IaC Guide For ACP

Terraform Edition

Alauda Team

Alauda Team

Table of contents

1. Introduction

In modern cloud computing environments, infrastructure management and deployment have become increasingly complex. To address this complexity, Infrastructure as Code (IaC) has emerged as a widely adopted practice. This guide will introduce how to apply the principles of IaC to the ACP (Alauda Container Platform) and leverage Terraform, a powerful IaC tool, to achieve automated and standardized infrastructure management.

1.1 Overview of ACP Platform

ACP (Alauda Container Platform) is an advanced cloud platform product built on Kubernetes. It offers a rich set of cloud-native features, enabling organizations to deploy, manage, and scale their applications and services more efficiently. ACP exposes its resources through Kubernetes APIs, providing users with a unified and powerful interface for platform resource operations.

1.2 IaC Concept

Infrastructure as Code is an approach to defining, deploying, and managing infrastructure using code. It treats infrastructure configuration as software code, enabling version control, automated deployment, and consistent management. The key advantages of IaC include:

- Increased deployment speed and efficiency
- Reduced human error
- Enhanced configuration consistency and reproducibility
- Improved collaboration and version control
- Simplified infrastructure management and maintenance

1.3 Introduction to Terraform

Terraform, developed by HashiCorp, is a tool for IaC management that allows users to manage infrastructure and business applications using a declarative language. Terraform's support for a wide range of cloud service providers and its active community ecosystem make it an ideal choice for managing complex infrastructure.

1.4 The Perfect Combination of ACP and Terraform

As a Kubernetes-based cloud platform, ACP is naturally suited for IaC management approaches. By integrating Terraform with the ACP platform, users can easily achieve automated management and version control of ACP resources, making ACP management and maintenance more reliable and efficient.

2. Terraform and ACP Integration

The ACP (Alauda Container Platform) is a cloud platform product developed as an extension of Kubernetes. It exposes its resource management interfaces in the form of Kubernetes APIs. This architectural design allows ACP to seamlessly integrate with Kubernetes-related Providers in the Terraform ecosystem, enabling infrastructure-as-code management.

2.1 Configuring the Provider

In your Terraform configuration file, you need to specify alekc/kubectl as the required provider. Here's an example configuration:

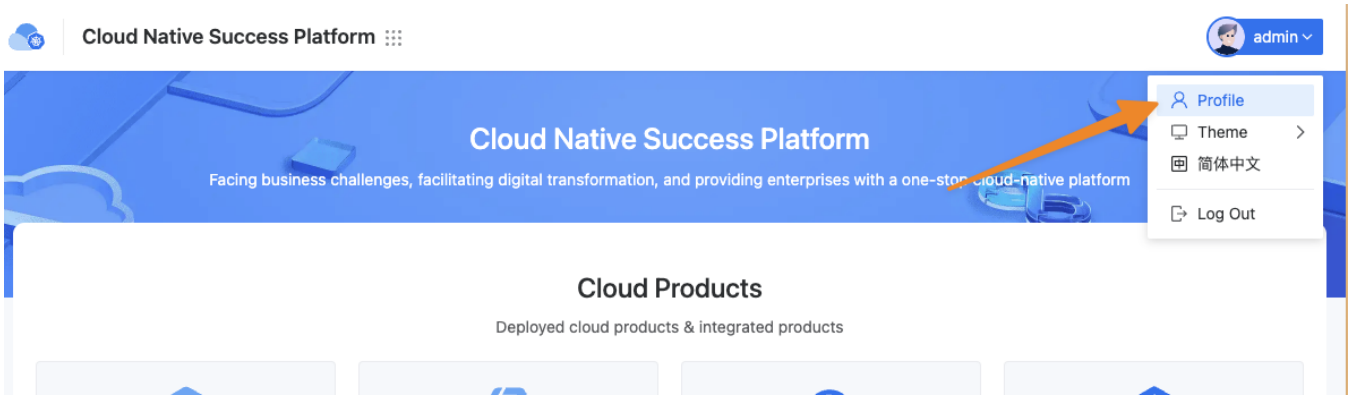
```
terraform {
  required_providers {
    acp = {
      source  = "alekc/kubectl"
      version = "~> 2.0"
    }
  }
}
```

Since ACP is a multi-cluster management platform, we recommend using the method of defining a provider with multiple aliases to define the provider, with each alias corresponding to a cluster name. This method allows you to manage resources for multiple clusters in the same Terraform configuration and improve the readability and maintainability of the configuration.

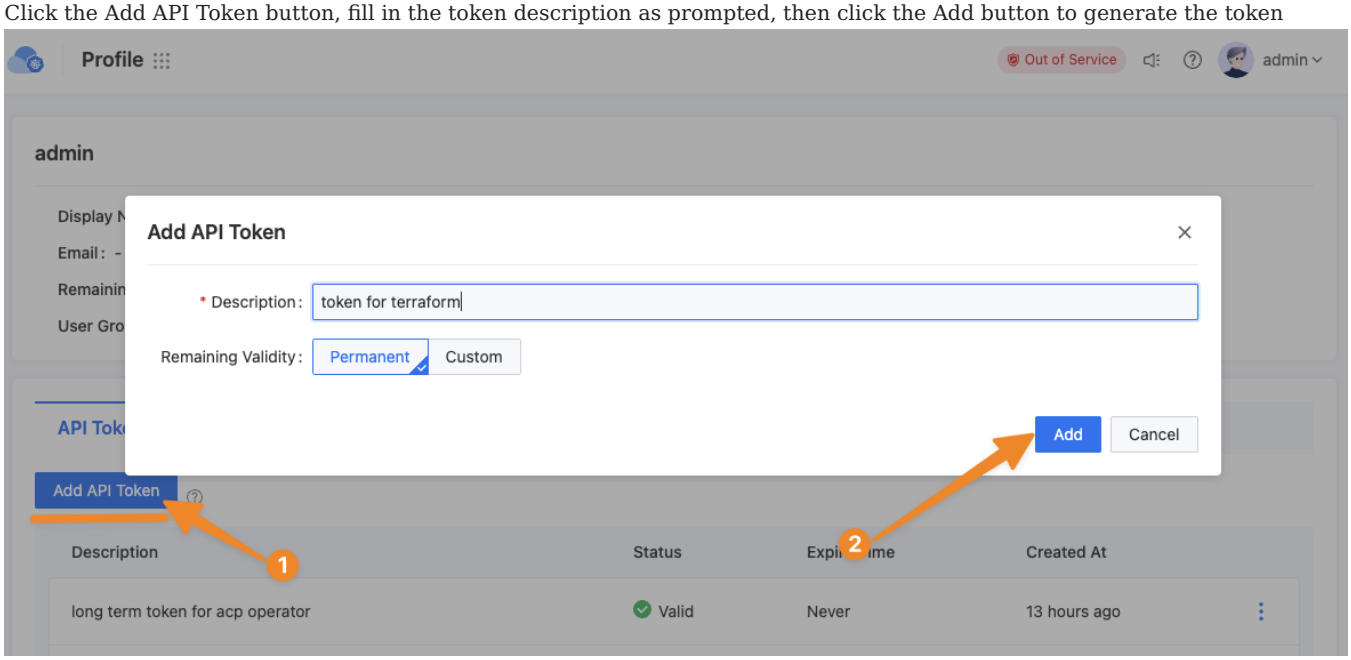
```
variable "acp_endpoint" {
  type string
}

variable "acp_token" {
  type string
}

provider acp {
  alias = "global"
  host = format("%s/kubernetes/global", trimsuffix(var.acp_endpoint, "/"))
  token = var.acp_token
  load_config_file = false
  insecure = false # if acp doesn't have a valid https certificate, set this to true
}
provider acp {
  alias = "cluster-1"
  host = format("%s/kubernetes/cluster-1", trimsuffix(var.acp_endpoint, "/"))
  token = var.acp_token
  load_config_file = false
  insecure = false # if acp doesn't have a valid https certificate, set this to true
}
```

The input parameters in the above example are explained as follows:

- `alias`: Represents the alias for different clusters. It's recommended to keep this consistent with the cluster names in the ACP platform.
- `host`: The access address for the Kubernetes cluster. ACP provides a unified proxy for different business clusters. By using the ACP platform address with the `/kubernetes/<cluster-name>` path, you can proxy to the Kubernetes service of the corresponding business cluster. This method allows uniform access to different business clusters using the ACP user's token.
- `token`: The access token for the ACP platform. It can be obtained through the following steps:
- Access the ACP platform and click the Profile button in the avatar to enter the user's details page



- Click the Add API Token button, fill in the token description as prompted, then click the Add button to generate the token



- `load_config_file`: Determines whether to load the local `kubeconfig` configuration. We recommend setting this to `false` and using only the `host` and `token` method to configure the cluster.
- `insecure`: Set this parameter to `false` if your ACP environment has a valid HTTPS certificate. Otherwise, set it to `true`.

3. ACP Resource Management

This section provides a detailed overview of managing ACP resources using Terraform. We offer three primary methods, each with its specific use cases and advantages. Users can choose the most suitable approach based on their requirements.

3.1 Using Terraform ACP Modules

Our company has encapsulated Terraform modules for several ACP resources. We recommend users to prioritize these modules for managing ACP resources. Using ACP Modules offers the following advantages:

- **Simplified Configuration**: Modules encapsulate complex resource configurations, allowing users to manage ACP resources through simple parameter settings.
- **Best Practices**: These modules are designed according to ACP-recommended best practices, making IaC writing elegant and efficient.
- **Maintenance Support**: These modules are continuously maintained and updated by Alauda Cloud, ensuring compatibility with the ACP platform.

ACP Modules are hosted in the alauda/terraform-acp-modules repository. The README document in the repository introduces the basic usage of the modules to help users get started quickly. Additionally, the README document for each module in the repository provides detailed parameter descriptions and usage examples, allowing users to select appropriate modules based on their needs.

3.2 Managing ACP Resources Using the Kubectl Provider

For users requiring greater flexibility, the `alekc/kubectl` provider can be used to directly manage ACP resources. This method allows users to precisely control all parameters of ACP resources and is particularly suitable for users familiar with ACP resource APIs. It serves as an effective complement, especially in scenarios not covered by our provided Terraform modules.

> If you find that certain scenarios are not covered by existing ACP modules, please feel free to contact us. We actively expand and improve ACP modules to meet more use cases.

ACP resources are provided through Kubernetes APIs. This means we can directly use the Kubectl Provider to operate these resources, just like operating regular Kubernetes resources. The following example demonstrates how to use the `alekc/kubectl` provider to manage ACP resources:

3.2.1 Scenario Introduction

In this example, we assume there are two business clusters on the ACP platform:

- `dev`: Cluster for the development environment
- `prod`: Cluster for the production environment

We will deploy different versions of the same application in these two clusters, with specific configurations for each environment:

- Development environment: Uses a newer image version, single replica deployment, and configures specific development environment variables
- Production environment: Uses a stable image version, multi-replica deployment, and configures specific production environment variables

This example demonstrates how to use Terraform with ACP to manage applications across multiple clusters simultaneously, achieving environmental consistency and configuration differentiation.

3.2.2 Example Code

```
terraform {
  required_providers {
    acp = {
      source  = "alekc/kubectl"
      version = "~> 2.0"
    }
  }
}

variable "acp_endpoint" {
  type string
}

variable "acp_token" {
  type string
}

provider acp {
  alias = "dev"
  host = format("%s/kubernetes/dev", trimsuffix(var.acp_endpoint, "/"))
  token = var.acp_token
  load_config_file = false
}
provider acp {
  alias = "prod"
  host = format("%s/kubernetes/prod", trimsuffix(var.acp_endpoint, "/"))
  token = var.acp_token
  load_config_file = false
}

# application for dev environment
resource "kubectl_manifest" "dev_application" {
  provider = acp.dev
  yaml_body = <<YAML
apiVersion: app.k8s.io/v1beta1
kind: Application
metadata:
  name: my-app-dev
  namespace: default
spec:
  assemblyPhase: Succeeded
  componentKinds:
    - group: apps
      kind: Deployment
    - group: ""
      kind: Service
  descriptor: {}
  selector:
    matchLabels:
      app.cpaas.io/name: default/my-app-dev
YAML
}

resource "kubectl_manifest" "dev_application_deployment" {
    provider = acp.dev
    yaml_body = <<YAML
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.cpaas.io/name: default/my-app-dev
  name: my-app-dev
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      service.cpaas.io/name: default/my-app-dev
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app.cpaas.io/name: default/my-app-dev
        service.cpaas.io/name: default/my-app-dev
    spec:
      containers:
        - image: my-app:latest
          name: app
YAML
}

resource "kubectl_manifest" "dev_application_service" {
    provider = acp.dev
    yaml_body = <<YAML
apiVersion: v1
kind: Service
metadata:
```

```
  name: my-app-dev
  namespace: default
spec:
  selector:
    app.cpaas.io/name: default/my-app-dev
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 80
YAML
}

# application for prod environment
resource "kubectl_manifest" "prod_application" {
  provider = acp.prod
  yaml_body = <<YAML
apiVersion: app.k8s.io/v1beta1
kind: Application
metadata:
  name: my-app-prod
  namespace: default
spec:
  assemblyPhase: Succeeded
  componentKinds:
    - group: apps
      kind: Deployment
    - group: ""
      kind: Service
  descriptor: {}
  selector:
    matchLabels:
      app.cpaas.io/name: default/my-app-prod
YAML
}

resource "kubectl_manifest" "prod_application_deployment" {
    provider = acp.prod
    yaml_body = <<YAML
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.cpaas.io/name: default/my-app-prod
  name: my-app-prod
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      service.cpaas.io/name: default/my-app-prod
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app.cpaas.io/name: default/my-app-prod
        service.cpaas.io/name: default/my-app-prod
    spec:
      containers:
        - image: my-app:1.23.4
          name: app
YAML
}

resource "kubectl_manifest" "prod_application_service" {
    provider = acp.prod
    yaml_body = <<YAML
apiVersion: v1
kind: Service
metadata:
  name: my-app-prod
  namespace: default
spec:
  selector:
    app.cpaas.io/name: default/my-app-prod
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 80
YAML
}
```

The usage of Provider declarations in the example can be referred to in the introduction of Terraform and ACP Integration.

3.3 Encapsulating Custom Terraform Modules

For enterprise users with specific requirements, encapsulating custom Terraform Modules that fit the company's internal use cases is an ideal choice. This approach allows for module design completely based on the company's specific needs and best practices, providing a unified resource management standard for internal use, further improving team efficiency and reducing repetitive work.

We suggest users fork and modify the alauda/terraform-acp-modules repository, or create a new repository referencing the structure and recommended practices of alauda/terraform-acp-modules for development. This repository not only provides implementations of existing modules but also details how to develop ACP modules in the Development document, along with important principles and suggestions to follow during development. This can help you create high-quality, easily maintainable custom ACP modules.

> If you encounter any issues or need further guidance during development, please feel free to contact our support team. We are more than happy to assist you in fully leveraging the powerful features of the ACP platform and Terraform.

4. FAQ

4.1 Why choose alekc/kubectl as the Provider?

We recommend using the alekc/kubectl provider instead of the hashicorp/kubernetes provider. This recommendation is based on the following considerations:

- **Broader Compatibility**: The alekc/kubectl Provider manages resources by directly executing kubectl commands. This approach supports all types of Kubernetes resources, including those provided through aggregated API servers. In contrast, the official Kubernetes Provider relies on server-side apply capabilities, which may be limited in certain situations.
- **More Flexible Deployment Process**: During the `terraform plan` phase, the official Kubernetes provider requires that the CRDs (Custom Resource Definitions) for all resources being used already exist in the target cluster. This means users cannot deploy resource CRDs and resource instances simultaneously within the same Terraform repository. The alekc/kubectl provider, however, does not need to verify the existence of CRDs during the plan phase, offering greater flexibility.
- **Simplified Resource Management**: The alekc/kubectl provider allows direct use of native Kubernetes YAML definitions to manage resources. For users familiar with Kubernetes, this can significantly reduce the learning curve and simplify the resource management process.
- **Better Version Compatibility**: Since the alekc/kubectl provider relies on the kubectl command-line tool, it typically adapts better to different versions of Kubernetes clusters, reducing the occurrence of version compatibility issues.