provide a single service implementation that provides specific functions, use Terraform modules. Modules represent the most basic reusable code that provides specific functions, like building blocks that can be combined with other modules to create more complex infrastructure solutions. Modules are typically used by technical professionals with Terraform skills who build custom infrastructure solutions and understand both Terraform syntax and the underlying cloud services.

Review the following module implementations:

- A resource group module that conditionally creates a resource group.
- A Red Hat OpenShift module that deploys a standard Red Hat OpenShift cluster on IBM Cloud.

Create deployable architectures for pre-configured service combinations

When you want to provide a common infrastructure pattern that combines multiple services, implement a deployable architecture. They can be consumed directly through the catalog with just a few clicks, making them ideal for teams who need to deploy infrastructure quickly without Terraform expertise.

An example of an implementation is the IBM Cloud Landing Zone Deployable Architecture, which deploys an opinionated VPC topology that is aligned with the IBM Cloud Framework for Financial Services. It also includes compute resources like Virtual Server Instances (VSIs) or a Red Hat OpenShift cluster, providing the necessary infrastructure to run applications and workloads.

Create stacks for end-to-end solutions

When you need to deliver a full infrastructure solution that addresses complex scenarios, implement stacks. Stacks combine multiple deployable architectures to create comprehensive solutions that can be managed as one unit and even shared through the catalogs themselves. Stacks are ideal for platform engineering teams and DevOps teams who need to build and maintain sophisticated infrastructure solutions that meet specific business needs.

An example is the Retrieval Augmented Generation stack. This stack combines multiple deployable architectures to deploy a full suite of IBM Cloud services. The services are configured to work together, which supports a compliant and secure retrieval augmented generation application pattern on IBM Cloud.

Understand root and reusable modules

When you write Terraform code, whether for a stand-alone module or as part of a deployable architecture, you need to understand the distinction between the two module types in the Terraform ecosystem:

- **Root modules**: Root modules are the Terraform configuration that you can actually run with `terraform apply` commands. In IBM Cloud, these configurations can run in Schematics or packaged as deployable architectures. They contain the provider configuration, like region and authentication, needed for execution.
- **Reusable modules**: Reusable modules are building blocks that can be imported into root modules or other reusable modules. They cannot be directly run with `terraform apply`. They provide specific functions without provider configuration, making them flexible and reusable. Reusable modules are essentially building blocks that can be used to create more complex infrastructure configuration, such as the root module used as part of a deployable architecture.

The distinction between root and reusable modules isn't enforced through specific Terraform syntax or keywords. The main technical difference is that root modules include a provider definition, which makes them stand-alone or executable, while reusable modules don't. By convention, this provider definition is often placed in a file that is named provider.tf, but it can be in any Terraform configuration (.tf file). For example:

```
$ provider "ibm" {
  region           = "us-south"
  ibmcloud_api_key = "your_api_key"
}
```

Define a clear scope

Defining a clear scope for your IaC implementation is crucial for modularity and reusability. Review the following recommendations for each implementation.

Scoping reusable Terraform modules

Make sure that you have a focused scope that targets specific functions in a reusable Terraform module:

- **Abstract a single service**: Design your module around a single IBM Cloud service or a closely related set of resources. For example, the terraform-ibm-cbr module provides a focused abstraction for context-based restrictions (CBR) by combining related resources (`ibm_cbr_rule`, `ibm_cbr_zone` and `ibm_cbr_zone_addresses`) into a cohesive reusable unit.
- **Avoid thin wrappers**: Don't create modules that wrap only a single unique Terraform provider resource. If the module name matches the main Terraform resource type inside it, the module doesn't create any new abstraction and adds unnecessary complexity. In such cases, the caller can use the Terraform provider resource directly.

Scoping deployable architectures

A well-designed deployable architecture groups resources that are commonly deployed and managed together, which typically includes resources that share similar access requirements, lifecycle management, and operational patterns. For complete recommendations and an example, see Deployable Architecture scoping guidelines.

Build for composability

Composability refers to designing modular and reusable infrastructure code, which integrates components to create larger and more complex solutions. Explore the best practices for each approach, as composability is a key principle that applies differently to reusable modules and deployable architectures.

Reusable module composability

Reusable modules are meant to be composed together to create larger IaC solutions, such as root modules or deployable architectures. Review the following best practices to maximize composability for reusable modules.

Maximize the number of inputs and outputs : The more inputs and outputs your module provides, the more flexible and integrable it is in a broad variety of configurations. For more implementation details, see Input variables recommendations.

Document inputs and outputs: Provide clear descriptions for all variables. Consider advanced Terraform structure types, like custom object, to add structure to your parameters. For implementation details and examples, see Type definitions.

Choose clear variable names: Have empathy for your module's consumers and do not assume that they are aware of all acronyms you're using. Using longer, descriptive input and output names is often better than using short and cryptic names.

Follow consistent naming conventions: Either create your own consistent naming convention or, better, borrow the names of similar inputs and outputs from the most popular community modules at Terraform IBM Modules (TIM). While no official naming convention is documented, the conventions used in the TIM modules are considered as reference. For more information, see Consistent naming conventions.

Use validation blocks: Add validation to verify the assumptions made on inputs. See Custom conditions for more details. For example:

```
variable "region" {
    description = "IBM Cloud region where resources will be created. Only us-south and us-east allowed."
    type = string
    validation {
        condition =  can(regex("^(us-south|us-east)$"), var.region)
        error_message = "Region must be one of: us-south, us-east"
    }
}
```

Apply dependency inversion: When your module needs to work with prerequisite resources (like VPCs or resource groups), design it to accept references to these resources rather than creating them. For example, a Red Hat OpenShift cluster module can take VPC and resource group IDs as inputs. This provides flexibility in how the VPC and resource group are created, whether through other Terraform modules, direct calls to the IBM Cloud Terraform provider, or by passing IDs for existing infrastructure, while also improving reuse.

Avoid conditional resource creation: Rather than adding logic to check for existing resources and create them if they're missing, design your module to accept existing resources as input variables. This approach aligns with Terraform's declarative model, where the desired state is explicitly defined. It also helps ensure idempotency, meaning running the module multiple times produces the same result, by keeping resource management predictable. Conditional creation can introduce unpredictability by altering resource creation based on external state, which might cause issues during updates or in larger configurations.

Deployable architecture composability recommendations

A fundamental principle of a deployable architecture is composability, which enables the creation of a broader deployable architecture stack by combining multiple deployable architectures. This modular approach allows for maximum flexibility and reusability of automation resources.

For detailed guidance on creating composable deployable architectures that can be combined into stacks, see Composability. These practices enable architectures to work together effectively and maintain their independence.

Enhance consumability for users

Consumability is about making your infrastructure code easy to understand, configure, and use by its intended audience. While reusable modules and deployable architecture serve different purposes and audiences, users need to easily adopt both.

Consumability for reusable modules

As building blocks meant to be composed into larger solutions, reusable modules need to be well-documented and version compatible. Review the following consumability practices:

Comprehensive documentation: Clear and comprehensive documentation reduces the cognitive burden for users and helps them quickly understand and adopt a module. By providing well-structured README.md files with clear descriptions, usage instructions, and variable details, you minimize the barriers to getting started. Documentation also supports troubleshooting and ensures smooth integration into larger solutions. The Module documentation section provides further details on the sections of a well-structured `README.md`.

Maximize dependency version compatibility: When modules are composed together, their version constraints must be compatible. Use broad version ranges for providers, and specify the minimum version to the lowest supported version that provides the capabilities that are required by your module. Specify the maximum supported version to the next (excluding) major version of the provider or Terraform version. This approach assumes providers maintain backward compatibility across minor versions, which applies to the IBM Cloud Terraform provider and most other providers. The following is an example of version ranges that are declared for the terraform-ibm-base-ocp-vpc module:

```
terraform {
  required_version = ">= 1.3.0, < 2.0.0"

  required_providers {
    # Use "greater than or equal to" range in modules
```

```
    ibm = {
      source  = "ibm-cloud/ibm"
      version = ">= 1.70.0, < 2.0.0"
    }

    null = {
      source  = "hashicorp/null"
      version = ">= 3.2.1, < 4.0.0"
    }

    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = ">= 2.16.1, < 3.0.0"
    }
  }
}
```

Provide working examples: Include Terraform samples that can be applied directly using the Terraform CLI. Each example serves as a root module with a provider definition, making it executable with terraform apply. These examples demonstrate how reusable modules fit into a broader solution:

- Demonstrate integration with other modules, like VPC and security groups
- Show how to handle dependencies between modules
- Provide a working end-to-end implementation that can be directly "terraform applied" for demo or investigation purposes

More details on examples can be found in Include examples.

License: For your module or code, consider a permissive license, such as the Apache v2 license, to encourage reuse and contributions and ensuring compatibility with a wide range of projects.

Consumability for deployable architecture

For guidance on making your deployable architecture easily consumable through the IBM Cloud catalog, including prerequisites, documentation requirements, and configuration best practices, see the Deployable Architecture consumability guidelines.

Simplify large configurations

Using a single, overly complex or monolithic Terraform root configuration can lead to significant operational challenges. Break down configurations into smaller, modular root modules to enhance flexibility during development and streamline operations in the following critical ways.

Improved risk management

> Smaller configurations inherently have a reduced blast radius, meaning that any potential issues or errors from changes are contained within a limited scope. This way, it's easier to manage risks that are associated with infrastructure changes and enhances overall system stability. Also, this separation allows for more precise role-based access control (RBAC), ensuring that team members have permissions relevant to only their specific modules.

Faster execution

> Large configuration can significantly slow down Terraform operations such as plan and apply. By breaking down configurations, you minimize the size of state files, leading to faster execution times for these commands, a short feedback loop, and iterative development.

Enhanced collaboration and role segmentation

> Smaller configurations enable multiple team members to work simultaneously without stepping on each other's toes. Reducing the likelihood of state-locking issues and conflicts allows for smoother collaboration among teams. When changes are isolated within modules, it becomes easier to track modifications and ensure that they do not inadvertently affect other parts of the infrastructure.

Breaking down configurations provides numerous advantages, it can also introduce some complexities:

- **Increased orchestration needs**: For example, instead of running a single `terraform apply` command, teams must now run `terraform plan` and `terraform apply` for each individual configuration, increasing the complexity of the deployment process.
- **Higher risk of configuration drift and inconsistencies** : For instance, updating a security group rule in one environment but forgetting to update it in others can lead to security vulnerabilities.
- **Greater difficulty in tracking changes and dependencies between configurations** : For example, understanding the impact of a change to a shared module on multiple dependent configurations can be challenging.

To mitigate these challenges, use IaC management tools like Terragrunt or IBM Cloud projects. Terragrunt acts as a CLI wrapper for Terraform, enabling users to manage multiple configurations and dependencies with enhanced flexibility and reduced complexity. IBM Cloud projects, on the other hand, introduce the concept of "configurations", which allows users to break down large solutions into smaller, manageable pieces in the form of a stack. Both have unique features and approaches to solving the challenges of managing multiple configurations.

Simplifying IaC solutions with Terragrunt

Terragrunt is a tool that enables the execution of broken-down configurations and provides a wrapper around Terraform. This wrapper offers several benefits, including promoting DRY principles, simplifying configuration management, enhancing dependency management, facilitating environment-specific configurations, and generating consistent code. By using Terragrunt, users can create a hierarchical structure for their Terraform configurations, manage dependencies between modules, and maintain a consistent structure across various environments.

Review the following usage and features for Terragrunt to keep configurations DRY and improve the comprehension of the Terraform configurations:

- Quick Start
- Units Definition
- Stacks Definition
- Use Auto-Init to initialize Terraform
- Keep Terraform code DRY
- Keep Terraform remote state configuration DRY
- Keep Terraform CLI arguments DRY
- Execute Terraform commands on multiple modules
- Define custom actions with before, after, and error hooks
- Control runtime features, errors, and excludes

Simplifying IaC solutions with IBM Cloud projects

An IBM Cloud project allows users to effectively run IaC solutions that are broken down into smaller, manageable configurations by IaC developers. Each configuration is mapped to a dedicated Schematics workspace and its own state configuration file, facilitating streamlined execution and management of modular solutions.

> ⚠ **Important:** While IBM Cloud projects makes it easier to manage and run modular solutions, it does not provide tools to break down an existing monolithic configuration into smaller components. Instead, developers are responsible for creating and structuring each component of the solution before it is managed through IBM Cloud Project.

By structuring solutions into smaller configurations, the size of state files is minimized, leading to faster execution times for Terraform operations. The separation of configurations into dedicated Schematics workspaces also reduces the blast radius of potential issues or errors, making it easier to manage risks associated with infrastructure changes.

If a team implements a large-scale IaC solution, rather than creating a single monolith configuration, design it into smaller configurations for each layer. For example:

1. Creating a VPC: This configuration can focus on setting up the network infrastructure, including the VPC, subnets, and security groups.
2. Creating a cluster on top of that VPC: This configuration can focus on deploying a Red Hat OpenShift cluster on top of the VPC created in the previous step.
3. Configuring a load balancer that points to that cluster ingress: This configuration can focus on setting up a load balancer that points to the ingress of the cluster created in the previous step.

Managing dependencies between configurations: IBM Cloud Project also supports referencing values between configurations. This mechanism allows wiring configuration input and outputs, effectively managing dependencies between configurations. Configurations can be linked together by using the outputs of one configuration as the inputs in another. For example, the cluster configuration can use the VPC ID output from the VPC configuration as an input to deploy the cluster on top of that VPC. Similarly, the load balancer configuration can use the cluster ingress output from the cluster configuration as an input to point the load balancer to the cluster ingress. To achieve this, you can add a reference to an input or an output from another configuration. You can also reference parameters from an environment. When you add a reference, the value is pulled from the input, output, or environment and used as the input value in the architecture that you're configuring.

Simplifying orchestration and configuration management with stacks: To further simplify the management of complex deployments, IBM Cloud Project introduces the concept of Stacks. A Stack is a collection of configurations that are linked together to form a cohesive deployment. Project can automatically plan and apply all configurations in the Stack, considering their dependencies, and eliminating the need for manual intervention.

Consider the previous example of three configurations: creating a VPC, creating a cluster on top of that VPC, and configuring a load balancer that points to that cluster ingress. By grouping these configurations into a Stack, users can deploy all three configurations with a single action. Projects automatically apply the configurations in the correct order, ensuring that the VPC is created before the cluster, and the cluster is created before the load balancer.

With stacks, you can simplify configuration management defining common inputs that can be shared across multiple configurations. For example, you can define a single input for an API key or a prefix that multiple configurations within the stack can use, eliminating the need to repeat the same input for each configuration. The configuration burden and the risk of configuration errors or mismatches is reduced.

Example of a deployable architecture stack

A concrete example of a deployable architecture stack is the Retrieval Augmented Generation Pattern Deployable Architecture , available in the IBM Cloud catalog. This stack consists of 11 individual deployable architectures, each specializing in setting up specific aspects of the end-to-end infrastructure. Aspects include configuring an IBM Cloud account, security and observability services, setting up databases and SaaS services, configuring a VPC, deploying an Red Hat OpenShift cluster, and implementing a CI/CD toolchain. Each deployable architecture within the stack is deployed in its own Schematics workspace, which provides isolation for the state file and other resources. The stack defines a limited set of input parameters at the top level, which each member of the stack uses to avoid redundant input requirements.

To view a project with the Retrieval Augmented Generation (RAG) stack, see Figure 7.

Best practices for coding Terraform-based solutions

The following recommendations apply to both Terraform modules and deployable architecture implementations, unless indicated. These guidelines aim to enhance quality, consistency, and reliability in IaC solutions.

Note: Stacks are not implemented directly by using Terraform, but rather as an assembly of existing deployable architectures. While these best practices might not apply directly, they create a foundation for high-quality stacks when modules and deployable architecture are combined.

Follow consistent coding standards

Maintaining consistent coding standards is essential for creating clear, reliable, and maintainable code. Following standards helps ensure that other developers can easily understand configurations and align with organizational requirements. The following practices fall under the broader umbrella of coding standards:

Format code consistently

Use linting and code formatting to maintain clean, readable code. Standardizing code styles reduces errors and improve collaboration. Use tools like terraform_fmt for formatting Terraform files and black for Python. For a detailed list of tools, see [Deployable architecture code quality practices{: external}, which apply to any type of Terraform-based solution.

Validate static configurations

Configuration validation checks the syntax and consistency of your Terraform configuration before deployment, reducing the risk of deployment errors. Tools such as terraform_validate, tflint for Terraform are used for this purpose. For a detailed list, consult the Deployable architecture Configuration validation practices

Validate security configurations

Make sure that the infrastructure that is created from your Terraform code is secure by identifying vulnerabilities and enforcing compliance with security policies. For a list of tools, see Development-level security scanning.

Test your solutions

Terraform-based solution testing does not involve pure testing in the way application development does. Instead, the testing strategy is to deploy the infrastructure to a real environment, validate its functions, and then destroy it.

Testing and continuous integration are critical in ensuring code quality. Some of the available tools for testing:

- **Terratest**: Terratest is one of the oldest and most widely adopted frameworks for infrastructure testing. It is a framework for writing comprehensive tests in Go. Most of the supported modules at Terraform IBM Modules incorporate Terratest-based testing automation.
- **Terraform Test Command**: Starting from version 1.6, HashiCorp introduced the terraform test command, enabling developers to write and run tests for their Terraform directly.

Key test type includes deployment tests, destruction tests, idempotency tests, and version upgrade tests. For more information about these strategies, see Testing, which applies to any Terraform-based solution.

Enable continuous integration (CI)

To ensure the reliability, consistency, and maintainability of your Terraform code, use a shift-left approach where quality checks and testing are integrated

early in the development cycle. This approach helps detect errors and defects early, reducing the likelihood of downstream problems and improving overall quality.

CI enables automated testing and validation early in the development cycle. Pre-commit hooks and CI pipelines help catch issues early and ensure consistent quality across deployments. For a detailed guide to integration CI with Terraform workflow, see Deployable architecture Continuous Integration best practices.

Versioning and release management

Adopt semantic versioning

Semantic versioning (SemVer) is a widely adopted standard that is used to categorize the impact of changes in software into major, minor, and patch versions. This system provides a clear framework for understanding the scope and impact of changes by using the format MAJOR.MINOR.PATCH. For example, version 1.2.3 indicates a major release (1), with minor updates (2) and patch-level bug fixes (3).

Adopting semantic versioning for your own Terraform module, deployable architecture, and stacks. This structured approach enables your users to understand changes at a glance and confidently plan upgrades. Adhering to semantic versioning ensures that infrastructure changes are predictable and well communicated.

Follow these guidelines when you release a new major, minor, or patch version of your IaC solution:

- **Major version changes** indicate breaking change that can disrupt existing infrastructure or workflow. These updates might include modification to existing resources that require manual intervention, such as migrating Terraform state files, or change that destroy deployed resources from the previous version. Such update must be thoughtfully documented, offering clear migration paths and documentation to guide the users of your module or deployable architecture through the upgrade process.
- **Minor version changes** introduce new features or enhancement that are compatible with an earlier version. These might include adding more and new inputs or outputs, modifying default values in ways that do not affect existing deployments, or incorporating new capabilities. Even though these changes do not break infrastructure, they must be clearly documented to help users adopt new functions.
- **Patch version changes** focus on fixing bugs or addressing nonfunctional aspects of the module. These changes are intended to be safe and nondisruptive for users, allowing them to apply updates with minimal risk.

Automate compatibility with an earlier version testing

To maintain consistency and reliability, the module maintainer must test all version updates to ensure compatibility with an earlier version. Design automated tests to detect scenarios where migrating from a patch or minor version to the next patch or minor version never results in infrastructure resources to be destroyed. Testing framework provide more details and tools that can be used for this purpose.

Document changes

Detailed changelogs and documentation must accompany every release to help users understand and adopt updates effectively:

- Mention new capabilities, and any bug fixes
- For breaking changes, provide clear guidance on how existing user can migrate to a new major version, for example: documentation on the manual steps, migration scripts, and so on.

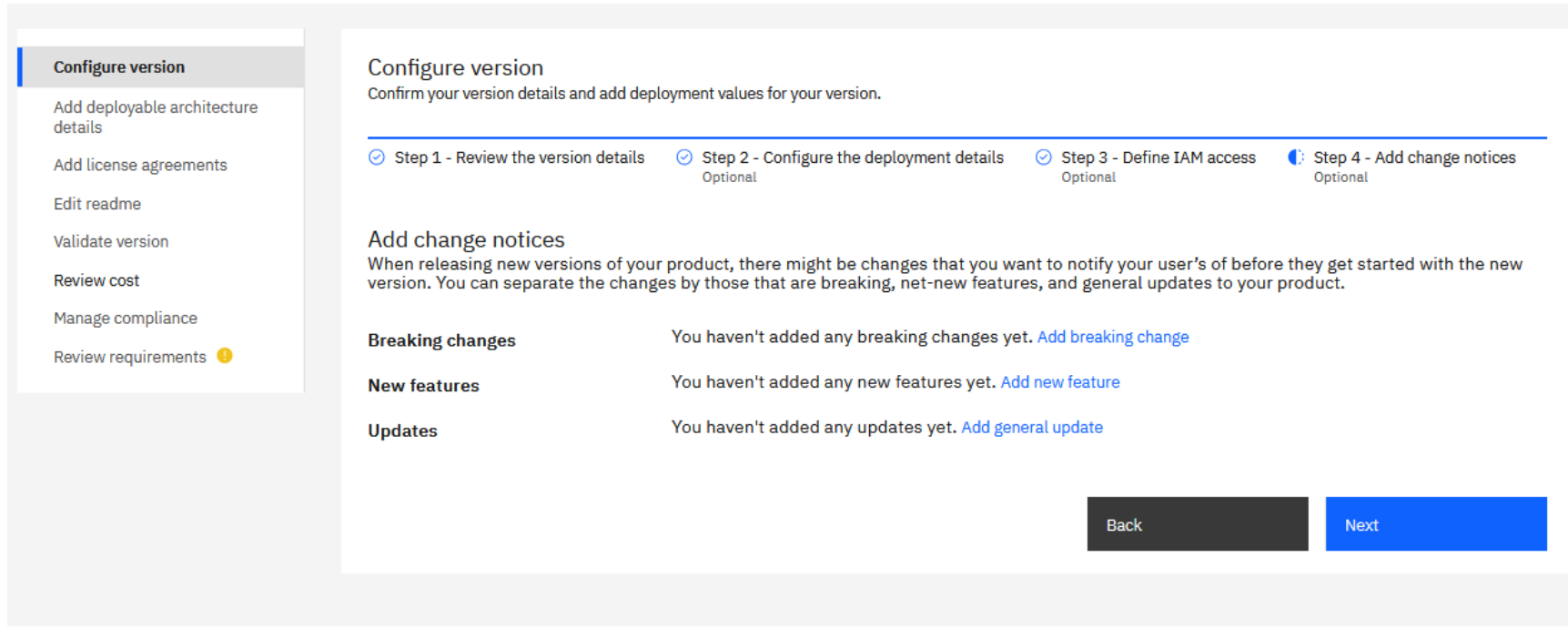
The following are examples of release notes for a deployable architecture and a Terraform module:

- Secure Landing Zone Deployable Architecture release notes
- Terraform module release notes in GitHub (Code Engine module example)

Release processes for module and deployable architectures

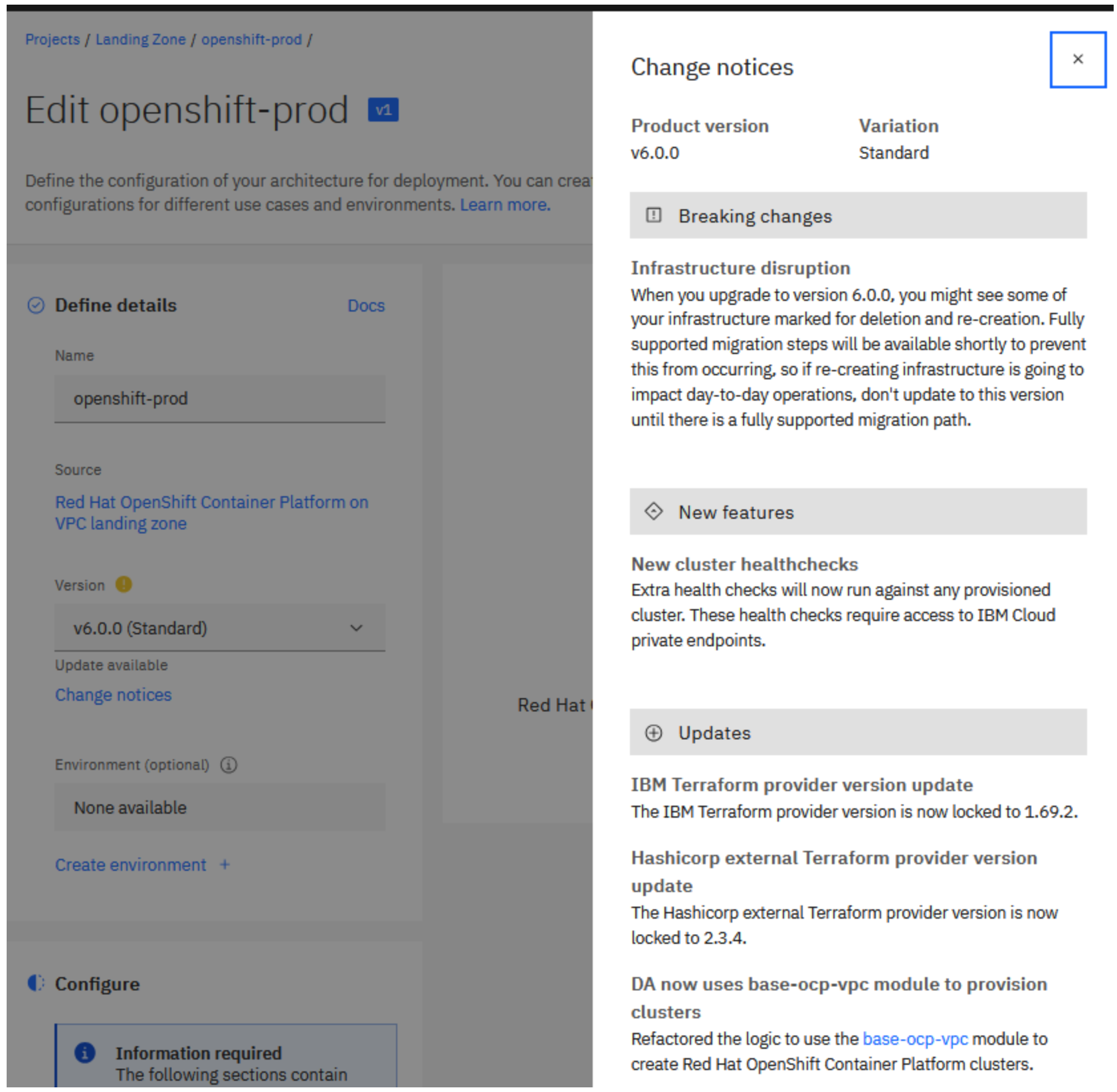
The release process for Terraform modules and deployable architectures (DA) involves distinct steps, though they are often interconnected:

- **Modules**: Releases are typically managed through source control platforms, such as GitHub. Maintainers use features such as the GitHub release section to attach release notes, highlight breaking changes, and document new features. The Terraform modules available in the Terraform IBM Module community use this release process.
- **Deployable architectures**: A deployable architecture release typically begins with an archive file, such as a .tar file, that contains the [Terraform logic and IBM Cloud specific metadata{: external} that is generated as part of a source control release. For example:
  - The deployable architecture source code is packaged into a .tar file during the GitHub release process, including all necessary artifacts and metadata.
  - This .tar file is then onboarded into the IBM Cloud catalog, where more you can provide release details, like notes and breaking changes.



Step-by-step configuration interface for defining details of a version of a deployable architecture.

- Users of the IBM Cloud catalog access the deployable architecture directly, with release information, and artifact derived from the source control release.



Users of the IBM Cloud catalog access the deployable architecture directly, with release information, and artifact derived from the source control release.

- When a new version of a deployable architecture is onboarded, IBM Cloud Project automatically notifies users of the previous versions to inform them of the availability of the update. To view an example of the console notifications, see Figure 6.

This workflow ensures consistency between the source control system and the IBM Cloud catalog, streamlining the transition from development to distribution. By aligning these processes, you can ensure transparency for users and reduce the risk of discrepancies between the source and published versions.

Automate module releases

For Terraform modules, tools like semantic-release can simplify and streamline the release process. This tool automates tasks such as determining the next version number, generating release notes based on commit messages. It is designed for managing releases in GitHub and serves as a starting point for deployable architecture release when the GitHub .tar file is used for catalog onboarding. For more information, refer to the semantic-release documentation.

As an example, Terraform IBM Modules make extensive use of semantic-release as part of the release process of every community module.

Manage input variables

Effectively managing input variables in Terraform is crucial for maintaining clarity, composability, and security in your infrastructure code. The following best practices are broadly applicable to both reusable Terraform modules and deployable architectures (DAs). While the initial recommendations are universal, nuances specific to DAs are addressed in a separate subsection.

The following guidelines ensure that input variables are consistent, reusable, and maintainable across all Terraform-based solutions.

Consistent naming conventions

Use clear and descriptive names for your variables to enhance readability and usability. Consistent naming conventions help other users, or yourself, understand the purpose of each variable quickly.

- **Use full names rather than acronyms** so that users unfamiliar with the products or services can understand what they refer to. For example, use `secrets_manager` instead of `sm` or `event_notification` instead of `en`. However, industry-wide naming conventions, such as `vpc` for Virtual Private Cloud or `cos` for Object Storage, are acceptable and can be used to maintain consistency with established standards.
- **Use underscores for separation**. Instead of `camelCase`, use `snake_case` for better alignment with Terraform conventions.
- **Be consistent in the ordering of similar terms** . For example, `id` or `name` is always at the end of a variable name: `resource_group_id` and not `id_resource_group`.
- **Start boolean variable names with verbs**. Examples: `use_private_endpoint`, `force_delete_storage`, `disable_outbound_traffic_protection`
- **Avoid ambiguity about resource creation** . When it's not obvious, clearly indicate when a variable refers to an existing infrastructure resource, as opposed to a resource that the module or deployable architecture creates. A convention is to prefix the name of the variable with `existing_`. For example, `existing_kms_instance_name` for a variable that takes as input the name of a key management service that exists in the account. The prefix reenforces that the module does not create the KMS instance, as a name like `kms_instance_name` might indicate.
- **Contextualize inputs**: For multi-service modules or DAs, use the name of the service as the prefix of the variable names. An example is the ocp_all_inclusive module that allows to create a Red Hat OpenShift cluster and configure various side aspects of it, such as the log analysis aspects. All inputs related to log analysis are prefixed with `log_analysis`, such as `log_analysis_secret_name`, `log_analysis_instance_region`. However, for modules that handle only one type of resource, the context is clear and prefixing variable name is not recommended to avoid redundancy.

Organize and optimize variables

- **Group variable definitions:** Use a dedicated `variables.tf` file to organize all input variables, improving discoverability and reusability. For more information on module structure, see Module and deployable architecture structure guidelines.
- **Colocate related variables**: Group related inputs logically to help the reader discover related variables quickly. For example, `variables.tf` in the `ocp-all-inclusive` module clearly groups variables that are related to VPC, cluster definition, key management, and log analysis.
- **Order variable for usability**: Place frequently used variables, like `resource_group`, and `region`, at the top of the `variables.tf` file.

Avoid hardcoding values

- **Maximize the number of inputs** : Maximize the number of inputs for generic reusable modules to make them more flexible and adaptable. However, this practice does not apply to deployable architectures, where the goal is to minimize the number of inputs. See Deployable Architecture-Specific Recommendations.
- **Use defaults**: Define sensible defaults for most variables to simplify adoption without requiring detailed user input. Defaults enable users to quickly start using the module with just a few required variables to set.
- **Use defaults that align with common usage patterns** : If multiple common use cases are identified, consider implementing variations of the module defaults tailored to each use case. This approach is widely adopted in the community-maintained modules at terraform-ibm-module. For example, the terraform-ibm-icd-postgresql module defines defaults that allows most users to quickly start with a cost-effective and publicly accessible PostgreSQL database instance. Also, the sub-module located in the fscloud directory defines stricter defaults that are designed for compliance-focused use cases, such as configuring the instance to restrict public internet access by default.

Protect sensitive variables

Variables that store sensitive data, such as passwords or API keys, must be marked as `sentisive = true` in Terraform, which prevents their values from

being displayed in logs or output and reduce the risk of accidental exposure.

```
$ variable "ibmcloud_api_key" {
  type        = string
  description = "The IBM Cloud API Key used to authenticate. All resources are created under the account associated with this API
key."
  sensitive   = true
}
```

See Protect sensitive input variables for further details.

Explicitly define variable types

Explicitly define variable types to improve validation and reduce ambiguity. Use types like list, map, or object for structured inputs. For example:

```
$ variable "resource_tags" {
  type        = list(string)
  description = "A list of tags applied to the resources created by the module."
  default     = []
}
```

In this example, the resource tag variable is defined as a list of string type. This removes any ambiguity on how to pass the tags like `["tag1", "tag2", ...]` (a list of string, as opposed to any other format such as comma-separated strings).

For enumerations, use custom objects instead of maps. Objects provide clear, self-documenting key names and enable better static validation, helping catch errors early.

Example:

```
$ variable "addons" {
  type = object({
    debug-tool                = string
    image-key-synchronizer    = string
    openshift-data-foundation = string
    vpc-file-csi-driver       = string
    static-route              = string
    cluster-autoscaler        = string
    vpc-block-csi-driver      = string
    ibm-storage-operator      = string
  })

  description = "Map of OCP cluster add-on versions to install ..."

  nullable = false

  default = {
    debug-tool                = ""
    image-key-synchronizer    = ""
    openshift-data-foundation = ""
    vpc-file-csi-driver       = ""
    static-route              = ""
    cluster-autoscaler        = ""
    vpc-block-csi-driver      = ""
    ibm-storage-operator      = ""
  }
}
```

The custom object in this example explicitly defined the values that are accepted as keys, which is not the case when you use a simple map:

```
$ variable "addons" {
  type        = map(string)
  description = "Map of OCP cluster add-on versions to install ..."
  nullable    = false
  default     = {}
}
```

Implement validation rules

Implement validation rules within variable declarations to enforce constraints on accepted values. Validation rules can help detect errors early in the deployment process. For example, in IBM Cloud, you might validate that a VPC ID is in the correct format by using a regular expression. Terraform 1.9 improved possible validation rules with the introduction of cross-object referencing. For more information, see the Terraform 1.9 blog post.

The following example checks that the `cluster_config_endpoint_type` variable value is only set to one of the 4 following valid strings: "default", "private", "vpe" or "link".

```
$ variable "cluster_config_endpoint_type" {
  description = "Specify which type of endpoint to use for for cluster config access: 'default', 'private', 'vpe', 'link'."
  type        = string
  validation {
    error_message = "Invalid Endpoint Type! Valid values are 'default', 'private', 'vpe', or 'link'"
    condition     = contains(["default", "private", "vpe", "link"], var.cluster_config_endpoint_type)
  }
}
```

Document variables

Provide clear and accurate descriptions for each variable:

- **Go beyond naming**: Avoid just rewording the name of the variable. Use the description to highlight further relevant details.
- **Link external reference**: Include links to external documentation for advanced details.
- **Mention nontrivial or nonobvious behaviors** as necessary.

The following is an example description that exhibits the previous three characteristics:

```
$ variable "addons" {
  ...
  description = "Map of OCP cluster add-on versions to install (NOTE: The 'vpc-block-csi-driver' add-on is installed by default
for VPC clusters and 'ibm-storage-operator' is installed by default in OCP 4.15 and later, however you can explicitly specify it
here if you wish to choose a later version than the default one). For full list of all supported add-ons and versions, see
/docs/containers?topic=containers-supported-cluster-addon-versions"
}
```

Manage input variables for deployable architectures

While the preceding recommendations provide universal guidance for input variables, deployable architectures have more unique requirements. Deployable architectures are designed to be reusable and usable for less-technical users to use through projects in the IBM Cloud console. They need a more user friendly input approach than modules built for Terraform developers.

Display only commonly modified arguments

To simplify adoption, display only variables that users are mostly like to modify. Avoid implementing deployable architectures with many input variables that can overwhelm users. For advanced users, consider providing a single JSON input field for further customization. For example, the VPC Landing zone deployable architecture shows the field `override_json_string` to allow full customization, and hides advanced parameters for casual users. For more information, see the VPC Landing zone deployment guide.

Use advanced typing capabilities

Use the advanced input types of deployable architecture. IBM Cloud supports more input variable types beyond the ones that are available by default in standard Terraform. IBM Cloud projects render appropriate input widgets based on the type, simplifying the user input experience. Example of types:

- The type `vpc` allows users to select a VPC by name from a list.
- The type `Platform resource` allows users to select an instance resource from a list for the type of resource that the deployable architecture author specifies
- The type `cluster` requires users to select a Kubernetes Service or Red Hat OpenShift cluster

For more information and types, see Locally editing the catalog manifest values.

Carefully select 3rd party dependencies

In the context of Terraform-based infrastructure, a dependency refers to any external component that your IaC setup relies on for functions. Dependencies include Terraform providers, modules, Helm chart, container images, or other artifacts and tools that are referenced and used within your Terraform code.

While dependencies can greatly simplify development and accelerate delivery, poorly maintained or untrustworthy ones can lead to security vulnerabilities, compatibility issues, and maintenance challenges. By carefully selecting and managing dependencies, you help ensure that your IaC setup remains stable, secure, and maintainable.

Ensure that a dependency is actively maintained

Before you decide on a 3rd party Terraform provider, module, or any dependency, verify that the dependency is actively maintained. Actively maintained dependencies are more likely to receive updates, bug fixes, and security fixes.

Look for the following key indicators:

- **Recent updates**: Review the release history of the provider or module. Has it been updated recently? Are there regular releases with new features, bug fixes, and security fixes?
- **Source history**: Examine the source code history on GitHub or other version control platforms. How many contributors are actively working on the project? Are there many open issues, or are they being addressed promptly?
- **Community engagement**: Look for signs of community engagement, such as active discussions, issues, and pull requests, which indicates a well-maintained project.

The **Insights** section in GitHub for a project is a great place to get these types of indicators. Insights provide a dashboard view of a project's activity, including:

- **Contributors**: See who is actively contributing to the project and how many contributors are involved.
- **Commits**: View the commit history and see how frequently the project is being updated.
- **Issues**: See how many open issues are present and how quickly they are being addressed.
- **Pull requests**: View the number of open pull requests and how quickly they are being merged.

By reviewing these metrics, you can get a sense of the project's activity level and whether it is being actively maintained.

Ensure that you trust the maintainers

Trust in the maintainers is essential to ensure the dependency's reliability and security. Evaluate maintainers based on:

- **Availability in** HashiCorp registry: Official or partner providers and modules provide a higher level of trustworthiness. However, some trusted dependencies might not appear in the registry as "partner" due to associated cost. For example, the IBM provider and module are not listed as partners in the HashiCorp registry, but are still trustworthy. Look for modules with a strong reputation, an active community, and transparent development process.
- **Reputation**: Research the maintainer's reputation online. Have they contributed to other reputable open source projects? Do they have a history of responding to issues and pull requests?
- **Transparency**: Evaluate the maintainer's transparency. Are they open about their development process, release schedules, and security practices?

Check the license for IaC automation

The dependency's license must support its use within your automation workflow. Specifically, ensure that the license allows:

- **No cost use and modification**: The license must allow no cost use and modification of the provider or module for IaC automation purposes.
- **Redistribution**: The license must allow for redistribution of the provider or module, either as part of your IaC setup or as a stand-alone component.

Look for permissive licenses that are widely used in the IaC community, such as **Apache, MIT or BSD license**, that facilitate modification and redistribution without restrictive limitations.

Monitor dependencies

Regularly monitor dependencies to ensure that they remain updated and secure. Automating this process helps to scale dependency management as your IaC grows.

Tools like Renovatebot automatically monitor dependencies for new versions and creates pull requests for updates. For more details, refer to Automate dependency version management.

Use a consistent structure

Adopting a consistent and standard structure for Terraform modules and deployable architectures makes it easier for developers to understand, use, and extend the code. It reduces onboarding time for new contributors, simplifies contribution by ensuring predictability where parts of the module implementation are located, and fosters a better collaboration by aligning everyone on the same conventions.

Choose a module structures based on your use case:

1. Minimal structure: A basic setup that provides a solid foundation that is suited for simple use cases or prototyping.
2. Production-grade structure: A structure that incorporates more features and best practices to ensure high-quality and reliable modules.

Include the following Terraform configuration files at the top level of the repository:

- **main.tf**: The main.tf file is the central location for the core logic of your Terraform module, describing the infrastructure to be created. The main.tf file is typically where other developers start to get familiar with your module codebase.
    - For small modules, this file might contain the full logic, including all resource and data block definitions and necessary variable and output declarations.
    - For larger modules, it's often beneficial to break out the logic into multiple files and nested modules and use the main.tf file as the core integration point. In this case, the main.tf file contains the logic that brings all the pieces together, defining how the various components interact and depend on each other.
- **variables.tf**: Groups definition of all the input variables for the module. Having all input variables in a single file helps in terms of discoverability and maintainability.
- **outputs.tf**: Groups definition of all outputs of the module.
- **version.tf**: This file contains the required_providers block, which specifies the providers, including their version required by the Terraform module. It also contains the required_version specification, defining the Terraform version that is compatible with this module. For more information about specifying version range and providers, see Reusable module consumability.

For larger modules and deployable architectures, consider breaking down the code base into nested modules to organize your code, placing them in the `modules/` directory and calling them from the parent module as needed. This approach helps keep your code organized, makes it easier to manage larger modules and deployable architectures, and enables code reuse within the same root module.

Minimal structure

The minimal structure is the typical starting point for a functional module or deployable architecture, with the Terraform code and the minimum documentation in a readme file. This structure aligns with the Hashicorp structure.

```
$ ├── examples/
| ├── example-name/
| | ├── main.tf
| | ├── provider.tf
| | ├── .../
| ├── .../
├── modules/
| ├── nestedModuleA/
| | ├── README.md
| | ├── variables.tf
| | ├── main.tf
| | ├── outputs.tf
| ├── nestedModuleB/
| ├── .../
├── reference-architectures/
| ├── architecture-example.md
| ├── example-architecture-diagram.svg
├── README.md
├── main.tf
├── outputs.tf
├── variables.tf
├── version.tf
```

Production-grade structure

The production-grade structure builds upon the minimal structure, incorporating extra features and best practices to achieve high-quality and reliable modules.

```
$ ├── examples/
| ├── example-name/
| | ├── main.tf
| | ├── provider.tf
| | ├── .../
| ├── .../
├── modules/
| ├── nestedModuleA/
| | ├── README.md
| | ├── variables.tf
```

```
│ │ ├── main.tf
│ │ ├── outputs.tf
│ ├── nestedModuleB/
│ ├── .../
├── reference-architectures/
│ ├── architecture-example.md
│ ├── example-architecture-diagram.svg
├── README.md
├── main.tf
├── outputs.tf
├── variables.tf
├── version.tf
├── tests/
│ ├── test_main.tf
│ ├── test_variables.tf
│ ├── .../
├── docs/
│ ├── user-guide.md
│ ├── api-documentation.md
│ ├── architecture-diagram.svg
├── ci/
│ ├── pipeline.yml
│ ├── .../
├── scripts/
│ ├── build.sh
│ ├── deploy.sh
│ ├── .../
├── common-dev-assets/
│ ├── workflows/
│ │ ├── workflow_file1.yml
│ │ ├── .../
│ ├── commitlint.config.js
│ ├── renovate.json
│ ├── .../
├── .github/
│ ├── workflows/
│ │ ├── workflow_file1.yml
│ │ ├── .../
│ ├── CODE_OF_CONDUCT.md
│ ├── CONTRIBUTING.md
│ ├── .../
├── .pre-commit-config.yaml
├── .releaserc
├── .secrets.baseline
├── LICENSE
├── Makefile
├── module-metadata.json
```

IBM uses the production-grade structure for reusable modules and deployable architectures in Terraform IBM Modules (TIM), which is refined by real-world usage. By following this structure, you can make sure that your own modules and deployable architectures are built to a high standard, making them more maintainable, scalable, and supportable. This structure is used in production environments and is a good starting point for any support modules or deployable architectures.

The production-grade structure uses other pre-configured development tools available in the TIM common-dev-assets to implement an opinionated development flow. This flow includes checks for code quality and configuration validation to make sure that the code is consistent, formatted, and secure.

The development flow includes two main types of checks: code quality and configuration validation. Code quality checks make sure that code is formatted and consistent using tools such as `terraform_fmt` and `go-fmt`, helping to detect errors and warnings early in the development process. Configuration validation checks make sure that configuration is valid and secure using tools such as `terraform_validate` and `checkov`, which helps to prevent downstream problems and helps ensure that the code is reliable and maintainable. More details can be found at Pre-commit hooks.

Get started with a template for this production-grade structure by going to the terraform-ibm-module-template repository. The tools are referenced through a Git submodule in the directory structure. Some top-level files, such as the makefile and pre-commit configuration, are soft links to files from the common-dev-assets Git submodule.

Include module examples

Include at least one example of how to use your module. Examples are root Terraform modules that use the module and demonstrate its usage. Place examples in the `examples/` directory.

- **Simple and basic**: Include at least one simple and basic example that helps users understand the module's usage and make it easier for them to get started.
- **Stand-alone**: Make sure that examples are stand-alone. Stand-alone examples can be run without manual setup of underlying infrastructure or other dependencies, other than an IBM Cloud account and an API key. This makes it easy for users to get started with the example and reduces the barrier to entry. For example, the terraform-ibm-base-ocp-vpc module has examples that can be run without requiring manual setup of underlying infrastructure.
- **Comprehensive**: Add as many examples as necessary to cover the most common scenarios for your module. It's helpful for users understand how to use your module in different contexts.
- **Educational**: Create examples to help developers understand how to use your module in their Terraform code so that they can integrate your module into their projects and reduce the learning curve.

The examples directory for the terraform-ibm-base-ocp-vpc module repository illustrates the previous practices through concrete examples that demonstrate various usage of the reusable `base-ocp-vpc` module.

Include a reference architecture for deployable architectures

If you're publishing your module as a deployable architecture on IBM Cloud, include a reference architecture that describes the design. Author the reference architecture in Markdown and include an architecture diagram. These files can be placed in the `reference-architectures/` directory.

Include module `README.md`

Document your module in the `README.md` file at the same level as the `main.tf` file for each of the modules in a repository, including the submodules, and the example modules. For consistency across modules, follow the same structure in all `README.md` files.

Create a `README.md` that includes:

1. Clear module description and purpose
2. Detailed usage section
3. Important assumptions or prerequisites
4. Module status (incubating, ready for early adopter, stable)
5. Complete input/output variable documentation

Use the following terraform README.md template as a starting point, and automated tools like terraform-docs to keep the documentation up to date.

An example of README.md following these practices can be found in the module for setting up a Red Hat OpenShift VPC cluster.

Adopting IaC for existing infrastructure

When you work with existing infrastructure, also known as brownfield deployments, adopting IaC helps you manage and version previously unmanaged or manually configured resources. Using Terraform, you define infrastructure in code and link resources to a state file. When complete, all future updates are made through code. This process has two main steps:

Step 1: Implementing Terraform code

Create Terraform code that matches the deployed state of your infrastructure. Terraform v1.5+ introduced an experimental feature that is called configuration generation, which simplifies this process. Before v1.5, this step is done manually. With configuration generation, you can declare the IDs of your infrastructure resources and their corresponding Terraform resources, and then generates the code for you. For more information, refer to the HashiCorp Terraform documentation: Generating Configuration.

Step 2: Populating the Terraform state file

To populate the Terraform state file with data from your existing infrastructure, use the `terraform import` command or import blocks. For detailed instructions and examples, refer to the official HashiCorp documentation:

- Terraform Import Command
- Terraform Import Blocks

The IBM Cloud Terraform provider documentation provides resource-specific guidance on using the `terraform import` command. Typically, you can find this information at the end of each resource's documentation page. For example, see the documentation for importing an IBM Cloud Object Storage bucket: Importing an IBM COS bucket

Example: Importing an existing Object Storage instance and bucket

This example demonstrates how to generate Terraform code for an existing Object Storage (COS) instance and a COS bucket, starting from existing deployed resources in an IBM Cloud account.

1. Initialize the IBM Cloud Provider.

   To begin, initialize the IBM Cloud provider in the Terraform configuration. Define a variable for the IBM Cloud API key and configure the provider to use it.

```
$ variable "ibmcloud_api_key" {
  type        = string
  description = "The IBM Cloud API key used to authenticate and authorize Terraform to provision and manage IBM Cloud
resources."
  sensitive   = true
}

terraform {
  required_providers {
    ibm = {
      source = "IBM-Cloud/ibm"
    }
  }
}

provider "ibm" {
  ibmcloud_api_key = var.ibmcloud_api_key
}
```

2. Add imports blocks for the resources.

   Next, add two import blocks - one for the COS instance and one for the COS bucket. The `to` attribute refers to the "address" of the resource in the IaC representation. In this case, use the IBM Cloud Terraform provider resources (`ibm_resource_instance` and `ibm_cos_bucket`) and the resource name.

   The `id` attribute specifies the ID of the existing infrastructure, which might vary depending on the resource type. The IBM Cloud provider documentation for each resource describes the id needed for import. For instance for the COS bucket, the IBM Cloud provider documentation for ibm_cos_bucket indicates that the format is `$CRN:meta:$buckettype:$bucketlocation` .

```
$ import {
  to = ibm_resource_instance.cos_instance
  id = "crn:v1:bluemix:public:cloud-object-storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-
2733993acba5::"
}

import {
  to = ibm_cos_bucket.my_storage
  id = "crn:v1:bluemix:public:cloud-object-storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-
2733993acba5:bucket:1129-my-family-photos-2024:meta:rl:us-south:public"
}
```

3. Generate the code.

   Run `terraform plan -generate-config-out=generated.tf` to generate the code in a new file named `generated.tf` . This provides a good starting point for the Terraform configuration.

```
$ # __generated__ by Terraform
# Please review these resources and move them into your main configuration files.
# __generated__ by Terraform from "crn:v1:bluemix:public:cloud-object-
storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-2733993acba5::"

resource "ibm_resource_instance" "cos_instance" {
location          = "global"
name              = "my-storage-service"
parameters        = null
parameters_json   = null
plan              = "standard"
resource_group_id = "848b91f5cc844415abeded89bb340a4d"
service           = "cloud-object-storage"
service_endpoints = null
tags              = []
}

# __generated__ by Terraform from "crn:v1:bluemix:public:cloud-object-
```

```
storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-2733993acba5:bucket:1129-my-family-photos-
2024:meta:rl:us-south:public"

resource "ibm_cos_bucket" "my_storage" {
allowed_ip            = []
bucket_name           = "1129-my-family-photos-2024"
cross_region_location = null
endpoint_type         = "public"
force_delete          = null
hard_quota            = 0
key_protect           = null
kms_key_crn           = null
object_lock           = null
region_location       = "us-south"
resource_instance_id  = "crn:v1:bluemix:public:cloud-object-storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-
49f1-b7be-2733993acba5::"
satellite_location_id = null
single_site_location  = null
storage_class         = "standard"
}
```

4. Refine the code to make it more concise, modular, and maintainable.

   Remove any unnecessary input. For example, the inputs that are set to their default values or the resource_group_id, which is set to the default resource group in this case. Link back Terraform resources and replace the hardcoded CRN of the `resource_instance_id` with the output of the `ibm_resource_instance` resource.

   After some minor editing, you get the following:

```
$ resource "ibm_resource_instance" "cos_instance" {
location          = "global"
name              = "my-storage-service"
plan              = "standard"
service           = "cloud-object-storage"
}

resource "ibm_cos_bucket" "my_storage" {
bucket_name           = "1129-my-family-photos-2024"
endpoint_type         = "public"
region_location       = "us-south"
resource_instance_id  = ibm_resource_instance.cos_instance.id
storage_class         = "standard"
force_delete = false
}
```

5. Import the resources.

   Given that you have the import blocks in the code, the next step is to run `terraform apply` to import the existing resources into the Terraform state file.

   Terraform imports the existing resources and potentially change the infrastructure if the code does not match the current state of the infrastructure. It's essential to review the Terraform plan carefully before you apply to ensure that it doesn't introduce unintended changes.

```
$ $ terraform apply

ibm_resource_instance.cos_instance: Preparing import... [id=crn:v1:bluemix:public:cloud-object-
storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-2733993acba5::]
ibm_resource_instance.cos_instance: Refreshing state... [id=crn:v1:bluemix:public:cloud-object-
storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-2733993acba5::]
ibm_cos_bucket.my_storage: Preparing import... [id=crn:v1:bluemix:public:cloud-object-
storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-2733993acba5:bucket:1129-my-family-photos-
2024:meta:rl:us-south:public]
ibm_cos_bucket.my_storage: Refreshing state... [id=crn:v1:bluemix:public:cloud-object-
storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-2733993acba5:bucket:1129-my-family-photos-
2024:meta:rl:us-south:public]

Terraform will perform the following actions:

## ibm_cos_bucket.my_storage will be imported
resource "ibm_cos_bucket" "my_storage" {
```

```
....
}

## ibm_resource_instance.cos_instance will be imported
resource "ibm_resource_instance" "cos_instance" {
....
}

Plan: 2 to import, 0 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

ibm_resource_instance.cos_instance: Importing... [id=crn:v1:bluemix:public:cloud-object-
storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-2733993acba5::]
ibm_resource_instance.cos_instance: Import complete [id=crn:v1:bluemix:public:cloud-object-
storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-2733993acba5::]
ibm_cos_bucket.my_storage: Importing... [id=crn:v1:bluemix:public:cloud-object-
storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-2733993acba5:bucket:1129-my-family-photos-
2024:meta:rl:us-south:public]
ibm_cos_bucket.my_storage: Import complete [id=crn:v1:bluemix:public:cloud-object-
storage:global:a/7aa6f7b185f2e3170fac9919aa1769ee:b72c7f3c-f9a3-49f1-b7be-2733993acba5:bucket:1129-my-family-photos-
2024:meta:rl:us-south:public]

Apply complete! Resources: 2 imported, 0 added, 0 changed, 0 destroyed.
```

From this point, the infrastructure can be managed "as code" through Terraform. You can remove the import blocks from the code to clean up.

Note: You might come across a tool called Terraformer. Terraformer doesn't fully support IBM Cloud resources and doesn't have plans to expand this support. Import existing resources and generate corresponding code by using the Terraform import command and code generation feature previously described.

Best practices for securing IaC implementations

IaC is a powerful approach to managing cloud infrastructure, offering automation, consistency, and scalability. However, adopting IaC requires deliverable security measures to protect sensitive data, ensure compliance, and mitigate risks. The following sections present best practices for both developers and users of deployable architectures on IBM Cloud.

Enforce least privilege across operations and development

The principle of least privilege is a core security concept that grants users and services only the necessary privileges to perform their tasks, eliminating unnecessary access. In IBM Cloud and Terraform, this principle ensures that the API key for the IBM Cloud Terraform provider has only the necessary permissions to run the configuration. Limiting privileges reduces the attack surface and minimizes security risks.

Review the following operational best practices for enforcing least privilege.

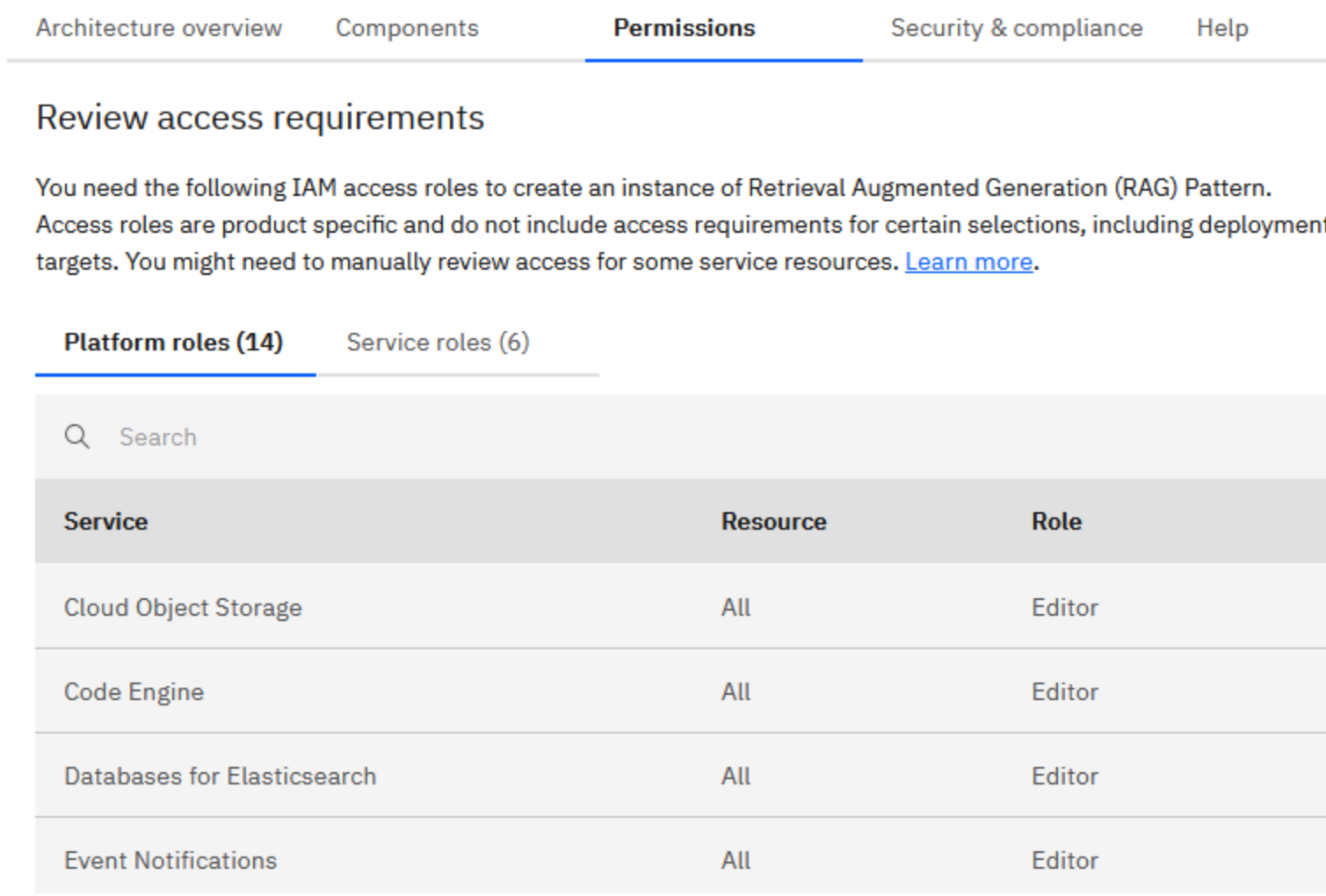
Use access groups to implement the least privilege

Access groups are a powerful feature in IBM Cloud that allow you to manage access to resources and services in a structured way.

Create dedicated access groups specifically for running one or more IaC solutions. Grant these group only the minimum permissions required to provision and manage the infrastructure. For more information, see Setting up access groups.

Refer to the module or deployable architecture documentation for guidance on the exact permissions needed. Don't use existing access groups that are assigned to operators or administrators, as these often include broader permissions than necessary and potentially exposes your environment.

For deployable architecture, the list of necessary permissions is available under the Permissions tab.



Access requirements page listing Cloud Identity and Access Management platform roles needed for various services, to configure a deployment.

Use a service ID API key

Use an API key that is associated with a Service ID in IBM Cloud, rather than a personal API key linked to a specific user. Service ID API keys are designed for use with automated processes and services and can be easily managed and rotated as needed. Using a Service ID API key provides several benefits, including:

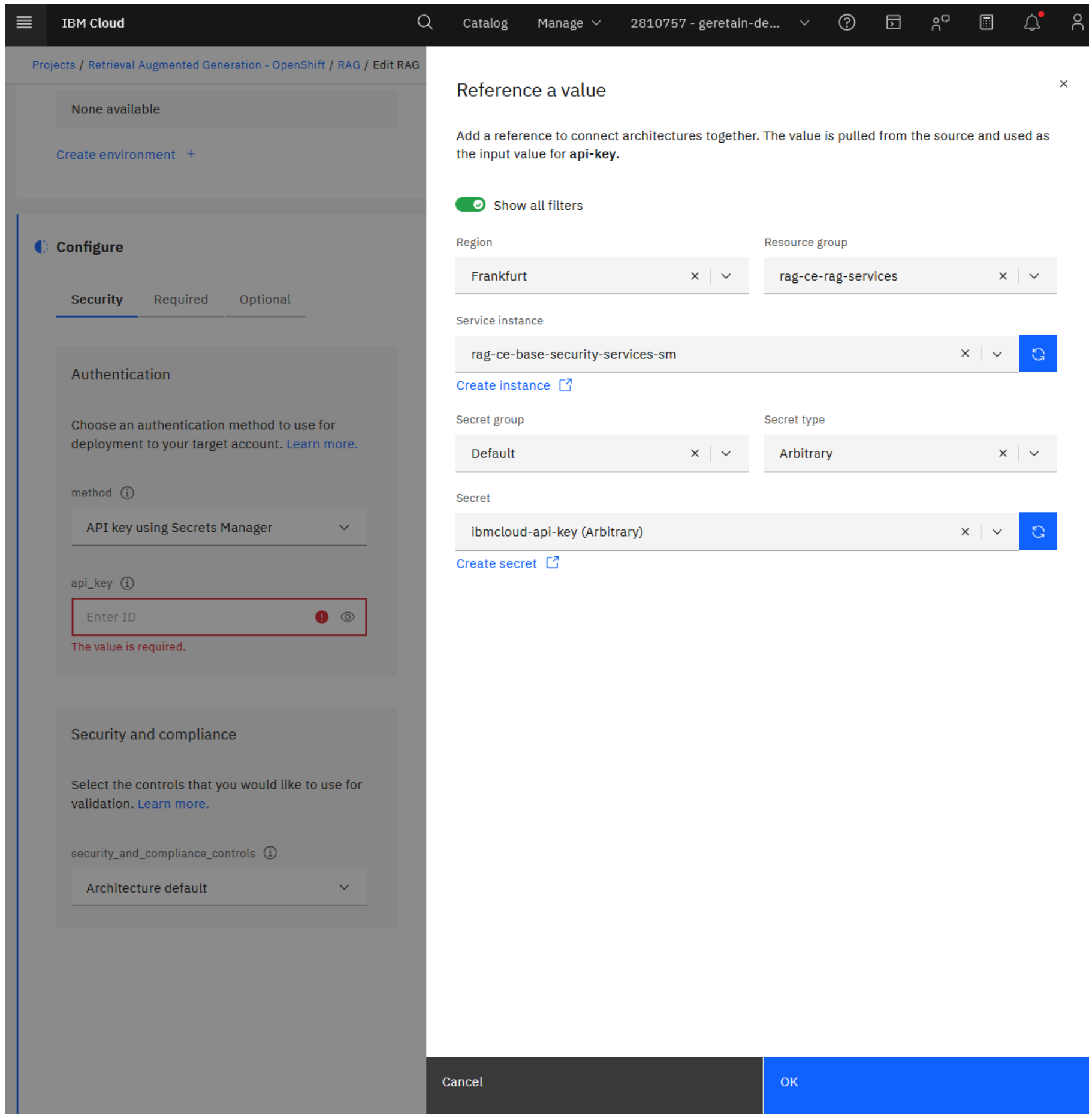
- **Role-based access control (RBAC):** Service IDs can be assigned specific access groups that are tailored to the IaC solution needs, ensuring adherence to the principle of least privilege
- **Credential isolation**: Each IaC solution can have its own Service ID and API Key, preventing shared credentials and reducing the risk of unintended access.
- **Operational continuity**: Service IDs are not tied to individual users, so running the IaC solution remains functional even if a user leaves the organization or their account is modified
- **Simplified key rotation**: Service IDs can have multiple API keys, allowing for seamless key rotation without disrupting application functions.
- **Enhanced auditing and monitoring**: Service IDs provide detailed logging of actions that are performed by the associated API Keys, improving traceability and security monitoring.

Store API Keys in IBM Cloud Secrets Manager

To further enhance security, store the API key in IBM Cloud Secrets Manager. Using Secrets Manager provides an extra layer of protection for the API key and ensures that it is not stored in plain text. Using a secret manager provides several benefits, including:

- **Automated rotation**: Secrets Manager can automatically rotate API keys, which reduce the risk of API key compromise. In addition, integration with IBM Cloud Event Notifications enable the operation team to be notified through various channels when secrets are about to expire.
- **Improved security**: Secrets Manager provide a secure way to store sensitive data, such as API keys. The data is encrypted by default at rest, including with the ability to integrate with services such as IBM Key Protect and Hyper Protect Crypto Services to have fully control on the encryption keys.
- **Centralized management**: Secrets Manager provide a centralized way to manage sensitive data, which makes it easier to audit, track and monitor access. Centralized management is accomplished by using audit events that are integrated into IBM Cloud Activity Tracker

IBM Cloud projects provide the capability for immediate integration with Secrets Manager, which allows the user to select an API Key from Secrets Manager. There is no need for the user to copy-paste an API key to deploy a deployable architecture, which might be a potential security attack vector.



IBM Cloud Project allows selecting API Key stored in IBM Cloud Secrets Manager.

Review the following development best practices for enforcing least privilege.

Document minimal permissions

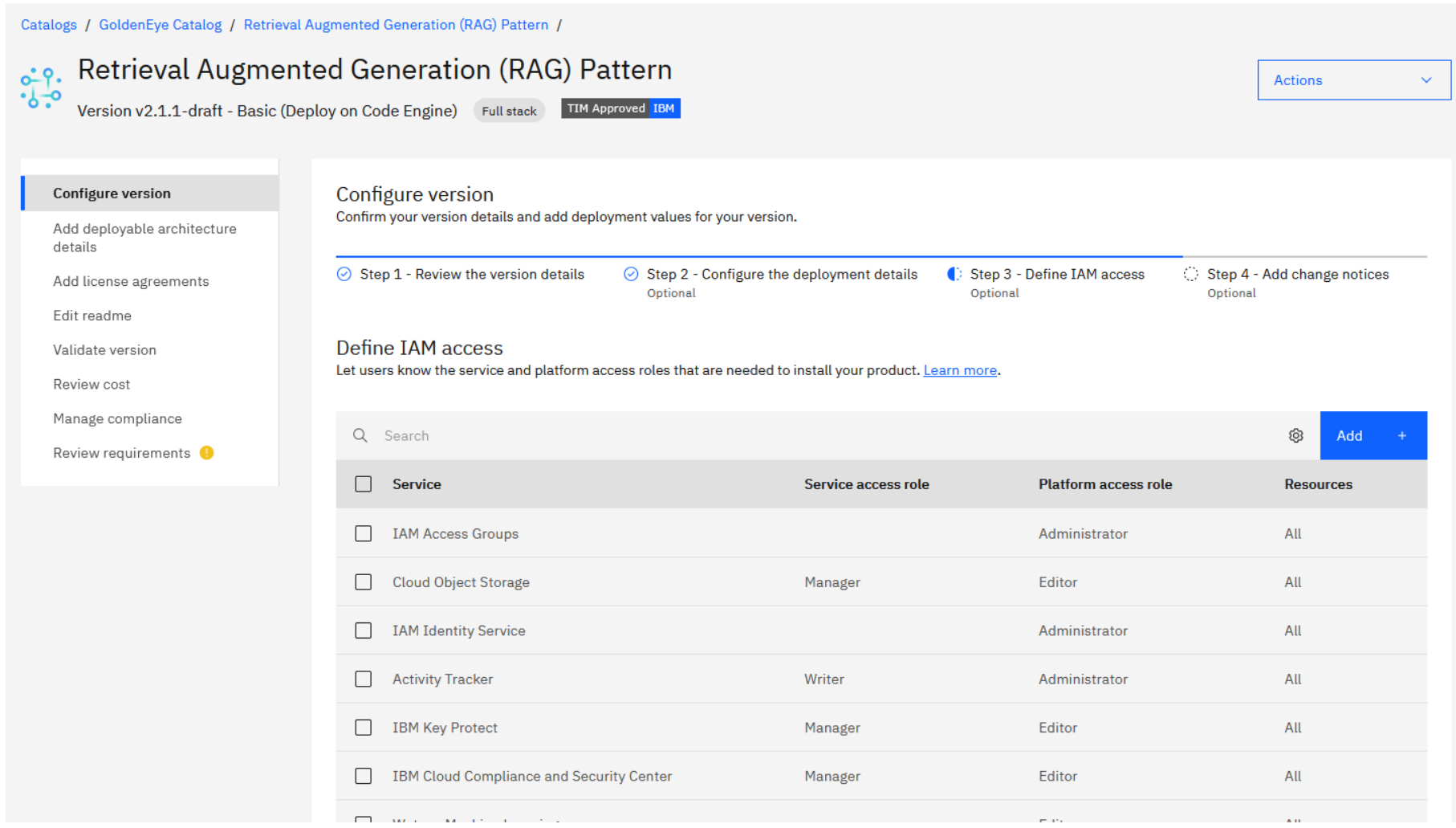
As a deployable architecture author, document the minimal API key permissions needed to apply the IaC configuration. Use IBM Cloud® Identity and Access Management terminology to help users create policies that match your documentation.

For each permission, ensure that the documentation contains:

- Indicate the service name
- Indicate the resources scope for the service (typically "All resources")
- Indicate the roles and actions for both Service and Platform access as applicable for the Service (for example: Reader, Writer, Administrator, ...)

Check out an example of such documentation in the `README.md` for the IBM Cloud COS Terraform reusable module :

For deployable architectures, use the built-in support to document the necessary permissions when you onboard into the IBM Cloud catalog.



Configuration screen allowing deployable architecture (DA) authors to document the necessary permissions to run the DA.

Validate minimal permissions

Regularly test the API keys that are used in CI automation to ensure the documented minimal permissions are sufficient for applying the solution.

1. Use the minimal set of documented permissions on the API keys used by the CI automation.
2. Exercise both `apply` and `destroy` operations to ensure that the minimal documented permissions are sufficient for the entire lifecycle of the solution. `apply` and `destroy` operations might require different sets of permissions, so validate both separately to ensure that the minimal documented permissions are accurate.

Secure your Terraform state

When you use Terraform to manage your infrastructure, it creates a state file that tracks the resources it creates. This state file is used to determine what changes need to be made to your infrastructure when you run Terraform again.

The state file is the internal representation or view of the world, as seen by Terraform. It is a JSON file that contains information about the resources that Terraform creates, such as their IDs, IP addresses, and other attributes.

Depending on the specific resources and providers that you're using, the Terraform state file might also contain secrets, such as API keys, passwords, and other sensitive data. For example, when you use the ibm_resource_key resource to create credentials, the generated credentials are stored in the Terraform state file. The Terraform state file contains sensitive data, such as API keys or passwords, that can be used to access the IBM Cloud resources.

It's essential to ensure that the Terraform state file is properly secured and protected to prevent unauthorized access to sensitive data:

- Terraform provides a native way to store the state remotely, which can be used to store the state in an encrypted Object Storage bucket by using the s3 backend or in a Kubernetes secret. This approach requires effort, including creating and securing the Object Storage bucket, managing access and encryption keys, handling state file backups and restores, and updating the Terraform configuration as needed.
- **Recommended**: A simpler way to manage your Terraform state securely is by using IBM Cloud Schematics. It provides support for managing Terraform state, making it easier to keep your IaC secure and up to date.

Using COS to store Terraform state

Although Amazon S3 is a de facto standard for object storage, many storage systems, including IBM Cloud Object Storage, implement S3-compatible APIs. This way, you can use Terraform's built-in S3 support with some modifications. To avoid calling Amazon-specific APIs, pass specific flags to the backend configuration.

The following configuration stores the Terraform state file in an IBM Cloud Object Storage bucket. The following configuration is validated for Terraform 1.9.

```
$ backend "s3" {
```

```
  bucket = "bucket-name"
  key    = "terraform.tfstate"
  endpoints = {
    s3 = "https://s3.eu-gb.cloud-object-storage.appdomain.cloud"
  }
  region = "eu-gb"
  skip_region_validation      = true
  skip_credentials_validation = true
  skip_metadata_api_check     = true
  skip_requesting_account_id  = true
  skip_s3_checksum            = true
  shared_credentials_file     = "credentials.txt"

}
```

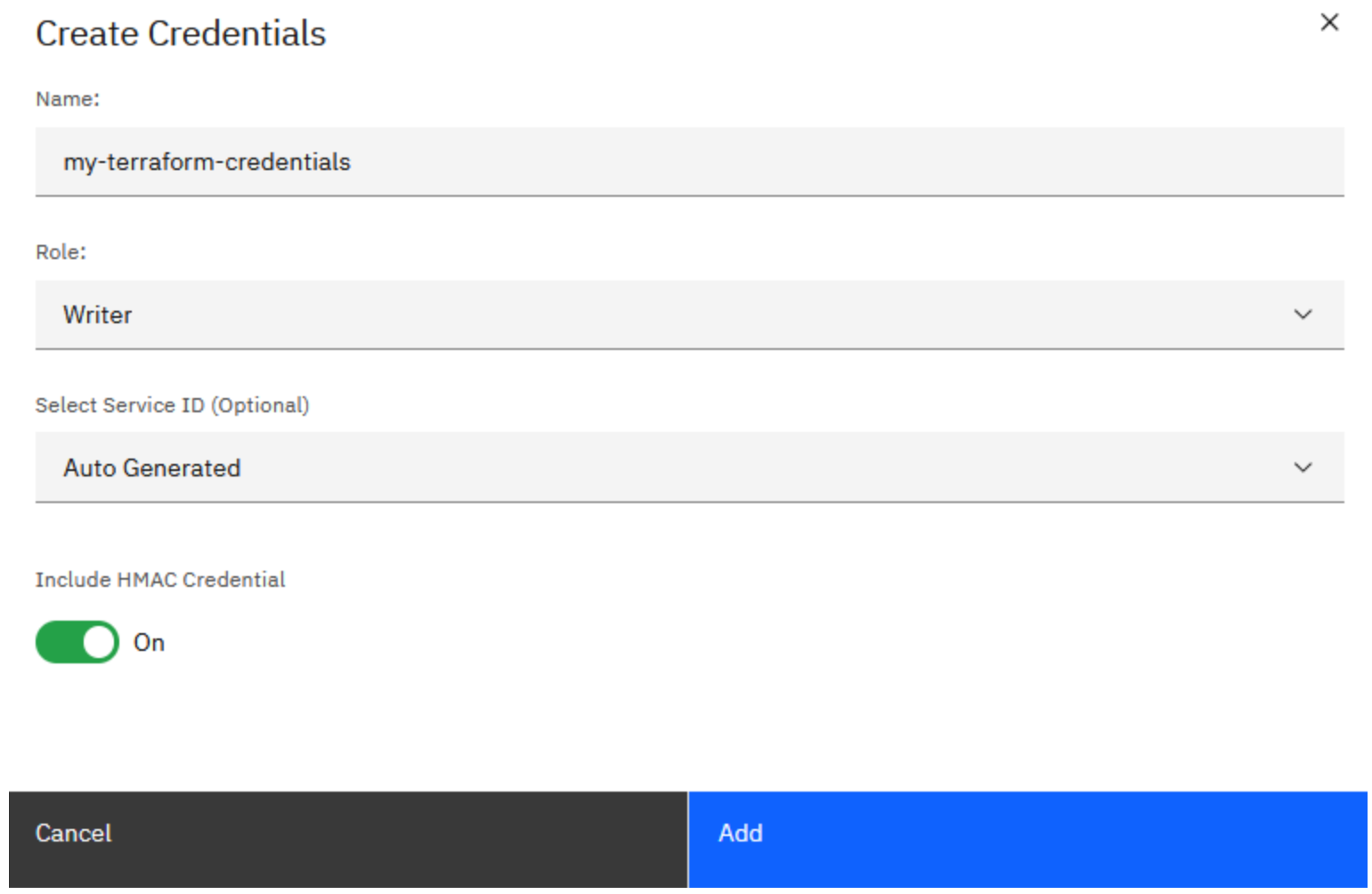
- `bucket`: The global name of your bucket.
- `key`: The name of the state file in your bucket, which the backend creates after the first apply.
- `endpoints`: Points to the IBM Cloud Object Storage API endpoint, which is regional. You can find the endpoint in the Object Storage configuration window.
- `region`: Set to `eu-gb`, although this value is not used by the backend in this case. It is marked as mandatory as otherwise causes a validation error.
- `skip_` statements: Necessary to indicate to the backend to skip calls to Amazon S3-specific functions that do not exist on IBM Cloud.
- `shared_credentials_file`: Points to the file that contains the credentials to access the Object Storage bucket (read and write access).

The following is an extract of the `credentials.txt` file

```
$ [default]
aws_access_key_id = < your key id >
aws_secret_access_key = < secret key >
```

Note: The key names contain the term "aws", but it is not an error and you can still use your IBM Cloud key and secret values here.

The credentials are created at the Object Storage instance level, under the "Service credentials" tab. Refer to Using HMAC credentials for steps. Make sure to select "Include HMAC Credential" when you create credentials for use with the Terraform S3 backend. The access and secret access key that is used by the S3 backend are HMAC credentials.



Create Credentials screen highlighting the option to include HMAC credentials

In addition to configuring Terraform to store the state in a Object Storage bucket, follow these other security best practices:

Encrypt data

IBM Cloud Object Storage encrypts data by default by using a randomly generated key that is managed by IBM Cloud. Integrate IBM Cloud Object

Storage with KMS system like Key Protect and HPCS for more control over encryption key management

Control access

Review and control access to the Object Storage bucket by using IBM Cloud® Identity and Access Management policies , applying a least-privileged approach.

Use a regional or cross-regional bucket

Ensure high availability and resiliency by using a bucket that is at minimum regional (that is: replicated across multiple data-centers in the same region) or cross-regional. Avoid using a single data center bucket.

Enable versioning

Enable versioning on the Object Storage bucket to revert to a previous version of your state file if corruption occurs.

Using Schematics to store Terraform state with encryption by default

As described in IBM Cloud Schematics section, IBM Cloud Schematics manages and stores the Terraform state file for any Terraform execution in a Schematics workspace. By default, all data, including the state file, is encrypted by using a randomly generated IBM®-managed encryption key. Encryption happens automatically, ensuring that all state files generated from Schematics are encrypted by default.

You can have further control over the encryption key that is used by Schematics by integrating with IBM Cloud Key Management Systems (KMS): Key Protect and Hyper Protect Crypto Services (HPCS). This approach helps meet compliance requirements or ensures that a key that only you fully own or control can decrypt the data, rather than relying on an IBM®-managed key.

These services offer different levels of control over the storage of encryption keys:

- **Key Protect**: Implements a Bring Your Own Key (BYOK) model where you can generate and control your own encryption key. You have control over the lifecycle of your encryption keys, but ultimately hand them over to IBM Cloud for management.
- **Hyper Protect Crypto Services**: Implements a Keep Your Own Key (KYOK) model, where users generate and control their key, which is stored in a dedicated Hardware Security Module (HSM) rather than on shared IBM Cloud infrastructure.

To enable KMS integration in Schematics, follow the steps that are outlined in KMS integration for BYOK or KYOK. For more information on how Schematics stores your data, see Securing your data with encryption

Control access to unencrypted data : Encrypting data is not enough; you must also control who can access the unencrypted data, including the state file. Schematics integrates with IBM Cloud® Identity and Access Management (Cloud Identity and Access Management) to control access. Further details on the Schematics roles can be found at Managing user access.

⚠ **Important:** Be cautious when you grant the Reader service role on Schematics resources, as it can lead to elevation of privilege problems. Although Reader is the lowest level of permission in Schematics, users with this role can access to workspaces, including the state file, which might contain sensitive information like API Keys. Users with the Reader service role might use the command `ibmcloud schematics state pull --id WORKSPACE_ID --template TEMPLATE_ID` to download locally the full Terraform state file. This information can then be used to access other resources, potentially leading to an elevation of privilege problem.

Document compliance claims for deployable architectures

Deployable architecture authors play a critical role in helping ensure transparency by documenting the compliance claims associated with their solutions. Documenting the compliance claims involves declaring the security and compliance controls that the infrastructure created by the automation meets. Providing clear documentation of these claims empowers consumers of the deployable architecture to assess its suitability for their environments and regulatory requirements.

IBM Cloud Security and Compliance Center (SCC) and IBM Cloud catalog offer a structured approach for specifying and communicating these compliance claims. By using compliance profiles, deployable architecture authors can map their architectures to established frameworks, such as NIST SP-800-53, PCI-DSS, or specific industry-focused framework like the IBM Cloud Framework for Financial Services.

Example of compliance documentation

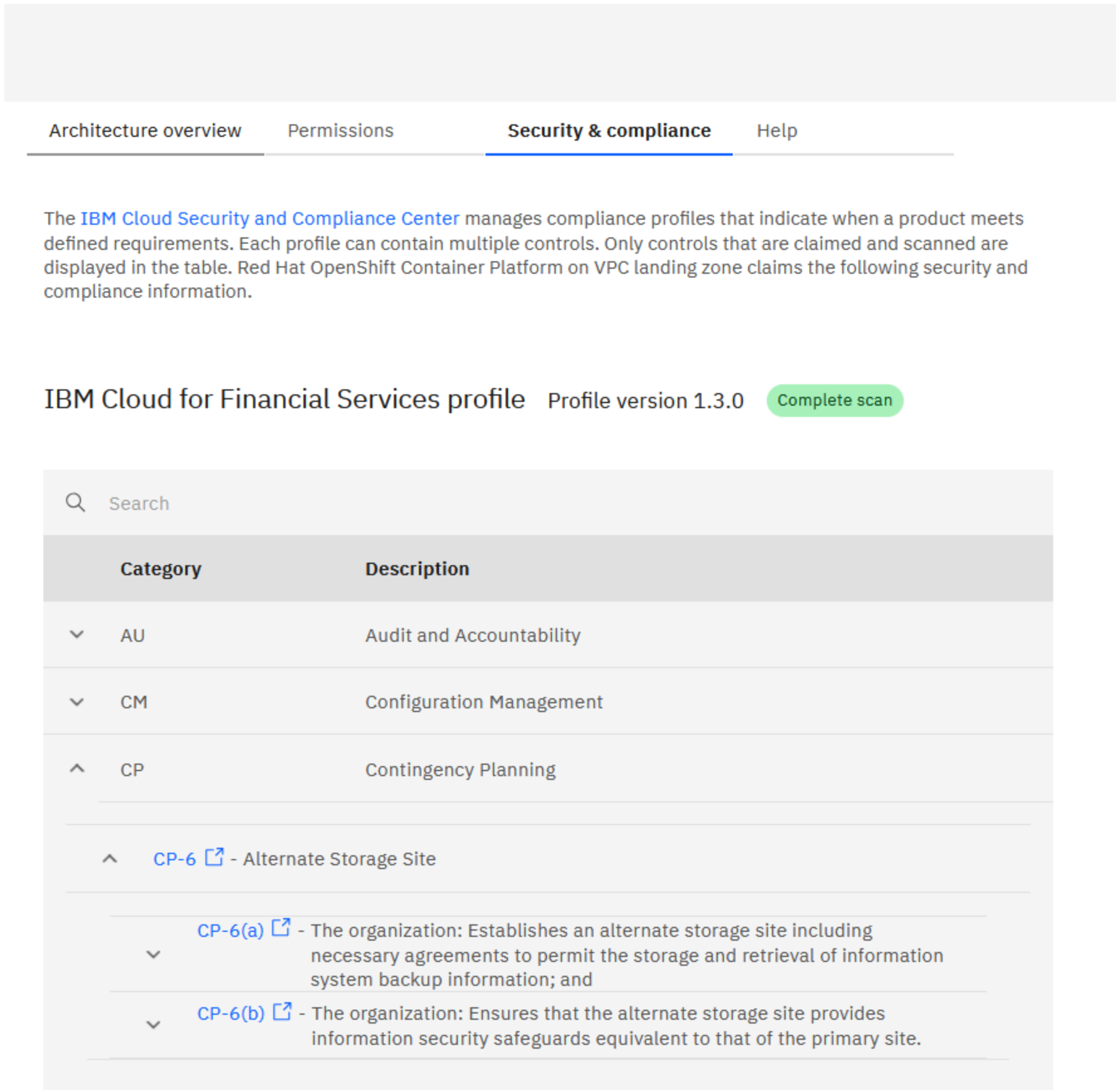
For step-by-step guidance on specifying compliance, see Managing compliance information for your deployable architecture .

The following example illustrates how the authors of the IBM Cloud Landing Zone DA declare compliance with the SCC IBM Cloud Framework for Financial Services profile.

```
$ "compliance": {
  "authority": "scc-v3",
  "profiles": [
```

```
    {
      "profile_name": "IBM Cloud for Financial Services",
      "profile_version": "1.3.0"
    }
  ]
}
```

This manifest ensures that consumers can easily identify which compliance measures the deployable architecture meets and evaluate its compatibility with their security policies. Users can access these compliance details in the IBM Cloud catalog under the Security & Compliance tab, enabling informed decision-making.



End-users can access compliance details for a deployable architecture in the IBM Cloud Catalog under the Security & Compliance tab

Key tips for deployable architecture authors:

- Use standard compliance profiles whenever possible to simplify understanding
- Include only the controls met by the infrastructure resources from the deployable architecture
- Keep compliance claims up to date as the deployable architecture evolves

Use compliance scans on infrastructure created by automation

From an operational standpoint, it is equaly critical to ensure that the infrastructure that is provisioned by the deployable architecture remains compliant over time. Continued compliance involves scanning the deployed resources with IBM Cloud Security and Compliance Center (SCC) regularly to detect misconfigurations or drift from the declared compliance standards.

Start with the deployable architecture compliance claims

Use the compliance profile specified by the deployable architecture as a starting point for scanning your infrastructure. Deployable architecture authors can claim a list of SCC compliance controls when you onboard a deployable architecture to the IBM Cloud catalog. The Security & Compliance section of a deployable architecture surfaces the list of controls that are scanned and passing within the profile. See Documenting compliance claims for details.

Establish a regular scanning schedule

Scan your infrastructure at least once a week, or more frequently based on regulatory obligations. For critical environments, daily scans can help promptly identify and address compliance regressions. However, consider balancing scan frequency with associated costs.

Set up event notifications for compliance alerts

Set up event notifications to detect regressions early, allowing quick response and remediation to minimize risks and maintain compliance.

Simplify setup with preconfigured deployable architectures

The IBM Cloud Essential Security and Observability Services deployable architecture streamlines the process of configuring SCC and its supporting services, such as storage and observability. While this deployable architecture does not configure compliance scanning, it provides a significant portion of the setup and configuration, leaving only the scanning profile to be configured within SCC.

Scan code for security vulnerabilities

Integrating security scanning tools into the development workflow ensures that vulnerabilities are detected early, reducing the risk of costly issues for the consumers of your solution.

Several tools are available to scan your code for security vulnerabilities and compliance issues:

- detect-secrets to detect secrets in code. This tool can be used to detect secrets that may have been inadvertently left in your code, such as an API Key, before committing to a repository.
- IBM Code Risk Analyzer to check for compliance issues by analyzing the Terraform plan output. This detects noncompliance before you deploy the infrastructure. It specializes in analyzing compliance for IBM Cloud resources specifically.
- checkov to check for security and compliance issues in Terraform code and Helm chart configuration. Unlike IBM Code Risk Analyzer, it does not have specific support for IBM Cloud resources, but is recommended for scenarios where the IaC solution uses extra assets, such as Helm chart definitions.
- tfsec: A security scanner for Terraform that checks for security vulnerabilities.

By incorporating these tools in CI pipelines, you can enforce a "shift-left" security philosophy and minimize downstream risks.

Refer to Terraform IBM Modules (TIM): curated and opinionated development tools and templates for a wider range of CI practices.

Manage dependency versions in IaC solutions

The following section is intended for authors of IaC solutions to guide the proper management of dependency versions. It is not intended for operators of the solution who consume modules or deployable architectures.

Managing dependencies in your IaC solution is critical to maintaining stability, security, and predictability in your deployments. Dependencies, such as Terraform provider, reusable modules, Helm charts, or container images, must be carefully managed through version pinning and automated updates to help ensure reliable operations. For more information about dependencies, see Selecting 3rd party Terraform providers and modules.

Pin versions for stability and predictability

Pinning versions for all dependencies ensures stability by preventing unexpected changes due to updates. It also provides reproducibility, allowing your infrastructure to be re-created identically in the future, and enables controlled upgrades after testing.

- **Modules**: Always specify the version for referenced modules to avoid pulling breaking changes.
- **Terraform providers**: Specify provider version explicitly in root modules (such as those used in deployable architectures). For reusable modules, use ranges to maximize consumability.
- **Other dependencies**: Specify explicit versions for Helm charts, container images, and other external dependencies.

Review the following code examples for pinning:

- Module

```
$ module "cbr_rule" {
  source  = "terraform-ibm-modules/cbr/ibm//modules/cbr-rule-module"
  version = "1.29.0" # Pin to a specific module version
  ...
```

```
}
```

- Terraform provider

```
$ terraform {
  required_providers {
    ibm = {
      source  = "IBM-Cloud/ibm"
      version = "1.70.0" # Pin to a specific provider version
    }
  }
  required_version = ">= v1.0.0, < 2.0.0" # Pin Terraform CLI version range
}
```

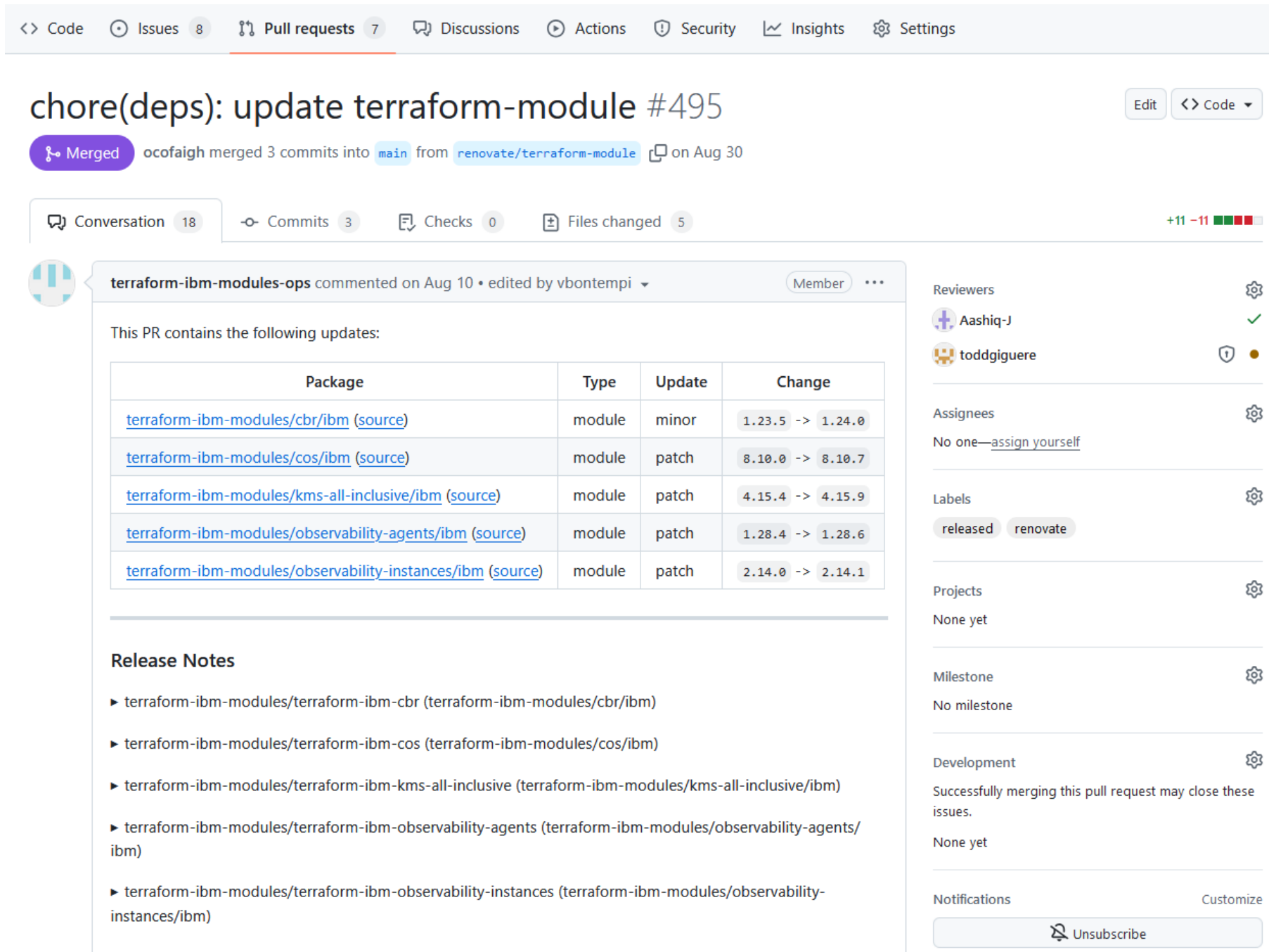
Automate dependency version management

When versions are pinned, regularly updating them is essentials to keep your dependencies secure and compatible. Automation tools, such as renovatebot streamline this process by:

- Detecting new versions of pinned dependencies and opening pull requests to update them.
- Running CI pipelines to validate the updates and detect potential regressions
- Allow manual review before you merge updates

The following is an example workflow that uses renovatebot:

1. Renovatebot detects new versions of your pinned dependencies
2. It opens a pull request, such as this PR, indicating the updated versions



Pull request opened by renovatebot for a Terraform module, detailing updates to various packages, including version changes, types of updates (minor/patch), and associated release notes for each dependent module.

3. CI pipelines validate the changes, ensuring that no regressions occur.
4. A human reviewer approves the changes before they merge.

About the authors

Vincent Burckhardt

Architect, IBM Cloud Platform

https://www.linkedin.com/in/vburckhardt/

Jay Warfield

IBM Cloud CoE Architect

https://www.linkedin.com/in/jaywarfield/

IBM Corporation
New Orchard Road
Armonk, NY 10504

Produced in the United States of America
2025-04-02