AI VOID FIELD GUIDE

KubeVela for Platform Teams: Building an Application Delivery Control Plane

Learn how KubeVela simplifies application delivery on Kubernetes for platform teams, leveraging the Open Application Model (OAM) to build a robust control plane.

AI VOID

2026-06-22 18:55 IST

Contents

01	Beyond Raw Kubernetes: Why Platform Teams Need an Application Delivery Control Plane	3
02	Setting Up KubeVela and Unpacking the Open Application Model (OAM)	16
03	Your First Application: Defining Components with KubeVela	28
04	Adding Capabilities: Scaling and Exposing Applications with Traits	39
05	Defining Your Delivery Pipeline: Mastering KubeVela Workflows	49
06	Governing Applications: Applying Policies for Compliance and Security	63
07	Empowering Platform Engineers: Customizing KubeVela with Definitions and Addons	75
08	KubeVela in Practice: Real-World Scenarios and Ecosystem Comparisons	92

CHAPTER 01

Beyond Raw Kubernetes: Why Platform Teams Need an Application Delivery Control Plane

Introduction

Welcome to a deep dive into KubeVela, an application delivery control plane designed to simplify the complexities of deploying and managing applications on Kubernetes. If you're an experienced software engineer or part of a platform team grappling with the operational overhead of modern cloud-native environments, you're in the right place.

This chapter will explain the fundamental challenges KubeVela solves, introduce its core concepts, and illustrate why it's becoming an indispensable tool for building robust internal developer platforms. We'll explore how KubeVela provides an application-centric abstraction that streamlines deployments across hybrid and multi-cloud scenarios, ultimately empowering developers while giving platform engineers greater control.

By the end of this chapter, you'll understand the "why" behind KubeVela and be ready to explore its practical applications. There are no prerequisites for this chapter, as we'll start from first principles.

The Kubernetes Paradox: Power Meets Complexity

Kubernetes revolutionized container orchestration, offering unparalleled power and flexibility for managing microservices at scale. Yet, this power comes with a significant cost: complexity. For platform teams, exposing raw Kubernetes manifests to developers often leads to a myriad of challenges:

- **YAML Overload:** Developers spend excessive time writing and maintaining verbose Kubernetes YAML files for deployments, services, ingresses, and more.
- **Operational Burden:** Platform engineers constantly define, standardize, and enforce operational concerns like scaling, traffic routing, and security policies across diverse applications.

- **Inconsistent Deployments:** Without a strong abstraction layer, developers might adopt different deployment patterns, leading to inconsistencies and increased troubleshooting effort.
- **Cognitive Load:** Understanding the intricate web of Kubernetes resources (Deployments, StatefulSets, Services, Ingresses, ConfigMaps, Secrets, RBAC, etc.) is a steep learning curve for many application developers.
- **Hybrid/Multi-Cloud Headaches:** Managing applications consistently across different clusters, cloud providers, or on-premises environments becomes a monumental task.

Platform teams often resort to building custom internal platforms, which, while effective, can be resource-intensive to develop and maintain. This is where KubeVela steps in to offer a more standardized and extensible solution.

KubeVela: An Application Delivery Control Plane

KubeVela is an open-source, Kubernetes-native application delivery control plane that abstracts away the underlying infrastructure complexities, presenting an application-centric interface to developers. It's built on the Open Application Model (OAM), which provides a clear separation of concerns between platform engineers and application developers.

📌 `Key Idea:` KubeVela acts as a "smart orchestrator" on top of Kubernetes, allowing platform teams to define how applications should be delivered and operated, while developers focus on what their application does.

Why KubeVela Matters to Platform Teams

KubeVela addresses the Kubernetes paradox by providing:

- **Higher-Level Abstraction:** Developers interact with an `Application` resource that bundles everything needed for their service, dramatically reducing the YAML they write.
- **Standardized Delivery:** Platform engineers define reusable building blocks (components, traits, policies, workflows), ensuring consistent and compliant deployments.
- **Extensibility:** KubeVela is highly extensible, allowing platform teams to integrate any Kubernetes-native capability (CRDs, Helm charts, operators) and expose them as simple, self-service options.

- **Hybrid/Multi-Cloud Capabilities:** Designed for managing applications consistently across multiple clusters and environments from a single control plane.
- **Reduced Cognitive Load:** Developers no longer need deep Kubernetes expertise; they simply consume predefined capabilities, focusing on their business logic.

Open Application Model (OAM): The Foundation

The Open Application Model (OAM) is the conceptual backbone of KubeVela. Its core philosophy is to separate the operational concerns from the application definition. This separation empowers platform engineers to define and manage the operational aspects (e.g., scaling, networking, observability) while developers focus solely on their application code and its required components.

🧠 `Important:` OAM defines an application as a collection of components that fulfill its business logic, along with traits that describe operational behaviors, policies for governance, and workflows for delivery. This separation of concerns is fundamental to building scalable and maintainable platforms.

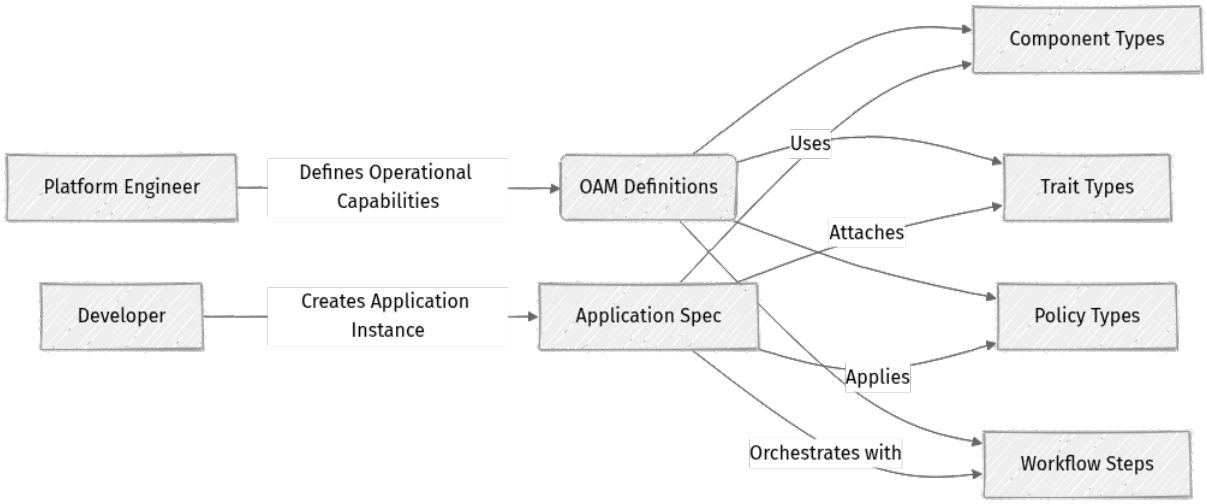


Figure 1.1: OAM's Separation of Concerns between Platform Engineers and Developers

Notice how the platform engineer defines the types of capabilities, and the developer uses instances of these types. This creates a powerful contract.

KubeVela's Core Concepts

Let's break down the fundamental building blocks within KubeVela that implement the OAM principles.

1. Application

The `Application` is KubeVela's top-level resource. It defines a complete application by combining components, traits, policies, and workflows. For a developer, the `Application` manifest is the single source of truth for their service. It's the "what" of their application, not the "how."

```yaml
# A simplified KubeVela Application manifest
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-web-app
spec:
  components:
    - name: frontend-service
      type: webservice # 'webservice' is a predefined component type
      properties:
        image: myregistry/frontend:v1.0 # The Docker image for the service
        port: 80 # The port the service listens on
      traits:
        - type: autoscaler # 'autoscaler' is a predefined trait
          properties:
            min: 2 # Minimum replicas
            max: 10 # Maximum replicas
            cpuUtil: 80 # CPU utilization percentage to trigger scaling
  workflow:
    steps:
      - name: deploy-frontend
        type: deploy # A basic deployment step
```

In this example, the developer specifies an `Application` named `my-web-app` that consists of a `frontend-service` component. They don't need to write a Kubernetes Deployment, Service, or Horizontal Pod Autoscaler directly. KubeVela, guided by the `webservice` component type and `autoscaler` trait, translates this high-level definition into the necessary Kubernetes resources, abstracting away the boilerplate.

2. Components

A `Component` defines a deployable unit of your application, such as a microservice, a database, or a worker process. KubeVela provides several built-in component types (like `webservice`, `worker`, `helm`), and platform engineers can define custom ones using `ComponentDefinition` CRDs.

When a developer uses a `webservice` component, KubeVela knows how to provision a Kubernetes Deployment and Service to run their containerized application. This is where the platform engineer's definition of "webservice" comes into play, mapping high-level properties to low-level Kubernetes constructs.

3. Traits

`Traits` are operational behaviors that can be attached to components. They enhance or modify how a component runs without changing its core definition. Think of them as modular add-ons for your components. Examples include:

- **Scaling:** `autoscaler` (Horizontal Pod Autoscaler)
- **Networking:** `ingress` (Kubernetes Ingress), `route`
- **Observability:** `metrics` (Prometheus metrics configuration), `logging`
- **Storage:** `pvc` (Persistent Volume Claim)

Platform engineers define `TraitDefinition` CRDs to expose these capabilities. This allows developers to add features like auto-scaling or an ingress simply by adding a few lines to their application manifest, completely decoupled from the component's core definition.

4. Policies

`Policies` enforce governance and compliance rules across applications. They ensure that all applications adhere to organizational standards and best practices. Policies can cover aspects like:

- **Security:** `firewall-policy`, `network-policy`
- **Cost Management:** `cost-allocation-policy`
- **Deployment Strategy:** `affinity-policy` (e.g., anti-affinity for high availability)
- **Placement:** `cluster-selector` (deploy to specific clusters or regions)

Platform engineers define `PolicyDefinition` CRDs. When an `Application` is deployed, KubeVela checks if it complies with the attached policies, preventing non-compliant deployments.

5. Workflows

`Workflows` define the delivery process of an application. This allows platform teams to orchestrate complex deployment pipelines directly within KubeVela, moving beyond simple "apply" operations. A workflow consists of a sequence of steps, which can include:

- **Deploying components**
- **Running pre/post-deployment hooks**
- **Manual approvals** (e.g., for production deployments)
- **Data migrations**
- **Integrations with external systems** (e.g., notifying Slack, updating a CMDB)

This powerful feature allows for blue/green deployments, canary releases, and other advanced strategies to be built into the application definition, managed by the platform, and consumed by developers.

6. Addons

`Addons` are pre-packaged extensions for KubeVela. They allow you to easily install and enable common tools and capabilities, extending KubeVela's out-of-the-box functionality. Examples include:

- **Observability tools:** Prometheus, Grafana, Loki
- **GitOps tools:** Argo CD (KubeVela can integrate with it)
- **Cloud provider integrations:** AWS S3, Azure Blob Storage
- **Advanced traits and policies:** Custom scaling rules, traffic management features

Addons simplify the process of extending KubeVela's functionality, making it easy to integrate with your existing ecosystem without manual YAML management.

KubeVela in Context: Comparisons

Understanding KubeVela's value often comes from comparing it to other tools in the cloud-native landscape.

KubeVela vs. Raw Kubernetes Manifests

- **Raw Kubernetes:** Developers manage low-level resources directly (Deployments, Services, etc.). This leads to high cognitive load, verbose YAML, and deployments prone to inconsistencies.
- **KubeVela:** Provides an application-centric abstraction. Developers define an `Application` with `Components` and `Traits`. This results in much less YAML, a lower cognitive load, and platform-enforced consistency.

`⚡ Real-world insight:` A typical microservice might require 5-10 Kubernetes YAML resources to define its deployment, service, ingress, and scaling. With KubeVela, this can often be condensed into a single `Application` resource, simplifying the developer experience significantly while still providing comprehensive operational control.

KubeVela vs. Helm

- **Helm:** A package manager for Kubernetes. It bundles Kubernetes resources into charts for easy deployment and versioning. It's excellent for packaging third-party applications or reusable sets of Kubernetes resources.
- **KubeVela:** An application delivery control plane. It can use Helm charts as a `Component` type, but its scope is much broader. KubeVela focuses on defining the entire application delivery process, including multi-cluster deployments, complex workflows, and policy enforcement, which goes beyond Helm's packaging capabilities.

`🔥 Optimization / Pro tip:` Many platform teams use Helm charts to package complex Kubernetes operators or common application patterns, then expose these Helm charts as `ComponentDefinition` types within KubeVela. This combines the best of both worlds: Helm for packaging and KubeVela for application-centric delivery and orchestration.

KubeVela vs. Argo CD

- **Argo CD:** A declarative GitOps continuous delivery tool for Kubernetes. It synchronizes the desired state defined in Git with the actual state in the cluster. It's excellent for GitOps workflows and ensuring cluster consistency.

- **KubeVela:** An application-centric control plane. While KubeVela can integrate with GitOps principles (e.g., storing `Application` manifests in Git and using Argo CD to sync them), its primary focus is on defining how applications are composed, delivered, and operated through OAM. KubeVela's workflows and policies provide a higher level of orchestration and governance over the entire application lifecycle, whereas Argo CD focuses on the synchronization mechanism. They are complementary, not mutually exclusive.

KubeVela vs. Custom Internal Platforms

- **Custom Internal Platforms:** Many organizations build their own platforms to abstract Kubernetes complexity. This offers maximum control but requires significant engineering effort for development, maintenance, and keeping up with the rapidly evolving cloud-native ecosystem.
- **KubeVela:** Provides an open-source, extensible framework for building internal platforms. It offers the core capabilities (OAM, workflows, policies, addons) out-of-the-box, significantly reducing the "reinventing the wheel" burden. Platform teams can customize KubeVela to fit their specific needs, integrating their existing tools and defining unique capabilities without starting from scratch.

Step-by-Step Implementation: Installing KubeVela

Let's get KubeVela up and running on your Kubernetes cluster. For this guide, we'll assume you have a running Kubernetes cluster and `kubectl` configured to access it.

As of `2026-06-22`, the latest stable version of KubeVela is assumed to be `v1.10.2`. Please always verify the absolute latest stable release on the official KubeVela documentation or its GitHub releases page at the time of your installation.

Prerequisites

- A Kubernetes cluster (version `1.19+`).
- `kubectl` installed and configured to access your cluster.

Install KubeVela CLI (Velacli)

First, install the KubeVela command-line tool, `velacli`. This tool simplifies interacting with KubeVela.

```
# For macOS / Linux (using Homebrew, recommended)
brew install kubevela
```

⚡ Quick Note: If you're on Linux and Homebrew is not an option, you can download the `vela` CLI binary directly from the KubeVela GitHub releases page (e.g., `<https://github.com/kubevela/vela/releases/download/v1.10.2/vela-v1.10.2-linux-amd64.tar.gz>`), extract it, and move the `vela` executable to a directory in your system's `PATH` (like `/usr/local/bin`).

Verify the installation of the CLI:

```
vela version
```

You should see output similar to this, indicating the client version:

```
CLI Version: v1.10.2
Core Version: Not installed
```

The "Core Version: Not installed" is expected at this stage because we haven't deployed the KubeVela control plane into your Kubernetes cluster yet.

Install KubeVela Control Plane on Kubernetes

Now, let's deploy KubeVela's core components into your Kubernetes cluster. This will install the KubeVela controller, CRDs, and other necessary components into the `vela-system` namespace.

```
# Install KubeVela (this command will deploy the latest stable version by default)
vela install

# Or, to specify a particular version (e.g., v1.10.2, assumed latest for this guide)
# vela install --version v1.10.2
```

This command will take a few minutes to complete as it deploys several components. You can monitor the deployment status by watching the pods in the `vela-system` namespace.

```
kubectl get pods -n vela-system
```

Wait until all pods in the `vela-system` namespace are in the `Running` state. You should see something similar to:

```
NAME                                          READY   STATUS    RESTARTS
AGE
kubevela-core-7b9c9f7d4c-abcde                 1/1     Running   0
2m
vela-cluster-gateway-55c8c5c7d-fghij           1/1     Running   0
2m
velaux-86d6d6c9f-klmno                         1/1     Running   0
2m
```

Once installed, verify the KubeVela version again:

```
vela version
```

Now, you should see both client and core versions matching:

```
CLI Version: v1.10.2
Core Version: v1.10.2
```

Congratulations! You have successfully installed KubeVela on your Kubernetes cluster. You're now ready to start defining applications in a whole new way.

Mini-Challenge: Explore KubeVela Definitions

Now that KubeVela is installed, take a moment to peek behind the curtain. KubeVela comes with many built-in `ComponentDefinition`, `TraitDefinition`, and `PolicyDefinition` resources that platform engineers have at their disposal.

Challenge: List all available `ComponentDefinition` resources in your cluster.

Hint: Think about how you would list any custom resource in Kubernetes using `kubectl`. KubeVela's definitions are just CRDs.

```
# Your command here
```

💡 **SOLUTION**

```
kubectl get ComponentDefinition
```

You should see a list of predefined component types like `webservice`, `worker`, `helm`, etc. These are the high-level building blocks that developers can immediately use to define their applications, abstracting away the underlying Kubernetes complexity.

What to observe/learn: Notice the names of the components and consider how each one might map to a specific type of application or service. This exercise gives you a concrete understanding of the abstractions KubeVela provides out-of-the-box.

Common Pitfalls & Troubleshooting

⚠️ `What can go wrong:` Installing new software on Kubernetes can sometimes be tricky. Here are a few common issues and how to approach them.

1. **`vela install` hangs or fails:**
    - **Issue:** Network connectivity problems preventing KubeVela container images from being pulled, or insufficient cluster resources.
    - **Troubleshooting:**
        - Check `kubectl get events -n vela-system` for detailed error messages, which often point to image pull failures or scheduling issues.
        - Ensure your cluster has enough CPU and memory. KubeVela components require some resources to run efficiently.
        - Verify your Kubernetes cluster is healthy and `kubectl` can communicate with it without issues.
        - If using a private or custom container registry, ensure it's accessible from your cluster and correctly configured.
2. **`vela version` shows "Core Version: Not installed" after `vela install`:**
    - **Issue:** The KubeVela control plane pods might still be starting, or they failed to start for some reason.
    - **Troubleshooting:**
        - Run `kubectl get pods -n vela-system` repeatedly to check the status of all pods.
        - If pods are stuck in a `Pending` state, describe them (`kubectl describe pod <pod-name> -n vela-system`) to see why they aren't scheduling (e.g., resource constraints, taint/toleration issues).
        - If pods are in `CrashLoopBackOff`, check their logs (`kubectl logs <pod-name> -n vela-system`) for runtime errors.

3. **`command not found: vela`:**
    - **Issue:** The `velacli` executable was not installed correctly, or its installation path is not included in your system's `PATH` environment variable.
    - **Troubleshooting:**
        - Re-run the `brew install kubevela` or manual installation command carefully, paying attention to any error messages.
        - Check your shell's `PATH` variable (`echo $PATH`). The `vela` executable should reside in one of the directories listed. If not, you might need to manually add the installation directory (often `/usr/local/bin` for Homebrew or `~/.kubevela/bin` for manual installs) to your `PATH` configuration (e.g., in `.bashrc` or `.zshrc`).

Summary

In this chapter, we've laid the groundwork for understanding KubeVela as a powerful application delivery control plane for Kubernetes. We explored the inherent complexities of raw Kubernetes and how KubeVela provides an elegant solution.

Here are the key takeaways:

- **Kubernetes Complexity:** Raw Kubernetes presents significant challenges for platform teams and developers due to its low-level abstractions and verbose configurations, leading to operational overhead and cognitive load.
- **KubeVela's Role:** It acts as an application-centric control plane, simplifying application deployment and management across hybrid/multi-cloud environments by offering a higher-level abstraction.
- **OAM Foundation:** The Open Application Model (OAM) is KubeVela's conceptual core, separating concerns between platform engineers (defining capabilities) and developers (consuming them via high-level `Application` manifests).
- **Core Concepts:** `Application`, `Components`, `Traits`, `Policies`, `Workflows`, and `Addons` work together to define and orchestrate the entire application lifecycle in a modular and extensible way.

- **Key Advantages:** KubeVela offers higher abstraction, standardized delivery, extensibility, and reduced cognitive load compared to raw Kubernetes, while complementing tools like Helm and Argo CD. It serves as an excellent framework for building custom internal developer platforms without reinventing the wheel.
- **Installation:** We successfully installed the `velacli` and the KubeVela control plane into your Kubernetes cluster, setting the stage for hands-on application delivery.

In the next chapter, we'll dive deeper into the Open Application Model by defining our first `ComponentDefinition` and `TraitDefinition`. This will empower you to extend KubeVela's capabilities and build your own platform abstractions tailored to your organization's needs. Get ready to start coding!

References

- KubeVela Official Documentation
- Open Application Model (OAM) Specification
- KubeVela GitHub Repository
- KubeVela Addons
- KubeVela CLI (Velacli) Installation Guide

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 02

Setting Up KubeVela and Unpacking the Open Application Model (OAM)

Introduction: From Kubernetes Complexity to Application Simplicity

When managing applications on Kubernetes, platform teams often face a dilemma: how do you give developers the power of Kubernetes without overwhelming them with its inherent complexity? Raw Kubernetes manifests can be verbose, Helm charts offer packaging but often require deep Kubernetes knowledge to customize, and building a custom internal platform is a massive undertaking. This is where KubeVela steps in, offering an application delivery control plane that abstracts away the underlying infrastructure details.

In this chapter, we'll shift from the "why" of KubeVela to the "how." You'll get your hands dirty by setting up KubeVela in your Kubernetes cluster. More importantly, we'll dive deep into the heart of KubeVela: the Open Application Model (OAM). Understanding OAM is crucial because it's the language KubeVela uses to describe and deliver applications, providing a clean separation of concerns between platform engineers and application developers.

By the end of this chapter, you'll have KubeVela running and a solid grasp of OAM's core concepts. This foundational knowledge will empower you to define and manage applications in an application-centric way, paving the path for truly scalable and maintainable delivery pipelines.

Prerequisites

Before we begin, please ensure you have:

- A running Kubernetes cluster (any version compatible with KubeVela, typically `v1.23+`).
- `kubectl` configured to access your cluster. We'll assume `kubectl` version `v1.32.x` for this guide, compatible with recent Kubernetes releases.

Setting Up KubeVela: Your Application Control Plane

KubeVela transforms your Kubernetes cluster into an application delivery hub. It provides an extensible framework that allows platform engineers to define the capabilities and guardrails for application deployment, which developers then consume through a simplified, application-centric API.

Installation

Installing KubeVela is straightforward. We'll use the `vela` command-line tool, which provides a convenient way to interact with KubeVela. As of 2026-06-22, the latest stable release of KubeVela is `v1.11.0`.

First, let's install the `vela` CLI:

```
# Download the vela CLI for your OS (example for Linux AMD64)
curl -fsSL https://kubevela.io/script/install.sh | bash

# Verify installation
vela version
```

You should see output similar to this (versions might vary slightly):

```
# Example output for 'vela version'
CLI Version: 1.11.0
Core Version: N/A
```

The `Core Version` is `N/A` because we haven't installed the KubeVela controller into your cluster yet. Let's do that now.

```
# Install KubeVela into your Kubernetes cluster
vela install --version v1.11.0

# Check the installation status
vela status -n vela-system
```

The `vela install` command deploys the KubeVela control plane components into the `vela-system` namespace. This includes the KubeVela controller, which is responsible for reconciling `Application` resources and other OAM definitions.

`vela status` will show you the progress. It might take a few minutes for all components to become ready. Once ready, you should see something like:

```
# Example output for 'vela status'
NAMESPACE      NAME                          STATUS   HEALTHY  REASON
```

```
vela-system    kubevela-cluster-gateway     Running  True
vela-system    kubevela-core                Running  True
vela-system    kubevela-workflow            Running  True
...
```

Congratulations! Your Kubernetes cluster is now powered by KubeVela.

Unpacking the Open Application Model (OAM)

At the heart of KubeVela is the Open Application Model (OAM). OAM is a specification that aims to make application development and operations easier by separating concerns. Think of it as a blueprint for building and running applications on cloud-native platforms.

What is OAM and Why Does It Exist?

📌 **Key Idea:** OAM separates the "what" (application definition by developers) from the "how" (operational capabilities provided by platform engineers).

Before OAM, developers often had to deal with low-level Kubernetes details like `Deployment` replicas, `Service` ports, `Ingress` rules, and `HorizontalPodAutoscaler` configurations. This created a steep learning curve and made it hard to enforce consistent operational practices.

OAM solves this by introducing a clear contract:

- **For Developers:** Focus on defining their application's components and their desired operational characteristics (e.g., "I need 3 replicas," "I need an ingress," "I need to scale based on CPU"). They don't need to know the intricate Kubernetes YAML.
- **For Platform Engineers:** Focus on defining the capabilities and policies that fulfill those operational characteristics. They create reusable building blocks that developers can consume.

This separation promotes consistency, reduces developer toil, and allows platform teams to evolve the underlying infrastructure without impacting developers' application definitions.

The Pillars of OAM: Building Blocks for Applications

OAM defines several core concepts that work together to describe a complete application. Let's explore them:

1. Application

The `Application` is the top-level OAM resource in KubeVela. It represents a complete, deployable application. An `Application` aggregates all other OAM building blocks (Components, Traits, Policies, Workflows) to define how your software runs.

It's the single source of truth for your application's deployment, lifecycle, and operational behaviors.

2. Components

A `Component` defines a core building block of your application. This could be a microservice, a database, a message queue, or even a static website. A `Component` typically describes the workload type and its basic configuration.

For instance, a component might specify a Docker image to run, environment variables, and resource requests. It's the "what" of your application's pieces.

3. Traits

`Traits` are operational capabilities that you attach to Components. They define how a component should behave or be operated. Think of them as modifiers that add functionality without changing the component's core definition.

Examples of traits include:

- `scaler`: To define the number of replicas for a workload.
- `ingress`: To expose a service via HTTP/HTTPS.
- `autoscaler`: To enable automatic scaling based on metrics.
- `rollout`: To define deployment strategies like canary or blue/green.

Platform engineers define these traits, and developers simply declare which traits they need for their components.

4. Policies

`Policies` define guardrails and governance rules for your applications. They ensure compliance, security, and resource management across your deployments. Policies can apply to the entire application or specific components.

Examples of policies:

- `health-check`: To define application health probes.
- `topology`: To specify where components should be deployed (e.g., specific clusters, regions).
- `security`: To enforce network policies or image scanning.

Policies are often defined by platform engineers to enforce organizational standards.

5. Workflows

`Workflows` define the steps involved in delivering and managing an application. They orchestrate the entire deployment process, from building and testing to deploying and releasing. Workflows are highly customizable and can include manual approvals, data transformations, or complex multi-stage rollouts.

Examples of workflow steps:

- `deploy`: Deploy components to a cluster.
- `manual-approval`: Pause for human intervention.
- `data-patch`: Modify resources during deployment.
- `webhook`: Trigger external systems.

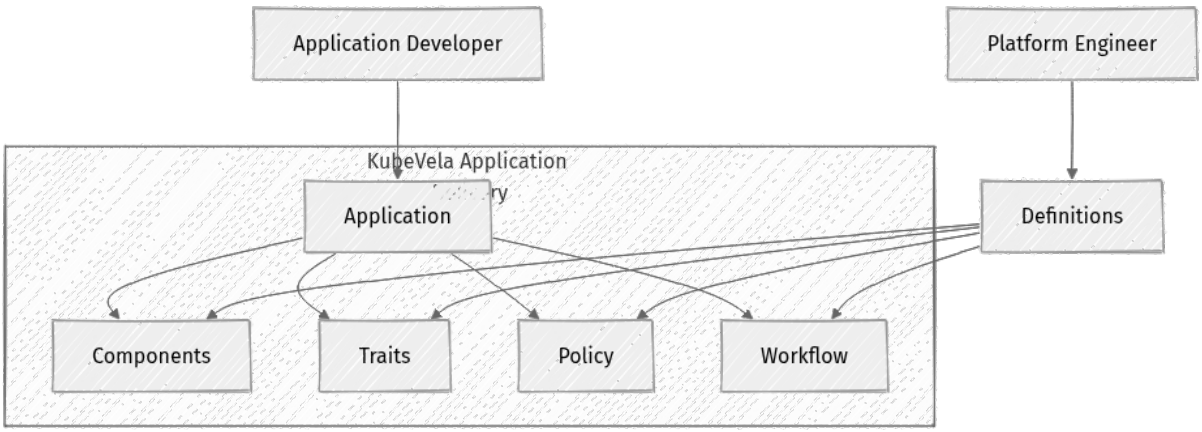
6. Addons

`Addons` extend KubeVela's capabilities by providing pre-packaged sets of components, traits, policies, and workflows. They allow KubeVela to integrate with other tools (like Argo CD, Flux, Prometheus, Grafana) or provide new application delivery features.

Addons are how KubeVela remains flexible and integrates with the broader cloud-native ecosystem.

OAM in Action: A Visual Flow

This diagram illustrates how these core OAM concepts come together within a KubeVela `Application`.



- **Developer** focuses on the `Application`, defining its `Components` and attaching desired `Traits`, `Policies`, and `Workflows`.

- **Platform Engineer** defines the underlying `Definitions` (Component, Trait, Policy, Workflow definitions) and enables `Addons`. These definitions are the "recipes" that KubeVela uses to fulfill the developer's requests.
- KubeVela then uses these definitions to deliver the application according to the specified `Workflows` and `Policies`.

Step-by-Step Implementation: Your First KubeVela Application

Let's put OAM into practice by deploying a simple "hello-world" web service using KubeVela. We'll start with a basic component and then add operational traits incrementally.

1. Define a Simple Component

Create a file named `my-first-app.yaml`. We'll start with just the `Application` and a single `Component`. This component will be a simple Nginx web server.

```yaml
# my-first-app.yaml
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-first-webapp
spec:
  components:
    - name: web-server
      type: webservice # This is a KubeVela built-in component type
      properties:
        image: nginx:1.25.3 # Using a specific Nginx version as of 2026-06-22
        port: 80
```

Let's break down this small YAML:

- `apiVersion: core.oam.dev/v1beta1` and `kind: Application`: This identifies our resource as a KubeVela `Application` using the OAM API.
- `metadata.name: my-first-webapp`: This gives our application a name.
- `spec.components`: This section lists the building blocks of our application.
- `- name: web-server`: This is our first component, named `web-server`.
- `type: webservice`: This tells KubeVela that our component is a `webservice`. KubeVela comes with several built-in component types, and `webservice` is a common one that typically abstracts a Kubernetes `Deployment` and `Service`.

- `properties`: These are the specific configurations for our `webservice` component.
  - `image: nginx:1.25.3`: Specifies the Docker image for our web server.
  - `port: 80`: The port our web server listens on.

Apply this application to your cluster:

```
kubectl apply -f my-first-app.yaml
```

Now, check the status of your KubeVela application:

```
vela status my-first-webapp
```

You should see output indicating the application is running and healthy:

```
# Example output for 'vela status my-first-webapp'
About:

  Name:         my-first-webapp
  Namespace:    default
  Created time: 2026-06-22 10:30:00 +0000 UTC
  Status:       running

Components:

  NAME        TYPE        HEALTHY STATUS  ...
  web-server  webservice  healthy         ...

Workflow:

  MODE        STATUS  STEP    MESSAGE
  DAG         succeed deploy  Workflow is finished.

Resources:

  (v1.Deployment)  web-server
  (v1.Service)     web-server
```

Notice that KubeVela automatically created a Kubernetes `Deployment` and a `Service` for our `webservice` component. We didn't have to write any raw Kubernetes YAML for them!

2. Add a Trait: Scaling Your Component

Our web server is running, but what if we need more replicas? This is where `Traits` shine. Let's add a `scaler` trait to specify 3 replicas.

Edit `my-first-app.yaml` and add the `traits` section under `web-server`:

```
# my-first-app.yaml (modified)
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-first-webapp
spec:
  components:
    - name: web-server
      type: webservice
      properties:
        image: nginx:1.25.3
        port: 80
      traits: # <--- New section for traits
        - type: scaler # <--- Using the built-in scaler trait
          properties:
            replicas: 3 # <--- Desired number of replicas
```

Apply the updated application:

```
kubectl apply -f my-first-app.yaml
```

Check the status again:

```
vela status my-first-webapp
```

You'll see the application reconcile, and if you inspect the underlying Kubernetes resources, you'll find the `Deployment` now has 3 replicas:

```
kubectl get deployment web-server
```

```
# Example output for 'kubectl get deployment web-server'
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
web-server   3/3     3            3           2m
```

See how easy that was? We added a single `trait` line, and KubeVela handled the underlying Kubernetes `Deployment` changes. The developer just asks for "3 replicas," and the platform (via KubeVela) knows how to deliver it.

3. Add Another Trait: Exposing Your Service with Ingress

Our web server is running with 3 replicas, but it's not accessible from outside the cluster yet. Let's add an `ingress` trait to expose it.

Edit `my-first-app.yaml` again and add another trait:

```yaml
# my-first-app.yaml (modified again)
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-first-webapp
spec:
  components:
    - name: web-server
      type: webservice
      properties:
        image: nginx:1.25.3
        port: 80
      traits:
        - type: scaler
          properties:
            replicas: 3
        - type: ingress # <--- New trait for exposing the service
          properties:
            domain: my-first-app.example.com # <--- Your desired domain
            http:
              "/": 80 # <--- Route all requests to port 80 of the component
```

⚡ **Quick Note:** For this `ingress` trait to fully function, you'll need an Ingress Controller (like Nginx Ingress Controller or Traefik) installed in your cluster. KubeVela will create the `Ingress` resource, but the controller is what makes it work. Also, `my-first-app.example.com` needs to resolve to your Ingress Controller's external IP. For local testing, you might need to modify your `/etc/hosts` file.

Apply the updated application:

```
kubectl apply -f my-first-app.yaml
```

Check the status one last time:

```
vela status my-first-webapp
```

You'll now see an `Ingress` resource listed under `Resources`:

```
# Example output for 'vela status my-first-webapp'
...
Resources:

  (v1.Deployment)  web-server
  (v1.Service)     web-server
  (v1.Ingress)     web-server-ingress # <--- New Ingress resource
```

You've successfully deployed a scalable, externally accessible web application with just a few lines of KubeVela YAML, leveraging its application-centric OAM model!

Mini-Challenge: Add a Health Check Policy

You've seen how to add traits. Now, let's try adding a `Policy` to ensure our application is robust.

Challenge: Modify your `my-first-app.yaml` to include a `health-check` policy. This policy should define a readiness probe for your `web-server` component, checking the `/` path on port `80` every 5 seconds, with an initial delay of 5 seconds.

Hint:

- You'll need to add a `policies` section at the top level of your `Application` spec, similar to `components`.
- Look for the `health-check` policy type and its properties for defining probes. KubeVela's built-in `health-check` policy usually maps to Kubernetes' `readinessProbe` and `livenessProbe`.

What to observe/learn: How policies are defined at the application level and how they influence the operational behavior of components without modifying the component's core definition. After applying, check `vela status my-first-webapp` and potentially `kubectl describe deployment web-server` to see the added probe.

Common Pitfalls & Troubleshooting

Even with KubeVela's simplicity, you might encounter issues. Here are a few common ones:

1. **KubeVela Controller Not Running**: If `vela status` shows `N/A` for `Core Version` or components in `vela-system` are not healthy, the KubeVela controller might not be running correctly.
    - **Solution**: Check `kubectl get pods -n vela-system` for any failing pods. Use `kubectl logs <pod-name> -n vela-system` to inspect logs for errors. Ensure your cluster has enough resources.

2. **Application Not Reconciling**: If `vela status my-first-webapp` shows errors or hangs in a pending state.
    - **Solution**: Double-check your YAML syntax. Even a small indentation error can cause issues. KubeVela's controller logs (from the `kubevela-core` pod in `vela-system`) can provide detailed error messages.
3. **Trait/Component Type Not Found**: If you get an error like "component type 'webservice' not found" or "trait type 'scaler' not found."
    - **Solution**: Ensure KubeVela is correctly installed and its definitions are loaded. You can list available definitions with `vela def ls`. If you're using a custom definition, make sure it's applied to the cluster.
4. **Ingress Not Working**: You've applied the `ingress` trait, but you can't access your application from your browser.
    - **Solution**: Verify an Ingress Controller is running in your cluster. Check the `Ingress` resource created by KubeVela (`kubectl get ingress web-server-ingress`) for its IP/Hostname. Ensure your DNS or `/etc/hosts` file correctly points to the Ingress Controller's external IP.

Summary

In this chapter, you've taken a significant step into the world of application delivery with KubeVela.

Here are the key takeaways:

- **KubeVela Installation**: You successfully installed the `vela` CLI and the KubeVela control plane (`v1.11.0`) into your Kubernetes cluster.
- **Open Application Model (OAM)**: You learned that OAM is the foundational specification for KubeVela, designed to separate concerns between developers and platform engineers.
- **OAM's Core Concepts**: You explored the roles of `Application`, `Component`, `Trait`, `Policy`, `Workflow`, and `Addon` in defining and delivering applications.
- **Hands-on Application Deployment**: You deployed a simple web application using KubeVela, incrementally adding `scaler` and `ingress` traits to modify its operational behavior without touching raw Kubernetes manifests.
- **Troubleshooting**: You're now aware of common pitfalls and how to approach debugging KubeVela applications.

You've built a solid foundation for understanding KubeVela's application-centric approach. In the next chapter, we'll dive deeper into how platform engineers can extend KubeVela by creating custom `ComponentDefinitions` and `TraitDefinitions`, unlocking the true power of KubeVela's extensibility.

References

- KubeVela Official Documentation
- Open Application Model (OAM) Specification
- KubeVela GitHub Repository
- KubeVela Component Definitions
- KubeVela Trait Definitions

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 03

Your First Application: Defining Components with KubeVela

Imagine you're a developer, eager to get your new microservice running on Kubernetes. You've written the code, but now you face the dreaded YAML jungle: Deployments, Services, Ingresses, ConfigMaps, Secrets – the list goes on. What if you could define your application in a much simpler, application-centric way?

Welcome back! In our previous discussions, we explored why KubeVela acts as an essential application delivery control plane for Kubernetes, simplifying complex deployments for platform teams and developers alike. We understood its role in abstracting away Kubernetes complexities and providing an application-centric view.

Now, it's time to bridge theory with practice. In this chapter, you'll learn how to define and deploy your very first application using KubeVela, focusing on its fundamental building block: the **Component**. By the end, you'll have a running web service in your Kubernetes cluster, managed by KubeVela, and a clear understanding of how it simplifies your deployment workflow.

The KubeVela Component: Your Application's Building Block

At the heart of the Open Application Model (OAM) and KubeVela lies the concept of a **Component**. Think of a Component as a modular, self-contained unit that represents a distinct piece of your application. This could be a microservice, a database, a message queue worker, or even a static file server.

📌 `Key Idea:` A KubeVela Component is an abstraction layer that empowers developers to declare what their application needs (e.g., "I need a web service running this Docker image") without getting bogged down in the low-level Kubernetes details like Deployments, Services, or Pods. It's the "what" of your application, not the "how."

Why Components? The Power of Abstraction

Before KubeVela, deploying even a simple web service on Kubernetes often involved writing several verbose YAML manifests: a `Deployment` for the pods, a `Service` for network access, and perhaps an `Ingress` for external routing. This quickly becomes a burden, especially for application developers whose primary focus is their code, not infrastructure boilerplate.

KubeVela Components solve this challenge by introducing a powerful level of abstraction:

- **Developer Focus:** Developers specify their application's requirements in terms of high-level component types (like `webservice` or `worker`), using simple, intuitive properties such as `image` and `port`. They don't need to know the intricacies of Kubernetes API objects.
- **Platform Engineer Control:** Platform engineers define `ComponentDefinition`s (a concept we'll explore in later chapters), which map these high-level component types to specific Kubernetes resources and operational best practices. This separation of concerns ensures consistency, compliance, and security across all applications.
- **Reduced Boilerplate:** Developers write significantly less YAML. Instead of managing multiple Kubernetes manifests, they interact with a single, application-centric definition. This dramatically reduces the cognitive load and potential for errors.

Consider that simple web service. With raw Kubernetes, you'd define a `Deployment` and a `Service`. With KubeVela, you define a single `webservice` component, and KubeVela automatically provisions and manages the necessary Kubernetes resources for you. This is a game-changer for developer productivity.

Built-in Component Types for Immediate Use

KubeVela comes equipped with several useful built-in component types, ready for immediate use. These cover common application patterns:

- **`webservice`**: Ideal for stateless web applications that require HTTP/HTTPS access. When you declare a `webservice` component, KubeVela typically provisions a Kubernetes `Deployment` to manage your application pods and a `Service` to expose it internally.
- **`worker`**: Designed for background services, batch jobs, or long-running processes that don't need direct HTTP exposure. Depending on its configuration, it might provision a `Deployment` or a `Job`.

- `raw`: For scenarios where you need to directly specify a raw Kubernetes resource (e.g., a custom CRD, a specific `StatefulSet`) as a component. This offers maximum flexibility when you need to drop down to the Kubernetes API level.

We'll start our hands-on journey with the `webservice` component, as it's a common pattern and an excellent way to grasp KubeVela's core abstraction.

Step-by-Step Implementation: Deploying Your First Component

Let's get our hands dirty and deploy a simple web service using KubeVela. This will solidify your understanding of components in action.

1. Prerequisites Check

Before proceeding, ensure you have the following:

- **A running Kubernetes cluster:** This could be a local cluster (like Kind or Minikube) or a cloud-managed Kubernetes service (e.g., GKE, EKS, AKS).
- **`kubectl` configured:** Your `kubectl` command-line tool must be configured to access your Kubernetes cluster. You can verify this by running:

```
kubectl cluster-info
```

```
You should see information about your cluster, indicating a successful
connection.
```

2. Install the KubeVela CLI (Vela CLI)

The `vela` CLI is your primary interface for interacting with KubeVela. It allows you to install KubeVela, deploy applications, check their status, and manage component definitions.

As of **2026-06-22**, we'll install `vela` CLI version `1.12.0`. Please always check the official KubeVela documentation for the absolute latest stable release if `1.12.0` is outdated.

```
# For Linux/macOS users:
curl -fsSL https://kubevela.io/script/install.sh | bash

# For Windows users (using PowerShell):
```

```
# Invoke-WebRequest -Uri https://kubevela.io/script/install.ps1 -OutFile
install.ps1; .\install.ps1; Remove-Item install.ps1
```

After installation, it's good practice to verify that the `vela` CLI is correctly installed and accessible in your path:

```
vela version
```

You should see output similar to this, confirming the CLI version:

```
CLI Version: v1.12.0 # (Or the specific version you installed)
GitCommit:   a1b2c3d
...
```

3. Install KubeVela into Your Cluster

Now that you have the `vela` CLI, let's install the KubeVela control plane into your Kubernetes cluster. This process deploys the core KubeVela components (like the controller and webhook) into a dedicated namespace, typically `vela-system`.

```
vela install --version v1.12.0 # Assuming v1.12.0 is the current stable
version as of 2026-06-22
```

The installation might take a few minutes, as KubeVela sets up its custom resource definitions (CRDs) and deploys its controller. Once the command completes, you can verify that KubeVela's components are running:

```
kubectl get pods -n vela-system
```

You should observe several pods in the `Running` state, including `vela-core`, `vela-cluster-gateway`, and potentially others, indicating a healthy KubeVela control plane.

4. Defining Your First Application

In KubeVela, applications are defined using the `Application` Custom Resource. This is the top-level object that acts as the blueprint for your entire application, orchestrating its components, traits, policies, and workflows.

Let's create a file named `my-first-app.yaml` with the following content. This YAML defines a simple web service.

```
# my-first-app.yaml
apiVersion: core.oam.dev/v1beta1
```

```
kind: Application
metadata:
  name: my-first-webapp
spec:
  components:
    - name: welcome-service
      type: webservice
      properties:
        image: oamdev/hello-world
        port: 8000
```

Let's dissect this YAML manifest line by line to understand what each part means:

- **`apiVersion: core.oam.dev/v1beta1`**: This line specifies the API version for the `Application` resource. KubeVela leverages the Open Application Model (OAM) specification, which defines these core APIs.
- **`kind: Application`**: This crucial line tells Kubernetes that we are defining an `Application` resource, which KubeVela's controller is designed to understand and manage.
- **`metadata:`**: This standard Kubernetes section holds metadata about our resource.
    - **`name: my-first-webapp`**: This is the unique name of our KubeVela application within the cluster.

- `spec:`: This is where the core definition of our application's desired state resides.
    - `components:`: This is a list where we define all the individual components that comprise our application. In this simple example, we have just one.
        - `- name: welcome-service`: This is the logical name for our specific component within the `my-first-webapp` application.
        - `type: webservice`: This is a key declaration! We're telling KubeVela that this component is of the `webservice` type. KubeVela, through its pre-defined `ComponentDefinition`s, knows how to translate a `webservice` into concrete Kubernetes resources like a `Deployment` and a `Service`.
        - `properties:`: These are the specific configuration parameters that are relevant to our `webservice` component type.
            - `image: oamdev/hello-world`: This specifies the Docker image that our web service will run. `oamdev/hello-world` is a lightweight Go application that serves a simple "Hello KubeVela" message.
            - `port: 8000`: This defines the port on which our web service inside the container will listen. KubeVela will configure the Kubernetes `Service` to expose this port.

5. Deploying the Application

With our `my-first-app.yaml` file prepared, it's time to deploy it using the `vela` CLI:

```
vela up -f my-first-app.yaml
```

You'll see output indicating the application is being deployed. The `vela up` command is KubeVela's intelligent way to apply or update an application definition. It figures out what underlying Kubernetes resources need to be created, updated, or deleted based on your `Application` manifest.

6. Verifying the Deployment

Once deployed, KubeVela will spring into action, provisioning the underlying Kubernetes resources. You can check the status of your KubeVela application to observe its progress:

```
vela status my-first-webapp
```

You should see output similar to this (it might take a few moments for all resources to become ready):

```
About this application:

  Name:         my-first-webapp
  Namespace:    default
  Created time: 2026-06-22 10:00:00 +0000 UTC
  Updated time: 2026-06-22 10:00:05 +0000 UTC

Components:

  - Name: welcome-service
    Type: webservice
    Health Status: Healthy
    Traits:
    Resources:
      - Kind: Deployment
        Name: welcome-service
      - Kind: Service
        Name: welcome-service
```

Notice the power of KubeVela here: it explicitly tells you that it created a `Deployment` and a `Service` for our single `webservice` component! This is the abstraction in action.

You can also use `kubectl` to directly inspect the underlying Kubernetes resources that KubeVela created:

```
kubectl get deployment,service -l app.oam.dev/name=my-first-webapp
```

You should find a `Deployment` and a `Service`, both named `welcome-service`, demonstrating KubeVela's orchestration.

To access your newly deployed web service, you can use `kubectl port-forward`:

```
kubectl port-forward service/welcome-service 8080:8000
```

Now, open your web browser and navigate to `<http://localhost:8080` >. You should proudly see the "Hello KubeVela!" message.

⚡ Real-world insight: This simple `webservice` component effectively replaces at least two distinct Kubernetes manifests (`Deployment` and `Service`), abstracting away the boilerplate. For developers, this means less YAML to write, fewer Kubernetes concepts to master, and a clearer focus on their application's logic.

Mini-Challenge: Customize Your Web Service

Now that you've successfully deployed your first KubeVela application, let's make a small change to solidify your understanding of updates.

Challenge: Modify the `my-first-app.yaml` file to use a different Docker image and expose it on a different port.

- Try using `nginxdemos/hello` as the image. This is a simple Nginx server that serves "Hello Nginx!".
- Change the `port` to `80`.

Hint: You only need to modify the `properties` section of your `welcome-service` component.

Here's how your modified `my-first-app.yaml` should look:

```
# my-first-app.yaml (modified)
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-first-webapp
spec:
  components:
    - name: welcome-service
      type: webservice
      properties:
        image: nginxdemos/hello # Changed image
        port: 80              # Changed port
```

After modifying the file, apply the changes using the `vela` CLI, just like you did for the initial deployment:

```
vela up -f my-first-app.yaml
```

What to observe/learn:

1. How quickly KubeVela detects the changes and initiates an update.

2. After the update, check `vela status my-first-webapp` again. Does it show `Healthy`?
3. Port-forward to the new port (`kubectl port-forward service/welcome-service 8080:80`).
4. Visit `<http://localhost:8080>` in your browser. Do you see "Hello Nginx!"?

This exercise demonstrates the agility of KubeVela's application-centric approach. You made a simple change to your high-level application definition, and KubeVela intelligently handled the underlying Kubernetes resource updates (e.g., performing a rolling update of the `Deployment`) for you.

Common Pitfalls & Troubleshooting

Even with KubeVela's streamlined approach, you might encounter issues. Here are a few common mistakes and how to debug them:

- **YAML Syntax Errors:** KubeVela, like Kubernetes, is very strict about YAML formatting. A small indentation error, a misplaced dash, or a missing colon can cause parsing failures.
    - **Solution:** Use a YAML linter (many IDEs like VS Code have excellent YAML extensions) or carefully review your file for syntax. The `vela up` command will usually provide helpful error messages indicating the line number where the issue occurred.
- **Image Pull Failures:** If your specified Docker image doesn't exist, is misspelled, or requires authentication (and credentials aren't provided), your pods won't start successfully.
    - **Solution:** Check `kubectl describe pod <pod-name>` for events related to `ImagePullBackOff` or `ErrImagePull`. Ensure the image name is correct and accessible from your cluster.
- **`vela status` shows `Running` but `Health Status` is `Unhealthy`:** This indicates that while KubeVela has successfully deployed the underlying Kubernetes resources, the application inside your pods isn't ready or healthy according to its defined health checks.
    - **Solution:** Use `kubectl logs <pod-name>` to check your application's logs for any runtime errors. Additionally, `vela status my-first-webapp --debug` can provide more verbose insights into the component's health status and any associated issues.

- **`vela up` fails with connection errors:** This typically means your `kubectl` context isn't correctly pointing to a running Kubernetes cluster, or KubeVela itself isn't fully installed or healthy in the `vela-system` namespace.
    - **Solution:** Verify your cluster connection with `kubectl cluster-info` and check the health of the KubeVela control plane with `kubectl get pods -n vela-system`.

⚠️ `What can go wrong:` Always remember that KubeVela builds on top of Kubernetes. If your underlying Kubernetes cluster has networking issues, resource constraints, or misconfigured storage, KubeVela applications will also be affected. KubeVela simplifies the definition and delivery but doesn't magically fix fundamental cluster problems.

Summary

Congratulations! You've successfully deployed your first application using KubeVela and truly experienced its core abstraction. Here are the key takeaways from this chapter:

- **Components are the core:** They are the fundamental building blocks of your application in KubeVela, abstracting away low-level Kubernetes details like Deployments and Services.
- **The `Application` CRD:** This is KubeVela's top-level resource, serving as the single source of truth for defining your entire application, including its components.
- **`webservice` component type:** A powerful built-in component that allows developers to quickly define stateless web applications with minimal configuration.
- **The `vela` CLI:** Your essential command-line companion for installing KubeVela, deploying applications, and efficiently checking their status.
- **Simplified Application Delivery:** KubeVela significantly reduces the complexity of deploying and managing applications by providing an application-centric abstraction, empowering developers to focus on their code.

You've seen how KubeVela empowers developers to focus on their application code and configuration, while the platform handles the underlying infrastructure intricacies. This separation of concerns is crucial for modern, scalable application delivery.

In the next chapter, we'll dive deeper into enhancing these components by attaching **Traits**. Traits allow you to add operational capabilities like scaling, ingress exposure, and more, without modifying the component's core definition, further extending KubeVela's flexibility and power.

References

- KubeVela Official Documentation
- KubeVela CLI Installation Guide
- KubeVela Application Definition
- Open Application Model (OAM) Specification

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 04

Adding Capabilities: Scaling and Exposing Applications with Traits

Introduction

In the previous chapter, we learned how to define the fundamental building blocks of our applications using KubeVela Components. We created a simple backend service, but it was just a raw deployment. In the real world, applications need more than just a running container; they need operational capabilities like scaling, network exposure, health checks, and more.

This chapter dives into KubeVela **Traits**, the powerful mechanism that allows platform engineers to embed these operational capabilities directly into the Application definition. We'll explore what Traits are, why they are essential for abstracting Kubernetes complexity, and how they empower developers to declare desired operational behaviors without becoming Kubernetes experts. By the end of this chapter, you'll be able to scale your applications and expose them to the network using KubeVela Traits.

Core Concepts: KubeVela Traits

Imagine you're building a house. The **Component** is like the foundational structure – the walls, roof, and floors. But a house needs more to be livable: plumbing for water, electricity for power, and a front door to welcome guests. In KubeVela, these additional features and operational aspects are handled by **Traits**.

What are Traits?

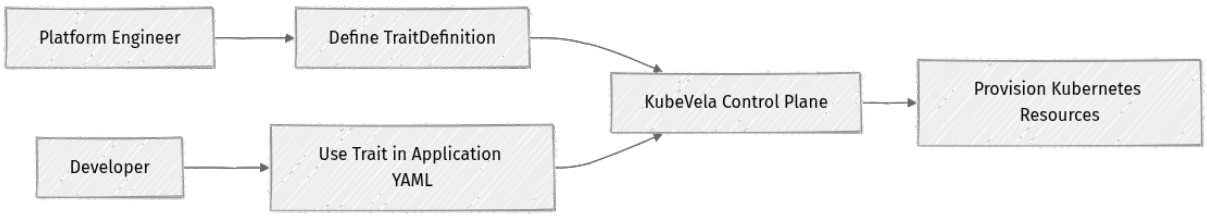
A **Trait** in KubeVela is an operational capability that can be attached to a Component. It describes how a Component should behave or what operational features it should have. Traits are a core part of the Open Application Model (OAM), designed to separate the concerns of application developers from those of platform operators.

- **What it is:** A declarative specification for an operational aspect of a component.

- **Why it exists:** To abstract away the complex, low-level details of Kubernetes resources (like `HorizontalPodAutoscaler`, `Service`, `Ingress`) and present them as simple, high-level parameters to developers.
- **What problem it solves:** It frees developers from needing deep Kubernetes knowledge for common operational tasks, allowing them to focus on application logic. Platform engineers, in turn, can define and standardize these capabilities.

The OAM Philosophy of Traits: Separation of Concerns

The brilliance of OAM, and by extension KubeVela, lies in its clear separation of responsibilities:



- **Platform Engineers:** They define `TraitDefinition`s. These definitions specify what a trait does, what parameters it accepts, and how it translates into underlying Kubernetes resources. This allows them to curate a catalog of reusable operational capabilities.
- **Developers:** They consume these pre-defined traits by simply listing them in their `Application` YAML and providing the necessary parameters. They don't need to know the intricate details of the Kubernetes resources being provisioned.

This separation ensures consistency, reduces errors, and speeds up application delivery.

Common Trait Examples

KubeVela comes with a rich set of built-in traits, and platform engineers can extend this by defining custom traits. Let's look at a couple of fundamental ones we'll use today. (Note: These are standard traits available in KubeVela `v1.11.0` as of 2026-06-22).

- **`scaler` Trait:** This trait allows you to define the number of replicas for your component. It directly influences the `replicas` field of a Kubernetes `Deployment` or can even integrate with a `HorizontalPodAutoscaler` for dynamic scaling.

- `ingress` **Trait:** This trait is used to expose HTTP/HTTPS services to the outside world, typically by creating Kubernetes `Service` and `Ingress` resources. It handles routing external traffic to your application component.
- `service-expose` **Trait:** For internal service exposure within the cluster, this trait creates a Kubernetes `Service` resource, making your component discoverable by other applications inside the cluster.

Deep Dive: The `scaler` Trait

The `scaler` trait is your go-to for controlling how many instances (replicas) of your application component should be running.

When you add a `scaler` trait with `replicas: 3` to a component, KubeVela will ensure that the underlying Kubernetes Deployment for that component maintains three pods. If one pod fails, Kubernetes will automatically start another to maintain the desired count.

How it works: The `scaler` trait primarily manipulates the `replicas` field of the Kubernetes `Deployment` or `StatefulSet` that KubeVela creates for your component. In more advanced scenarios, a `scaler` trait definition could also be configured to create a `HorizontalPodAutoscaler` (HPA) resource, allowing for automatic scaling based on CPU or memory utilization.

Deep Dive: The `ingress` Trait

To make your web application accessible from outside your Kubernetes cluster, you need an Ingress. The `ingress` trait simplifies this process dramatically.

Instead of writing complex Kubernetes `Service` and `Ingress` YAML, you just tell KubeVela the host, path, and port you want to expose. KubeVela then generates and manages the necessary Kubernetes resources for you.

How it works: When you apply an `ingress` trait, KubeVela typically performs two main actions:

1. **Creates a Kubernetes `Service`:** This internal service provides a stable network endpoint for your component's pods within the cluster.
2. **Creates a Kubernetes `Ingress`:** This resource defines how external HTTP/HTTPS traffic should be routed to the `Service` created in step 1, based on rules like hostnames and URL paths.

⚠️ **What can go wrong:** For the `ingress` trait to work, your Kubernetes cluster must have an Ingress controller (like Nginx Ingress, Traefik, or Istio Gateway) installed and running. Without it, the `Ingress` resource will be created but will not actually route traffic.

Step-by-Step Implementation: Scaling a Component

Let's take our simple `my-web-app` component from the previous chapter and add the ability to scale it.

First, let's look at our base application (assuming you have a similar `my-app.yaml` from the previous chapter).

```
# my-app.yaml (Initial version from previous chapter)
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-web-app
spec:
  components:
    - name: backend-service
      type: webservice
      properties:
        image: crccheck/hello-world
        port: 8000
```

Now, let's add the `scaler` trait to our `backend-service` component. We'll set the `replicas` to `3`.

1. **Modify your `my-app.yaml`:** Open `my-app.yaml` and locate the `backend-service` component. Add a `traits` section under the `components.backend-service` block, as shown below.

```
    # my-app.yaml (with scaler trait)
    apiVersion: core.oam.dev/v1beta1
    kind: Application
    metadata:
      name: my-web-app
    spec:
      components:
        - name: backend-service
          type: webservice
          properties:
            image: crccheck/hello-world
            port: 8000
          # --- Start of new addition ---
          traits:
            - type: scaler
              properties:
                replicas: 3
```

```
          # --- End of new addition ---
```

```
**Explanation:**
- `traits:`: This new section within the `backend-service` component is where
we declare all the operational capabilities for this component.
- `- type: scaler`: We're adding a trait of type `scaler`. KubeVela recognizes
this type and knows how to handle it.
- `properties:`: These are the parameters specific to the `scaler` trait.
- `replicas: 3`: We're telling KubeVela that we want 3 instances of our
`backend-service` running.
```

1. **Apply the updated Application:** Save the `my-app.yaml` file and apply it to your Kubernetes cluster using `kubectl`.

```
    kubectl apply -f my-app.yaml
```

```
You should see output similar to `application.core.oam.dev/my-web-app
configured`.
```

1. **Observe the changes:** Now, let's check if KubeVela has correctly scaled our application. We can inspect the underlying Kubernetes Deployment.

```
    kubectl get deployment backend-service -o wide
```

```
You should see that the `READY` and `AVAILABLE` columns show `3/3`, confirming
that three replicas are running.
```

```
    NAME              READY   UP-TO-DATE   AVAILABLE   AGE
CONTAINERS        IMAGES                      SELECTOR
    backend-service   3/3     3            3           2m      backend-
service   crccheck/hello-world       app.oam.dev/component=backend-service
```

```
You can also get a high-level view of your KubeVela application:
```

```
    vela ls
```

```
This will show your `my-web-app` and its components and traits.
```

Step-by-Step Implementation: Exposing a Component with Ingress

Now that our application can scale, let's make it accessible from outside the cluster using the `ingress` trait. We'll expose it on a specific hostname and path.

1. **Modify your `my-app.yaml` again:** Add another trait, this time the `ingress` trait, to your `backend-service` component. We'll use `my-app.example.com` as the hostname and `/` as the path. You'll also need to ensure the `port` matches the one your application listens on (which is `8000` in our `webservice` component properties).

```
    # my-app.yaml (with scaler and ingress traits)
    apiVersion: core.oam.dev/v1beta1
    kind: Application
    metadata:
      name: my-web-app
    spec:
      components:
        - name: backend-service
          type: webservice
          properties:
            image: crccheck/hello-world
            port: 8000
          traits:
            - type: scaler
              properties:
                replicas: 3
            # --- Start of new addition ---
            - type: ingress
              properties:
                domain: my-app.example.com
                http:
                  "/": 8000 # Map root path to component's port 8000
            # --- End of new addition ---
```

```
**Explanation:**
- `- type: ingress`: We're adding the `ingress` trait.
- `properties:`: Parameters for the ingress.
- `domain: my-app.example.com`: The hostname under which our application will
be accessible.
- `http:`: Defines HTTP routing rules.
- `"/": 8000`: This rule says that any request to the root path (`/`) on `my-
app.example.com` should be routed to port `8000` of our `backend-service`
component.
```

1. **Apply the updated Application:** Save the `my-app.yaml` file and apply it.

```
    kubectl apply -f my-app.yaml
```

1. **Observe the changes:** KubeVela will now create an `Ingress` resource. You can check for it:

```
    kubectl get ingress
```

```
You should see an Ingress resource named something like `backend-service-
ingress`.
```

```
    NAME                      CLASS    HOSTS                 ADDRESS
PORTS   AGE
    backend-service-ingress   <none>   my-app.example.com    <pending>
80      1m
```

```
The `ADDRESS` column might show `<pending>` or an IP address depending on your
Ingress controller setup.

**To test connectivity:**
You'll need to resolve `my-app.example.com` to the IP address of your Ingress
controller.
- If you're running locally (e.g., Minikube, Docker Desktop), you might get an
IP from `minikube ip` or your cluster's external IP.
- You can modify your local `/etc/hosts` file (or `C:
\Windows\System32\drivers\etc\hosts` on Windows) to map `my-app.example.com`
to your Ingress controller's IP. For example:
```

```
        <INGRESS_CONTROLLER_IP> my-app.example.com
```

```
- Once the hostname resolves, you can use `curl`:
```

```
        curl http://my-app.example.com
```

```
    You should receive the "Hello World" response from your application!
```

Mini-Challenge

You've successfully scaled and exposed your application externally. Now, let's imagine another internal service needs to communicate with `backend-service` without going through the public internet. This is where the `service-expose` trait comes in handy.

Challenge: Add a `service-expose` trait to your `backend-service` component. Configure it to expose your component on port `8000` (which is its internal listening port).

Hint: You can inspect the `service-expose` trait definition using `vela def show service-expose` to understand its parameters. You'll likely need to specify the component's port.

What to observe/learn: After applying the updated application, check for a new Kubernetes `Service` resource that KubeVela creates. Note its name and cluster IP. This service will allow other pods within your Kubernetes cluster to reach your `backend-service` using its internal DNS name (e.g., `backend-service.default.svc.cluster.local`).

Common Pitfalls & Troubleshooting

1. **Ingress Trait Not Working (Address Pending):**
    - **Symptom:** `kubectl get ingress` shows `<pending>` under the `ADDRESS` column, and you can't access your application externally.
    - **Cause:** Your Kubernetes cluster likely doesn't have an Ingress controller installed (e.g., Nginx Ingress Controller, Traefik, Istio). KubeVela creates the `Ingress` resource, but nothing is listening for it.
    - **Solution:** Install an Ingress controller in your cluster. For example, for Nginx Ingress Controller: `helm upgrade --install ingress-nginx ingress-nginx --repo <https://kubernetes.github.io/ingress-nginx> --namespace ingress-nginx --create-namespace`.
2. **Incorrect Trait Parameters:**
    - **Symptom:** Application fails to deploy, or the trait doesn't have the desired effect. KubeVela reports validation errors.
    - **Cause:** Typos in trait names, incorrect parameter names, or invalid values for trait properties. For example, specifying a non-existent `port` for the `ingress` trait.
    - **Solution:** Double-check the trait definition using `vela def show <trait-type>` (e.g., `vela def show ingress`) to verify expected parameters and types. Consult the KubeVela official documentation for built-in trait references.
3. **Application Status "Unhealthy" or "Running but Warning":**
    - **Symptom:** `vela ls` shows your application with an unhealthy status.
    - **Cause:** Underlying Kubernetes resources might be failing. This could be due to issues unrelated to traits (e.g., incorrect image name in the component, resource limits exceeded).
    - **Solution:** Use `vela status my-web-app --detail` to get a detailed breakdown of the application's health, including the status of its components and traits. This command will often point you to the specific Kubernetes resource that is having issues, which you can then inspect with `kubectl describe` and `kubectl logs`.

Summary

Congratulations! You've successfully leveraged KubeVela Traits to enhance your application's operational capabilities. Let's recap the key takeaways:

- **Traits abstract complexity:** They allow developers to declare high-level operational requirements (like scaling or exposing) without directly interacting with complex Kubernetes resources like `Deployment`, `Service`, or `Ingress`.
- **Separation of Concerns:** Traits embody the OAM principle of separating developer concerns (what the app does) from platform engineering concerns (how the app operates).
- **Extensibility:** KubeVela provides a rich set of built-in traits, and platform engineers can easily define custom traits to meet specific organizational needs.
- **Practical Application:** You've seen how to use the `scaler` trait to control replica counts and the `ingress` trait to expose your application to the external network.

By using Traits, KubeVela significantly reduces the cognitive load on developers, allowing them to focus on delivering business value while platform teams ensure consistent, standardized operational practices.

In the next chapter, we'll explore **Policies**, which allow platform engineers to enforce governance, security, and compliance rules across applications, adding another layer of control and automation to your application delivery pipeline.

References

- KubeVela Official Documentation - Traits
- KubeVela Official Documentation - Built-in Traits Reference
- KubeVela Official Documentation - TraitDefinition
- Open Application Model (OAM) Specification

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 05

Defining Your Delivery Pipeline: Mastering KubeVela Workflows

Imagine needing to deploy an application that requires multiple stages: first, setting up a database, then deploying your microservice, running integration tests, and finally, sending a notification to a Slack channel. How do you automate this complex, ordered sequence of operations consistently across different environments? This often involves a tangle of scripts, manual steps, or brittle CI/CD configurations.

This is precisely where KubeVela Workflows shine. For platform teams, Workflows transform KubeVela into a powerful application delivery control plane, allowing you to define, automate, and orchestrate the entire lifecycle of an application, from provisioning resources to post-deployment verification and promotion. They bring order and consistency to what would otherwise be a chaotic process.

In this chapter, we'll dive deep into KubeVela Workflows. You'll learn what they are, why they're indispensable for creating robust delivery pipelines, and how to build your own multi-stage workflows using KubeVela's flexible step system and the powerful CUE language. By the end, you'll be able to design sophisticated, automated application delivery processes that bring consistency and reliability to your deployments.

To get the most out of this chapter, you should be familiar with KubeVela's core concepts: Applications, Components, and Traits, as covered in previous sections. You'll also need a running Kubernetes cluster with KubeVela `v1.12.x` (as of 2026-06-22) installed and `kubectl` configured to access it.

The Power of KubeVela Workflows

When we talk about deploying applications, it's rarely a single, atomic operation. Modern applications often involve a series of interconnected steps: resource provisioning, service deployment, configuration, testing, and more. KubeVela Workflows provide the orchestration layer to manage these complexities directly within your application definition.

What is a KubeVela Workflow?

A KubeVela Workflow is an ordered sequence of steps that define the lifecycle of an application. Think of it as a customizable blueprint for how your application gets from code to production. Each step in a workflow performs a specific action, and they execute in a defined order, allowing you to build sophisticated delivery pipelines directly within your `Application` definition.

Unlike traditional CI/CD pipelines defined externally, KubeVela Workflows are declared as part of your `Application` resource. This means your application's delivery process becomes a first-class citizen alongside its components and traits, promoting a truly application-centric approach.

Why Workflows Matter for Platform Teams

For platform engineers, Workflows are a game-changer. They address several critical challenges that often plague application delivery:

- **Standardization:** Workflows enable you to encapsulate best practices for application delivery into reusable templates. Developers then consume these standardized pipelines, ensuring consistent deployments across the organization without needing to understand the underlying infrastructure complexities.
- **Automation:** Manual deployment steps are error-prone and slow. Workflows automate everything from resource provisioning to post-deployment checks, reducing human error, accelerating delivery, and freeing up developer time.
- **Extensibility:** KubeVela's workflow engine is incredibly flexible. You're not limited to built-in steps; you can define custom steps using CUE, allowing you to integrate with virtually any external system or internal tool, from cloud providers to custom CI/CD systems, security scanners, or notification services.
- **Observability:** KubeVela provides clear visibility into the execution status of each workflow step. This makes it easy to monitor progress, identify bottlenecks, and troubleshoot failures directly from the `vela` CLI or KubeVela UI.

Without Workflows, platform teams often resort to complex CI/CD scripts, custom Kubernetes operators, or manual interventions to manage application lifecycles. Workflows offer a declarative, application-centric, and Kubernetes-native approach to solve this.

Workflow Steps: The Building Blocks of Delivery

Each workflow is composed of individual **steps**. A step is an atomic unit of work that performs a specific action. KubeVela comes with several built-in step types, and you can extend these with custom definitions using `WorkflowStepDefinition` resources.

Common built-in step types you'll encounter include:

- `deploy`: This is the most common step, responsible for deploying components and their associated traits onto the Kubernetes cluster.
- `suspend`: Pauses the workflow until manually resumed or a specified condition is met. This is incredibly useful for implementing manual approvals, waiting for external systems, or staging rollouts.
- `apply-component`: Applies a specific component definition, allowing for fine-grained control over which components are deployed at which stage.
- `read-component`: Reads the output of a component, enabling subsequent steps to use information generated by a deployed component.
- `custom`: Allows you to define highly custom logic using CUE, making it possible to interact with external APIs, run shell commands, or perform complex data transformations.

Steps can pass information between each other using KubeVela's context mechanism, allowing you to build sophisticated pipelines where the output of one step becomes the input for the next. This enables complex, data-driven delivery processes.

Customizing Steps with CUE: The Secret Sauce

The real power and flexibility of KubeVela Workflows come from its deep integration with CUE. CUE (Configuration, Unification, and Execution) is a powerful open-source language that allows you to define, generate, and validate configurations. In KubeVela, CUE is used to:

- **Define Custom Workflow Steps:** You can write CUE templates that describe the logic and actions of a new, domain-specific workflow step. This allows you to integrate with external APIs (e.g., a cloud provider's database service, a security scanner), run shell commands inside a temporary pod, or perform complex data transformations.

- **Control Step Behavior:** CUE can be used within existing steps (like `deploy`) to dynamically modify resource configurations based on workflow context, application parameters, or even outputs from previous steps. This enables highly adaptive deployments.
- **Validate Inputs/Outputs:** Ensure that data flowing between steps and the parameters provided to custom steps meet expected schemas, enhancing reliability and preventing misconfigurations.

📌 **Key Idea:** CUE provides a robust, declarative way to extend KubeVela's capabilities without writing Go code or building custom Kubernetes operators. This significantly lowers the barrier for platform engineers to create powerful, tailored delivery experiences.

Visualizing the Workflow Execution Flow

To better understand how KubeVela Workflows operate, let's visualize a common delivery pipeline pattern.

Diagram unavailable in this PDF export.

In this flow:

- The `Application Definition` triggers the `Workflow`.
- A `Deploy Staging Component` step creates resources in a staging environment.
- `Run Integration Tests` then verifies the staging deployment.
- Based on the test results, the workflow conditionally proceeds:
  - If `Tests Pass`, it might `Request Manual Approval` (using a `suspend` step).
  - If `Tests Fail`, it could `Send Failure Alert` (a custom CUE step).
- Upon approval, `Deploy Production Component` executes.
- Finally, the workflow concludes, whether through success or failure paths.

This diagram illustrates sequential execution with conditional branching, a core capability of KubeVela Workflows. Each box represents a step, and the arrows show the flow of control.

Building Your First Delivery Pipeline

Let's put theory into practice by creating a multi-stage application delivery workflow. Our goal is to:

1. Define a simple Nginx web server component.
2. Implement a "pre-deployment" gate, simulating a manual approval before any deployment occurs.
3. Deploy the Nginx component.
4. Perform a "post-deployment" verification using a custom CUE step to simulate a health check.

Setting the Stage: Our Sample Application

First, let's define a basic KubeVela `Application` with an Nginx component. We'll add the workflow directly to this application.

Create a file named `nginx-app-workflow.yaml`:

```
# nginx-app-workflow.yaml
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: nginx-workflow-app
spec:
  components:
    - name: my-nginx-component
      type: webservice
      properties:
        image: nginx:1.25.3 # Using a recent stable Nginx version (as of
2026-06-22)
        port: 80
      traits:
        - type: ingress
          properties:
            domain: workflow.example.com
            http:
              /: 80
  # We'll add the workflow definition here!
```

This is a standard KubeVela application deploying an Nginx `webservice` component with an `ingress` trait. Nothing new here, just a foundation for our workflow.

Step 1: Initial Deployment (and a Pre-Deployment Pause)

Now, let's add a `workflow` section to our `Application`. Our first step will be a `suspend` step, simulating a manual approval or a pre-deployment gate. This step will pause the workflow until we manually resume it.

Add the `workflow` section to your `nginx-app-workflow.yaml` file, right after the `components` section:

```
# nginx-app-workflow.yaml (continued)
# ... (existing components and traits)
  workflow:
    steps:
      - name: pre-deployment-check
        type: suspend
        properties:
          message: "Manual approval required before deploying Nginx
component. Review changes."
      - name: deploy-nginx
        type: deploy
        properties:
          # This step will deploy all components defined in the application.
          # For granular control, you could specify:
          # component: my-nginx-component
```

Let's break down what we've added:

- `workflow`: This is the top-level key for defining our application's delivery pipeline.
- `steps`: A list of individual workflow actions that KubeVela will execute in order.
- `pre-deployment-check`: Our first step, of `type: suspend`. It will simply pause the workflow. The `message` property provides context to anyone viewing the workflow status, explaining why it's paused.
- `deploy-nginx`: This is a standard `deploy` step. By default, it will deploy all components defined in the `Application`. If you had multiple components and only wanted to deploy a specific one at this stage, you could use `component: <component-name>`.

Now, let's apply this application to your Kubernetes cluster and observe its state:

```
kubectl apply -f nginx-app-workflow.yaml
```

Once applied, check the application status, paying close attention to the workflow section:

```
vela status nginx-workflow-app --workflow
```

You should see output similar to this, indicating the workflow is paused:

```
App Status: running
...
Workflow Steps:
  - Name: pre-deployment-check
    Type: suspend
    Phase: running
    Message: Manual approval required before deploying Nginx component. Review
changes.
  - Name: deploy-nginx
    Type: deploy
    Phase: pending
```

Notice that `pre-deployment-check` is `running` and `deploy-nginx` is `pending`. The workflow is currently paused, waiting for your instruction!

To proceed and allow the deployment to happen, we need to manually resume the workflow:

```
vela workflow resume nginx-workflow-app
```

Now, check `vela status nginx-workflow-app --workflow` again. You should see the `deploy-nginx` step now running or completed, and your Nginx component being deployed onto the cluster. You can verify the Nginx deployment by checking `kubectl get deploy,svc,ing -l app.oam.dev/name=nginx-workflow-app`.

Step 2: Post-Deployment Verification with a Custom CUE Step

Deploying is one thing, but verifying that the deployment is healthy and functioning correctly is another crucial step. What if we want to perform a check after deployment, like verifying an external API endpoint or checking for specific logs? We can define a custom CUE step for this.

First, let's define a `WorkflowStepDefinition` that encapsulates our verification logic. We'll create a simple step that "checks" a condition and, for demonstration, can either succeed or fail based on a parameter.

Create a file named `verify-step.yaml`:

```
# verify-step.yaml
apiVersion: core.oam.dev/v1beta1
kind: WorkflowStepDefinition
metadata:
  name: verify-deployment
spec:
  schematic:
    cue: |
      parameter: {
        condition: bool @default(true) // Simulates a condition check: true
```

```
for success, false for failure
      }
      outputs: {
        result: "Verification " + (if parameter.condition {"succeeded"} else
{"failed"})
      }
      // In a real-world scenario, you would replace the simple 'condition'
parameter with actual logic:
      // - Making an HTTP request to a service endpoint to check its
availability.
      // - Querying Kubernetes resource status (e.g., all pods are 'Ready').
      // - Running a shell command inside a temporary pod to execute a test
script.
      // - Interacting with an external monitoring system to check metrics.
      // If the check fails, you could use `fail: "Reason for failure"` to
stop the workflow and trigger a rollback.
      // For this example, we'll just output a message based on the
'condition' parameter.
```

Apply this `WorkflowStepDefinition` to your cluster. This registers our new custom step type with KubeVela:

```
kubectl apply -f verify-step.yaml
```

Now, let's modify our `nginx-app-workflow.yaml` to include this new `verify-deployment` step after our `deploy-nginx` step.

```yaml
# nginx-app-workflow.yaml (modified)
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: nginx-workflow-app
spec:
  components:
    - name: my-nginx-component
      type: webservice
      properties:
        image: nginx:1.25.3
        port: 80
      traits:
        - type: ingress
          properties:
            domain: workflow.example.com
            http:
              /: 80
  workflow:
    steps:
      - name: pre-deployment-check
        type: suspend
        properties:
          message: "Manual approval required before deploying Nginx
component. Review changes."
      - name: deploy-nginx
        type: deploy
      - name: post-deployment-verify
        type: verify-deployment # Using our custom step!
```

```
        properties:
          condition: true # Set to 'false' here to simulate a verification
failure
```

Apply the updated application. Since the workflow for `nginx-workflow-app` might still be `running` or `terminated` from the previous execution, KubeVela will re-evaluate it. If it was `terminated`, it will run again. If it was `running` and already past the `suspend` step, it will continue with the new steps.

To ensure you see the new workflow in action from the beginning, it's often best to delete and re-create the application, and then resume it after the suspend step:

```
# Delete the existing application to clear its workflow state
kubectl delete application nginx-workflow-app

# Re-apply the updated application with the new verification step
kubectl apply -f nginx-app-workflow.yaml

# Resume the workflow after the initial suspend step
vela workflow resume nginx-workflow-app
```

Now, check the status again:

```
vela status nginx-workflow-app --workflow
```

You should see `post-deployment-verify` running or completed. This is where the power of custom steps becomes evident. You can inspect its logs to see the output from our CUE logic:

```
vela workflow logs nginx-workflow-app --step post-deployment-verify
```

The logs might show the output from our CUE step, like "Verification succeeded". If you had set `condition: false` in the `properties` for the `post-deployment-verify` step in `nginx-app-workflow.yaml` and reapplied, the output would reflect "Verification failed". In a real scenario, a failing CUE step could use `fail: "reason"` to stop the workflow or trigger a rollback.

Step 3: Observing Workflow Progress

KubeVela provides excellent tools to observe and manage your workflow's execution. These commands are invaluable for platform engineers managing complex delivery pipelines:

- `vela status <app-name> --workflow`: This is your primary command. It shows the current phase of each step (e.g., `running`, `pending`, `succeeded`, `failed`, `terminated`).
- `vela workflow logs <app-name> --step <step-name>`: Fetches logs for a specific step. This is crucial for debugging custom CUE steps, as it displays any `print` statements or errors from your CUE code.
- `vela workflow restart <app-name>`: Restarts the workflow from the beginning. Useful for re-running a deployment after fixing an issue.
- `vela workflow terminate <app-name>`: Stops a running workflow immediately. This can be used to halt a problematic deployment.
- `vela workflow suspend <app-name>`: Pauses a running workflow at its current step.
- `vela workflow resume <app-name>`: Resumes a suspended workflow, allowing it to continue from where it left off.

Mini-Challenge: Enhancing Your Workflow

You've built a basic multi-stage workflow. Now, let's make it even more robust by adding a notification step.

Challenge: Modify the `nginx-workflow-app` to include an additional workflow step:

1. After the `post-deployment-verify` step, add a new custom step called `send-slack-notification`.
2. This step should take a `message` parameter (e.g., "Nginx deployment to staging completed successfully!").
3. For simplicity, the CUE definition for this `send-slack-notification` step doesn't need to actually integrate with Slack. Just make it output a message like "Sending Slack notification: [your message]".
4. Apply the new `WorkflowStepDefinition` and then update your `Application`'s workflow to include this new step.

Hint:

- You'll need to create a new `WorkflowStepDefinition` YAML file (e.g., `slack-step.yaml`) similar to `verify-step.yaml`.
- Remember to define `parameter` and `outputs` in the CUE schematic.
- Then, add a new step to the `workflow.steps` list in your `nginx-app-workflow.yaml`, placing it after the `post-deployment-verify` step.
- Don't forget to `kubectl apply` both the new `WorkflowStepDefinition` and the updated `Application`. You might need to delete and re-apply the application to see the full workflow run from the start.

What to observe/learn: You'll see how easy it is to extend your delivery pipeline with new custom actions, building a comprehensive, automated process for your applications. This exercise reinforces the power of `WorkflowStepDefinition` for encapsulating reusable logic.

Troubleshooting Common Workflow Issues

Workflows, especially with custom CUE steps, can sometimes be tricky. Here are some common pitfalls and how to troubleshoot them:

- **Workflow Stuck in `running` or `pending`:**
  - **Diagnosis:** Use `vela status <app-name> --workflow`. This will show you which specific step is currently active or waiting.
  - **Solution:**
    - If it's a `suspend` step, you likely need to `vela workflow resume <app-name>`.
    - If it's a `deploy` step, check the underlying Kubernetes resources (Pods, Deployments, Services) for errors using `kubectl get events`, `kubectl describe` the relevant resources, or `kubectl logs` pods. The application's components might be failing to start.
    - If it's a custom CUE step, move to the next point.

- **CUE Syntax Errors or Logic Issues in Custom Steps:**
  - **Diagnosis:** When defining `WorkflowStepDefinition` with CUE, even a small syntax error or logical flaw can prevent the step from working. `vela workflow logs <app-name> --step <step-name>` is your best friend here. It will often output detailed CUE validation errors, runtime errors, or the results of your `print` statements.
  - **Solution:** Carefully review the CUE code in your `WorkflowStepDefinition`. Use a CUE linter or an IDE extension with CUE support during development to catch errors early. Incrementally build your CUE logic, testing small parts before combining them.
- **Incorrect Step Order or Dependencies:**
  - **Diagnosis:** Workflows execute steps sequentially as defined in the `steps` list. If a later step depends on an output from an earlier step, or if an action needs to happen before another, the order is critical.
  - **Solution:** Review your `workflow.steps` list. KubeVela allows `dependsOn` in steps for explicit ordering, which can be useful for non-linear flows, but for simple linear pipelines, the list order is generally sufficient. Ensure that any resource provisioning or information gathering steps occur before steps that rely on them.
- **Permissions Issues for Custom Steps:**
  - **Diagnosis:** If your custom CUE step attempts to create, modify, or delete Kubernetes resources (e.g., creating a `Secret`, updating a `ConfigMap`), or if it interacts with external services, the KubeVela controller's ServiceAccount needs the necessary RBAC permissions.
  - **Solution:** Ensure the `ClusterRole` and `ClusterRoleBinding` for KubeVela grant appropriate permissions for the resources your custom steps manage. If a CUE step calls an external API, ensure the KubeVela controller pod has network access and any required credentials (e.g., API keys) are correctly mounted as `Secret`s and referenced within your CUE logic.

⚡ **Real-world insight:** For complex CUE steps, it's often helpful to break down the logic into smaller, testable CUE files. You can even use the `cue eval` command-line tool to test parts of your CUE definitions independently before embedding them directly into a `WorkflowStepDefinition`. This significantly speeds up debugging.

Summary: Orchestrating Your Application's Journey

Congratulations! You've successfully navigated the world of KubeVela Workflows, a cornerstone for building robust and automated application delivery pipelines.

Here are the key takeaways from this chapter:

- **Workflows automate application lifecycle management:** They define an ordered sequence of steps for deploying, verifying, promoting, and even rolling back applications.
- **Platform teams leverage Workflows for standardization and extensibility:** They provide a declarative, application-centric way to build consistent and customizable delivery processes across environments.
- **Steps are the building blocks:** KubeVela offers powerful built-in steps like `deploy` and `suspend`, and you can create highly specialized custom steps.
- **CUE is crucial for customization:** It empowers platform engineers to define complex custom workflow logic, integrate with external systems, and dynamically control step behavior without writing Go code.
- **Observability is built-in:** Tools like `vela status --workflow` and `vela workflow logs` provide clear, real-time visibility into pipeline execution, making monitoring and debugging straightforward.

By mastering Workflows, you empower your team to move beyond simple deployments to fully automated, intelligent application delivery. This is a significant step towards building a truly self-service, cloud-native platform that reduces operational burden and accelerates development.

In the next chapter, we'll explore KubeVela Addons, which allow you to extend KubeVela's capabilities by installing pre-packaged features like advanced metrics, logging, or integration with GitOps tools. This will further enhance your platform engineering toolkit, showing you how to grow your KubeVela platform.

References

- KubeVela Official Documentation: Workflow
- KubeVela Official Documentation: WorkflowStepDefinition
- CUE Language Official Website
- KubeVela GitHub Repository (for latest releases)

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 06

Governing Applications: Applying Policies for Compliance and Security

Ensuring applications adhere to organizational standards for security, compliance, and resource usage can feel like an endless battle in a dynamic Kubernetes environment. Without proper governance, applications can quickly become sprawling, insecure, and costly. This is where KubeVela Policies step in, offering a powerful way for platform teams to regain control.

In this chapter, we'll explore how KubeVela provides an application-centric approach to enforce these critical rules. You'll learn exactly what KubeVela Policies are, why they are essential for platform teams, and how to apply them to your applications to maintain control and order. We'll move beyond raw Kubernetes resource-level policies and embrace KubeVela's higher-level abstractions for comprehensive application governance.

To get the most out of this chapter, you should have a basic understanding of KubeVela Applications, Components, and Traits, as covered in previous sections.

The Governance Challenge for Platform Teams

Imagine you're a platform engineer managing hundreds of applications across multiple Kubernetes clusters. Each application might consist of several Kubernetes Deployments, Services, ConfigMaps, and other resources. How do you ensure:

- All container images originate from approved, secure registries?
- No single application consumes more than its fair share of CPU or memory, preventing "noisy neighbor" issues?
- Sensitive data is never exposed via unencrypted connections, adhering to security best practices?
- Resources are cleaned up automatically and completely when an application is removed, avoiding orphaned resources and unnecessary cloud costs?

Applying these rules directly at the Kubernetes resource level (e.g., using Admission Controllers, raw `ResourceQuota` objects, or Pod Security Standards) becomes incredibly complex and fragmented. Developers would need to understand and apply these low-level policies, leading to cognitive overload, potential errors, and inconsistent application deployments.

📌 **Key Idea:** The core problem is that raw Kubernetes policies operate at the resource level, while platform teams need to govern at the application level.

KubeVela addresses this by elevating policy enforcement to the application layer. Instead of policing individual Kubernetes resources, you define policies that apply to entire KubeVela Applications. This provides a consistent, higher-level abstraction and a developer-friendly governance model.

KubeVela Policies: Application-Centric Control

KubeVela Policies are a core part of the Open Application Model (OAM), specifically designed to define and enforce rules and constraints on applications. They allow platform engineers to codify operational best practices, security requirements, and compliance mandates into reusable building blocks.

Essentially, a KubeVela Policy tells the KubeVela control plane how an application should behave or what conditions it must meet to be considered compliant and healthy. This separation of concerns is powerful: platform engineers can define these governance rules once, and developers can simply consume them by referencing policies in their application definitions, without needing to understand the intricate underlying Kubernetes details.

PolicyDefinition vs. Policy Instance

Just like Components and Traits, Policies in KubeVela are extensible. The framework distinguishes between two key concepts:

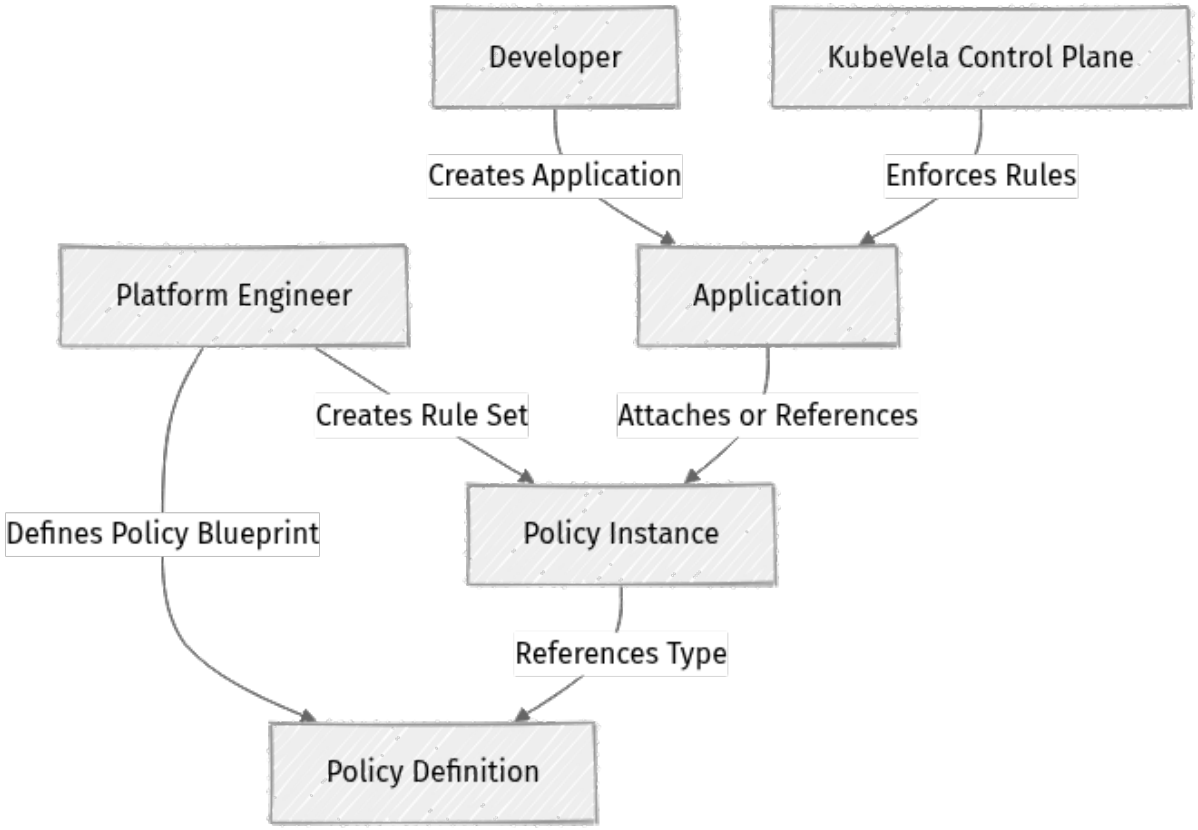
1. **`PolicyDefinition`**: This is a blueprint or template for a type of policy. Platform engineers create `PolicyDefinition` resources to define new types of policies tailored to their organization's specific needs. These definitions specify the policy's parameters and how it interacts with the application. KubeVela ships with several built-in `PolicyDefinition`s (e.g., for resource quotas, garbage collection).

2. **`Policy`**: This is an instance of a `PolicyDefinition`. Once a `PolicyDefinition` exists, developers or platform engineers can create `Policy` instances that refer to these definitions and apply them to applications. A `Policy` instance specifies the concrete values for the parameters defined in its `PolicyDefinition`.

🧠 **Important:** `PolicyDefinition` defines what kind of policy can exist, while a `Policy` instance defines a specific set of rules of that kind.

How Policies Integrate with Applications

The relationship between `PolicyDefinition`, `Policy` instances, and `Application`s is crucial for understanding KubeVela's governance model:



In this flow:

- **Platform Engineers** define `PolicyDefinition`s (reusable templates) and often create common `Policy` instances (specific rule sets).
- **Developers** (or platform engineers managing applications) then create `Application` resources.
- An `Application` resource includes a `policies` section, which can either embed a policy directly or, more commonly, reference an existing `Policy` object by name.

- The **KubeVela Control Plane** observes these policies attached to applications and enforces them during the application's lifecycle, ensuring compliance and desired behavior.

Common KubeVela Policy Types

KubeVela ships with several powerful built-in policy types, and its extensibility allows you to define many more:

- `ResourceQuotaPolicy`: Limits the total CPU, memory, or other compute resources that an application (or its components) can consume within a namespace. This is crucial for cost management and preventing resource exhaustion.
- `TopologyPolicy`: Specifies deployment constraints, such as where an application should be deployed (e.g., specific clusters, namespaces, regions, or even node labels). Ideal for multi-cluster deployments, disaster recovery, and compliance with data residency.
- `GarbageCollectPolicy`: Defines how resources associated with an application are cleaned up when the application itself is deleted. This prevents orphaned resources and reduces cloud costs by ensuring complete removal.
- `HealthCheckPolicy`: Establishes criteria for determining an application's overall health, going beyond simple Pod liveness/readiness probes to include dependencies or business logic.
- `OverridePolicy`: Allows overriding certain parameters in components or traits based on environmental conditions or other criteria, providing dynamic configuration.
- `SharedResourcePolicy`: Manages the lifecycle of resources shared across multiple applications.

These policies provide powerful ways to standardize application behavior, resource consumption, and deployment patterns across your fleet.

Step-by-Step: Applying a Resource Quota Policy

Let's put theory into practice. We'll apply a `ResourceQuotaPolicy` to an application to limit its resource consumption. This is a common requirement for platform teams to prevent "noisy neighbors" and manage cloud spend.

Prerequisites: Your KubeVela Environment

Ensure you have a running Kubernetes cluster with KubeVela installed. You should also have `kubectl` and `vela` CLI tools configured.

As of **2026-06-22**, KubeVela `v1.10.x` is the current stable release series. Your `vela` CLI and KubeVela control plane should be compatible.

```
# Verify KubeVela CLI version (should be v1.10.x or newer)
vela version

# Verify kubectl access to your cluster
kubectl cluster-info
```

Step 1: Define a Simple Application

First, let's define a basic KubeVela application. This application will deploy a simple Nginx server.

Create a file named `my-nginx-app.yaml`:

```
# my-nginx-app.yaml
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-nginx-app
spec:
  components:
    - name: nginx-server
      type: webservice
      properties:
        image: nginx:1.25.3
        port: 80
        cpu: "500m" # Requesting 500 millicores
        memory: "512Mi" # Requesting 512 MiB
```

Explanation:

- This is a standard KubeVela `Application` resource.
- It defines a single `webservice` component named `nginx-server`.
- We've explicitly set `cpu` and `memory` requests for this component. This is important because `ResourceQuotaPolicy` will typically act upon these requests to ensure they don't exceed defined limits.

Deploy this application:

```
vela up -f my-nginx-app.yaml --wait
```

The `--wait` flag ensures the command waits until the application is fully deployed and healthy according to its definition.

You can check its status:

```
vela status my-nginx-app
```

At this point, the application should be healthy and requesting its specified resources.

Step 2: Create a Resource Quota Policy Instance

Now, let's define a specific `Policy` instance that limits the total resources an application can request. We'll create a `ResourceQuotaPolicy` that enforces a maximum of `200m` CPU and `256Mi` memory for any application it's applied to.

Create a file named `low-resource-quota-policy.yaml`:

```
# low-resource-quota-policy.yaml
apiVersion: core.oam.dev/v1beta1
kind: Policy
metadata:
  name: low-resource-quota # This is the name we'll reference later
spec:
  type: resource-quota # This refers to the built-in
ResourceQuotaPolicyDefinition
  properties:
    cpu: "200m"
    memory: "256Mi"
```

Explanation:

- `kind: Policy`: This tells Kubernetes and KubeVela that we are defining a specific policy rule set.
- `metadata.name: low-resource-quota`: This is the unique name of this policy instance. We will use this name to reference it from our application.
- `spec.type: resource-quota`: This specifies that this policy is an instance of the `ResourceQuotaPolicyDefinition`. KubeVela knows how to interpret the `properties` based on this type.
- `spec.properties`: These are the parameters for this specific `resource-quota` policy. Here, we set a hard limit of `200m` CPU and `256Mi` memory.

Apply this `Policy` instance to your Kubernetes cluster using `kubectl`:

```
kubectl apply -f low-resource-quota-policy.yaml
```

⚡ **Quick Note:** Remember, `PolicyDefinition`s are applied using `vela def apply`, but `Policy` instances (which are standard Kubernetes custom resources) are applied using `kubectl apply`.

Step 3: Attach the Policy to the Application

Now, we need to tell our `my-nginx-app` to use the `low-resource-quota` policy we just created. We do this by adding a `policies` section to our `Application` manifest and referencing the policy by its name.

Modify `my-nginx-app.yaml` to include the policy reference:

```
# my-nginx-app.yaml (modified to reference Policy object)
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-nginx-app
spec:
  components:
    - name: nginx-server
      type: webservice
      properties:
        image: nginx:1.25.3
        port: 80
        cpu: "500m" # Still requesting 500 millicores
        memory: "512Mi" # Still requesting 512 MiB
  policies: # <--- New section for policies
    - name: my-app-resource-limit # A unique name for this policy binding
within the application
      type: resource-quota # The type of policy, matching the PolicyDefinition
      properties:
        policyRef: low-resource-quota # Referencing our Policy object by its
name
```

Explanation of the `policies` section:

- `policies:`: This top-level key within the `Application` spec is where you list all policies that apply to this application.
- `- name: my-app-resource-limit`: This is an internal name for this specific policy binding within this application manifest. It helps differentiate multiple policies of the same type if needed.
- `type: resource-quota`: This specifies that we are using a policy of the `resource-quota` type.
- `properties: policyRef: low-resource-quota`: This is the crucial part. It tells KubeVela to apply the rules defined in the standalone `Policy` object named `low-resource-quota` to this application. This promotes reusability, as the `low-resource-quota` policy can now be applied to many applications.

Now, apply the updated application manifest:

```
vela up -f my-nginx-app.yaml --wait
```

Step 4: Observe Policy Enforcement

After updating the application, KubeVela's control plane will reconcile the application with the attached policy. Our application requests `500m` CPU and `512Mi` memory, but the `low-resource-quota` policy limits it to `200m` CPU and `256Mi` memory. This means the policy should intervene.

Let's check the status of the application:

```
vela status my-nginx-app --detail
```

You should see output indicating that the policy `low-resource-quota` is affecting the application. KubeVela, through its `ResourceQuotaPolicy` implementation, will typically create a Kubernetes `ResourceQuota` object in the application's namespace. This `ResourceQuota` will then enforce the defined limits, preventing the application's Pods from requesting more than allowed.

Look closely at the `Policy` section of the `vela status --detail` output for any warnings or messages. You can also inspect the underlying Kubernetes `ResourceQuota` object:

```
kubectl get resourcequota -n default # Or the namespace where your app is
deployed
kubectl describe resourcequota low-resource-quota-my-app-resource-limit -n def
ault # KubeVela names it based on policy and app
```

You will likely observe that the `ResourceQuota` object shows a limit of `200m` CPU and `256Mi` memory. When your application's Pods attempt to request `500m` CPU, the Kubernetes Admission Controller (which `ResourceQuota` leverages) will prevent the Pod from being scheduled or created if it exceeds the quota. KubeVela might also show the application in a `unhealthy` or `progressing` state if the policy prevents full deployment.

⚡ **Real-world insight:** `ResourceQuotaPolicy` directly translates to Kubernetes `ResourceQuota` resources. This means the enforcement is handled by Kubernetes itself, with KubeVela providing the higher-level abstraction and application-centric binding.

Step 5: Clean Up

To clean up the resources created in this step:

```
vela delete my-nginx-app
kubectl delete -f low-resource-quota-policy.yaml
```

This will remove both the KubeVela Application and the standalone Policy object.

Mini-Challenge: Enforcing Resource Cleanup with `GarbageCollectPolicy`

Platform teams constantly battle resource sprawl, where deleted applications leave behind orphaned Kubernetes resources (like PersistentVolumeClaims, Secrets, or even Deployments that somehow got detached). This leads to unnecessary cloud costs and clutter.

Challenge: Modify the `my-nginx-app` (or create a new simple application) and apply a `GarbageCollectPolicy` to it. This policy should ensure that all Kubernetes resources created by the application are automatically and cleanly deleted when the application itself is deleted via `vela delete`.

Hint:

- You'll need to define a `Policy` of `type: garbage-collect`.
- The `garbage-collect` policy has properties like `strategy` and `rules`. A common strategy is `onAppDelete`, which ensures cleanup when the `Application` is deleted.
- Attach this `Policy` instance to your `my-nginx-app` (or a new test application) in its `policies` section.
- Deploy the application. Verify it creates some Kubernetes resources (e.g., `kubectl get deploy,svc,pod -n default`).
- Then, `vela delete` the application and observe if all associated Kubernetes resources are gone.

What to observe/learn: After running `vela delete <your-app>`, execute `kubectl get all -n <your-app-namespace>` (or the namespace you used). You should find that no resources or significantly fewer resources (e.g., only KubeVela's own internal tracking resources, not your application's deployments, services, etc.) remain. This demonstrates how KubeVela policies automate cleanup, reducing manual toil, preventing resource leakage, and ultimately saving cloud costs.

Common Pitfalls & Troubleshooting

Even with KubeVela's abstractions, policies can sometimes be tricky. Here are some common issues and how to approach them:

1. **Policy Not Taking Effect or Unexpected Behavior:**
    - **Incorrect `policyRef`:** Double-check the `metadata.name` in your `Policy` object and the `properties.policyRef` in your `Application` manifest. They must match exactly.
    - **Wrong `type`:** Ensure the `type` in your `Application`'s policy reference (e.g., `type: resource-quota`) matches the `type` defined in the `Policy` instance.
    - **Policy not applied:** Did you `kubectl apply -f your-policy.yaml` the `Policy` instance itself? If it's a `PolicyDefinition`, did you use `vela def apply`?
    - **Timing/Reconciliation:** KubeVela's control plane needs time to reconcile changes. Wait a few moments and check `vela status --detail` again. Sometimes, a `vela up -f <app.yaml> --refresh` can force a re-evaluation.
2. **Conflicting Policies:**
    - If multiple policies apply to the same resource or aspect of an application (e.g., two `ResourceQuotaPolicy` instances, or a `ResourceQuotaPolicy` and an `OverridePolicy` affecting resource requests), their interactions can be complex. KubeVela has a policy evaluation order, but it's generally best to design policies to be as orthogonal as possible to avoid ambiguity.
    - Check `vela status <app-name> --detail` carefully for any warnings or errors related to policy conflicts. KubeVela will often log if a policy is being overridden or if conflicts are detected.

3. **Debugging Policies:**
    - `vela status <app-name> --detail`: This is your primary diagnostic tool. It shows the status of all components, traits, workflows, and policies applied to the application. Look for the `Policy` sections and any `Health` or `Message` fields indicating issues or enforcement actions.
    - `kubectl get policy <policy-name> -o yaml`: Inspect the raw `Policy` object YAML to ensure its `spec` is correctly defined and matches your intent.
    - `kubectl describe policy <policy-name>`: Provides more detailed events and status information for the `Policy` object itself, which can reveal why it's not being applied or is in an error state.
    - `kubectl get policydefinition`: Verify that the `PolicyDefinition` for the type you're using (e.g., `resource-quota`) exists and is healthy in your cluster. If it's a custom policy, check its definition carefully.
    - **KubeVela Controller Logs:** For deeper issues, inspect the logs of the KubeVela core controllers (e.g., `vela-core` deployment in the `vela-system` namespace) for error messages related to policy reconciliation.

Summary

In this chapter, we've taken a significant step into the world of application governance with KubeVela Policies. This powerful feature allows platform teams to embed crucial operational, security, and compliance rules directly into the application delivery process.

Here are the key takeaways:

- **Application-Centric Governance:** KubeVela Policies enable platform teams to define and enforce rules at the application level, abstracting away low-level Kubernetes complexities for developers.
- **Separation of Concerns:** `PolicyDefinition`s act as reusable blueprints, while `Policy` instances define specific rule sets. Developers then consume these `Policy` instances by referencing them in their `Application` manifests.
- **Extensibility:** KubeVela's policy framework is highly extensible, allowing platform engineers to define custom policies tailored to unique organizational needs.

- **Built-in Policies:** KubeVela provides several powerful built-in policies, such as `ResourceQuotaPolicy` for resource management and `GarbageCollectPolicy` for automated cleanup.
- **Practical Enforcement:** We demonstrated how to apply a `ResourceQuotaPolicy` to limit application resource consumption, a critical aspect of cost control and stability in any Kubernetes environment.

By leveraging KubeVela Policies, platform teams can ensure consistency, compliance, and security across their application fleet. This empowers developers to focus on building features without getting bogged down in infrastructure-level governance details, ultimately leading to faster, safer, and more reliable application delivery.

Next, we'll dive into KubeVela Workflows, which allow you to define custom, multi-step application delivery processes, taking your application management to the next level.

References

- KubeVela Policy Documentation
- KubeVela Built-in Policies Guide
- Open Application Model (OAM) Specification
- Kubernetes Resource Quotas

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 07

Empowering Platform Engineers: Customizing KubeVela with Definitions and Addons

Empowering Platform Engineers: Customizing KubeVela with Definitions and Addons

Welcome back, intrepid platform builder! In our journey through KubeVela, we've explored its core application model and how it simplifies deployments for developers. But what if the built-in components and traits aren't enough for your organization's unique needs? What if you have a proprietary service or a specific operational pattern you want to expose to developers without them needing to touch raw Kubernetes?

This chapter is your deep dive into KubeVela's extensibility. We'll uncover how platform engineers wield the power of **Definitions** (for Components, Traits, Policies, and Workflows) and **Addons** to tailor KubeVela precisely to their platform's requirements. By the end, you'll understand not just how to extend KubeVela, but why it's a game-changer for building truly self-service, opinionated platforms.

The "Why" of Extensibility: Beyond Off-the-Shelf

Imagine your developers need to deploy an application that automatically integrates with your internal monitoring system and gets a specific type of database provisioned. With raw Kubernetes, they'd write multiple YAML manifests, possibly `ConfigMaps`, `Secrets`, and `ServiceMonitors`, each with intricate configurations. This is tedious, error-prone, and requires deep Kubernetes knowledge.

KubeVela's built-in components and traits abstract many common patterns, but they can't cover every unique system or operational requirement of your specific organization. This is where extensibility shines.

Platform engineers need the ability to:

- **Encapsulate Operational Best Practices:** Define custom components that deploy applications according to organizational standards (e.g., specific sidecars, security policies).
- **Integrate with Internal Systems:** Connect applications seamlessly with proprietary monitoring, logging, or secret management solutions.
- **Simplify Developer Experience:** Provide developers with high-level, application-centric abstractions that hide underlying complexity, letting them focus on business logic.
- **Standardize Delivery Workflows:** Define custom steps in application delivery workflows to enforce specific deployment, testing, or approval gates.

KubeVela's extensibility mechanisms empower platform teams to be the architects of their application delivery experience, providing a curated catalog of capabilities that developers can consume with ease.

KubeVela's Extensibility Foundation: Open Application Model (OAM) and CUE

At the heart of KubeVela's extensibility are the **Definitions** that enable you to extend the Open Application Model (OAM). These definitions are Kubernetes Custom Resource Definitions (CRDs) that KubeVela understands. They tell KubeVela how to translate a high-level application description into concrete Kubernetes resources.

The magic behind these definitions is often powered by **CUE**. CUE is an open-source language, a superset of JSON, designed for defining, generating, and validating configurations. KubeVela leverages CUE to:

- **Define Schema:** Specify the valid parameters and types for your custom components, traits, policies, and workflow steps. This ensures developers provide correct inputs.
- **Generate Kubernetes Manifests:** Dynamically create the underlying Kubernetes YAML based on the developer's inputs and your platform's logic.
- **Validate Configuration:** Enforce constraints and rules, preventing invalid deployments before they even hit the cluster.

Using CUE within definitions allows platform engineers to embed sophisticated logic and validation directly into their platform capabilities. This shifts the burden of knowing Kubernetes internals from developers to the platform, making the developer experience much smoother and more reliable.

Unpacking Definitions: Component, Trait, Policy, and WorkflowStep

KubeVela provides four primary types of Definitions for extensibility, each targeting a different aspect of the application delivery lifecycle:

1. **ComponentDefinition:**
    - **What it is:** Defines a new type of component that developers can use in their `Application` manifests. A component describes the core workload (e.g., a web service, a batch job).
    - **Why it exists:** To abstract how a specific type of workload is deployed. For example, you could define a `MyWebService` component that always deploys a `Deployment`, `Service`, and `Ingress` with specific defaults and sidecars.
    - **How it works:** It contains a CUE template that takes parameters from the developer's `Application` and renders the necessary Kubernetes resources.
2. **TraitDefinition:**
    - **What it is:** Defines a new operational capability that can be attached to a component. Traits modify or augment components (e.g., scaling, exposing, monitoring).
    - **Why it exists:** To separate operational concerns from the core workload definition. This allows platform engineers to provide reusable "add-ons" that developers can mix and match.
    - **How it works:** It takes parameters from the `Application`'s trait configuration and the component's generated resources, then renders additional Kubernetes resources or modifies existing ones.

3. **PolicyDefinition:**
    - **What it is:** Defines a new type of policy that can be applied to an `Application` or parts of it. Policies enforce governance, security, or compliance rules.
    - **Why it exists:** To allow platform engineers to codify and enforce organizational rules across applications, such as resource quotas, network access, or deployment windows.
    - **How it works:** It specifies validation logic and/or triggers actions based on the application's configuration or cluster state.
4. **WorkflowStepDefinition:**
    - **What it is:** Defines a new custom step that can be used within an `Application`'s workflow. Workflows orchestrate the delivery process.
    - **Why it exists:** To enable platform engineers to build highly customized, multi-stage delivery pipelines directly within KubeVela, integrating with external systems or internal tools.
    - **How it works:** It specifies the logic for a particular step, which can involve running scripts, calling APIs, or waiting for conditions, often leveraging CUE or even Go templates.

Step-by-Step Implementation: Crafting a Custom Trait for Internal Monitoring

Let's walk through creating a simple `TraitDefinition` that automatically injects a `ConfigMap` into a component to configure an imaginary internal monitoring agent. This trait will be called `internal-monitor`.

Prerequisites: Before we start, ensure you have KubeVela installed on your Kubernetes cluster. If you followed previous chapters, you should be all set. We're assuming KubeVela `v1.14.0` as of 2026-06-22 for this example.

Step 1: Define the Basic TraitDefinition Structure

First, let's create a file named `trait-internal-monitor.yaml`. We'll start with the standard Kubernetes object definition and KubeVela-specific metadata.

```
# trait-internal-monitor.yaml
apiVersion: core.oam.dev/v1beta1
kind: TraitDefinition
metadata:
  name: internal-monitor
  annotations:
    definition.oam.dev/description: "Injects an internal monitoring agent conf
iguration ConfigMap."
```

```
spec:
  appliesToWorkloads:
    - deployments.apps
    - statefulsets.apps
  schematic:
    cue:
      template: |
        #CUE_DEFINITION_TEMPLATE
        // The CUE template will go here, piece by piece.
```

Explanation:

- `apiVersion` and `kind`: Standard Kubernetes API version and resource type for a `TraitDefinition`.
- `metadata.name`: This is the name developers will use to refer to your custom trait in their `Application` manifests (e.g., `type: internal-monitor`).
- `annotations`: Provides a helpful description, which KubeVela UI tools can use.
- `spec.appliesToWorkloads`: This is critical! It tells KubeVela that this trait can be applied to workloads that result in Kubernetes `Deployment` or `StatefulSet` resources. If a developer tries to apply it to an unsupported workload, KubeVela will prevent it.
- `spec.schematic.cue.template`: This section is where we'll write our CUE logic. The `#CUE_DEFINITION_TEMPLATE` comment is a KubeVela convention to mark the start of the CUE template.

Step 2: Define the Trait's Parameters

Next, let's add the `parameter` block inside the `cue.template`. This defines what inputs a developer can provide when using our `internal-monitor` trait.

```
# trait-internal-monitor.yaml (updated)
apiVersion: core.oam.dev/v1beta1
kind: TraitDefinition
metadata:
  name: internal-monitor
  annotations:
    definition.oam.dev/description: "Injects an internal monitoring agent conf
iguration ConfigMap."
spec:
  appliesToWorkloads:
    - deployments.apps
    - statefulsets.apps
  schematic:
    cue:
      template: |
        #CUE_DEFINITION_TEMPLATE
        parameter: {
          // The name of the monitoring agent
```

```
          agentName: string @default("vela-monitor")
          // The endpoint for the monitoring agent to send data
          agentEndpoint: string @default("http://monitor-
server.internal:8080")
          // Labels to add to the ConfigMap
          labels?: [string]: string
        }
        // ... rest of the CUE template will follow
```

Explanation:

- `parameter`: This CUE block defines the schema for the trait's configuration.
- `agentName: string @default("vela-monitor")`: Defines a required `string` parameter named `agentName` with a default value.
- `agentEndpoint: string @default("http://monitor-server.internal:8080")`: Similar to `agentName`, for the monitoring server's URL.
- `labels?: [string]: string`: Defines an optional map (`?` makes it optional) where keys and values are strings. This allows developers to add custom labels to the generated `ConfigMap`.
- KubeVela automatically generates a Kubernetes CRD schema from this `parameter` block, ensuring developers get validation for their inputs.

Step 3: Access the Workload and Define Output Resources

Now, let's add the logic to access the underlying workload's pod specification and define the `ConfigMap` we want to create.

```
# trait-internal-monitor.yaml (updated)
apiVersion: core.oam.dev/v1beta1
kind: TraitDefinition
metadata:
  name: internal-monitor
  annotations:
    definition.oam.dev/description: "Injects an internal monitoring agent conf
iguration ConfigMap."
spec:
  appliesToWorkloads:
    - deployments.apps
    - statefulsets.apps
  schematic:
    cue:
      template: |
        #CUE_DEFINITION_TEMPLATE
        parameter: {
          agentName: string @default("vela-monitor")
          agentEndpoint: string @default("http://monitor-
server.internal:8080")
          labels?: [string]: string
        }
```

```
        // We process the existing workload (component) here.
        // `context.output` refers to the primary Kubernetes resource
generated by the component
        // (e.g., a Deployment). We access its PodSpec.
        workload: context.output.spec.template.spec

        // Define the ConfigMap to be created
        outputs: internalMonitorConfig: {
          apiVersion: "v1"
          kind: "ConfigMap"
          metadata: {
            name: context.name + "-monitor-config" // Dynamically name the
ConfigMap

labels: parameter.labels | {} // Merge with any provided labels or use an
empty map
          }
          data: {
            "monitor-agent.conf": """
              agent.name: \(parameter.agentName)
              agent.endpoint: \(parameter.agentEndpoint)
              # Other internal monitoring specific configurations can go here
              """
          }
        }
        // ... injection logic will follow
```

Explanation:

- `workload: context.output.spec.template.spec`: This line is crucial for traits. `context.output` represents the main Kubernetes resource generated by the component (e.g., the `Deployment`). We navigate to its `spec.template.spec` to get to the `PodSpec`, which contains volumes and containers.
- `outputs: internalMonitorConfig`: This CUE block declares a new Kubernetes resource that KubeVela should create.
  - `apiVersion`, `kind`: Standard fields for a `ConfigMap`.
  - `metadata.name: context.name + "-monitor-config"`: We dynamically generate the `ConfigMap`'s name using `context.name` (which is the component's name) to ensure it's unique and related to the application.
  - `labels: parameter.labels | {}`: This uses CUE's `|` (union) operator. It means "use the labels provided by `parameter.labels` if they exist, otherwise use an empty map `{}`."
  - `data`: This holds the actual content of our configuration file. Notice the `"""` for multi-line strings and `\(...)` for string interpolation, allowing us to embed `parameter.agentName` and `parameter.agentEndpoint` directly.

Step 4: Inject Volumes and Volume Mounts into the Workload

Finally, we'll modify the `workload` (the `PodSpec`) to inject our newly defined `ConfigMap` as a volume and mount it into the application containers.

```
# trait-internal-monitor.yaml (updated)
apiVersion: core.oam.dev/v1beta1
kind: TraitDefinition
metadata:
  name: internal-monitor
  annotations:
    definition.oam.dev/description: "Injects an internal monitoring agent conf
iguration ConfigMap."
spec:
  appliesToWorkloads:
    - deployments.apps
    - statefulsets.apps
  schematic:
    cue:
      template: |
        #CUE_DEFINITION_TEMPLATE
        parameter: {
          agentName: string @default("vela-monitor")
          agentEndpoint: string @default("http://monitor-
server.internal:8080")
          labels?: [string]: string
        }

        workload: context.output.spec.template.spec

        outputs: internalMonitorConfig: {
          apiVersion: "v1"
          kind: "ConfigMap"
          metadata: {
            name: context.name + "-monitor-config"
            labels: parameter.labels | {}
          }
          data: {
            "monitor-agent.conf": """
              agent.name: \(parameter.agentName)
              agent.endpoint: \(parameter.agentEndpoint)
              # Other internal monitoring specific configurations
              """
          }
        }

        // Now, we need to inject this ConfigMap as a volume and volumeMount
        // into the workload's pod template.

        // Add a new volume to the workload's PodSpec
        workload.volumes: [... (workload.volumes or []), {
          name: "monitor-config-volume"
          configMap: {

name: outputs.internalMonitorConfig.metadata.name // Reference our generated
ConfigMap
          }
        }]

        // Iterate through all containers and add a volumeMount
```

```
        workload.containers: [ for container in workload.containers {
          name: container.name
          volumeMounts: [... (container.volumeMounts or []), {
            name: "monitor-config-volume"
            mountPath: "/etc/monitor-agent"
            readOnly: true
          }]
          // Optional: Inject an environment variable to point to the config
          env: [... (container.env or []), {
            name: "MONITOR_AGENT_CONFIG_PATH"
            value: "/etc/monitor-agent/monitor-agent.conf"
          }]
          // This CUE idiom copies all other fields from the original
container,
          // ensuring we only add to it, not overwrite.
          _other: container
        }]
```

Explanation of Injection Logic:

- `workload.volumes: [... (workload.volumes or []), { ... }]`: This line adds a new volume definition to the `PodSpec`.
  - `...`: This is CUE's "list comprehension" or "splat" operator, used here to merge lists. It ensures that if `workload.volumes` already exists, our new volume is appended without overwriting existing ones. If `workload.volumes` is `null` or undefined, `(workload.volumes or [])` provides an empty list to start with.
  - The new volume is named `monitor-config-volume` and references our dynamically created `ConfigMap` by name.

- `workload.containers: [ for container in workload.containers { ... }]`: This is a CUE list comprehension. It iterates over each container already defined in the `PodSpec`. For each `container`:
  - `name: container.name`: We explicitly keep the container's original name.
  - `volumeMounts: [... (container.volumeMounts or []), { ... }]`: Similar to volumes, we append a new `volumeMount` for our `monitor-config-volume` to the container's existing volume mounts. It specifies the `mountPath` inside the container.
  - `env: [... (container.env or []), { ... }]`: We also add an environment variable `MONITOR_AGENT_CONFIG_PATH` to tell the agent where its configuration file is located.
  - `_other: container`: This is a powerful CUE idiom. It tells CUE to include all other fields from the original `container` object that we haven't explicitly mentioned or modified. This prevents us from accidentally removing important container properties (like `image`, `ports`, `args`, etc.).

Step 5: Deploy the TraitDefinition

Now that our `TraitDefinition` is complete, save the `trait-internal-monitor.yaml` file and apply it to your Kubernetes cluster using `kubectl`.

```
kubectl apply -f trait-internal-monitor.yaml
```

You should see output similar to:

```
traitdefinition.core.oam.dev/internal-monitor created
```

This command creates a new `CustomResourceDefinition` (CRD) in your cluster, making KubeVela aware of your custom `internal-monitor` trait.

Step 6: Create an Application Using Your Custom Trait

With the `TraitDefinition` deployed, developers can now use `type: internal-monitor` in their `Application` manifests. Create a new file named `application-with-custom-trait.yaml`:

```
# application-with-custom-trait.yaml
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-monitored-app
```

```
spec:
  components:
    - name: my-backend
      type: webservice # A built-in KubeVela component type
      properties:
        image: crccheck/hello-world:1.0
        port: 8000
      traits:
        - type: internal-monitor # Our custom trait!
          properties:
            agentName: "my-custom-agent"
            labels:
              app: my-backend
              environment: dev
        - type: ingress # Another built-in trait for exposing via Ingress
          properties:
            domain: my-monitored-app.example.com
            http:
              "/": 8000
```

Explanation:

- We define an `Application` with a `webservice` component named `my-backend`.
- Crucially, under `traits`, we specify `type: internal-monitor`.
- Under `properties` for our custom trait, we provide values for `agentName` and `labels`, demonstrating how developers configure the trait.

Deploy this application:

```
kubectl apply -f application-with-custom-trait.yaml
```

Step 7: Verify the Deployed Resources

After the application is deployed, let's inspect the generated Kubernetes resources to see the effect of our custom trait:

```
# Get the KubeVela application status
kubectl get app my-monitored-app -o yaml

# Get the Deployment created by the webservice component
kubectl get deployment my-monitored-app-my-backend -o yaml

# Get the ConfigMap created by our custom internal-monitor trait
kubectl get configmap my-monitored-app-my-backend-monitor-config -o yaml
```

You'll observe that KubeVela has:

1. Created a `Deployment` for `my-backend`.

2. Created a `ConfigMap` named `my-monitored-app-my-backend-monitor-config` with the `agentName` and `labels` you specified in the `Application` manifest.
3. Injected a volume and volume mount for this `ConfigMap` into the `Deployment`'s pod template.
4. Added the `MONITOR_AGENT_CONFIG_PATH` environment variable to the container.

All of this was achieved by simply adding `type: internal-monitor` to the application's traits, abstracting away the underlying Kubernetes details from the developer! This is the power of KubeVela definitions.

The Power of Addons: Packaging KubeVela Extensions

While Definitions allow you to create individual custom capabilities, **Addons** are KubeVela's way of bundling and distributing a collection of related capabilities. An Addon can contain:

- Multiple `ComponentDefinitions`, `TraitDefinitions`, `PolicyDefinitions`, `WorkflowStepDefinitions`.
- Helm charts for deploying controllers or operators that support these definitions.
- Raw Kubernetes manifests for other resources (e.g., `Namespace`, `ServiceAccount`).
- Even other custom resources.

Why Addons matter:

- **Discoverability:** Addons provide a centralized, versioned way to share and consume complex KubeVela extensions.
- **Ease of Installation:** Platform engineers can install a whole suite of features with a single command.
- **Version Management:** Addons can be versioned, allowing for controlled updates and rollbacks.
- **Ecosystem:** They foster an ecosystem where community and vendors can contribute ready-to-use platform capabilities.

For example, a "GitOps" Addon might include `WorkflowStepDefinitions` for interacting with Git repositories, `PolicyDefinitions` for enforcing GitOps principles, and even a Helm chart for deploying Argo CD or Flux CD if not already present.

You can browse available KubeVela addons and install them using the `vela addon` CLI command.

```
# List available addons
vela addon list

# Enable an addon (e.g., the velaux addon for the UI)
# Using KubeVela v1.14.0 as of 2026-06-22
vela addon enable velaux --version 1.14.0

# Disable an addon
vela addon disable velaux
```

⚡ **Real-world insight:** Platform teams often start with custom definitions for their unique needs, then package them into internal Addons for easier distribution and management across different environments or teams. This makes onboarding new projects or teams much more efficient and ensures consistency.

Comparing KubeVela Extensibility

Let's briefly compare KubeVela's extensibility model with other common approaches:

Feature	KubeVela Definitions + CUE	Raw Kubernetes (CRDs + Controllers)	Helm (Chart Hooks + Templates)	Argo CD (Lua/ Resource Hooks)
Abstraction	High-level, application-centric (OAM)	Low-level, Kubernetes resource-centric	Medium-level, package-centric	GitOps-centric, focused on sync/ health
Logic	CUE for schema, validation, resource generation	Go/Python for controllers, webhooks	Go templates for YAML generation	Lua scripts for health checks, resource modifications
Developer Exp.	Consume high-level traits/components	Directly write/ manage K8s YAML & custom resources	Install/ configure charts	Configure sync behavior, resource health
Complexity	CUE can have a learning curve, but powerful	High (Go programming, controller patterns, API design)	Moderate (Go templating, chart structure)	Moderate (Lua scripting, understanding Argo CD lifecycle)
Use Case	Define new application capabilities, policies, workflows for developers	Building new K8s APIs, operators	Packaging and deploying K8s applications	GitOps automation, advanced sync strategies

KubeVela's approach offers a sweet spot: powerful extensibility for platform engineers without requiring them to write full-blown Kubernetes controllers in Go, while still providing developers with a clean, application-centric interface.

Mini-Challenge: Create a Custom ComponentDefinition

Now it's your turn!

Challenge: Create a `ComponentDefinition` for a simple "Batch Job" that deploys a Kubernetes `Job` resource. This component should accept the following parameters:

- `image`: The container image for the job (string).
- `command`: A list of strings for the command to run (e.g., `["/bin/sh", "-c", "echo Hello && sleep 10"]`).
- `backoffLimit`: The number of retries before considering a job failed (integer, default to 3).

- `activeDeadlineSeconds`: The maximum time (in seconds) the job may be active before it is terminated (integer, optional).

After defining and applying the `ComponentDefinition`, create an `Application` that uses your new `batch-job` component type to run a simple `echo` command.

Hint:

- Start with a `ComponentDefinition` YAML structure, similar to our `TraitDefinition`.
- Inside `spec.schematic.cue.template`, define your `parameter` block with the specified inputs.
- Then, define the `outputs` block to render a `Job` resource. Remember to use `context.name` for the `Job`'s `metadata.name`.
- For the `Job`'s `spec.template.spec.containers` section, use the `image` and `command` from your parameters.
- Don't forget to include `backoffLimit` and `activeDeadlineSeconds` in the `Job` spec. For `activeDeadlineSeconds`, remember it's optional, so you might use a CUE conditional or a default of `_` (undefined) if not provided.

What to observe/learn: Pay attention to how the CUE template directly maps your high-level component parameters into the specific fields of the Kubernetes `Job` resource. This is the core of abstracting Kubernetes complexity for developers.

Common Pitfalls & Troubleshooting

1. **CUE Syntax Errors:** CUE is strict about syntax and types. Even a misplaced comma or incorrect indentation can lead to validation failures.
    - **Troubleshooting:** Use a CUE linter or validator (e.g., `cue vet` or an IDE plugin) to check your CUE template before applying. The KubeVela CLI also provides `vela def vet` for validating definitions, which is highly recommended.

2. **Incorrect `appliesToWorkloads`:** If your `TraitDefinition` doesn't list the correct workload types in `appliesToWorkloads`, KubeVela won't be able to attach the trait to the component, resulting in validation errors or the trait simply not applying.
    - **Troubleshooting:** Double-check the `apiVersion.kind` of the underlying Kubernetes resources your components generate (e.g., `deployments.apps`, `statefulsets.apps`). You can find these by inspecting the `WorkloadDefinition` of built-in components or the actual Kubernetes resources created.
3. **Missing `context.output` Reference (for Traits):** When writing CUE for traits, you often need to modify the primary resource generated by the component. Forgetting to reference `context.output` or referencing it incorrectly can lead to errors.
    - **Troubleshooting:** Remember `context.output` holds the primary resource. For a `Deployment`, its `PodSpec` is typically at `context.output.spec.template.spec`. Use `vela def render` with an example application to see the intermediate output.
4. **Addon Installation Issues:** Sometimes addons fail to enable due to network issues, missing CRDs, or conflicts with existing resources.
    - **Troubleshooting:** Use `vela addon status <addon-name>` to get detailed information about the addon's state. Check the KubeVela controller logs for more specific errors. You can usually find controller logs by `kubectl logs -n vela-system deploy/vela-core`.

Summary

Congratulations! You've navigated the powerful world of KubeVela extensibility. By now, you should have a solid understanding of:

- **Why extensibility is vital** for platform engineers to build opinionated, developer-friendly platforms.
- The role of the **Open Application Model (OAM)** and **CUE** in defining and rendering custom capabilities.
- The four primary types of **Definitions** (`ComponentDefinition`, `TraitDefinition`, `PolicyDefinition`, `WorkflowStepDefinition`) and their distinct purposes.
- How to craft a custom `TraitDefinition` to inject functionality into application components, abstracting Kubernetes details with CUE.

- The value of **Addons** for packaging and distributing collections of KubeVela extensions.
- How KubeVela's extensibility compares to other Kubernetes ecosystem tools.

You've moved beyond merely using KubeVela to shaping it, empowering your platform team to deliver a truly tailored application experience.

In the next chapter, we'll explore advanced deployment strategies and GitOps integration, bringing together the power of KubeVela's application model with modern continuous delivery practices.

References

- KubeVela Official Documentation (KubeVela v1.14.0)
- KubeVela Addons Overview (KubeVela v1.14.0)
- CUE Language Official Documentation
- KubeVela OAM Model for Platform Engineers (KubeVela v1.14.0)
- KubeVela Policy Definition (KubeVela v1.14.0)

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 08

KubeVela in Practice: Real-World Scenarios and Ecosystem Comparisons

KubeVela in Practice: Real-World Scenarios and Ecosystem Comparisons

Welcome back, seasoned engineers! In previous chapters, we laid the groundwork for understanding KubeVela's core components and its foundational Open Application Model (OAM). We explored how it empowers platform engineers to build a robust, extensible application delivery control plane. But where does KubeVela truly fit in the vast and often complex cloud-native landscape?

This chapter dives deep into KubeVela's practical applications, illustrating how it simplifies application deployment and management in real-world scenarios. More importantly, we'll clarify KubeVela's unique position by comparing it with other popular tools you're likely already using: raw Kubernetes manifests, Helm, Argo CD, and even custom internal platforms. By the end, you'll have a clear understanding of when KubeVela is the right tool for the job and how it complements your existing ecosystem.

KubeVela's Place in the Cloud-Native Ecosystem

Understanding KubeVela isn't just about learning its syntax; it's about discerning its value proposition amidst a rich ecosystem of tools. Let's break down how KubeVela differentiates itself and where it integrates.

KubeVela vs. Raw Kubernetes Manifests

Directly managing applications with raw Kubernetes manifests can quickly become a YAML nightmare. Why is this a problem for developers?

- **Verbosity:** Even a simple application often requires multiple manifest files (Deployment, Service, Ingress, ConfigMap, Secret).
- **Complexity:** Understanding the intricate relationships between these resources, especially for complex applications, is a steep learning curve.
- **Repetition:** Many configurations are boilerplate, duplicated across applications or environments.

- **Limited Abstraction:** Kubernetes resources are infrastructure-centric, not application-centric. They describe how to run containers, not what the application actually is or how it should be delivered.

KubeVela's Solution: Application-Centric Abstraction

KubeVela, built on OAM, provides a higher-level, application-centric abstraction. Instead of dealing with individual Kubernetes resources, developers interact with an `Application` resource. This `Application` combines `Components` (the core services), `Traits` (operational capabilities like scaling or routing), `Policies` (governance rules), and `Workflows` (delivery steps).

📌 **Key Idea:** KubeVela moves the focus from infrastructure details to application intent, drastically reducing developer burden and YAML boilerplate.

Why it Matters for Platform Teams:

Platform teams define these `ComponentDefinition`s, `TraitDefinition`s, and `PolicyDefinition`s once. Developers then consume these well-defined building blocks, leading to:

- **Improved Developer Experience:** Developers declare what their application needs, not how to provision every Kubernetes resource.
- **Standardization:** Consistent application definitions across the organization.
- **Reduced Errors:** Less manual YAML writing means fewer typos and configuration mistakes.
- **Faster Delivery:** Developers can deploy complex applications with minimal YAML.

Consider a simple web service. With raw Kubernetes, you might need a `Deployment`, a `Service`, and an `Ingress`. With KubeVela, this can be expressed as a single `Application` resource.

Here's a conceptual KubeVela `Application` for a web service. Notice how it encapsulates the details of a deployment, scaling, and ingress within a single, readable structure:

```
# KubeVela Application for a web service (conceptual example)
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-web-app
spec:
  components:
    - name: frontend
```

```
    type: webservice # 'webservice' is a pre-defined ComponentDefinition
    properties:
      image: my-registry/my-web-app:v1.0
      port: 8080
    traits:
      - type: scaler # 'scaler' is a pre-defined TraitDefinition
        properties:
          replicas: 3
      - type: ingress # 'ingress' is another pre-defined TraitDefinition
        properties:
          domain: my-web-app.example.com
          http:
            /: 8080
```

This single KubeVela `Application` manifest achieves what would typically require multiple, more verbose Kubernetes YAML files for `Deployment`, `Service`, and `Ingress`. KubeVela encapsulates the operational details, allowing developers to focus on their application code and intent.

KubeVela vs. Helm

What is Helm?

Helm is the de facto package manager for Kubernetes. It allows you to define, install, and upgrade even the most complex Kubernetes applications using "charts." A Helm chart is a collection of files that describe a related set of Kubernetes resources, often with templating capabilities.

KubeVela's Perspective: Complement, Not Replace

KubeVela doesn't aim to replace Helm; it often leverages Helm. Helm charts are excellent for packaging Kubernetes operators, controllers, or even entire applications. KubeVela can integrate these charts as `Components`, treating them as deployable units.

Key Differences:

- **Scope:** Helm is primarily a packaging and templating tool for Kubernetes resources. KubeVela is an application delivery control plane that orchestrates the entire lifecycle of an application, from definition to deployment, across environments and clusters.
- **Abstraction Layer:** Helm still operates close to raw Kubernetes manifests (though templated). KubeVela provides an application-centric abstraction on top of Kubernetes, using OAM.

- **Functionality:** KubeVela extends beyond mere packaging by offering:
  - **Workflows:** Define complex delivery pipelines (e.g., canary rollouts, manual approvals, custom steps).
  - **Policies:** Enforce governance rules (e.g., security, cost, compliance, environment-specific overrides).
  - **Multi-cluster/Hybrid-cloud:** Manage deployments across diverse infrastructures from a single control plane.
  - **Extensibility:** Easily define new `Component`s, `Trait`s, `Policy`s, and `WorkflowStepDefinition`s without modifying KubeVela's core.

⚡ **Quick Note:** A common pattern is for platform teams to define KubeVela `ComponentDefinition`s that wrap existing Helm charts. This allows developers to consume complex applications packaged by Helm through KubeVela's simpler, application-centric interface, while still benefiting from KubeVela's advanced delivery capabilities.

KubeVela vs. Argo CD (GitOps Tools)

What is Argo CD?

Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes. It automates the deployment of desired application states defined in Git repositories to specified Kubernetes clusters. Its core function is to synchronize the live state of applications with the state defined in Git.

How They Differ and Integrate:

- **Core Purpose:**
  - **Argo CD:** Focuses on GitOps synchronization—ensuring that what's in your Git repository is exactly what's running in your cluster. It's about maintaining desired state from a source of truth (Git).
  - **KubeVela:** Focuses on application definition and delivery logic—defining what an application is, how it should be delivered, and what policies govern its deployment. It's about abstracting the complexity of Kubernetes for developers and orchestrating advanced delivery.
- **Abstraction Level:** Argo CD operates on Kubernetes manifests (or Helm charts, Kustomize, etc.). KubeVela operates on its higher-level `Application` resource, which then renders into Kubernetes manifests.

- **Integration:** These tools are highly complementary and often used together.
    - You can define your KubeVela `Application` resources in a Git repository.
    - Argo CD can then be configured to synchronize these KubeVela `Application`s to your Kubernetes cluster.
    - KubeVela takes over from there, interpreting the `Application` and executing its defined `Workflow`s and `Policies` to deploy the actual Kubernetes resources.

🧠 **Important:** KubeVela defines the application model and delivery process. Argo CD ensures that the KubeVela application definition in Git is consistently applied to the cluster.

KubeVela vs. Custom Internal Platforms

The Challenge of Custom Platforms:

Many organizations, especially larger enterprises, end up building their own internal developer platforms (IDPs) to abstract Kubernetes complexity. This often involves:

- Developing custom YAML generators or abstraction layers.
- Writing bespoke controllers or operators.
- Maintaining complex scripts for deployment workflows.
- Significant engineering effort to build and maintain.

KubeVela's Role: An Open-Source Framework for Building Platforms

KubeVela offers an open-source, extensible framework that platform teams can use as the foundation for their IDP, rather than starting from scratch.

Benefits:

- **Reduced Development Burden:** KubeVela provides the OAM abstraction layer, component model, workflow engine, and policy engine out-of-the-box. This significantly reduces the amount of custom code platform teams need to write.
- **Standardization:** By adopting OAM, platform teams leverage an open standard, avoiding vendor lock-in and benefiting from a growing ecosystem.

- **Extensibility:** KubeVela is designed for extensibility. Platform teams can easily define custom `ComponentDefinition`s, `TraitDefinition`s, `PolicyDefinition`s, and `WorkflowStepDefinition`s to encapsulate their unique operational knowledge and best practices.
- **Community Support:** Being open-source, KubeVela benefits from community contributions, bug fixes, and feature enhancements.

🔥 **Optimization / Pro tip:** Instead of building a bespoke platform from the ground up, platform teams can adopt KubeVela, customize it with their specific definitions and workflows, and integrate it with their existing toolchain. This allows them to focus their engineering talent on unique business value rather than undifferentiated heavy lifting.

Real-World Application Scenarios with KubeVela (Step-by-Step Implementation)

Let's explore how KubeVela helps solve common challenges in modern application delivery through practical examples.

Multi-Environment Deployment

Imagine you have an application that needs to be deployed across development, staging, and production environments. Each environment might require different replica counts, resource limits, or ingress rules.

KubeVela Approach:

KubeVela excels here by allowing you to define environment-specific configurations using `policy` definitions directly within your `Application` resource. This keeps the application definition centralized but allows for environmental variations to be applied at deployment time.

Here's an example of an application with a workflow step that applies a `staging-policy`:

```
# Example: Application with environment-specific policy for 'staging'
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: multi-env-app
spec:
  components:
    - name: backend
      type: webservice
      properties:
        image: my-registry/app-backend:v1.0
  workflow:
    steps:
      - name: deploy-dev
```

```
        type: deploy
        properties:
          # This step deploys to the default cluster/namespace for 'dev'.
          # Policies would be applied based on the dev environment context if
defined.
      - name: manual-review
        type: suspend # Pauses the workflow, requiring manual intervention
      - name: deploy-staging
        type: deploy
        properties:
          # This step explicitly applies the 'staging-policy' for deployment.
          policy:
            - name: staging-policy
          cluster: staging-cluster # Assuming 'staging-cluster' is registered
with KubeVela
  policies:
    - name: staging-policy
      type: override # The 'override' policy type allows modifying component
properties
      properties:
        components:
          - name: backend
            traits:
              - type: scaler
                properties:
                  replicas: 2 # Staging gets 2 replicas
              - type: ingress
                properties:
                  domain: staging.my-app.example.com # Staging-specific domain
```

Explanation:

1. `components`: We define a `backend` component of type `webservice` with a base image.
2. `workflow`:
    - `deploy-dev`: A simple `deploy` step for the development environment.
    - `manual-review`: A `suspend` step, which pauses the workflow and requires a manual resume, often used for critical gates.
    - `deploy-staging`: Another `deploy` step. Crucially, it references `policy: - name: staging-policy` and specifies a `cluster: staging-cluster`. This tells KubeVela to apply the named policy and deploy to that specific cluster.

3. `policies`:
    - `staging-policy`: This is an `override` type policy.
    - `properties.components`: It targets the `backend` component.
    - `traits`: Within the `backend` component, it overrides the `scaler` trait to set `replicas: 2` and the `ingress` trait to set a `domain: staging.my-app.example.com`.

This setup ensures that when the `deploy-staging` step runs, the `backend` component is deployed with 2 replicas and the staging domain, even though the base component definition might specify something different or nothing at all for these traits.

Hybrid/Multi-Cloud Delivery

Modern applications often need to span multiple Kubernetes clusters, whether for disaster recovery, geographical distribution, or leveraging specific cloud provider services.

KubeVela Approach:

KubeVela simplifies multi-cluster management by allowing you to register external Kubernetes clusters. You can then use `topology` policies within your `Application` workflow to specify where components should be deployed.

```
# Example: Deploying components to different clusters
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: hybrid-app
spec:
  components:
    - name: api-gateway
      type: webservice
      properties:
        image: my-registry/api-gateway:v1.0
    - name: data-processor
      type: worker
      properties:
        image: my-registry/data-processor:v1.0
  workflow:
    steps:
      - name: deploy-gateway-us-east
        type: deploy
        properties:
          component: api-gateway
          policy:
            - name: us-east-placement
      - name: deploy-processor-eu-west
        type: deploy
        properties:
          component: data-processor
          policy:
```

```
            - name: eu-west-placement
  policies:
    - name: us-east-placement
      type: topology # The 'topology' policy specifies target clusters
      properties:
        clusters: ["us-east-cluster"] # Assumes 'us-east-cluster' is
registered with KubeVela
    - name: eu-west-placement
      type: topology
      properties:
        clusters: ["eu-west-cluster"] # Assumes 'eu-west-cluster' is
registered with KubeVela
```

Explanation:

1. `components`: We define two distinct components: `api-gateway` (a `webservice`) and `data-processor` (a `worker`).
2. `workflow`:
    - `deploy-gateway-us-east`: This step targets the `api-gateway` component and applies the `us-east-placement` policy.
    - `deploy-processor-eu-west`: This step targets the `data-processor` component and applies the `eu-west-placement` policy.
3. `policies`:
    - `us-east-placement`: A `topology` policy that restricts deployment to the `us-east-cluster`.
    - `eu-west-placement`: Another `topology` policy, restricting deployment to the `eu-west-cluster`.

⚡ **Real-world insight:** KubeVela's `topology` policy allows you to distribute application components across different clusters and regions from a single `Application` definition, abstracting the underlying infrastructure complexity for developers. This is invaluable for disaster recovery, data residency requirements, and latency optimization.

Customizing Delivery Workflows

Beyond simple deployments, real-world applications often require complex delivery workflows: canary releases, blue/green deployments, manual approvals, security scans, data migrations, or even custom notification steps.

KubeVela Approach:

KubeVela's `Workflow` engine is highly extensible. You can define custom `WorkflowStepDefinition`s to encapsulate any arbitrary logic, which can then be used as steps in your application's delivery workflow. This allows platform teams to codify complex operational procedures into reusable building blocks.

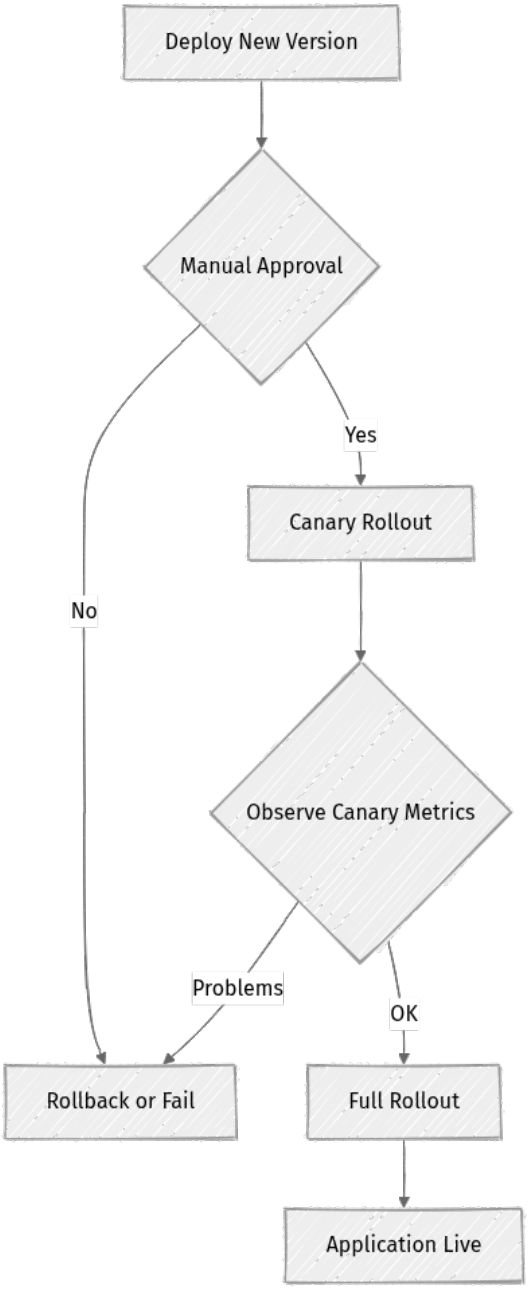
Here's an example of an application workflow that includes custom steps for manual approval and a canary rollout:

```
# Example: A workflow with a custom approval step and canary rollout
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: advanced-workflow-app
spec:
  components:
    - name: webservice
      type: webservice
      properties:
        image: my-registry/my-app:v1.1 # New version to deploy
  workflow:
    steps:
      - name: deploy-new-version
        type: deploy
        properties:
          component: webservice

# This step creates a new deployment for v1.1, potentially alongside the old
version.
      - name: manual-approval
        type: suspend
# This step pauses the workflow, requiring manual resume via 'vela workflow
resume <app-name>'
        properties:
          message: "New version deployed, please verify and approve for
canary rollout."
      - name: canary-rollout
        type: canary-rollout # Assuming 'canary-rollout' is a custom
WorkflowStepDefinition
        properties:
          component: webservice
          trafficWeight: 20 # Send 20% traffic to new version (v1.1)
          duration: 5m # Wait 5 minutes to observe behavior
      - name: observe-canary
        type: suspend # Another manual gate to monitor canary metrics
        properties:
          message: "Canary active. Monitor metrics. Approve for full rollout."
      - name: full-rollout
        type: rollout # Assuming 'rollout' is another custom
WorkflowStepDefinition
        properties:
          component: webservice
          trafficWeight: 100 # Shift all traffic to new version (v1.1)
```

Workflow Visualization:

To better understand the flow, let's visualize this with a Mermaid diagram:



Explanation:

1. **`deploy-new-version`**: This `deploy` step initiates the deployment of the `webservice` component with the `v1.1` image.
2. **`manual-approval`**: A `suspend` step is used to introduce a mandatory pause. The workflow will wait indefinitely until a platform engineer manually resumes it after verifying the initial deployment.
3. **`canary-rollout`**: This step (which would be defined by a custom `WorkflowStepDefinition`) initiates a canary deployment, routing 20% of traffic to the new `v1.1` version for a `5m` duration.

4. **`observe-canary`**: Another `suspend` step, providing a window for monitoring metrics and logs of the canary deployment.
5. **`full-rollout`**: If approved, this step (another custom `WorkflowStepDefinition`) shifts 100% of traffic to the new `v1.1` version, completing the rollout.

⚠️ **What can go wrong:** While powerful, overly complex workflows can become difficult to debug and manage. Design workflows with atomic, well-defined steps. Use `suspend` steps for critical manual gates, and ensure your custom `WorkflowStepDefinition`s are thoroughly tested and idempotent.

Mini-Challenge: Adapting an Application for a New Environment

Let's put your understanding to the test. This challenge will reinforce how KubeVela's policies provide a powerful mechanism to manage environment-specific configurations without duplicating entire application definitions. You'll see how a single application can be "mutated" at deployment time based on the target environment.

Challenge:

You have a simple KubeVela `Application` for a web service deployed to a `dev` environment. Now, adapt this application to deploy to a `staging` environment. The `staging` environment should have:

1. Exactly `2` replicas for the `webservice` component.
2. An `ingress` host of `staging.my-app-domain.com`.
3. The base image should remain `nginx:latest`.

You'll need to define an `Application` with a `workflow` that includes a `deploy` step specifically for staging, and a `policy` to apply these staging-specific overrides.

Hint:

Focus on defining an `override` `Policy` that targets your `frontend-service` component and modifies its `scaler` and `ingress` traits. Then, ensure your workflow step correctly applies this policy.

```
# Your starting point:
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: my-challenge-app
spec:
  components:
    - name: frontend-service
```

```
      type: webservice
      properties:
        image: nginx:latest
        port: 80
      traits:
        - type: scaler
          properties:
            replicas: 1
        - type: ingress
          properties:
            domain: dev.my-app-domain.com
            http:
              /: 80
  workflow:
    steps:
      - name: deploy-dev
        type: deploy
        properties:
          # This deploys to the default cluster/namespace.
          # You'll add another step and policy for staging below.
```

(Solution provided in next chapter or separate solution guide.)

Common Pitfalls & Troubleshooting in KubeVela Adoption

Adopting a new platform like KubeVela can come with its own set of challenges. Here are a few common pitfalls and how to navigate them.

Over-abstraction or Under-abstraction

What can go wrong:

Finding the right balance when defining your `ComponentDefinition`s and `TraitDefinition`s is crucial.

- **Over-abstraction:** If your definitions are too generic or hide too many details, developers might lose necessary control or clarity, leading to frustration. They might need to escape the abstraction to achieve specific Kubernetes configurations, defeating the purpose.
- **Under-abstraction:** If definitions expose too many Kubernetes-specific details, you haven't truly simplified the developer experience, and you're back to managing verbose YAML, just wrapped in OAM.

How to avoid/troubleshoot:

- **Iterate with Developers:** Work closely with your developer teams. Start with a few simple, high-value abstractions. Gather feedback on what details they need to control and what can be safely abstracted away.

- **Focus on Intent:** Definitions should describe the intent (e.g., "a scalable web service," "a public endpoint"), not the low-level Kubernetes resources (e.g., "a Deployment with HPA and an Ingress").
- **Leverage Existing Addons:** Before creating a custom definition, check if KubeVela's extensive addon library (available via `vela addon enable --help` or the KubeVela website) already provides a suitable solution.

Misunderstanding Component vs. Trait vs. Policy

What can go wrong:

The OAM model's separation of concerns (Component for primary workload, Trait for operational capabilities, Policy for governance) can be initially confusing. Misplacing configurations can lead to:

- **Unintended side effects:** A Trait trying to do a Component's job, or a Policy interfering with a Trait in unexpected ways.
- **Hard-to-debug applications:** When the responsibilities are blurred, tracing issues and understanding the application's true state becomes difficult.

How to avoid/troubleshoot:

- **Review OAM Principles:** Regularly revisit the core OAM tenets.
    - **Component:** Defines the application's core workload (e.g., a container, a Helm chart, a cloud service).
    - **Trait:** Adds operational capabilities to a component (e.g., scaling, ingress, volume mounts, monitoring). Traits modify or enhance a component.
    - **Policy:** Enforces governance rules or overrides across the application or its components (e.g., security constraints, cost limits, environment-specific changes, placement rules). Policies govern or mutate the application's definition or deployment.
- **Clear Definition Schemas:** Ensure your `Definition`s have clear schemas and descriptions that guide users on how to use them correctly.
- **Start Simple:** Begin with basic components and traits, then gradually introduce more complex ones as your team gains familiarity.

Workflow Failures

What can go wrong:

KubeVela workflows orchestrate the delivery process. Failures in a workflow step can halt deployments, especially if custom `WorkflowStepDefinition`s are involved. Debugging complex, multi-step workflows can be challenging without proper tools and practices.

How to avoid/troubleshoot:

- **Monitor Workflow Status:** Regularly check the status of your `Application` and its associated `Workflow` using `kubectl vela ls` or `kubectl vela status <app-name>`. These commands provide visibility into which step is currently executing or has failed.
- **Examine Events and Logs:** KubeVela's workflow engine leverages Kubernetes events. Check `kubectl describe application <app-name>` for event logs and the logs of the KubeVela controller (usually in the `vela-system` namespace) for detailed error messages.
- **Test Custom Steps:** If you create custom `WorkflowStepDefinition`s, thoroughly test them in isolated environments before integrating into critical production workflows. Implement robust error handling within these custom steps.
- **Idempotency:** Design workflow steps to be idempotent, meaning they can be run multiple times without causing unintended side effects. This is crucial for recovery from partial failures and for resuming workflows.

Summary

Congratulations! You've reached the end of our KubeVela journey. We've seen how KubeVela stands as a powerful application delivery control plane, built to simplify the complexities of Kubernetes for developers while empowering platform teams with a robust, extensible framework.

Here are the key takeaways from this chapter:

- **KubeVela abstracts Kubernetes:** It provides an application-centric view, reducing the YAML burden and cognitive load for developers compared to raw Kubernetes manifests.
- **Complements, not replaces Helm:** KubeVela can consume Helm charts as components, adding advanced delivery workflows and policies on top of Helm's packaging capabilities.
- **Integrates with GitOps tools like Argo CD:** KubeVela defines how an application is delivered and managed, while Argo CD ensures the application definition in Git is synchronized to the cluster. They work hand-in-hand.

- **A framework for custom platforms:** KubeVela offers an open-source, extensible foundation for building internal developer platforms, saving engineering effort compared to building from scratch.
- **Solves real-world challenges:** KubeVela provides elegant solutions for multi-environment deployments, hybrid/multi-cloud delivery, and complex custom delivery workflows.
- **Balance is key:** Successful KubeVela adoption requires finding the right level of abstraction and clearly defining component, trait, and policy responsibilities.

KubeVela represents a significant step forward in making cloud-native application delivery more accessible and manageable. By leveraging its capabilities, platform teams can provide a streamlined, consistent, and powerful experience for their developers, accelerating innovation and reducing operational overhead.

References

- KubeVela Official Documentation
- Open Application Model (OAM) Specification
- KubeVela Addons Overview
- Kubernetes Official Documentation
- Helm Official Documentation
- Argo CD Official Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.