Project Report: Automated Deployment of Scalable Applications on AWS EC2 with Kubernetes and Argo CD

DevOpsEngineer/VaibhavAgarwal

INTRODUCTION

In the modern landscape of software engineering, the shift from monolithic architectures to microservices has necessitated robust orchestration tools and automated deployment strategies. Manual deployment of distributed systems is error-prone and inefficient. This project addresses these challenges by implementing a **GitOps** workflow using **Kubernetes** and **Argo CD**. By treating infrastructure as code and using Git as the single source of truth, I established a deployment pipeline that is both auditable and self-healing.

ABSTRACT

This project demonstrates the end-to-end deployment of a multi-tier ”Voting Application” on a Kubernetes cluster hosted on AWS EC2. The application consists of five distinct microservices: a Python-based voting interface, a Redis queue, a .NET worker, a PostgreSQL database, and a Node.js results interface.

The core objective was to move beyond manual ‘kubectl‘ application and implement a **GitOps** continuous delivery model. I utilized **Kind** (Kubernetes in Docker) to bootstrap a lightweight cluster and **Argo CD** to synchronize the cluster state with a GitHub repository. The project successfully achieved automated syncing, where changes pushed to the repository were immediately reflected in the live environment without manual intervention.

TOOLS USED

The following technologies were utilized to build the infrastructure and deployment pipeline:

- **AWS EC2 (Ubuntu 22.04):** Served as the host infrastructure, simulating a cloud environment.
- **Docker:** Used as the container runtime to host the Kind cluster nodes.
- **Kind (Kubernetes in Docker):** Used to bootstrap a lightweight, production-like Kubernetes cluster.
- **Kubernetes (K8s):** The orchestration platform managing the microservices, deployments, and services.
- **Argo CD:** The GitOps continuous delivery tool responsible for monitoring the Git repository and syncing manifests.
- **GitHub:** Hosted the source code and Kubernetes manifest files (YAML), acting as the ”Source of Truth.”
- **Kubernetes Dashboard:** Provided a web-based user interface for visualizing cluster resources.

STEPS INVOLVED IN BUILDING THE PROJECT

Step 1: Infrastructure Provisioning

An AWS EC2 instance (t2.medium) was launched with Ubuntu 22.04. Essential dependencies were installed, including Docker (for container runtime) and 'kubectl' (for cluster interaction).

Step 2: Cluster Initialization

I utilized 'kind' to bootstrap the Kubernetes cluster. A configuration file ('config.yml') was used to define the cluster nodes. Verification was performed using 'kubectl get nodes' to ensure the control plane was ready.

Step 3: Kubernetes Dashboard Setup

To gain visual observability into the cluster, I deployed the standard Kubernetes Dashboard using its official YAML manifest. I created an 'admin-user' ServiceAccount with cluster-wide permissions and generated a secure token for authentication. The dashboard was accessed via port-forwarding, allowing me to monitor workloads, logs, and resource usage directly from the browser.

Step 4: Argo CD Installation

Argo CD was deployed into a dedicated namespace ('argocd') using its official manifest. To make the Argo CD UI accessible from the internet, I patched the service type to 'NodePort' and utilized port-forwarding techniques to tunnel traffic from the EC2 instance to the browser.

Step 5: GitOps Configuration

I accessed the Argo CD UI and configured a "New App." I connected the application to the GitHub repository hosting the microservices' YAML files ('vote-deployment.yaml', 'redis.yaml', etc.). The sync policy was set to "Automatic" with "Self-Heal" enabled.

Step 6: Deployment and Verification

Upon creation, Argo CD automatically pulled the manifests from GitHub and deployed the five microservices. I verified the deployment by accessing the Voting App and Result App via their respective ports. I further tested the GitOps flow by modifying the replica count in GitHub and observing the automatic scale-up in the cluster.

CONCLUSION

This project successfully demonstrated the power of GitOps in managing cloud-native applications. By integrating **Argo CD** with **Kubernetes**, the need for manual deployment scripts was eliminated, reducing the risk of human error.

The implemented architecture ensures that the cluster state always matches the configuration in Git. Features like **Self-Healing** (where Argo CD automatically recreates deleted resources) and **Automated Syncing** significantly improved the reliability and maintainability of the deployment process. This setup provides a solid foundation for scalable, production-grade DevOps pipelines.