Next.js AKS Deployment with DevSecOps Pipeline

This project demonstrates how to securely build, package, scan, and deploy a **Next.js** application to **Azure Kubernetes Service (AKS)** using a GitHub Actions pipeline. The deployment includes integration with **Azure Container Registry (ACR)**, **Helm charts**, **ingress controller**, and **security best practices**.

Prerequisites

Thanks! Based on the assignment PDF and your project structure, here's an updated version of the **"## Prerequisites"** section with a clear reference to **Task 1** and what was done to fulfill it:

Prerequisites

What was done in Task 1

In Task 1, I created an Infrastructure as Code (IaC) pipeline that provisions the core cloud infrastructure components required for deployment:

- An **Azure Kubernetes Service (AKS)** cluster
- An **Azure Container Registry (ACR)**
- An **NGINX Ingress Controller**, installed via Helm

Secrets

To securely deploy to Azure from GitHub Actions, the following secrets are configured in the repository under
Settings → Secrets and variables → Actions:

- `AZURE_CREDENTIALS`
  A JSON object containing credentials for a service principal with access to the Azure subscription.
  Used by the `azure/login@v1` action to authenticate to Azure.
  It includes:
  - `clientId`
  - `clientSecret`
  - `subscriptionId`
  - `tenantId`
- `ACR_USERNAME`
  The username for Azure Container Registry (ACR).
  This is typically the name of the ACR instance (e.g., `nextjsbasicapp`).
- `ACR_PASSWORD`
  The password (secret) for the ACR.

  az acr credential show
  --name nextjsbasicappprivate

--query "passwords[0].value"
--output tsv

CI/CD Pipeline Overview

There are two deployment pipelines in this project: both follow the same process, but the pipeline that deploys to a private AKS cluster uses a self-hosted runner that has network access to the cluster, while the other pipeline runs on a GitHub-hosted runner and targets a publicly accessible AKS cluster.

Steps Explained

1. **Checkout Code**
   Clones the repository to the GitHub Actions runner.

2. **Semantic Version Calculation** Fetches the latest Git tags, increments the patch version (0.0.X), and creates a new Git tag. This version is then used for the build artifacts.

3. **Node.js Setup & Install**
   Installs Node.js v22 and project dependencies via `npm install`.

4. **Security Checks**

   - `npm audit fix`: Automatically fixes known vulnerabilities.
   - `npm audit --audit-level=high`: Detects remaining high/critical issues.
   - `npm run lint`: Ensures code quality and coding standards.

5. **Build the App**
   Executes the Next.js build process (`npm run build`).

6. **Static Code Analysis**

   - **CodeQL Analysis**: Detects security vulnerabilities in TypeScript code using GitHub's built-in scanner.

7. **Build & Push Docker Image to ACR**

   - Uses `buildx` to build and tag the Docker image with `latest` and the version tag.
   - Pushes image to ACR.

8. **Inject Version into Helm**
   Updates the `Chart.yaml` and `values.yaml` with the correct version and image tag.

9. **Package & Push Helm Chart to ACR**

   - Packages Helm chart.
   - Logs in to ACR.
   - Pushes the chart to the OCI registry in ACR.

10. **Helm Deployment to AKS**

- Uses Helm to deploy the application to AKS using the updated chart.
- Waits for pod readiness and validates deployment.

Exposing the Application via Ingress

The application is exposed through an **NGINX Ingress Controller**, deployed using Helm.

On the AKS with external access, it provisions a public IP to access the app externally.

You can access the app at:

```
https://nextjsbasicapp.128.203.114.45.nip.io
```

> This URL uses `nip.io`, a free wildcard DNS service that automatically resolves subdomains based on embedded IP addresses.
> In this case, `nextjsbasicapp.128.203.114.45.nip.io` points directly to `128.203.114.45` — no DNS configuration or custom domain setup is needed.
> It's a convenient way to test applications deployed with dynamic or temporary IPs, especially in dev or demo environments.

In the private AKS deployment pipeline, I used an internal NGINX Ingress Controller, which provisions an internal load balancer. This means the application is only accessible within the same virtual network, such as from a jumpbox VM or other internal services connected to the same VNet.

Ingress Rule

The Ingress rule includes:

```
- path: /
  pathType: Prefix
```

This ensures **all traffic under** `/` (e.g., `/`, `/blog`, `/api`) is routed to the backend service — ideal for web apps with client-side routing.

TLS Support

TLS is enabled with a **self-signed certificate** for encrypted HTTPS access in dev/test environments.

> **Note:** In production, only HTTPS should be used with valid TLS certificates from a trusted Certificate Authority to ensure secure, encrypted, and authenticated communication. Self-signed certificates are suitable only for development and testing.

Helm-Templated Ingress

The Ingress resource is dynamically generated via a Helm template located at:

```
templates/ingress.yaml
```

This allows flexible customization of hostnames, TLS, annotations, and routing paths per environment.

Security Scans Overview

This project follows DevSecOps practices by integrating automated security checks directly into the CI/CD pipeline. These checks help identify issues early and ensure that the code and dependencies are secure before deployment.

1. npm audit --audit-level=high

Runs a stricter audit to fail the pipeline if any **high or critical** severity vulnerabilities remain unresolved.

```
npm audit --audit-level=high
```

This helps prevent deploying versions that contain serious known risks.

2. TypeScript Linter

Linting is used to enforce consistent coding styles and detect potential bugs or bad patterns. Running the linter before build time helps keep the codebase clean and maintainable.

```
npm run lint
```

3. GitHub CodeQL Analysis

CodeQL is a static analysis engine provided by GitHub. It scans the codebase for known security vulnerabilities such as:

- SQL injection
- Cross-site scripting (XSS)
- Hardcoded secrets
- Insecure deserialization

It has two steps:

- `init`: initializes the analysis engine with the appropriate language (TypeScript)
- `analyze`: runs the actual scan and uploads results to the GitHub Security tab

These scans improve the pipeline's trust level and provide visibility into secure coding practices over time.

Explaining the `uses:` Steps

In the pipelines, several steps use the `uses:` keyword. This is a **shortcut for plugging in prebuilt GitHub Actions** — reusable automation blocks provided by GitHub or the open-source community.

Instead of writing shell commands to install and configure tools, we can use `uses:` to pull in actions that do the job reliably and efficiently.

`uses:` Steps in Each Pipeline

Here are the `uses:` steps used in each pipeline:

- `actions/checkout@v3` — pulls the code from the repository.
- `actions/setup-node@v3` — installs and sets up Node.js version 22.
- `github/codeql-action/init@v3` — initializes GitHub CodeQL analysis.
- `github/codeql-action/analyze@v3` — runs CodeQL and uploads results.
- `docker/setup-buildx-action@v2` — sets up Docker Buildx for advanced image builds.
- `docker/login-action@v2` — logs into Azure Container Registry (ACR).
- `docker/build-push-action@v5` — builds and pushes Docker images to ACR.
- `azure/login@v1` — logs in to Azure using a service principal.

Why `uses:` Is Better Than Manual Installation

Using `uses:` has several advantages compared to installing tools manually with shell commands:

- **Faster**: GitHub-hosted runners often cache popular actions like `setup-node`, `checkout`, and `buildx`, so they run significantly faster than downloading and installing tools on every run. For example, when using:

```
- name: Set up Node.js
  uses: actions/setup-node@v3
  with:
  node-version: '22'
```

GitHub downloads a precompiled and pre-cached binary of Node.js version 22, instead of building it from source or installing it via apt.

- **Reliable**: These actions are versioned, tested, and maintained.
- **Clean and readable**: Your workflow stays short and easy to understand.
- **Reusable and consistent**: Actions are designed to work across different repositories and environments. You can reuse the same action in multiple projects, ensuring consistent behavior and reducing duplication.

Example: Manual Install vs. `uses:`

To install **Helm**, you could do this manually:

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |
bash
```

Or simply use:

```
- name: Set up Helm
  uses: azure/setup-helm@v3
```

In the pipeline that was used to deploy to private AKS, I removed this `uses:` step because Helm was already installed on the self-hosted runner.