Kubernetes, OpenShift, and Helm: A Beginner's Guide

Kubernetes, OpenShift, and Helm form a powerful trio for modern cloud-native development and operations. This guide introduces Kubernetes core concepts, compares OpenShift with upstream Kubernetes, and explores Helm for package management. Designed for tech-savvy beginners, it blends developer and DevOps perspectives with clear language, code examples, and practical tips to build a solid foundation.

A by Atish Kumar Sinha

Introduction to Kubernetes

Container Orchestration

Kubernetes (K8s) is an open-source platform that automates the deployment, scaling, and management of containerized applications, originally developed by Google and now maintained by the Cloud Native Computing Foundation.

Portability

Kubernetes abstracts away underlying infrastructure. You can run it on your laptop (Minikube), on-premises, or on cloud providers (AWS, Azure, GCP) with minimal changes.

Self-Healing

If a container or machine fails, Kubernetes can replace it automatically. It can also scale out more instances if demand increases, optimizing resource usage.

Core Kubernetes Concepts: Pods and Deployments

Pods

A Pod is the smallest deployable unit in Kubernetes. It represents one or more containers that should be managed as a single entity. Think of a Pod as a "wrapper" around containers.

All containers in a Pod share certain contexts like network (same IP address and ports) and optionally storage volumes. Pods are ephemeral - if a Pod dies, Kubernetes creates a new Pod with a new IP to replace it.

Deployments

A Deployment ensures a desired number of Pods are running, using a template for those Pods. It's primarily for stateless applications that don't need to remember state between restarts.

With Deployments, you declare your desired state (e.g., "I want 3 Pods running version 1.2 of my app"). If you update the Deployment, Kubernetes performs a rolling update, ensuring zero-downtime upgrades.

Core Kubernetes Concepts: Services and Configuration

Services

A Service is a stable endpoint (virtual IP and DNS name) that fronts a group of Pods. It provides a single IP address and automatically routes traffic to member Pods. Even if Pods come and go, the Service's IP stays the same, decoupling clients from Pods.

Service types include ClusterIP (internal), NodePort (node-level access), LoadBalancer (external), and ExternalName (DNS mapping).

ConfigMaps

ConfigMaps store non-confidential configuration data in key-value pairs. They decouple environment-specific configs from container images, allowing the same image to be used across environments with different configurations.

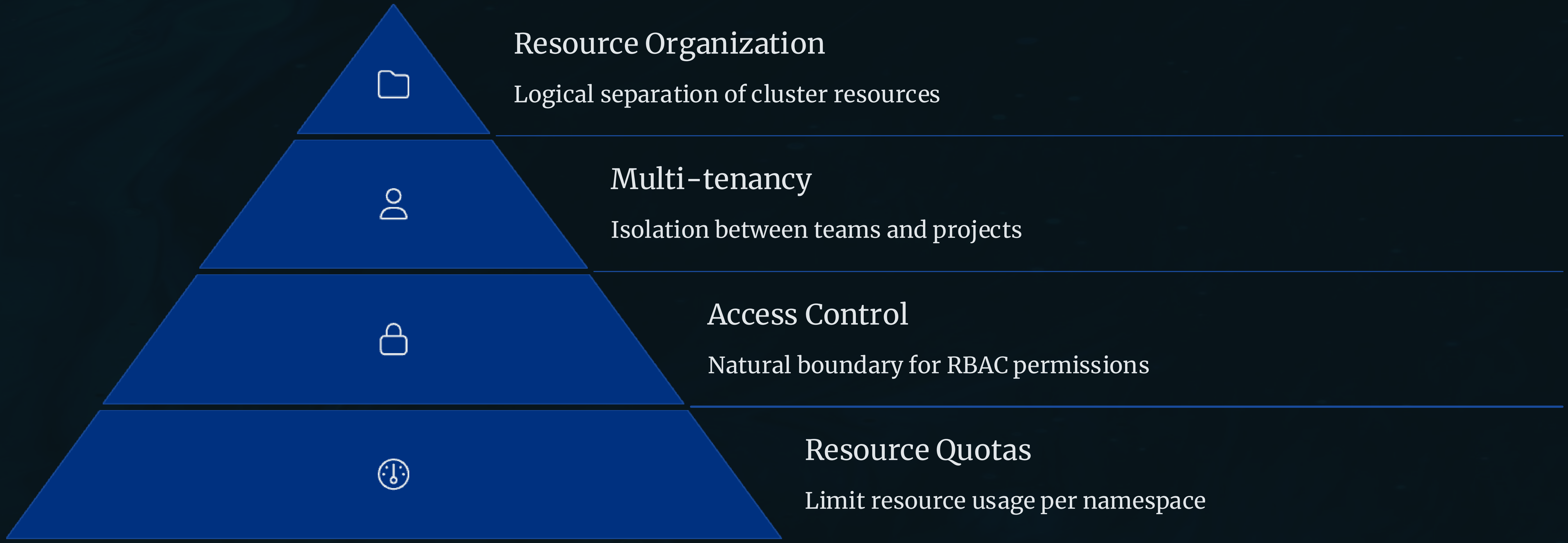
Pods can mount ConfigMaps as files or expose them as environment variables. They are not encrypted and should not contain sensitive data.

Secrets

Secrets store sensitive data like passwords, API tokens, and SSH keys. They're similar to ConfigMaps but intended for confidential information. By default, they're stored base64-encoded in etcd, though not truly encrypted without additional settings.

Using Secrets means you don't bake passwords into images, reducing risk of accidental leaks.

Namespaces and Resource Organization



Resource Organization

Logical separation of cluster resources

Multi-tenancy

Isolation between teams and projects

Access Control

Natural boundary for RBAC permissions

Resource Quotas

Limit resource usage per namespace

A Kubernetes Namespace is like a logical cluster inside a cluster – it isolates names of resources and helps divide cluster resources among multiple users or groups. Within a namespace, names must be unique, but different namespaces can have objects with the same name.

Namespaces are commonly used to separate environments (dev, staging, prod) or teams/projects. If you don't specify one, Kubernetes uses the "default" namespace. System namespaces include "kube-system" for system components and "kube-public."

OpenShift vs. Kubernetes: Similarities

Kubernetes Core

OpenShift uses Kubernetes as its orchestration engine with all core concepts intact

Container Images

Both use standard container images (Docker/OCI)

Multi-Environment

Both run on various infrastructures (cloud, on-prem, hybrid)

Cloud-Native Ecosystem

Both leverage the broader CNCF ecosystem tools

OpenShift includes Kubernetes at its core but adds many components to make a more complete enterprise platform. To an application developer, deploying to OpenShift feels like deploying to Kubernetes – you still write YAML manifests or use kubectl/oc commands.

In short: OpenShift is Kubernetes++ (Kubernetes plus extras). If you know Kubernetes, you're a long way toward understanding OpenShift. But the "extras" matter, especially for enterprises.

OpenShift vs. Kubernetes: Key Differences

Aspect	**Kubernetes**	**OpenShift**
Deployment & Setup	DIY, manual setup required	Automated installer, push-button installation
Networking	Pluggable (Calico/others)	Built-in SDN and route management
Security	Baseline security	Enforced security with stricter defaults
Registry	External needed	Included internal registry
CI/CD	External tools needed	Integrated Jenkins & Tekton, S2I, webhooks
GUI	Optional Dashboard	Full web console built-in
Support	Community support	Vendor SLA support with subscription

OpenShift's user interface is very developer-friendly with visualization of projects, workloads, and integrated source-to-image capabilities. It enforces stricter security defaults including Security Context Constraints (SCCs) to restrict running containers as root.

For enterprises, OpenShift provides integrated CI/CD with OpenShift Pipelines (Tekton) and Source-to-Image for building container images directly from source. It also includes monitoring, logging, and compliance features out of the box.

Deep Dive into Helm

Package Management

Helm is the package manager for Kubernetes

Templating

Parameterize YAML with variables and functions

Release Management

Track installations and handle upgrades/rollbacks

Chart Repository

Share and reuse application definitions

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications. It uses "charts" - collections of files that describe a related set of Kubernetes resources. Charts contain templates that generate standard Kubernetes YAML files when combined with values.

With Helm, you can install pre-packaged applications, customize them with values files, and manage their lifecycle including upgrades and rollbacks. Helm charts can be shared through repositories, making it easy to distribute and reuse application definitions across teams and organizations.

Enterprise Considerations and Next Steps

Learn Kubernetes

Master core concepts and YAML

Explore OpenShift

Understand enterprise additions

Practice with Helm

Package and deploy applications

Deploy to Production

Apply best practices at scale

When choosing between Kubernetes and OpenShift, consider your organization's needs. If you want pure flexibility, community support, and cutting-edge features, vanilla Kubernetes is great. If you need an integrated platform with enterprise support, OpenShift adds significant convenience despite its cost.

Both share Kubernetes DNA, so learning one helps with the other. OpenShift is a product with more pre-integrated features and support, while Kubernetes is a project requiring assembly of components. For enterprise production with stringent requirements, OpenShift can be attractive. For cloud-native teams who prefer custom stacks, vanilla Kubernetes may suffice.

Kubernetes App Deployment with YAML: A Comprehensive Guide

This presentation explores YAML files used in Kubernetes for app deployment.

You'll learn differences between Deployment and StatefulSet objects.

We'll cover configuration, examples, and practical usage tips.

A by Atish Kumar Sinha

Introduction to Kubernetes and YAML

Kubernetes

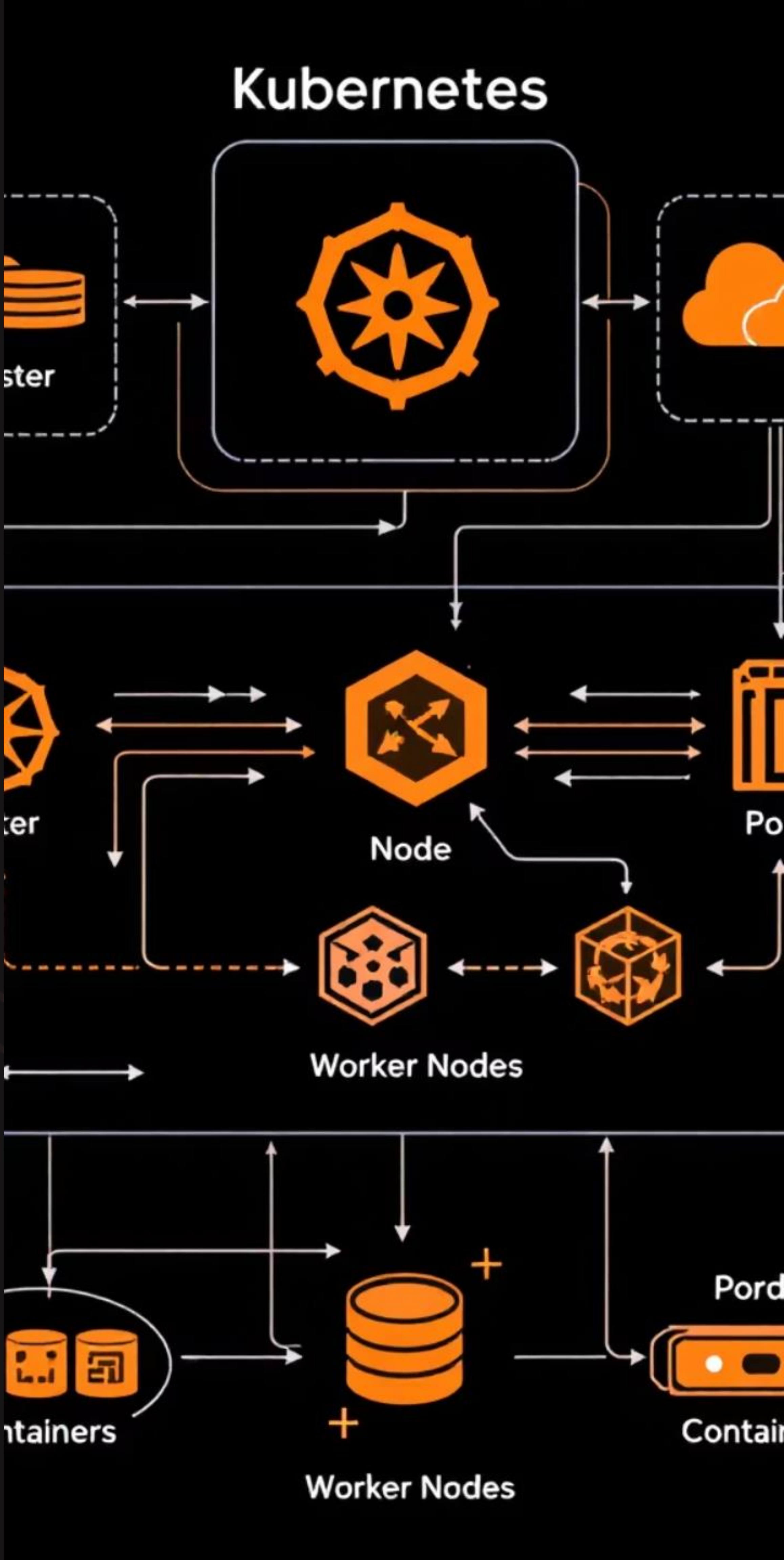
Open-source platform for automating app deployment and scaling

Key Components

Pods, Services, Deployments, StatefulSets manage container lifecycle

YAML Format

Human-readable files define Kubernetes objects declaratively



Core Kubernetes Objects Defined with YAML

Pods

Smallest deployable units encapsulating containers

Example: apiVersion v1, kind Pod, metadata name my-pod

Services

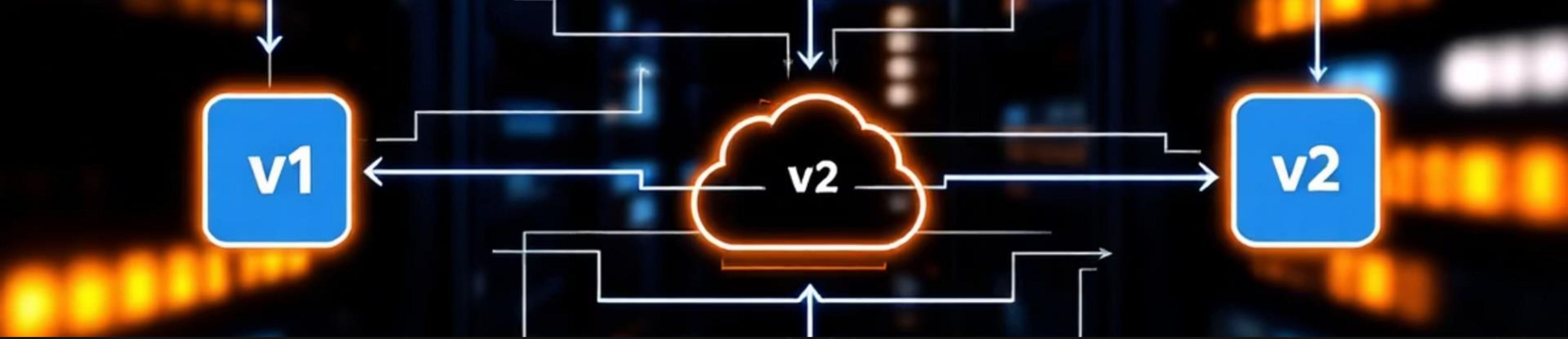
Expose applications running inside Pods to networks

Example: apiVersion v1, kind Service, spec ports 80 to 8080

Deployments

Manage replica sets for stateless applications

Example: apiVersion apps/v1, kind Deployment, replicas 3



Deployment YAMLs: Scaling and Updates

replicas

Number of desired Pod instances

selector

Matches Pods managed by the Deployment

strategy

RollingUpdate for gradual replacement; or Recreate for full restart

template

Defines Pod specification including containers and labels

StatefulSet YAMLs: Stable, Unique Network IDs

Stateful Applications

Manage databases and apps needing persistent identity and data

Persistent Volumes

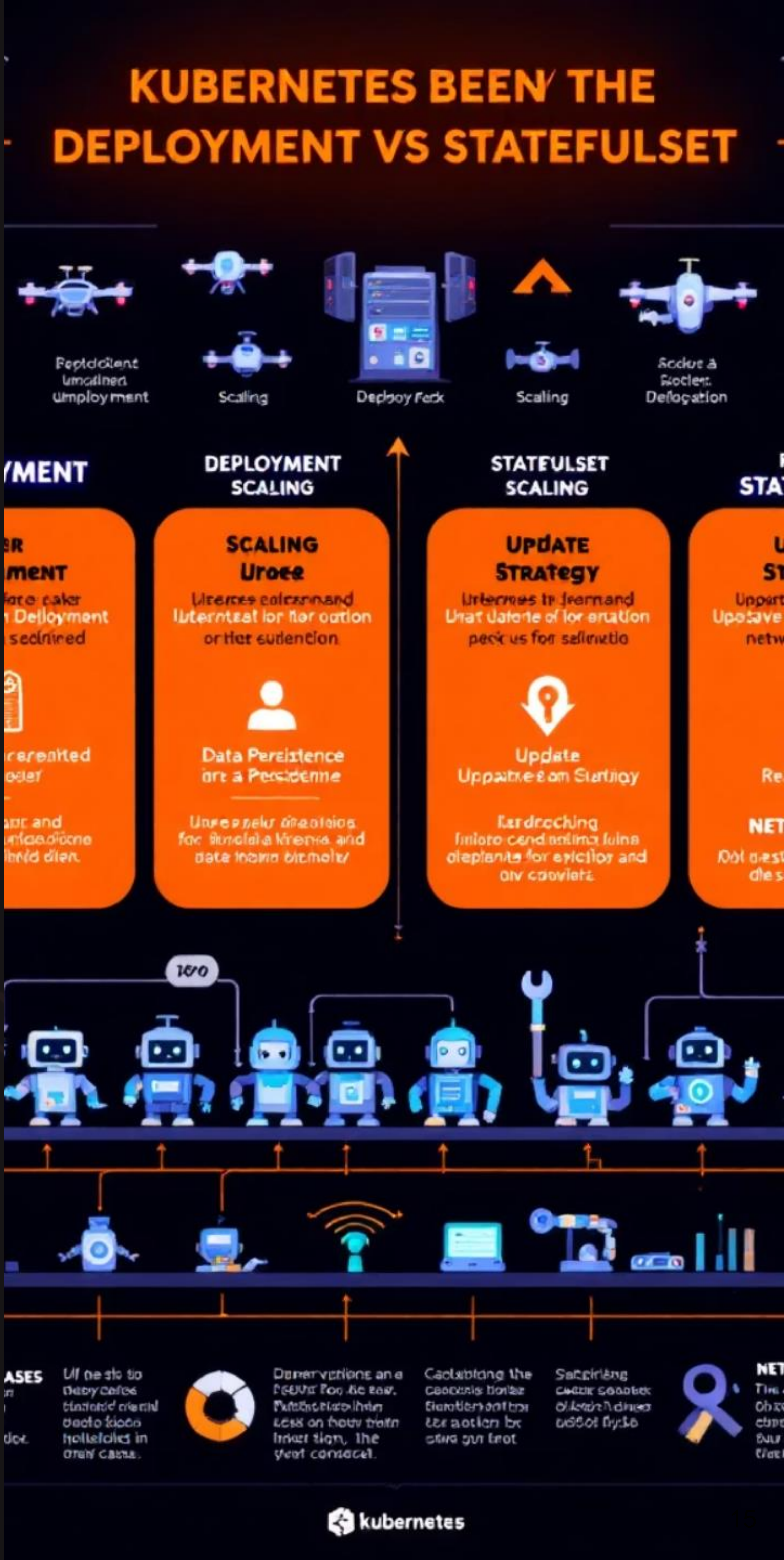
Each Pod gets its own persistent storage via volumeClaimTemplates

Stable Network IDs

Pods have ordered, unique names backed by a headless Service

Deployment vs. StatefulSet: Key Differences

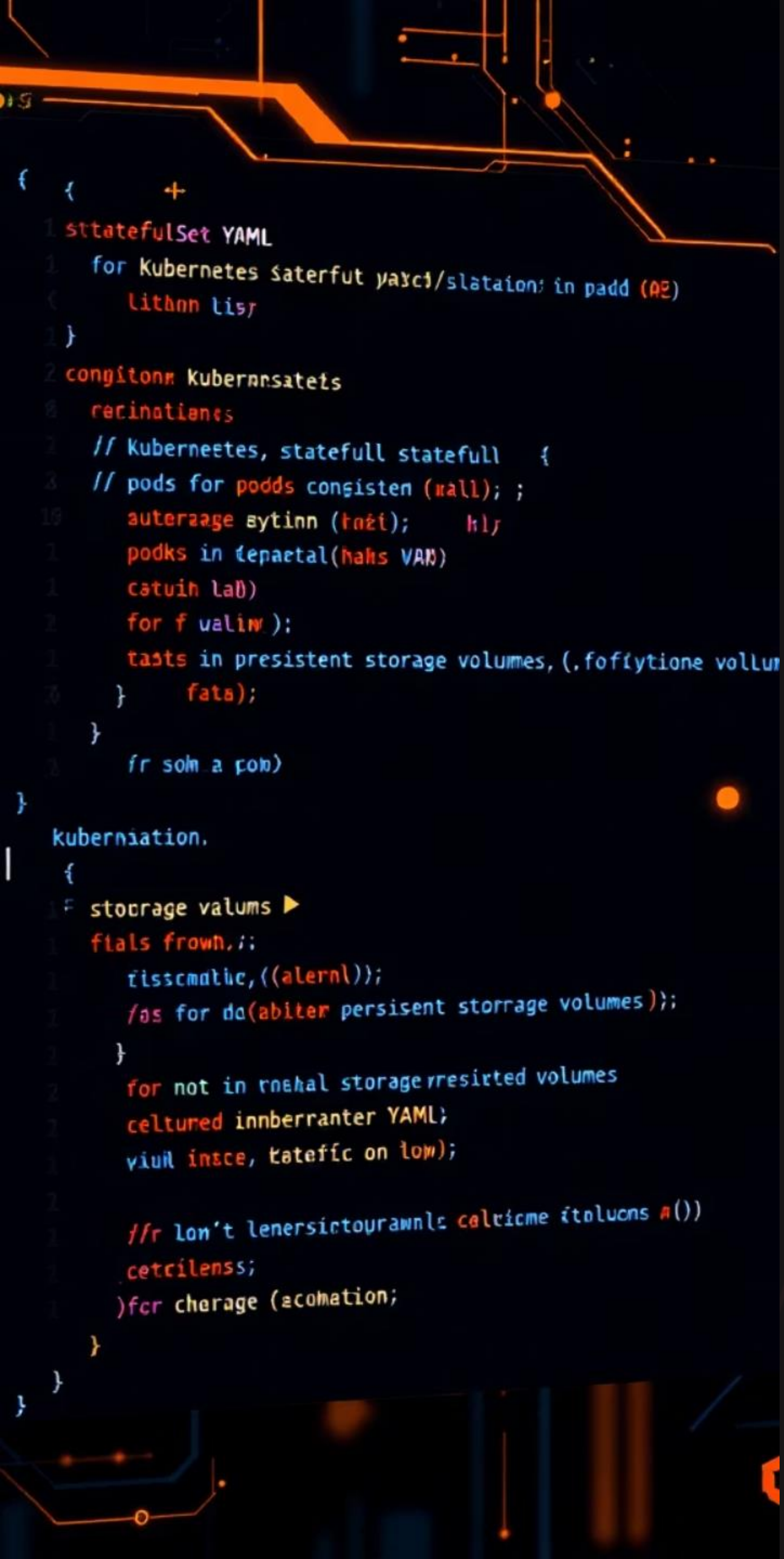
Feature	Deployment	StatefulSet
Pod Identity	Random names	Stable ordinal index
Volume Management	Ephemeral storage	Persistent volumes
Scaling	unordered scaling	Ordered scaling and updates
Use Case	Stateless apps	Stateful apps like databases



Example Deployment YAML

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: web-container
        image: nginx:latest
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
```

peroment (Kubernetes delpbwenc(
repliicas.: 3
repliicans: 3,
faft :: insttinge instning";
instaate lunning);
replica yAMm. idnanional kueperiatally
Kubernetes: yAMLL
rerpliction: (replicas. 3;)
this rolling updates undtate
(replient;
applications, atpliians_of_yall>
In inslitans;,
rolling trever tupdate
patter repliction,
talm:(con uption)!



Example StatefulSet YAML

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: db-statefulset
spec:
  serviceName: "db-service"
  replicas: 3
  selector:
    matchLabels:
      app: db
  template:
    metadata:
      labels:
        app: db
    spec:
      containers:
      - name: db-container
        image: postgres:latest
        volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql/data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Example of Deploying the Application

1

Prepare YAML Files

Create and validate Deployment or StatefulSet YAML file

2

Apply Deployment

Run kubectl apply -f deployment.yaml to deploy app

3

Verify Pods

Monitor pods with kubectl get pods and logs

4

Update or Scale

Edit YAML and reapply or use kubectl scale command

Helm Charts and Kubernetes Simplification

Helm Package Manager

Simplifies deployment by templating and packaging YAML files

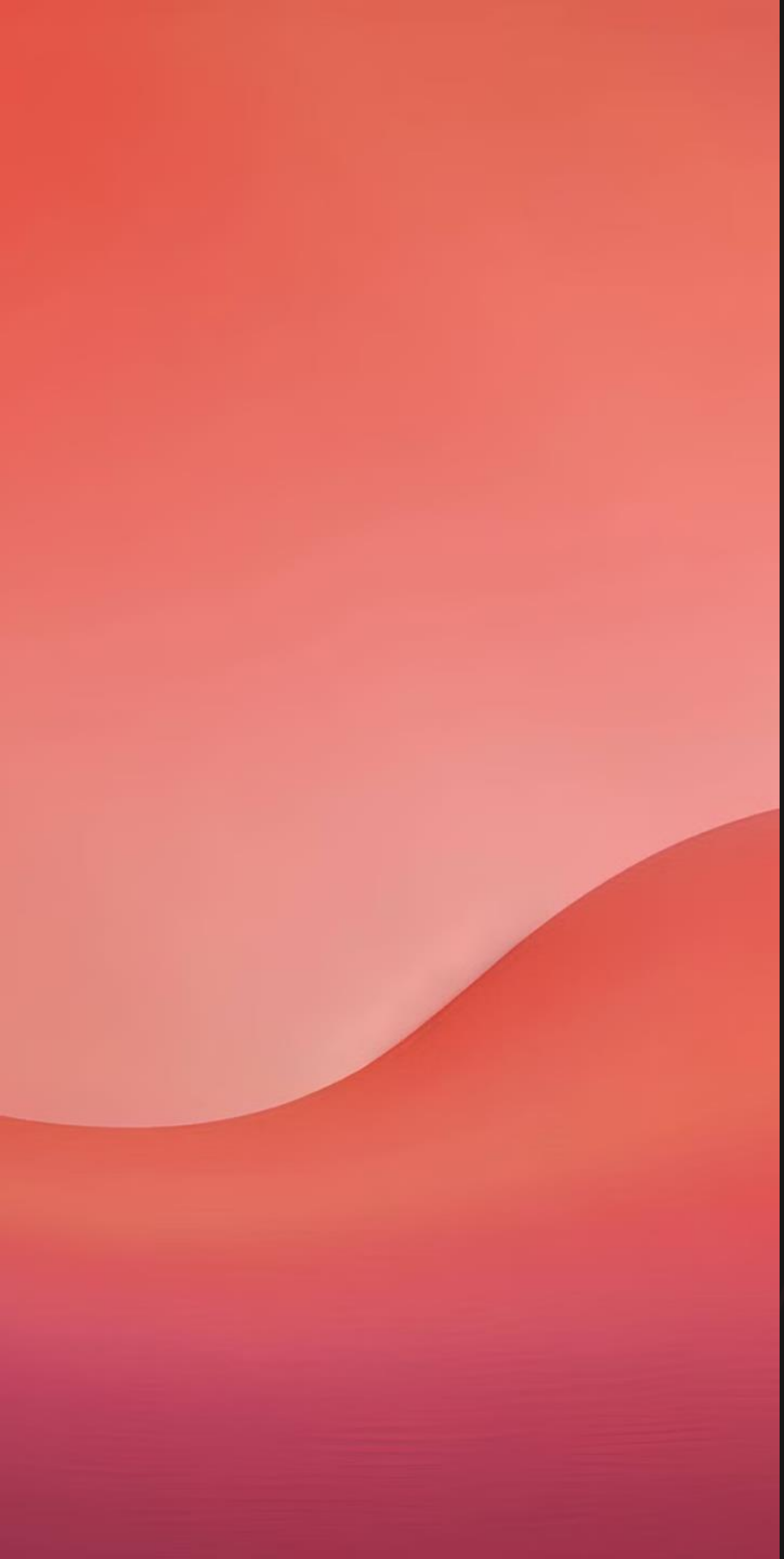
Reusable Configurations

Manage multiple environments and applications easily

Rollbacks and Upgrades

Enable version control and quick application rollback

Helm streamlines managing complex Kubernetes deployments efficiently.



Using values.yaml in Kubernetes Deployment

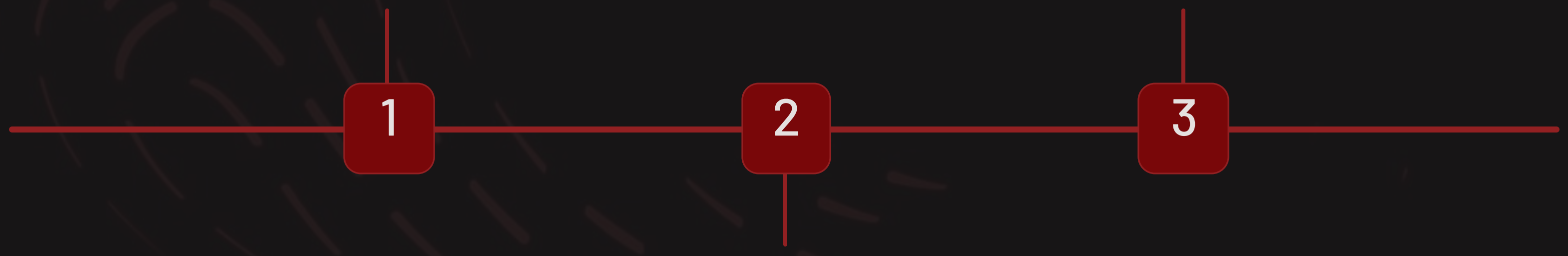
The values.yaml file is a key component in Kubernetes deployments, especially when using Helm charts. It allows you to externalize and manage configuration settings for your application, making it easier to customize deployments without modifying the core YAML manifests.

Define Configuration Variables

In your Helm chart, you can define various configuration variables in the values.yaml file, such as container image, resource limits, environment variables, and more.

Customize Deployments

When deploying your application, you can provide a custom values.yaml file that overrides the default settings, allowing you to tailor the deployment to your specific needs.



Inject Values into YAML

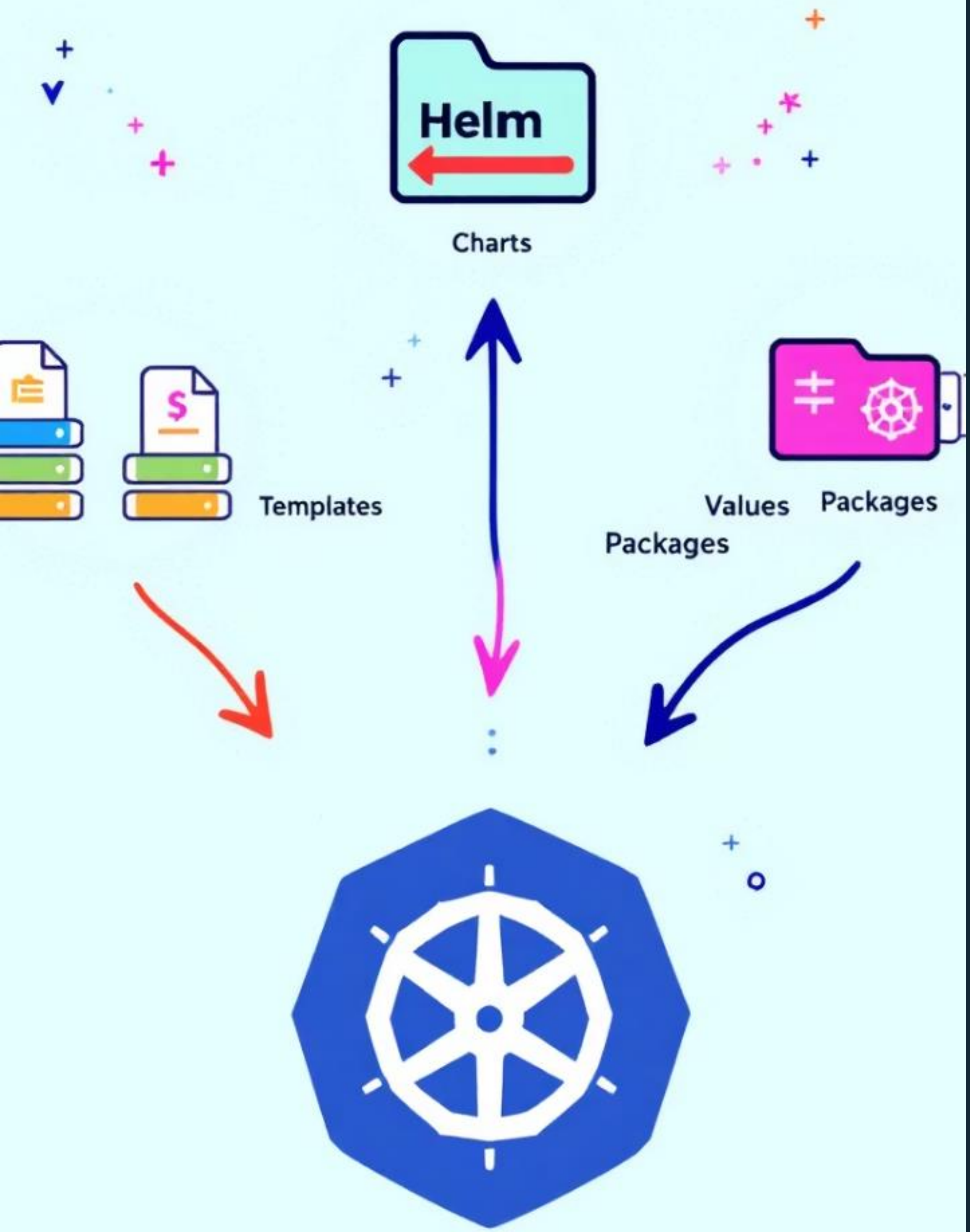
Using Jinja templating syntax, you can then inject these values into your Kubernetes YAML manifests, making them dynamic and reusable.

This approach promotes better separation of concerns, making it easier to manage configurations and deploy your application across different environments.

Helm: The Package Manager for Kubernetes

Helm streamlines deploying applications on Kubernetes by templatizing and grouping YAMLs. It bundles Kubernetes manifests into reusable packages called Charts, providing commands to install or upgrade these charts in a cluster.

A **by Atish Kumar Sinha**



What is Helm and Why It's Useful

Charts

Bundles of YAML templates that can be parameterized, like installable apps for for Kubernetes.

Templating

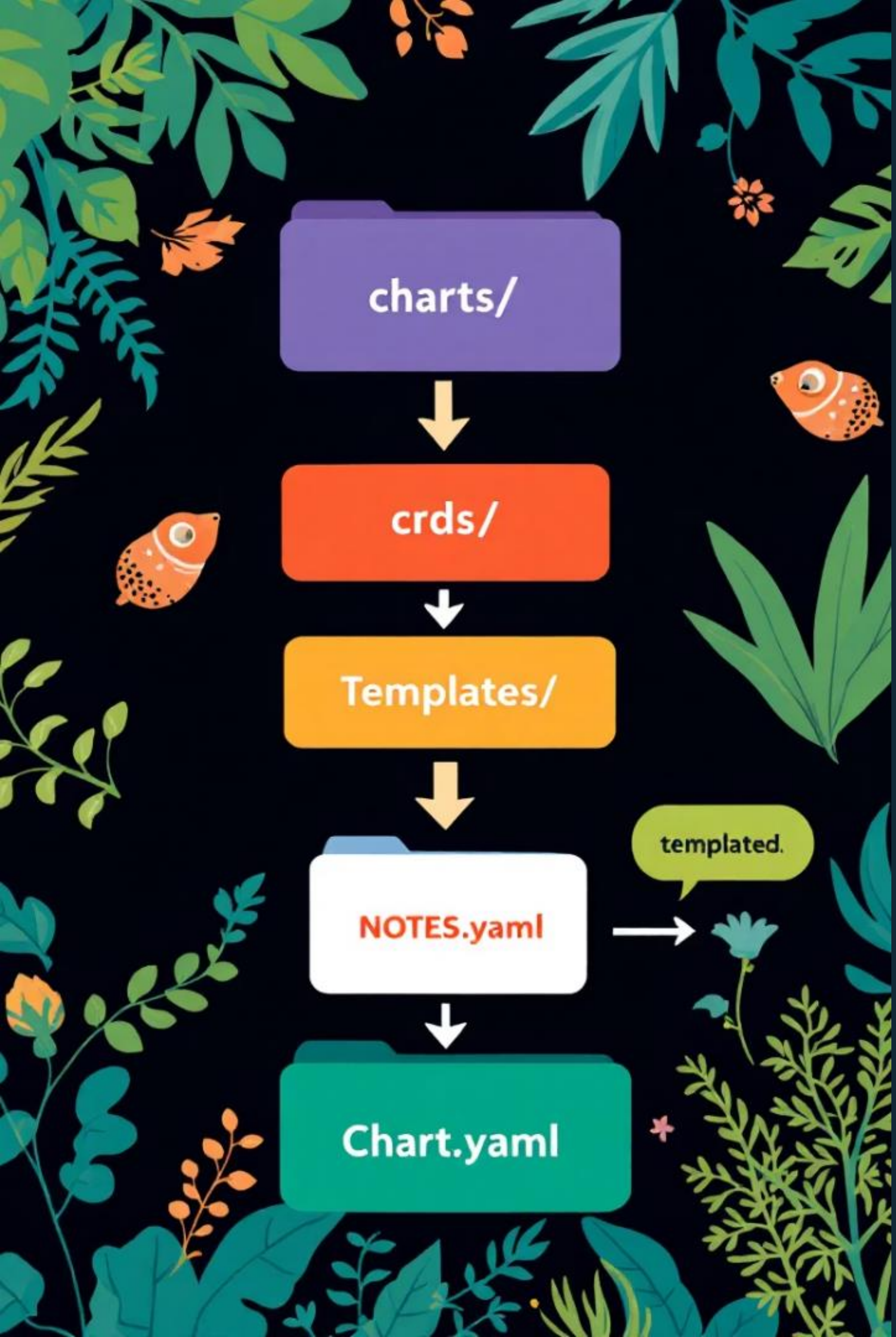
Charts include template files with variables. Values.yaml provides defaults that can be that can be overridden.

Reuse and Share

Teams can share charts via repositories. Artifact Hub hosts thousands of community and official charts.

Lifecycle Management

Helm tracks releases, enabling easy upgrades and rollbacks with a single command.



Helm Chart Structure

Chart.yaml

Contains metadata about the chart (name, version, description, app version).

values.yaml

Default configuration values for the chart templates. Users can override these when these when installing.

templates/

Contains Go-Template files that will be rendered into Kubernetes manifests.

charts/

Directory for subcharts (dependencies) that your chart may require.

Using Helm to Install Applications

Install Helm CLI

First, install the Helm command-line tool on your machine.

Find Charts

Add repositories and search for charts using helm search repo or helm search hub.

Install a Chart

Use helm install [release-name] [chart] with optional flags to customize values.

Upgrade and Rollback

Manage releases with helm upgrade and helm rollback commands.

Parameterizing and Injecting Values

Methods of Overriding

- --values (file): Provide your own YAML file
- --set (key=val): Quick overrides via CLI
- --set-file: Set a key's value from file content

Value Management

- Default values in values.yaml
- Values can be nested (accessed via .Values.)
- Use template functions for defaults
- Consider security for sensitive values

Creating Custom Helm Charts

Scaffold with helm create

Create a new chart structure with helm create myapp as a starting point.

Define Chart Metadata and Values

Adjust Chart.yaml with app details and values.yaml with configuration options.

Customize Templates

Modify deployment.yaml, service.yaml, and other templates for your application needs.

Test and Iterate

Use helm lint and helm install ----dry-run to validate before deploying.

Managing Helm Releases

List Releases

Use helm list to see all deployed releases in the cluster.

View History

Check revision history with helm history [release-name].

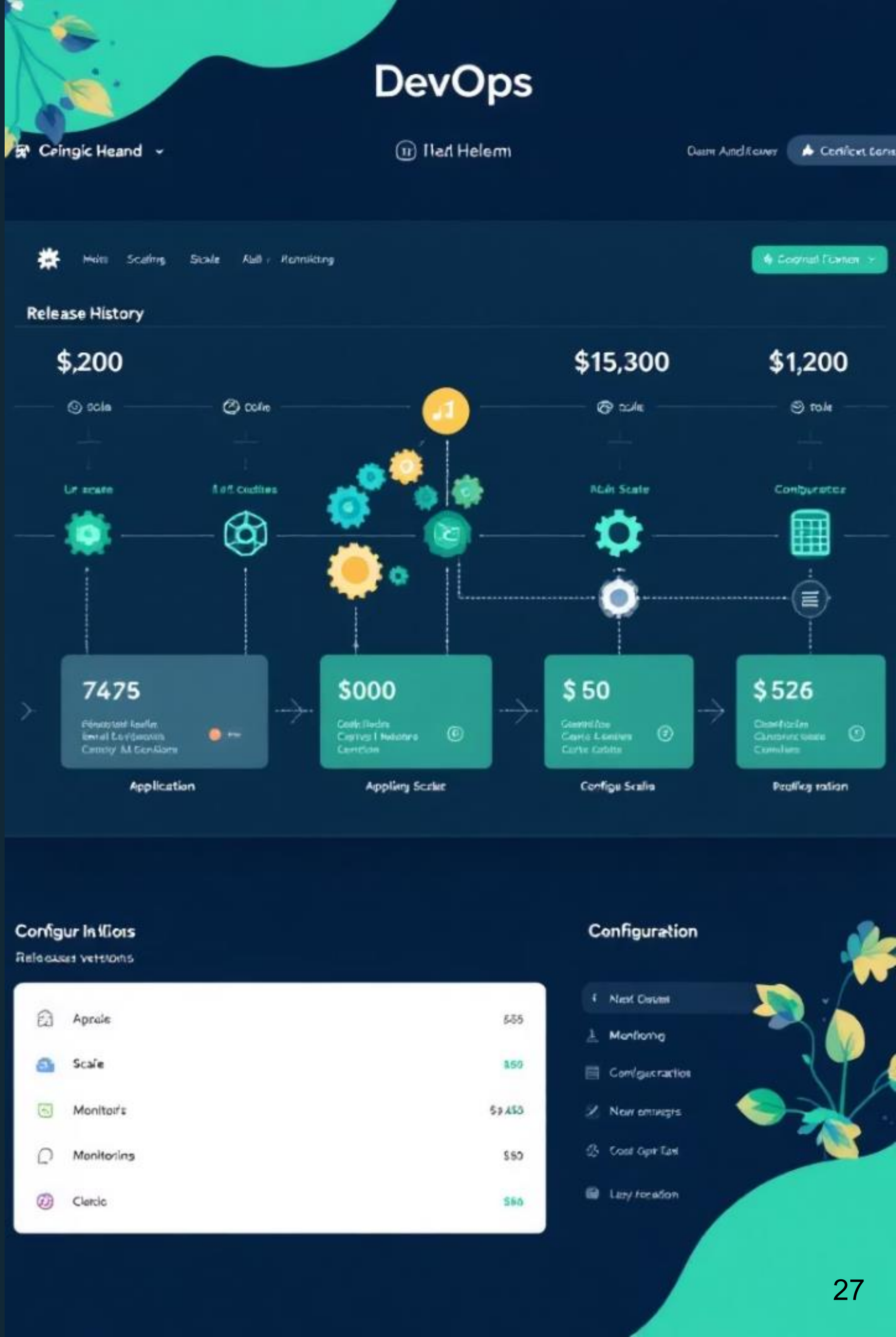
Rollback

Return to previous version with helm rollback [release-name] [revision]. [revision].

4

Uninstall

Remove a release with helm uninstall [release-name].



Helm Best Practices

Reuse Existing Charts

Don't reinvent the wheel. Check Artifact Hub for well-maintained charts.

Document Your Chart

Ensure README and values.yaml explain what each value does.

Secure Secrets

Use tools like helm-secrets or external external secret management for sensitive data.

Keep Charts Simple

One chart should manage one application or cohesive set of components.

Helm in Different Environments

AWS EKS

Works with IAM authentication. Can use use AWS-specific charts for services like like ALB Controller.

Minikube (Local)

Perfect for development. Use minikube minikube tunnel for LoadBalancer services.

OpenShift

Helm works on OpenShift too, with official official chart repositories available.

Introduction to Helm and Helm Charts

Helm is the Kubernetes package manager, simplifying application deployment on Kubernetes clusters. It functions much like apt or yum in the Linux world, streamlining installation, updates, and rollbacks.

Key concepts include **Chart** (the package format), **Release** (an instance of a instance of a chart), and **Repository** (a collection of charts). Helm boosts boosts productivity by managing complex Kubernetes resources efficiently. efficiently.

A **by Atish Kumar Sinha**

Anatomy of a Helm Chart: The Directory Structure

Basic Structure

- **mychart/**: Root directory for a chart
- **Chart.yaml**: Contains chart metadata
- **values.yaml**: Holds default configuration values
- **templates/**: Contains Kubernetes manifest templates

A well-organized chart ensures maintainability and clarity, helping teams collaborate seamlessly.

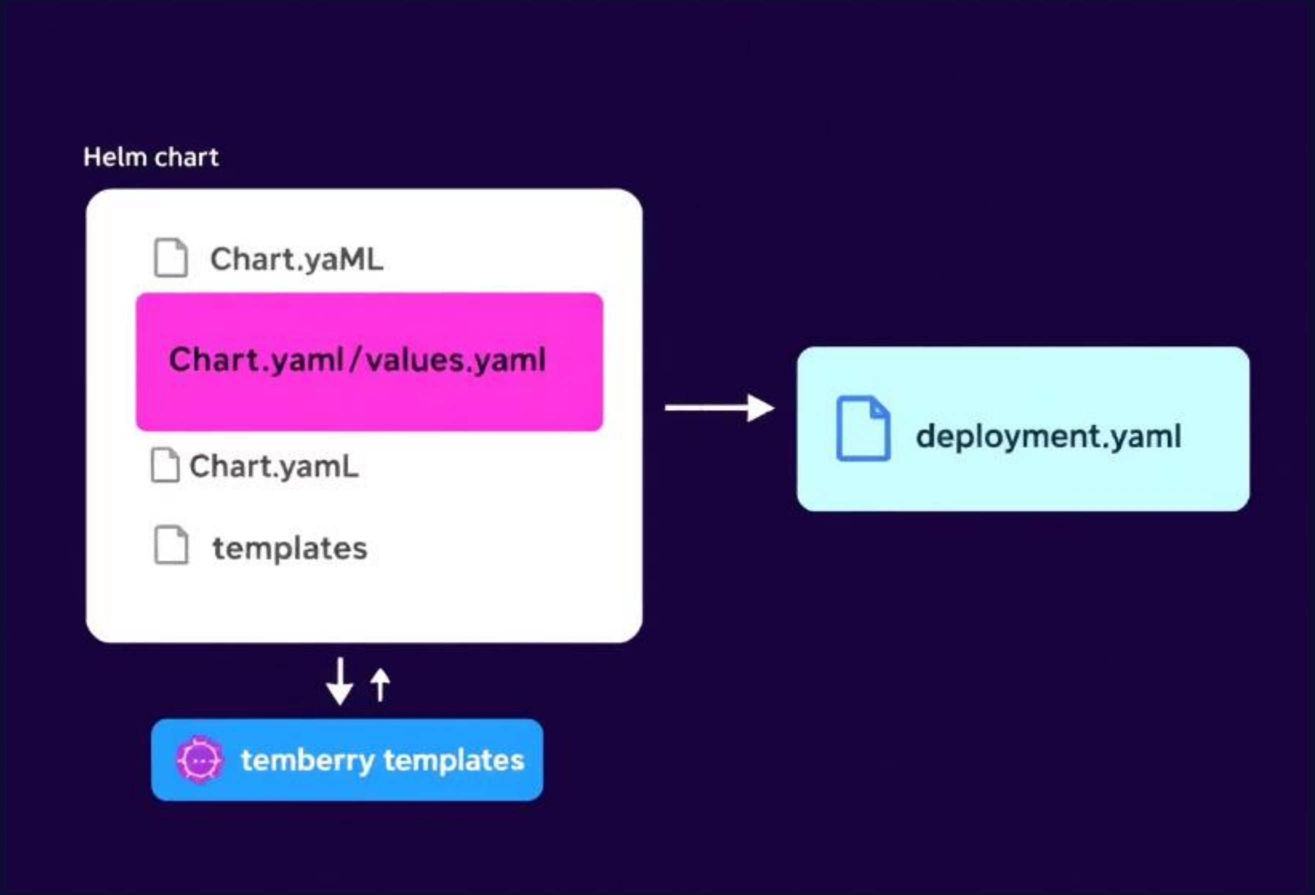


Chart.yaml: Defining Chart Metadata

Key Fields in Chart.yaml

apiVersion: Chart API version; **name**: Chart name;

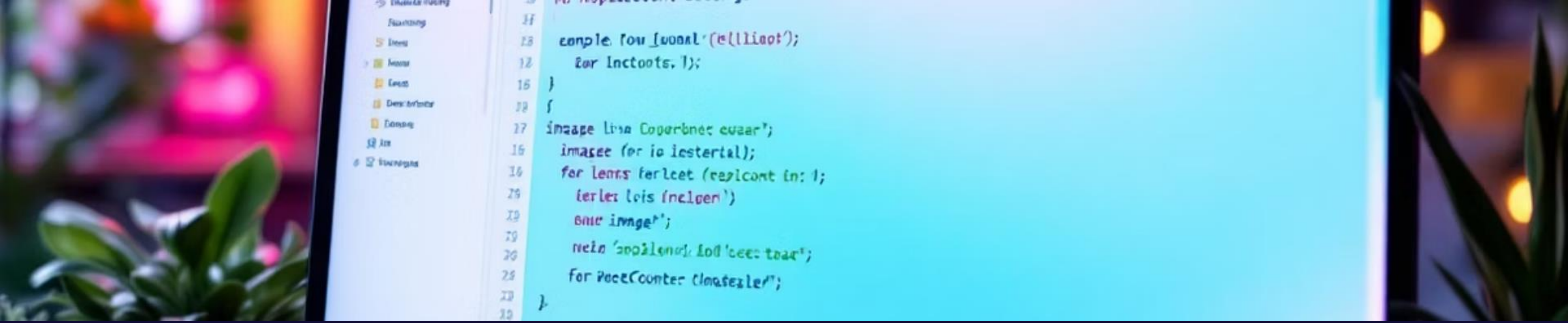
description: Brief description

version: Chart version using semantic versioning;

appVersion: Version of the app being deployed

Semantic versioning clearly communicates changes and compatibility to users.

Maintaining accurate metadata is critical for chart discoverability and lifecycle management.



values.yaml: Setting Default Values

Purpose

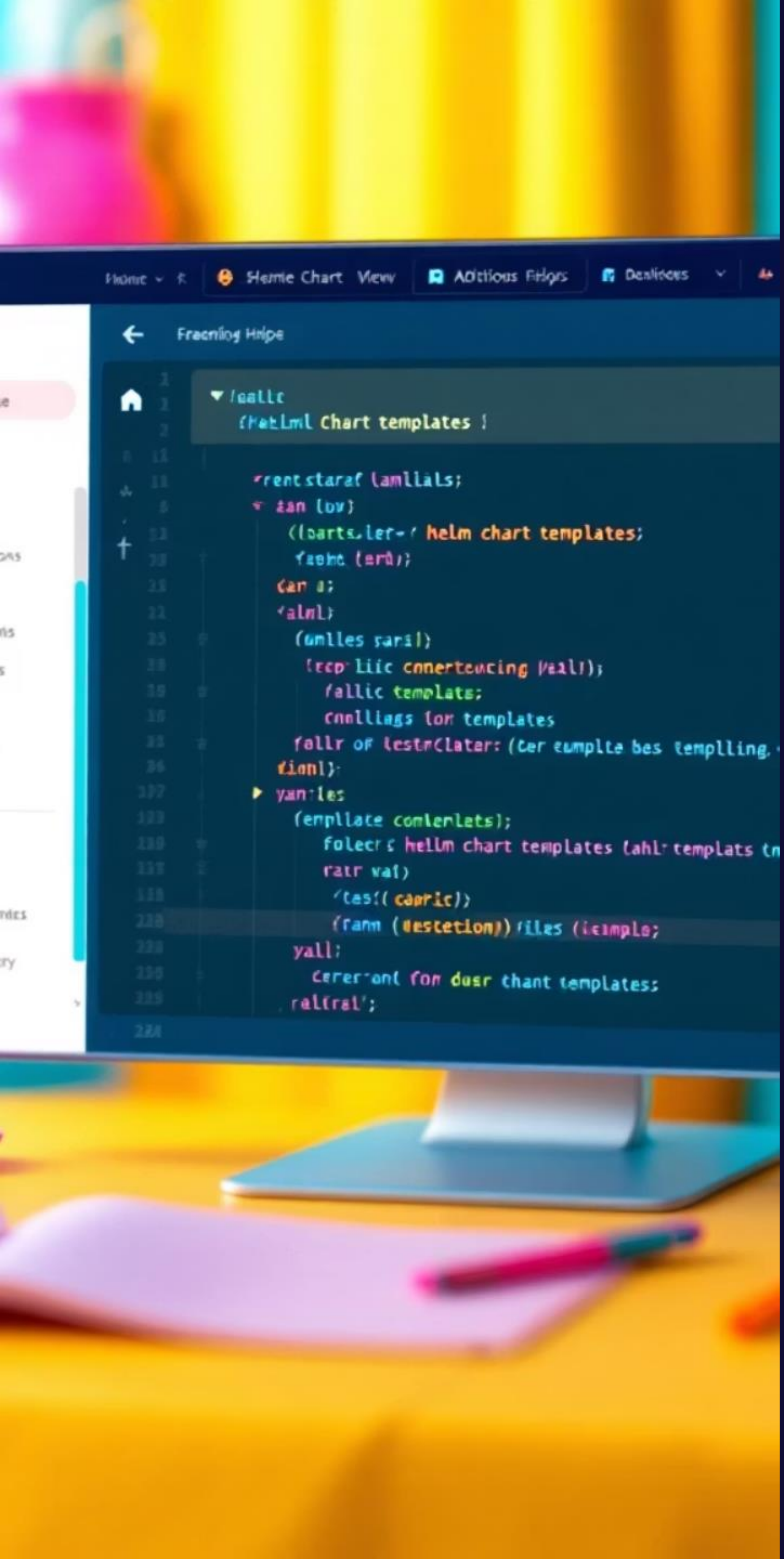
Defines default configuration values values for your chart, enabling easy easy customization without modifying modifying templates.

Typical Parameters

- replicaCount: Number of pod replicas
- image: Repository and tag of the container image
- servicePort: Port exposed by the service

Advantages

Customizing deployments is simplified simplified by overriding these values values during install or upgrade.



Deep Dive into the templates/ Directory

Role of templates/

Contains Kubernetes manifest templates that are are dynamically rendered with values to generate resource manifests.

Common Template Files Files

- deployment.yaml
- service.yaml
- ingress.yaml
- configmap.yaml

Rendering Process

Helm uses Jinja-like templating to substitute and control values, values, making charts reusable and configurable.

Template Example: deployment.yaml

Accessing Values

Use the **.Values** object to reference parameters from values.yaml.

- {{ .Values.replicaCount }} - number of replicas
- {{ .Values.image.repository }}:{{ .Values.image.tag }} - container image tag

Example snippet:

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: {{ .Values.replicaCount }}
  template:
    spec:
      containers:
        - name: app
          image: {{ .Values.image.repository }}:{{
.Values.image.tag }}
```

Template Example: service.yaml service.yaml

Configuring Service

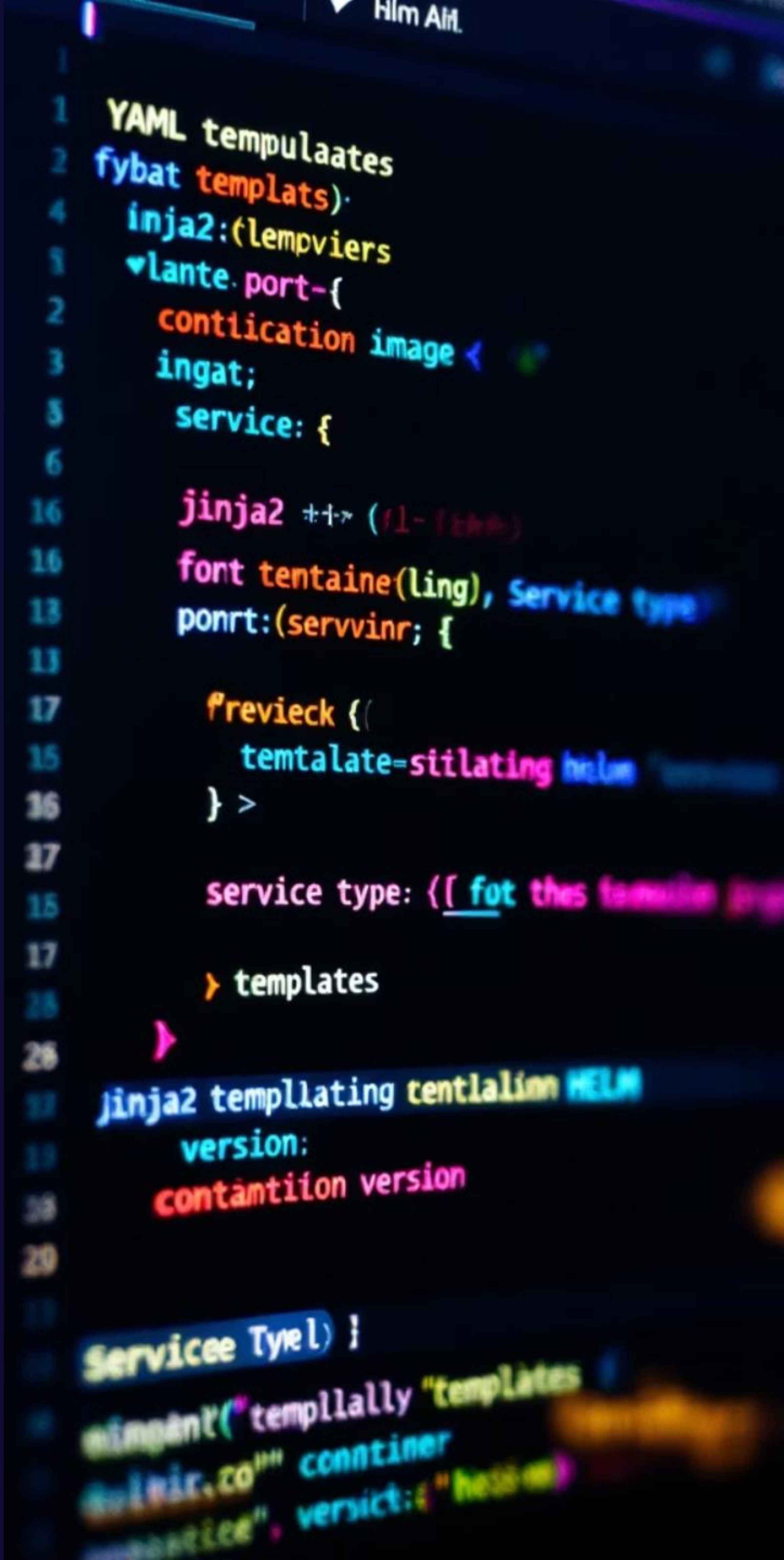
Configure service type and and port dynamically using using values from values.yaml.

Example Variables

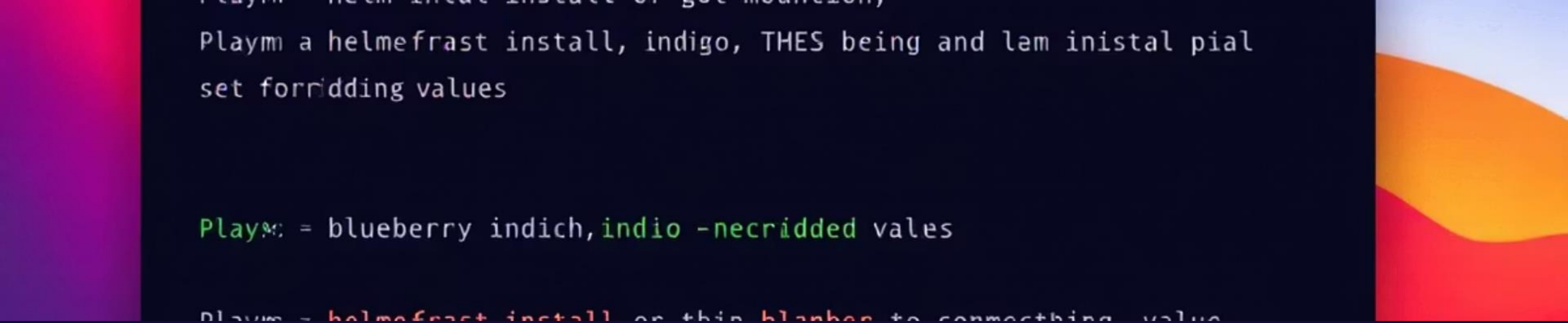
- {{ .Values.service.type }} - service type (ClusterIP, NodePort, LoadBalancer)
- {{ .Values.service.port }} - port the service exposes

Service Types

NodePort exposes the service on a static port across nodes. **LoadBalancer** provisions an external load balancer to route traffic. traffic.



```
YAML tempulaates
fybat templats)·
  inja2:(lempviers
 ▾lante. port-{
    contlication image
    ingat;
     service: {

      jinja2 ++ (
      font tentaine(ling), Service type
      ponrt:(servvinr; {

        frevieck {(
          temtalate-stilating helm
        } >

        service type: ({ fot thes

        › templates
    ›
jinja2 templlating tentlalinn HELM
     version;
   contamtiion version

Servicee Tyel) ]
```



```
Playm - helm incut install or got mountion)
Playm a helmefrast install, indigo, THES being and lem inistal pial
set forridding values


Play%: = blueberry indich,indio -necridded vales

Playm - helmefrast install or thin blanber to connecthing, value
```

Jinja Templating in Helm: Overriding Values

Jinja2 Templating Basics

Helm uses Jinja-like syntax: **{{ }}** for variables and **{{- -}}** to control whitespace.

Overriding Values

You can override values.yaml parameters at install time using the --set flag, e.g., **--set replicaCount=3,image.tag=latest**.

Example Command

helm install mychart --set replicaCount=3,image.tag=latest t dynamically sets replicas and image image tag without changing files.

Advanced Templating: Conditionals and Loops

Conditionals

Use **if/else** to conditionally render resources based on values. values.

- Example: Create Ingress only if enabled

Loops

The **range** function iterates over lists, e.g., creating multiple containers or volumes based on user input.

- Enhances flexibility and composition

Conclusion: Benefits and Best Best Practices

Helm Benefits

Simplifies Kubernetes deployment with reusable, reusable, versioned, and customizable charts.

Best Practices

- Keep charts modular and clean
- Use semantic versioning versioning consistently consistently
- Document charts thoroughly for users

Learn More

Leverage the official Helm documentation and explore public example example charts to deepen knowledge and improve skills.