Automated Deployment of a Containerized Web Application Using ArgoCD & GitHub Actions

Ayomide Ajayi

Introduction

This Project implemented an automated deployment of a containerized web application using **ArgoCD** and **GitHub Actions**, executed via a self-hosted runner on a local Windows machine. The environment is powered by **Docker Desktop**, with **Kubernetes managing a three-node cluster** to ensure efficient orchestration and deployment.

By implementing a **GitOps-driven deployment pipeline**, this project leverages **ArgoCD** and **GitHub Actions** to achieve automation, security, and reliability. Infrastructure is maintained declaratively in a Git repository, ensuring continuous reconciliation with the desired state while minimizing manual intervention and configuration drift.

Despite its simplicity, this project is directly applicable to **production environments**, offering individuals and organizations the flexibility to customize and enhance the implementation to meet their specific needs while benefiting from its inherent advantages.

Project Overview

- **Platform:** Windows 11 with **Docker Desktop Kubernetes (3-node cluster)**.
- **Deployment Tool: ArgoCD** (manually installed in the argocd namespace).
- **CI/CD: GitHub Actions** for the automating of deployments.
- **Application Exposure: NodePort services** for external access.
- **Access Method: Port Forwarding** for browser access.
- **Runner Type: Self-hosted GitHub Actions runner** for local execution.

Project Deliverables

This project delivers a **fully automated, GitOps-driven deployment pipeline** for Kubernetes applications, ensuring robust automation, security, consistency, and scalability. The following are the key deliverables and tangible outcomes of this project implementation:

1. **Automated Deployment System:** A fully automated pipeline using **ArgoCD** and **GitHub Actions**, enabling application updates and infrastructure changes to be deployed seamlessly via Git commits.
2. **Continuous Synchronization Mechanism:** A GitOps-based reconciliation process that ensures alignment between the **Git repository** and the **Kubernetes cluster**, preventing configuration drift.
3. **Enhanced Security & Auditability Framework:** Enforced **version control** and infrastructure audit trails, allowing for secure deployment management and traceability of changes.
4. **Configuration Consistency & Rollback Capability:** A mechanism to maintain application stability, continuously reconciling with the desired state and enabling **rapid rollbacks** to prevent downtime in case of failures.
5. **Scalable Microservices Deployment Strategy:** A Kubernetes architecture supporting **dynamic scaling**, optimized for microservices-based applications, ensuring efficient resource utilization.

This approach delivers a **robust, efficient, and resilient Kubernetes deployment strategy**, providing a **structured framework** for managing containerized applications with **ArgoCD and GitHub Actions**. It enables **seamless automation**, ensures **infrastructure consistency**, and enhances **operational reliability**, optimizing deployment workflows for scalability and maintainability.

Step-by-Step Implementation

Prerequisites

1. **Install Docker Desktop**
2. **Install Kubectl**
3. **Set Up Kubernetes on Docker Desktop**
4. **Verify 3-node Cluster Availability**
5. **Install ArgoCD Manually in the argocd Namespace.**

   - **Create the argocd namespace:** kubectl create namespace argocd.
   - **Apply ArgoCD installation manifest:**

   kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

   - **Verify ArgoCD pods are running:** *kubectl get pods -n argocd*
   - **Expose ArgoCD UI using NodePort/Port-forwarding:**

     Apply the service: *kubectl apply -f argoservice.yaml*

   - **Access ArgoCD UI via browser:** 👉 **http://localhost:8080**
   - **Retrieve ArgoCD admin password:** *using powershell commands*

6. **Create the manifests for infrastructure, services and ArgoCD Application**
7. **Automate Deployment Using GitHub Actions (Self-Hosted Runner)**

   ✓ GitHub Actions Workflow (.github/workflows/kube.yaml) Using a Self-Hosted Runner

Technical Overview

This project highlights the **interdependency between ArgoCD, GitHub Actions, and Git repositories**, demonstrating their combined role in deploying **scalable, secure, and automated infrastructure**. Efficient deployment of a **containerized web application** relies on a **seamlessly interconnected workflow**, where each component plays a distinct role in **automation, management, and maintaining deployment integrity.**

Architecture & Workflow

1. **The Git Repository: Source of Truth**

   The Git repository **acts as the single source of truth** for the containerized web application as it stores various artifacts including but not limited to **Application source code (containerized in this case), Kubernetes deployment manifests, ArgoCD application configurations and GitHub Actions workflow definitions. Hence, the key responsibilities of the Git repository include:**

   - ✓ **Version Control:** Git enables tracking changes, rollbacks, and collaborative development.
   - ✓ **Declarative Infrastructure:** Kubernetes manifests stored in Git define the desired state of the application.
   - ✓ **Integration with GitOps:** ArgoCD continuously monitors changes in the repo, ensuring deployments are still up to date.

2. **ArgoCD: Kubernetes Continuous Deployment**

   ArgoCD as a **GitOps tool facilitates the automation of continuous deployment and management of the** Kubernetes application using CICD and **Git repositories** as the control mechanism. Hence, ArgoCD offers key responsibilities that are central to:

ArgoCD ensures seamless GitOps-driven Kubernetes deployments through several key mechanisms:

- **Application Syncing:** It continuously monitors the Git repository and automatically synchronises Kubernetes manifests, ensuring deployments stay up to date.
- **Declarative Deployment:** The live cluster state is consistently aligned with the desired state defined in the Git repository, minimizing manual intervention.

- **Self-Healing:** If unauthorized changes are made directly in the Kubernetes cluster, ArgoCD detects and reverts them to maintain consistency.
- **Automated Rollbacks:** In case of deployment failures, ArgoCD can roll back to the last known stable version, preventing downtime and errors.

Interaction with Git in ArgoCD Deployment

ArgoCD utilizes **application manifests** to track designated branches and paths within a **Git repository**, continuously monitoring for changes pushed by **GitHub Actions**. When updates are detected, **automated synchronization** is triggered, ensuring that deployments remain secure, traceable, and fully aligned with the declared Kubernetes configuration. This approach guarantees a **self-healing, GitOps-driven workflow**, minimizing manual intervention while maintaining infrastructure integrity.

3. **GitHub Actions: Automating CI/CD**

GitHub Actions provided an automation layer for **building, testing, and updating application deployments**. These key responsibilities were achieved during this project implementation:

- ✓ **Trigger Deployments:** Runs workflows upon code changes (push, merge).
- ✓ **Manifest Updates:** Applies Kubernetes manifests from Git to trigger ArgoCD synchronization.
- ✓ **Port Forwarding and Service Exposure:** Automates Kubernetes service setup for accessibility.

Interaction Between Git & ArgoCD

- **GitHub Actions triggers workflows on code changes**, updating deployment manifests in the **Git repository**.
- **ArgoCD detects updates in Git**, pulls the latest manifests, and synchronizes changes with the **Kubernetes cluster**.
- **CI/CD pipeline ensures deployment integrity**, automating tests before ArgoCD applies updates, maintaining a secure and reliable deployment process.

4. **Rollback & Disaster Recovery**
    a. Every deployment is version-controlled, allowing easy rollback to previous states if needed.
    b. ArgoCD ensures self-healing deployments by reapplying desired configurations if accidental changes occur.

Deployment Process

1. **Containerization:**
    - The web application is packaged into a Docker container.
    - The container image is pushed to and stored in a local registry.
2. **Infrastructure as Code:**
    - Kubernetes manifests define application deployment, services, and ingress rules.
    - ArgoCD application configuration ensures automated synchronization.
3. **GitOps Workflow:**
    - Code pushed to GitHub, triggering GitHub Actions for deployment builds.
    - Updated manifests in the Git repository signal ArgoCD to deploy the new application version.
4. **Monitoring & Observability:**
    - ArgoCD UI provides real-time visibility into application deployment status.
    - Logging tools (such as Prometheus and Grafana) can be integrated for performance monitoring albeit not implemented.

Project Artefacts

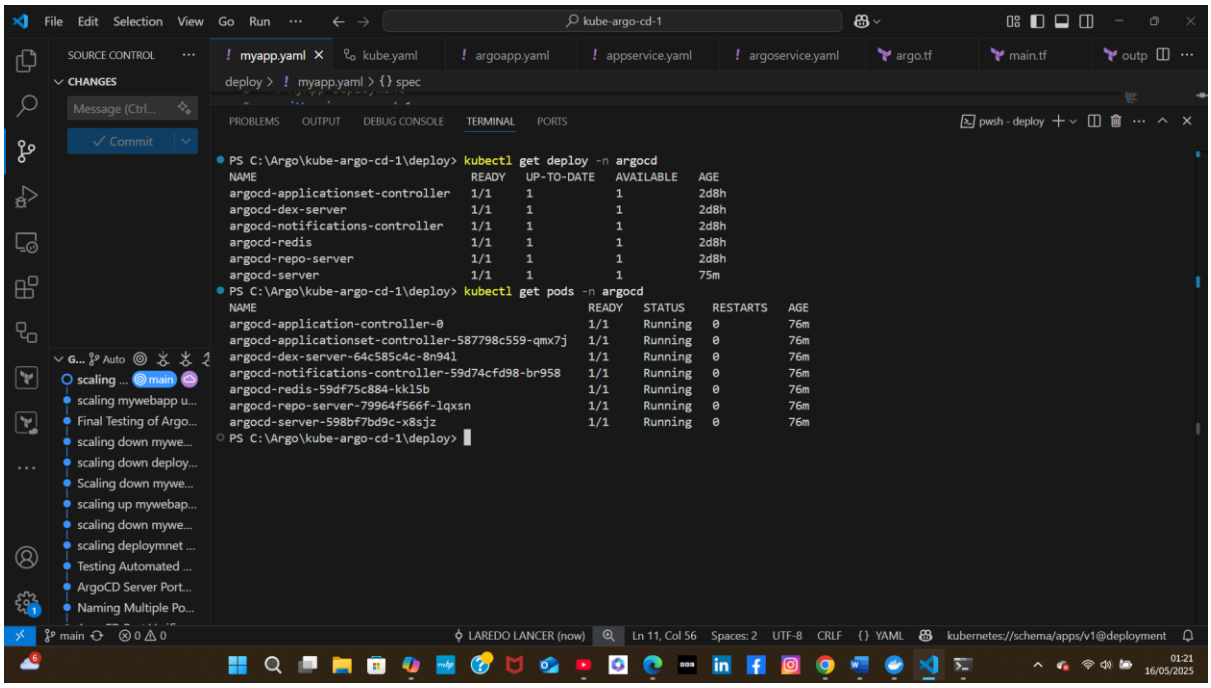
Project Outcomes: Testing ArgoCD for Web Application Deployment

To evaluate the functionality of ArgoCD in deploying a web application on Kubernetes, I conducted a structured assessment focusing on deployment accuracy, associated services, scaling behaviour, security measures, high availability, and fault tolerance. The following systematic testing approach ensured comprehensive validation of these key aspects.

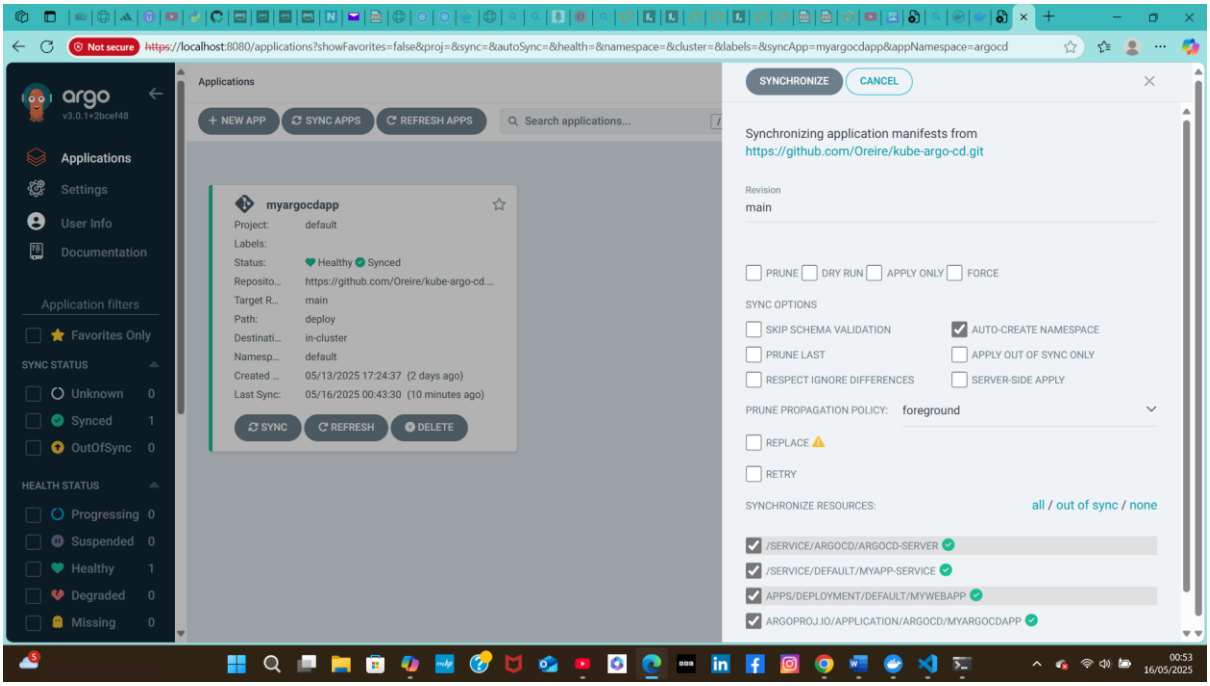
A. Initial Deployment Verification

Steps:

1. **Ensure ArgoCD is Installed & Running:** Confirm that all ArgoCD components (server, repo-server, controller, etc.) are running.

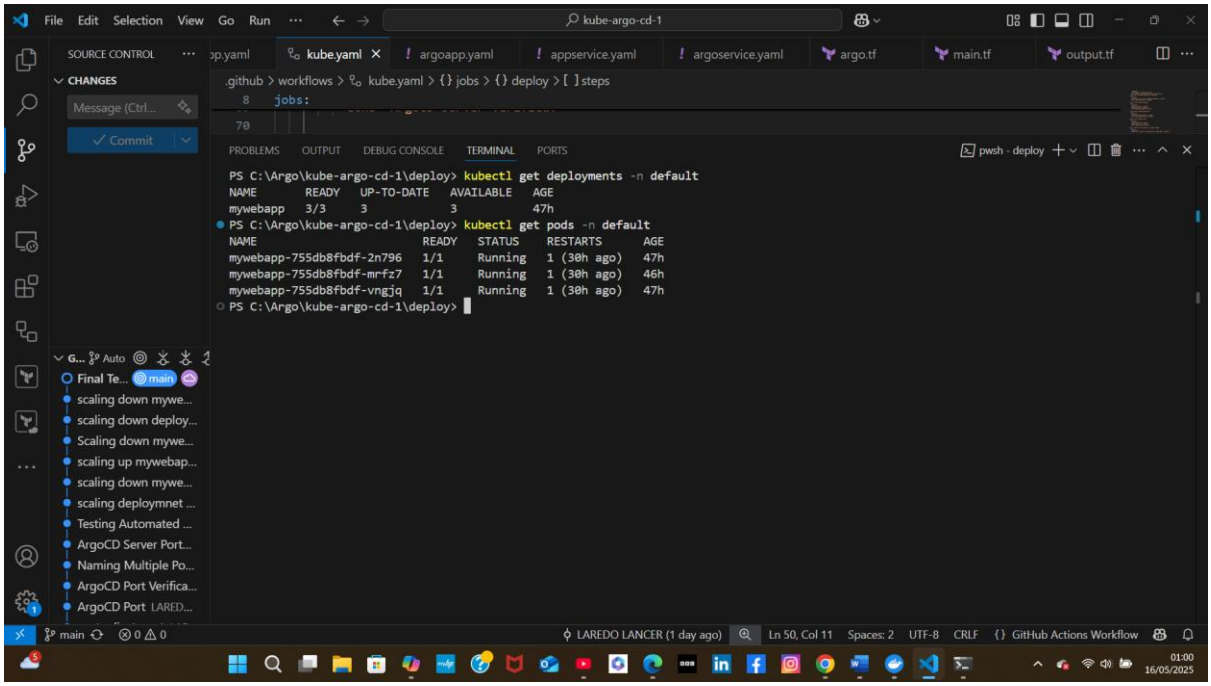


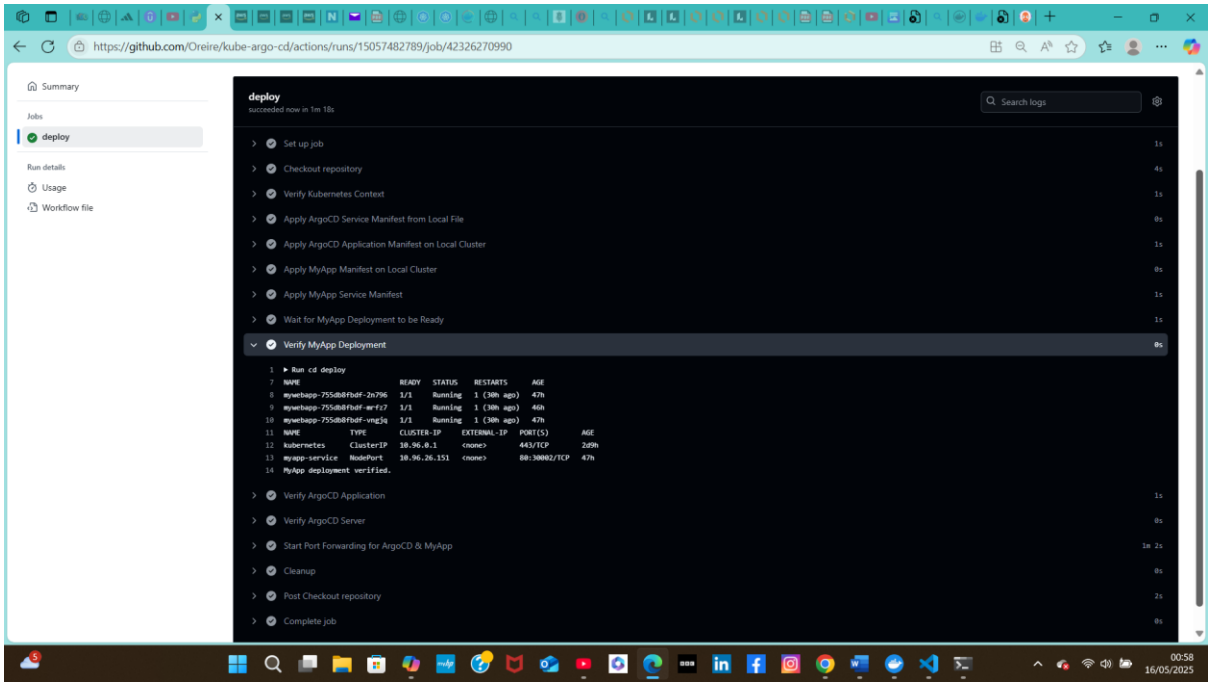
2. **Check Application Syncing** Apply an ArgoCD application manifest:



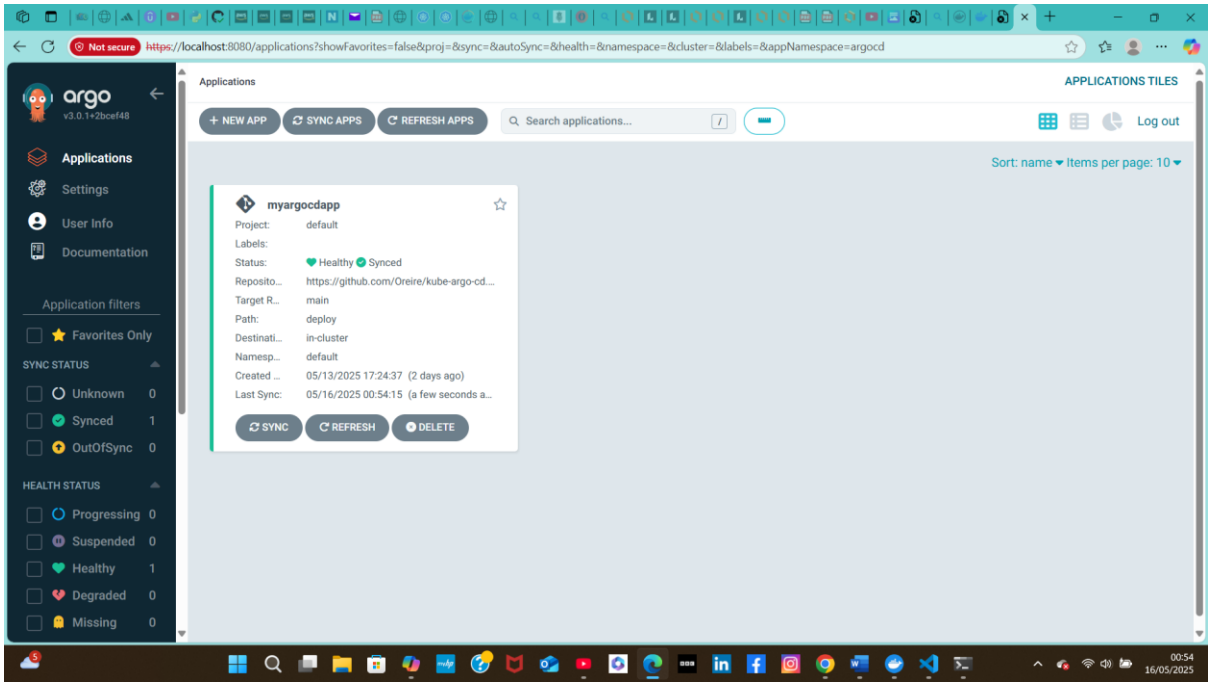
3. **Confirm Deployment Creation**

Check web Deployment and Pods Status





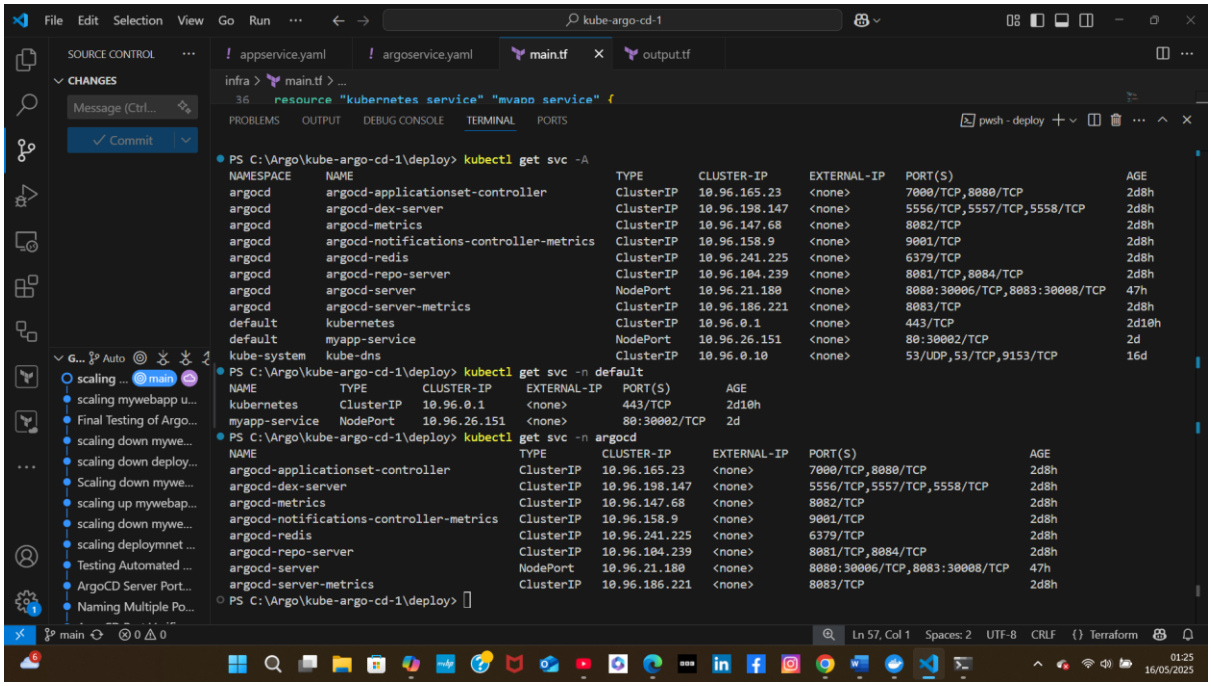
4. **Verify ArgoCD UI Sync**



Access https://localhost:8080

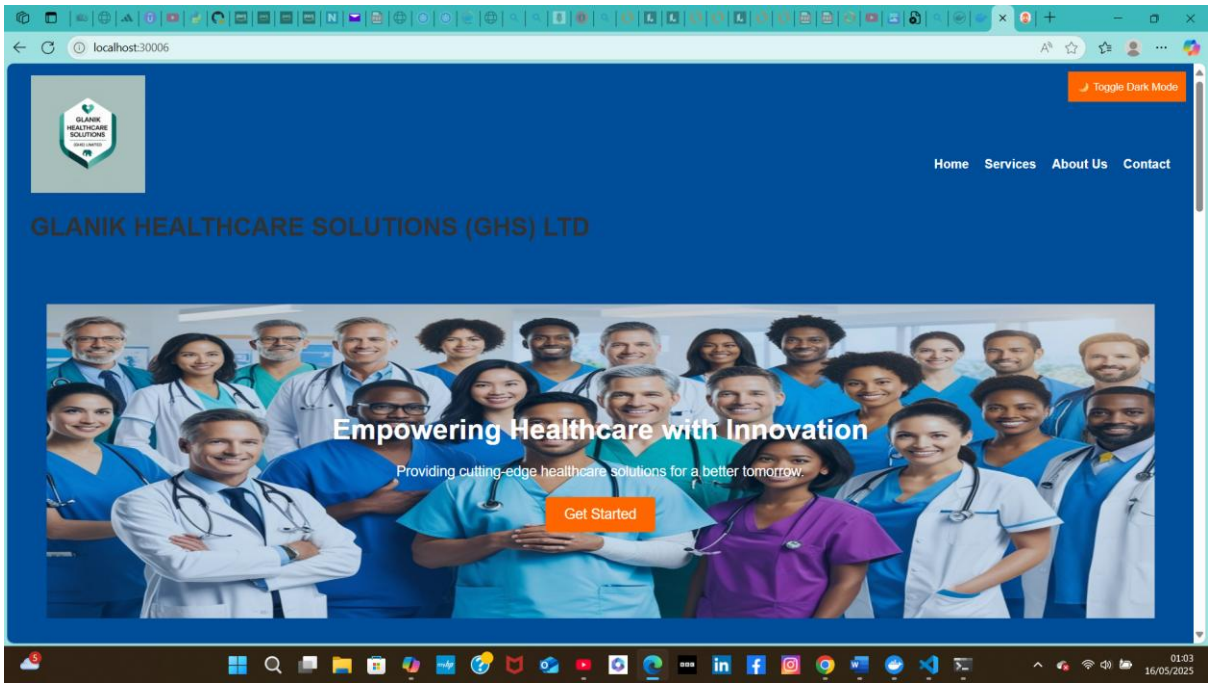
2. Validating Associated Service

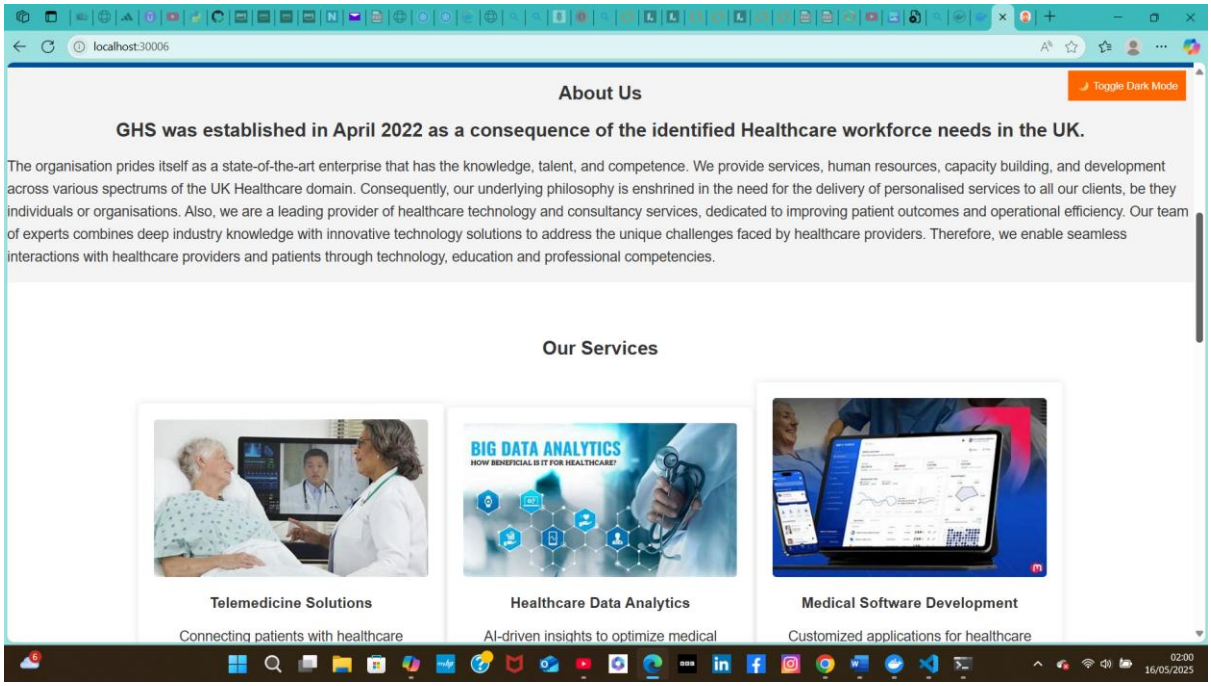
Check If the Web Application Service Is Running

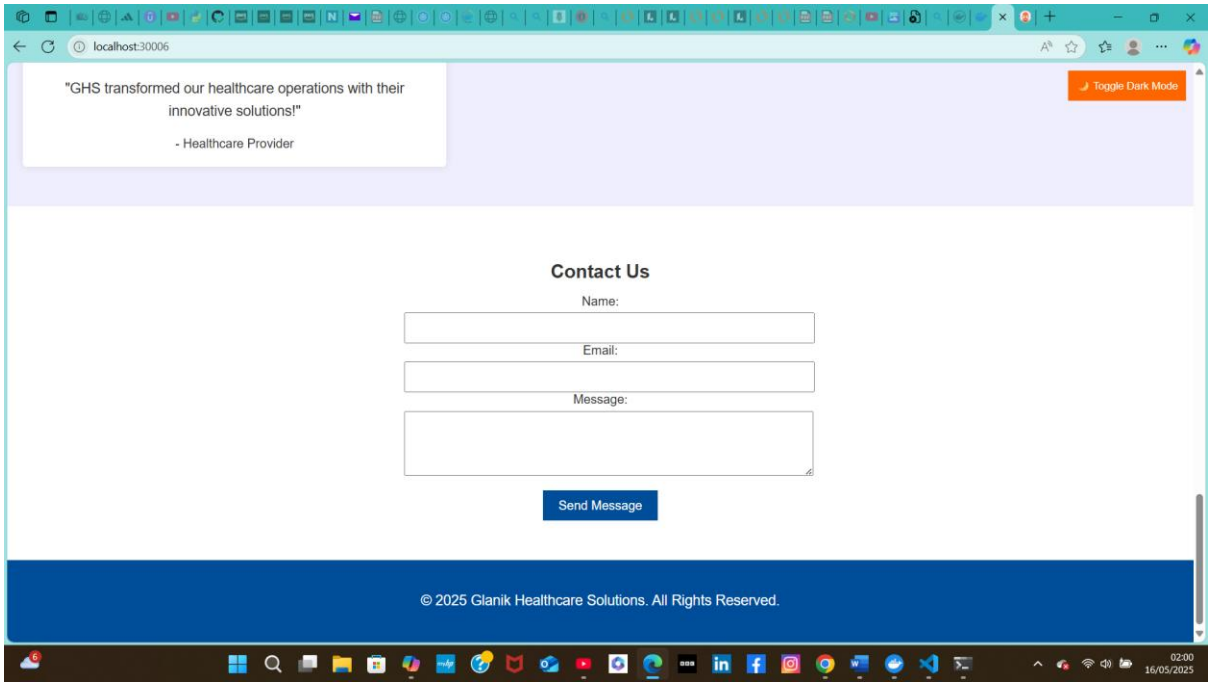


- Services correctly created and configured for external access.

B. Manually Access the Web Application (mywebapp)



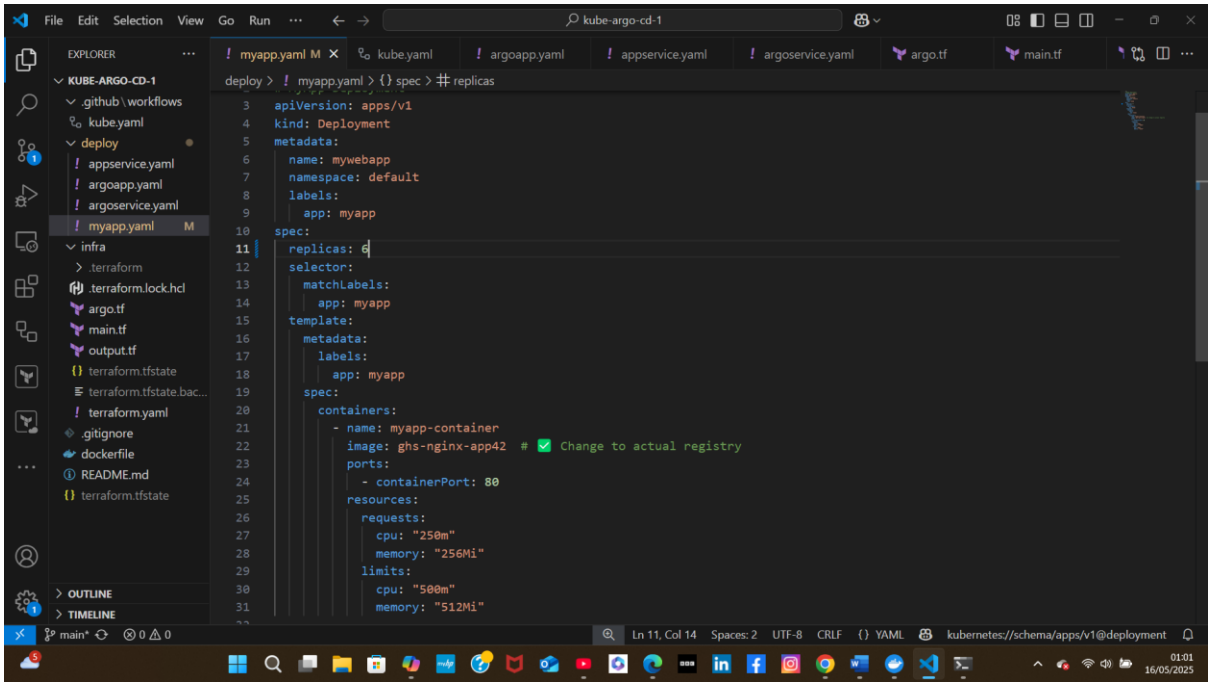




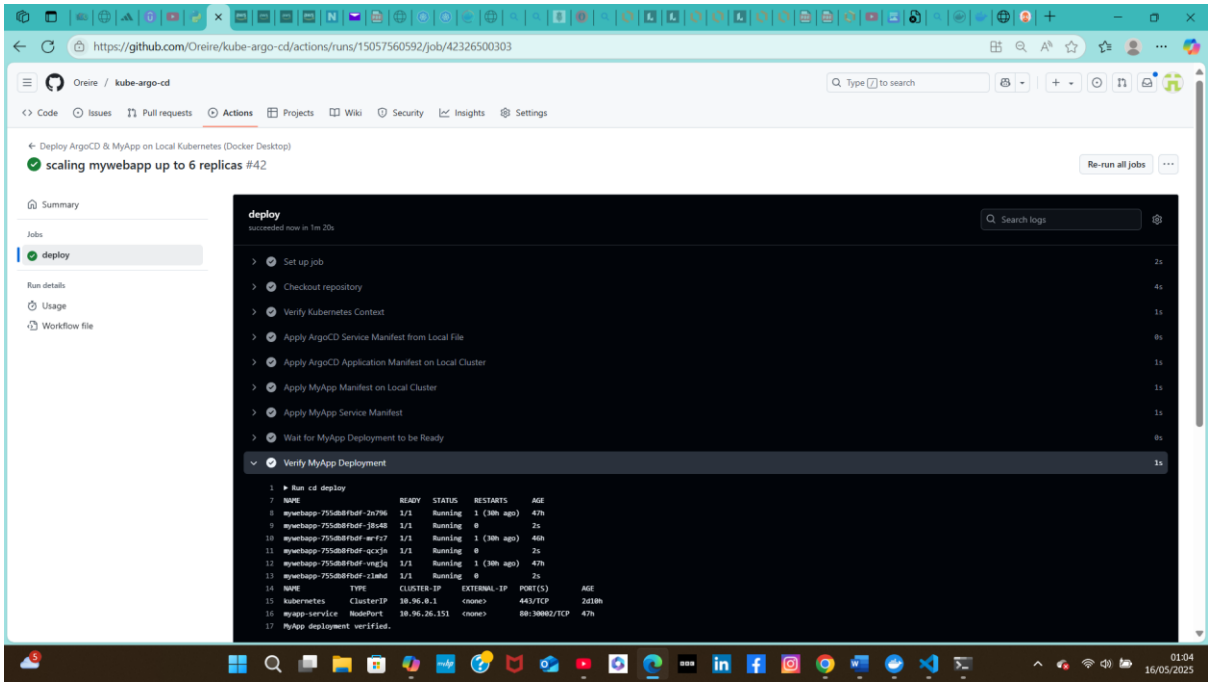
localhost:30006 (via port-forwarding)

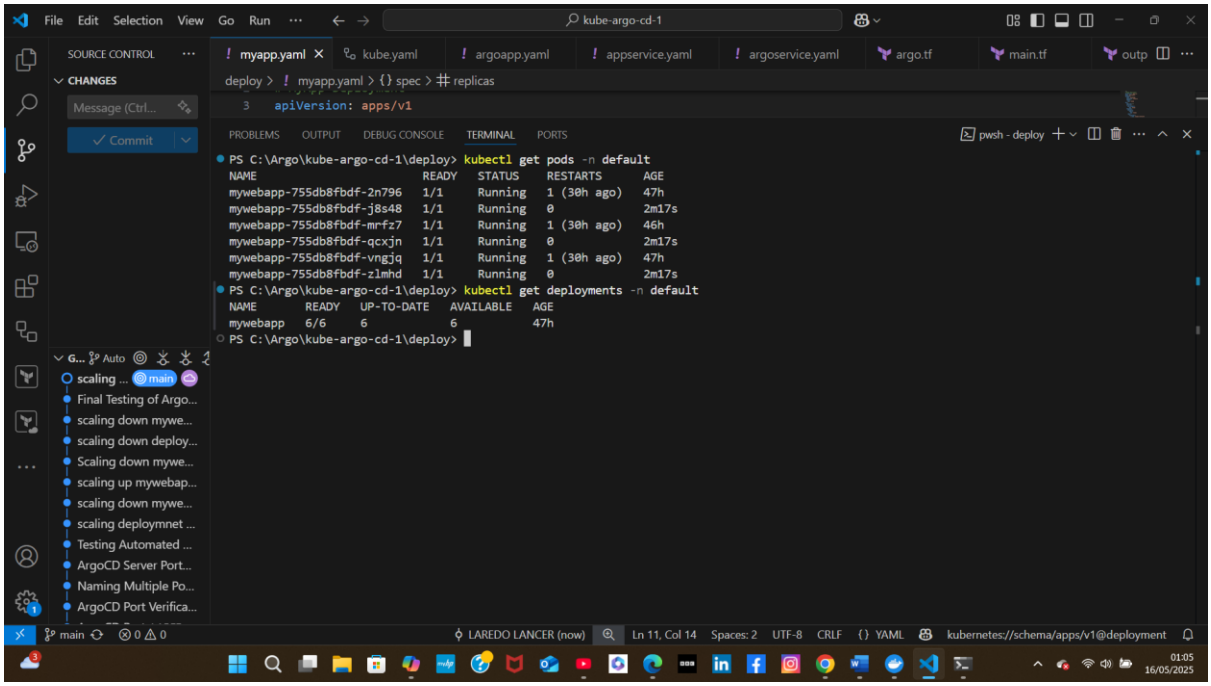
C. Scaling Tests for High Availability

Scaling Up Deployment



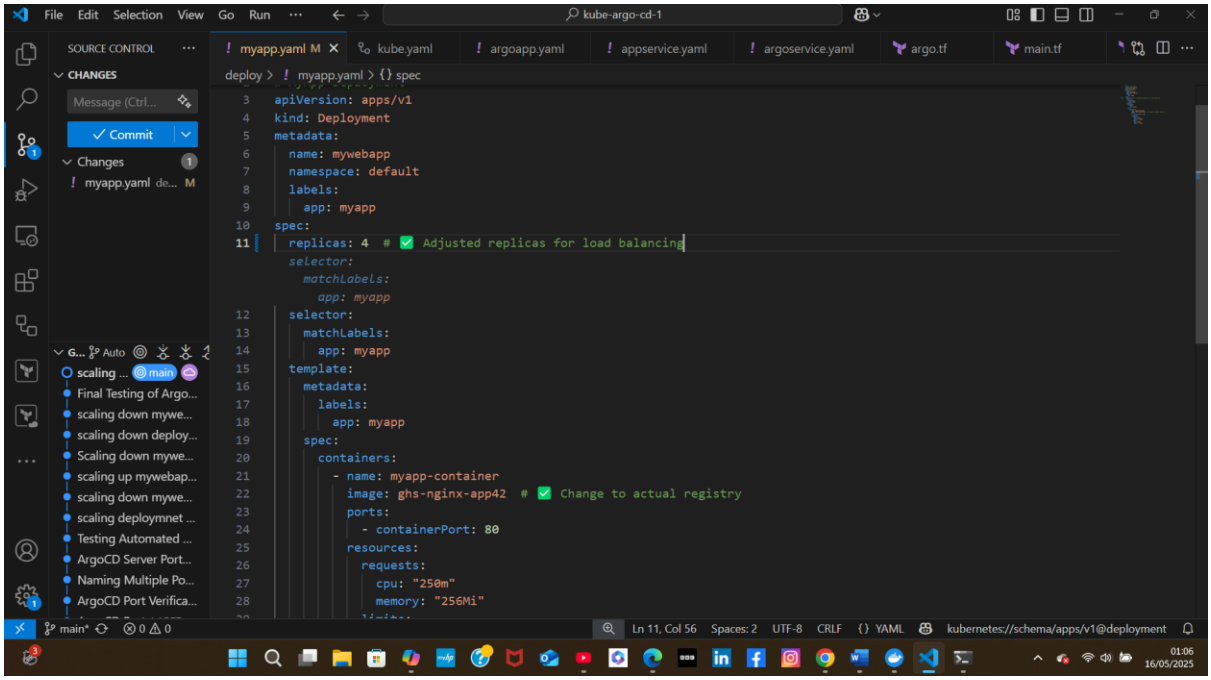
Scaling deployment up to 6 replicas: Pipeline



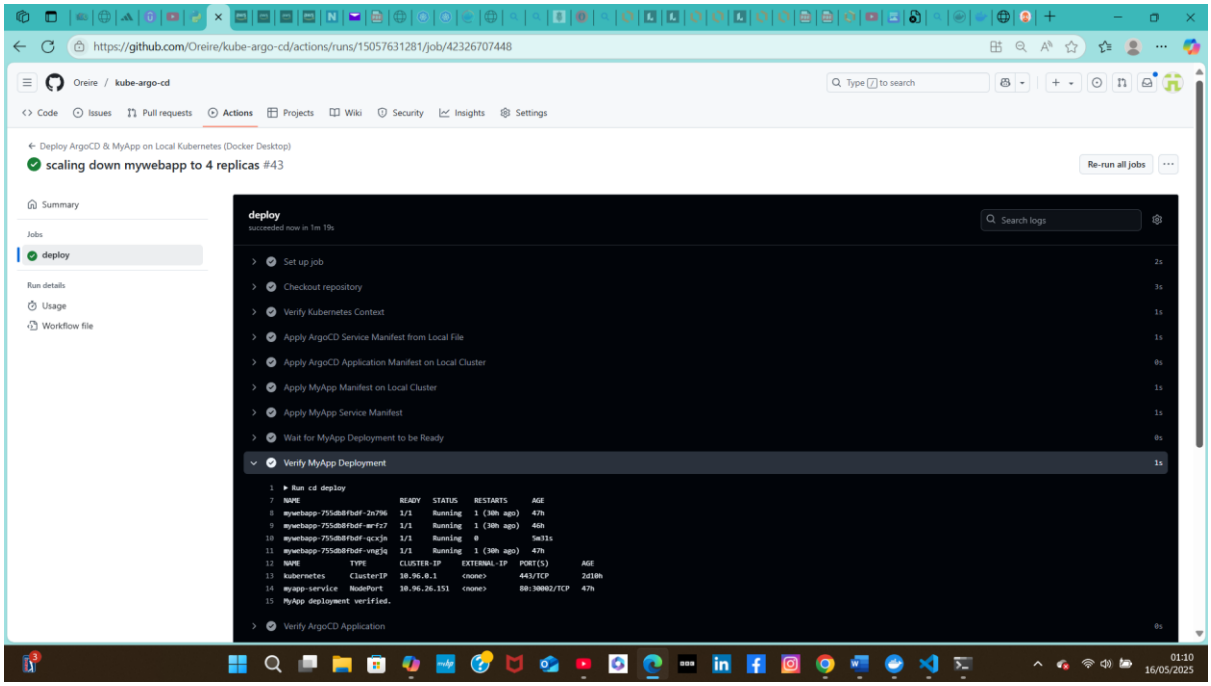


Scaling Down Deployment

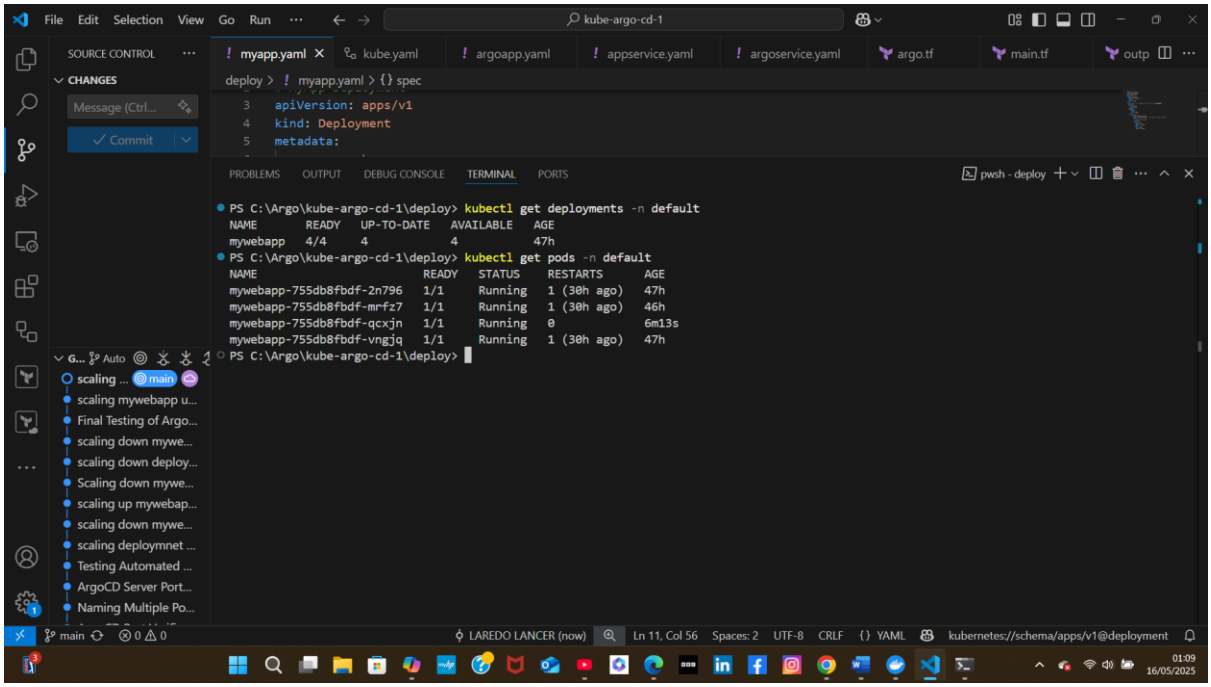
Scaling down Mywebapp deployment to 4 replicas



Git push to deploy via ArgoCD (pipeline)



Vscode verification

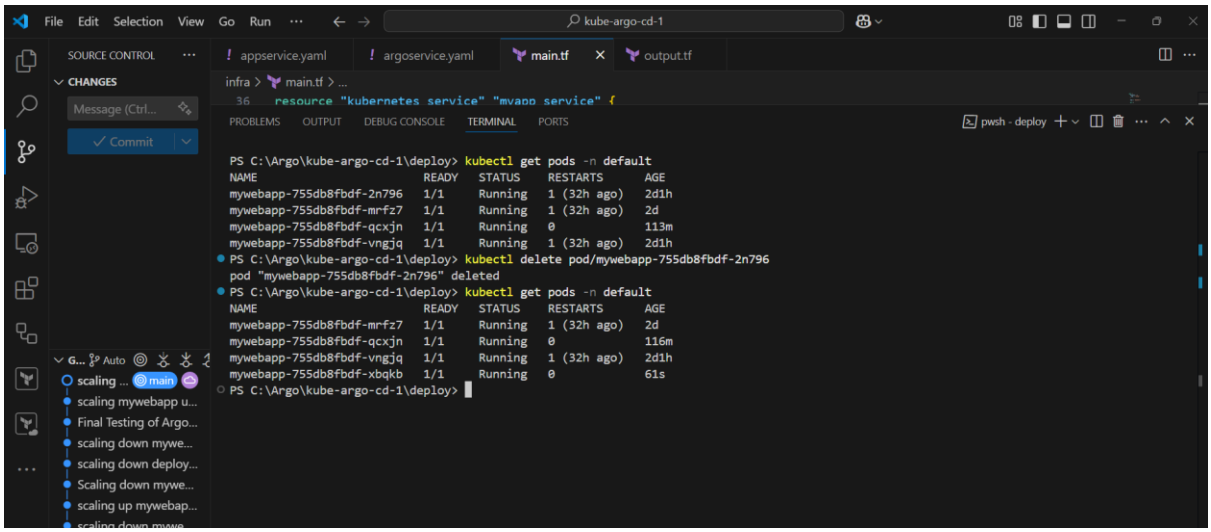


D. Fault Tolerance Tests

Forced Pod Failures to observe recovery:

Delete a pod and check if Kubernetes restarts it:

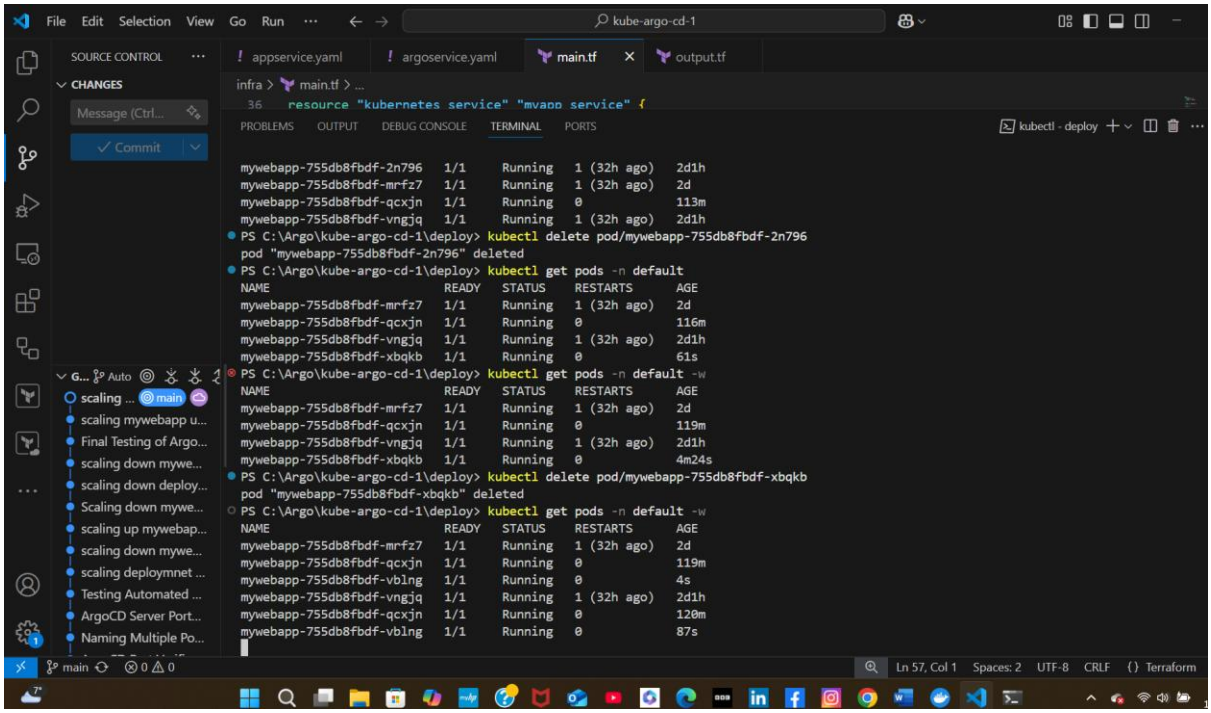
1. pod "mywebapp-755db8fbdf-2n796" deleted.
2. mywebapp-755db8fbdf-xbqkb (new pod created via self-healing and ArgoCD SYN)



Delete pod in namespace: kubectl delete pod <pod-name> -n default.

Monitor pod recreation:

kubectl get pods -n default -w (recreated pod: mywebapp-755db8fbdf-vblng)

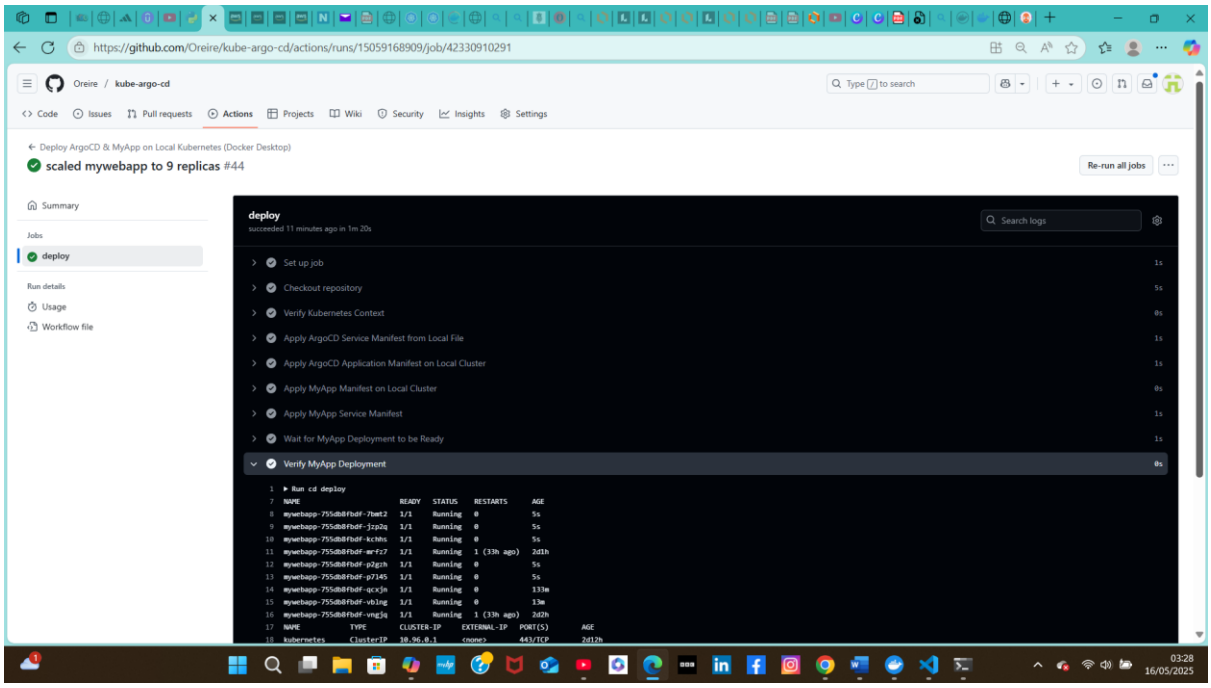


Network Disruptions Simulation

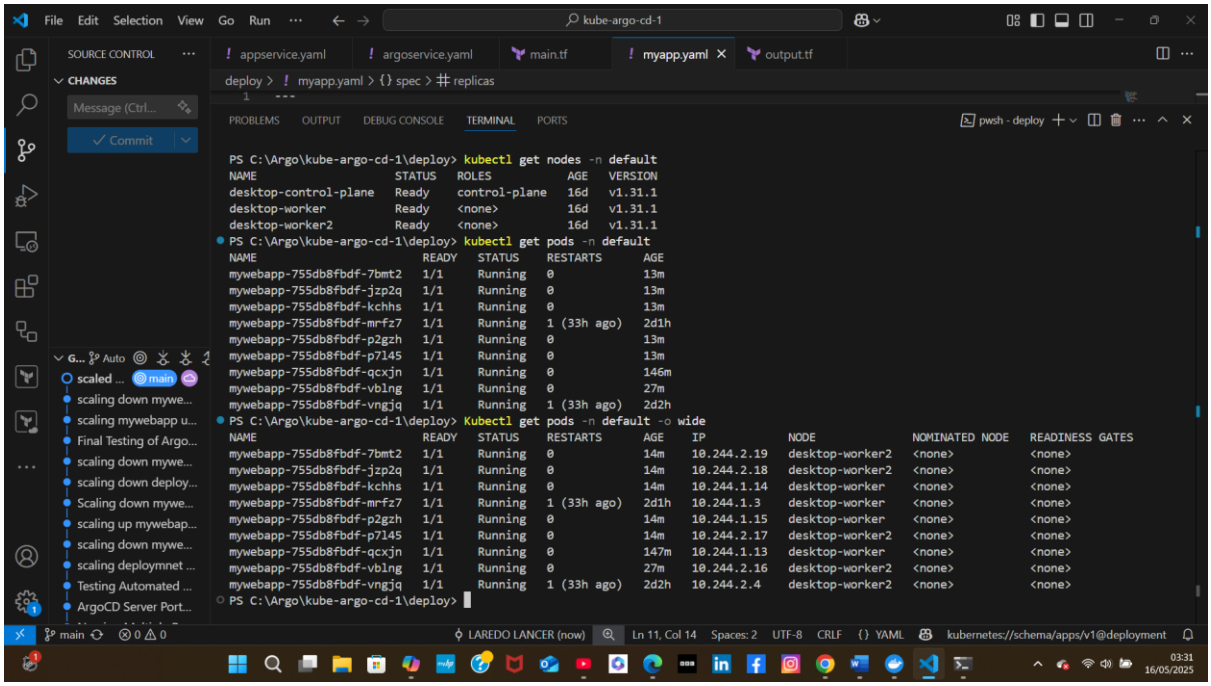
Testing Node Tainting

To simulate a **network outage**, node disconnection was achieved using **tainting**. However, before applying the taint, it was crucial to determine which pods were already scheduled on the affected nodes. To achieve this, I **scaled the mywebapp deployment to 9 replicas** and verified the **distribution of pods across worker nodes** to assess the impact of tainting on pod placement and scheduling behavior.

Replicas: 9



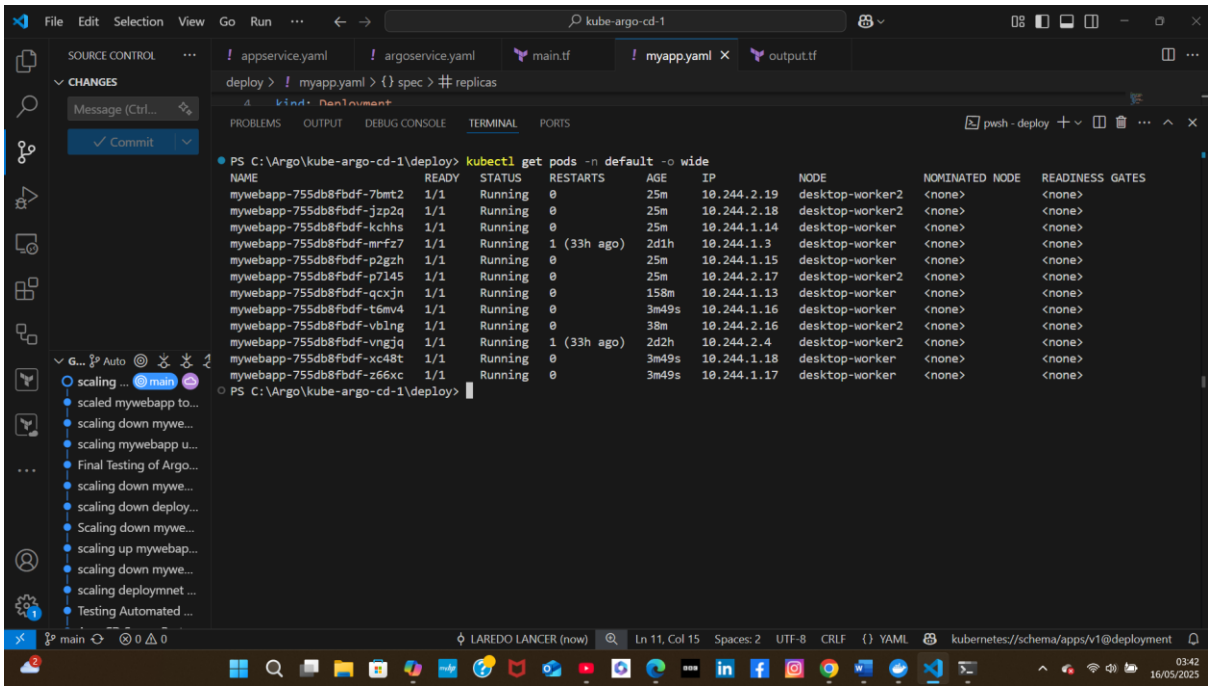
Pods Scheduled on Nodes

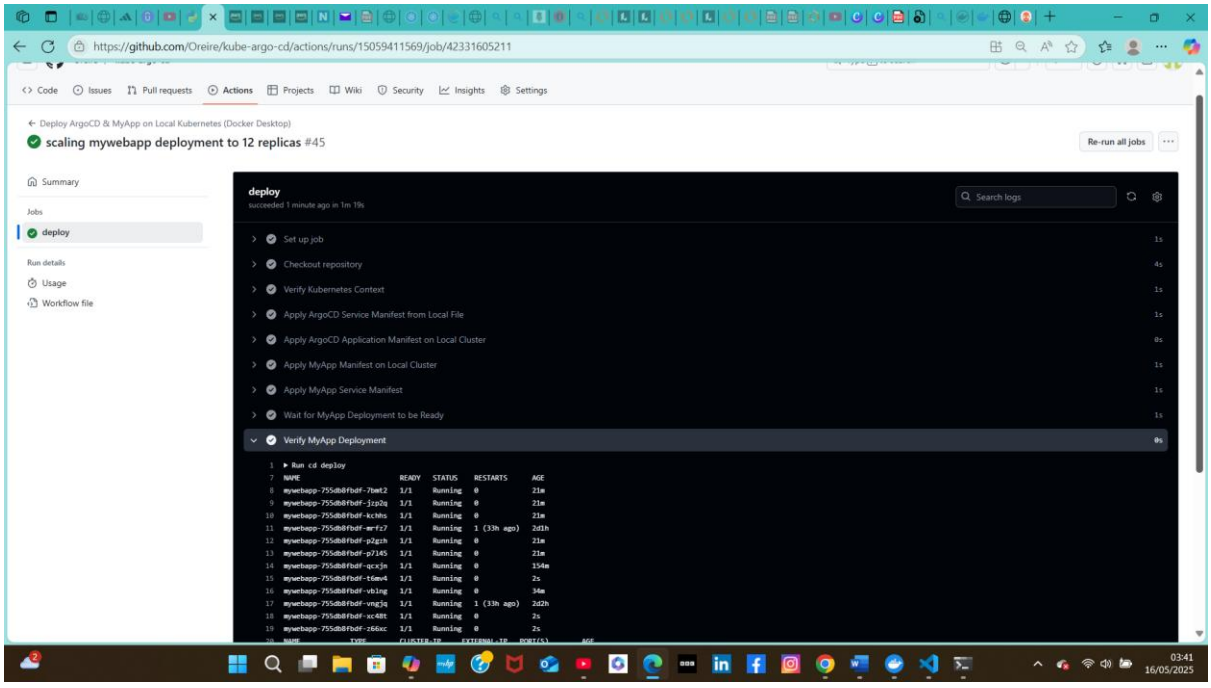


Tainting of desktop-worker2

kubectl taint nodes desktop-worker2 key=value:NoSchedule

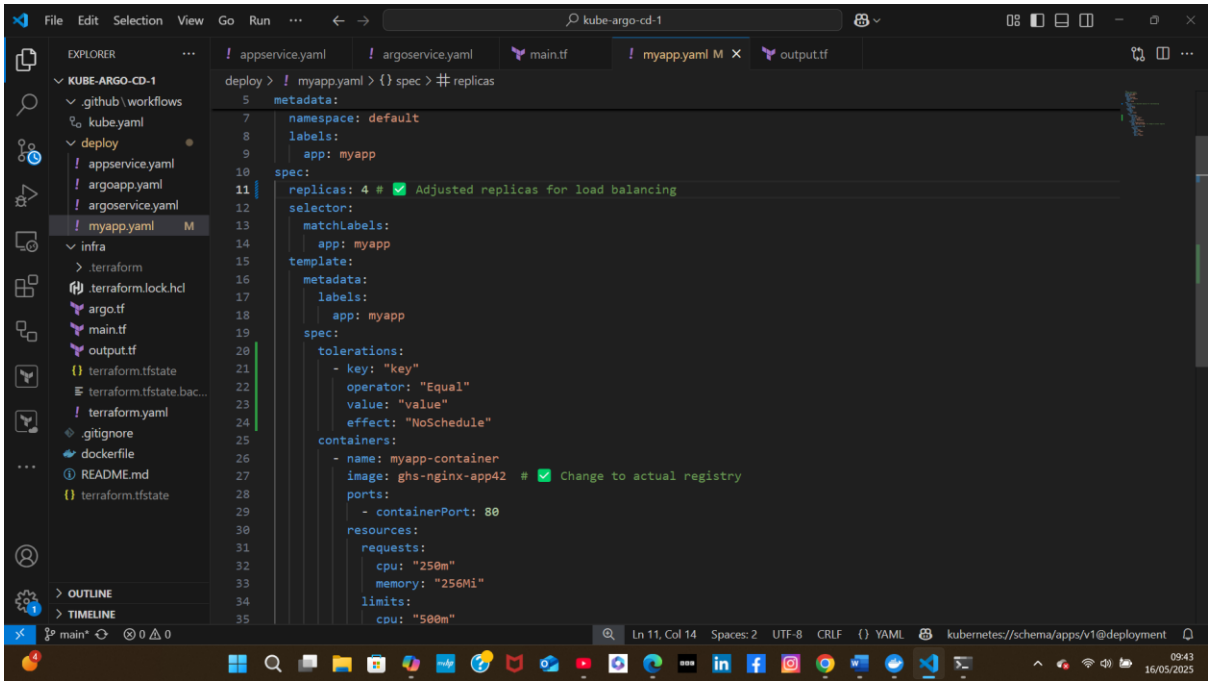
After applying the taint, I **scaled my web application deployment to 12 replicas** to accommodate an additional 3 pods. Based on the taint restrictions, these new pods should be **scheduled exclusively on available nodes**, ensuring they are provisioned only on desktop-worker while avoiding desktop-worker2. ArgoCD automatically reconciled the changes, enforcing deployment rules and rescheduling pods, accordingly, as shown in the results below.



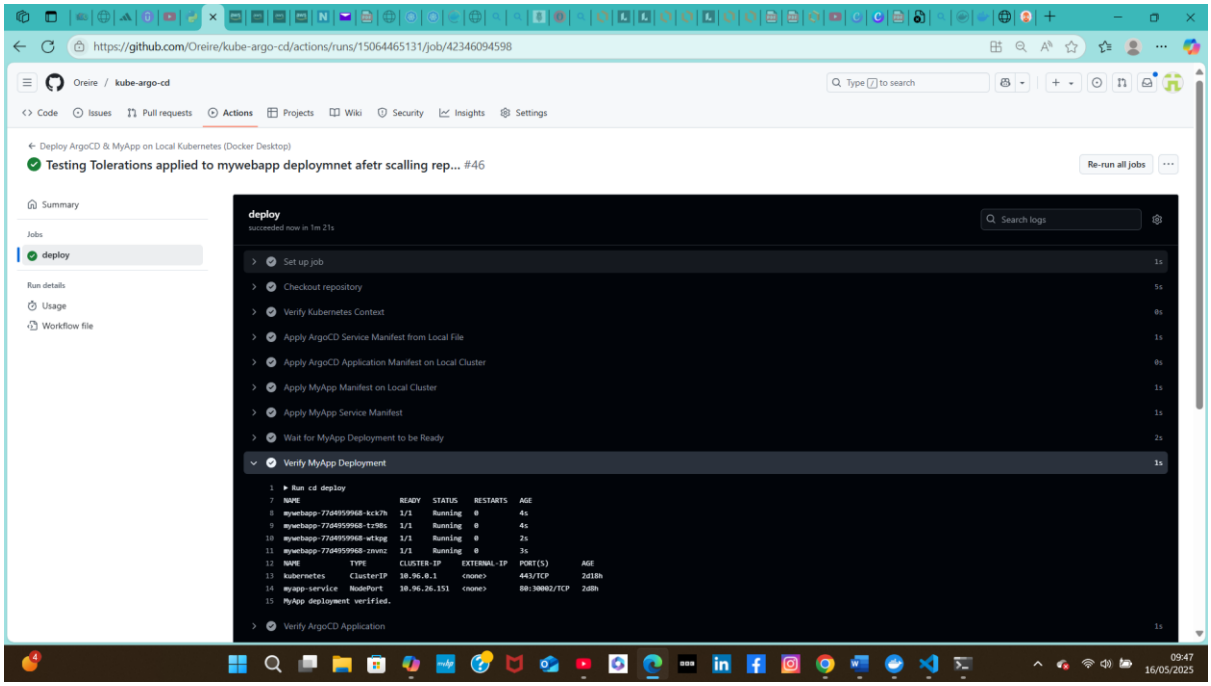


Testing Pod Tolerations

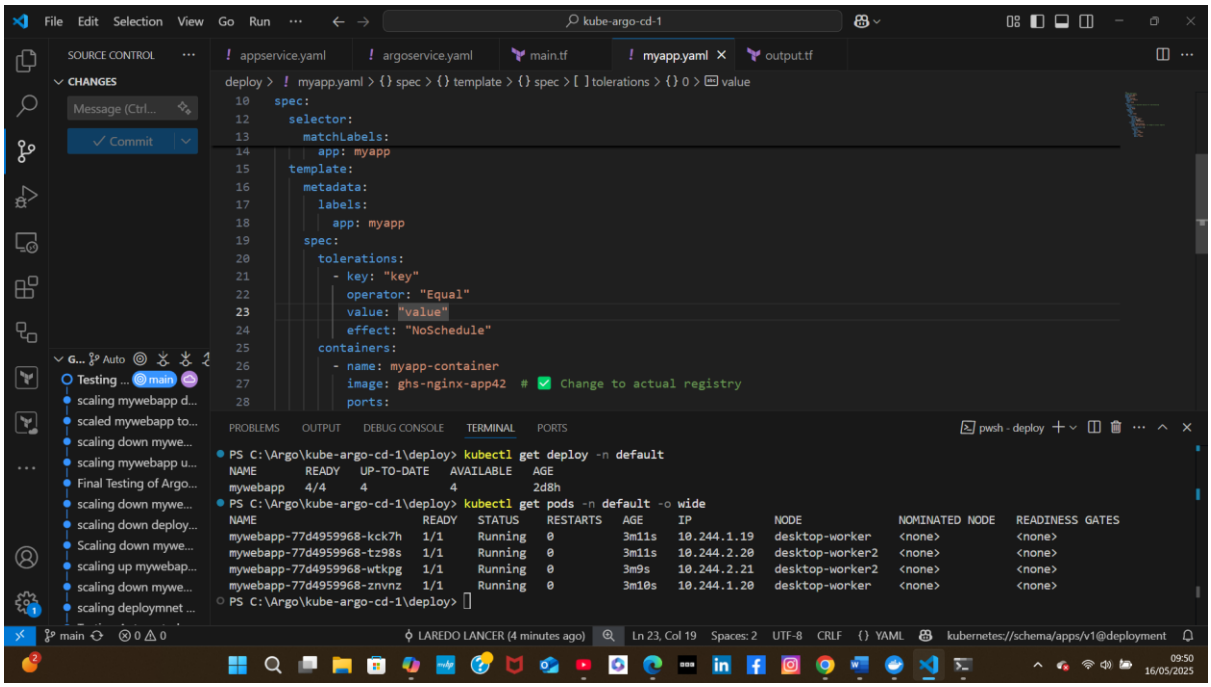
Scaling down deployment to 4 replica and applying tolerations



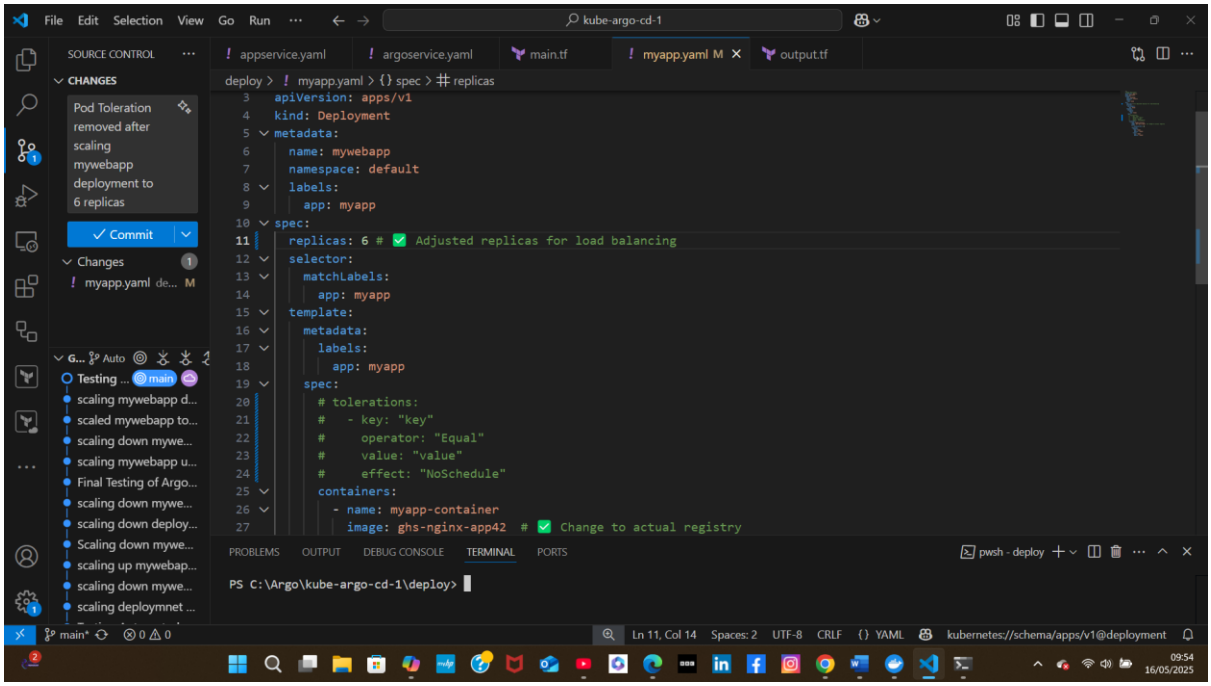
Pipeline Success



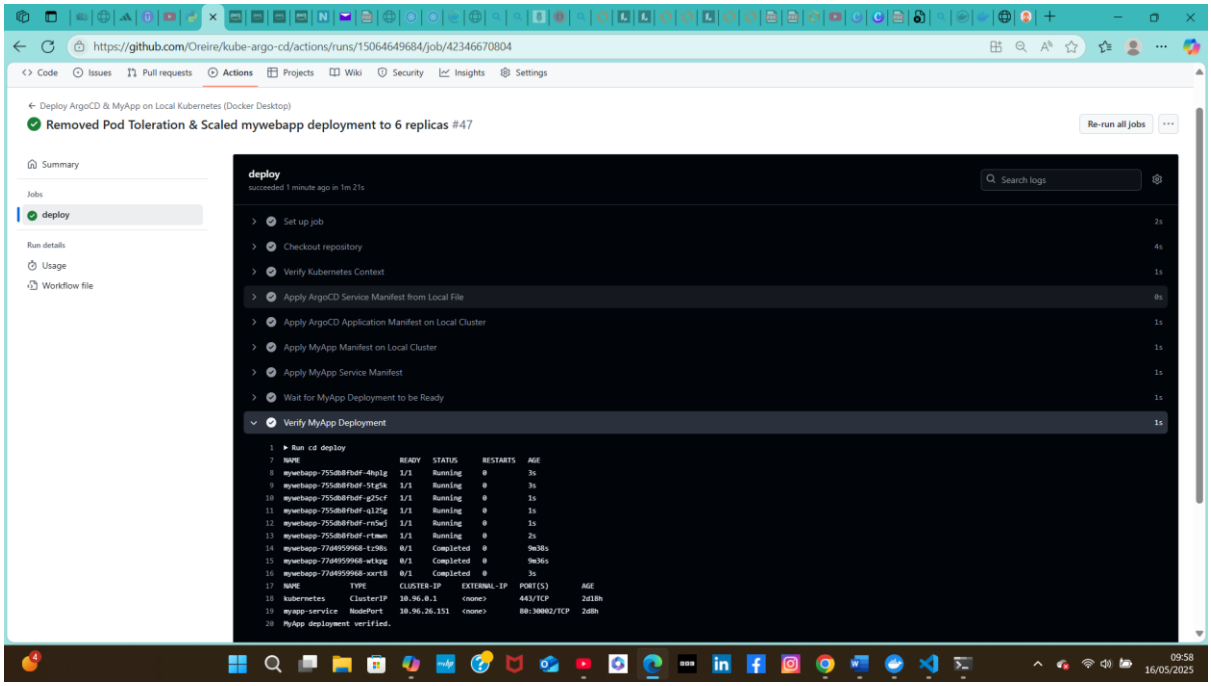
Two (2) Mywebapp pod with tolerations deployed on the tainted Desktop-worker2 Node.



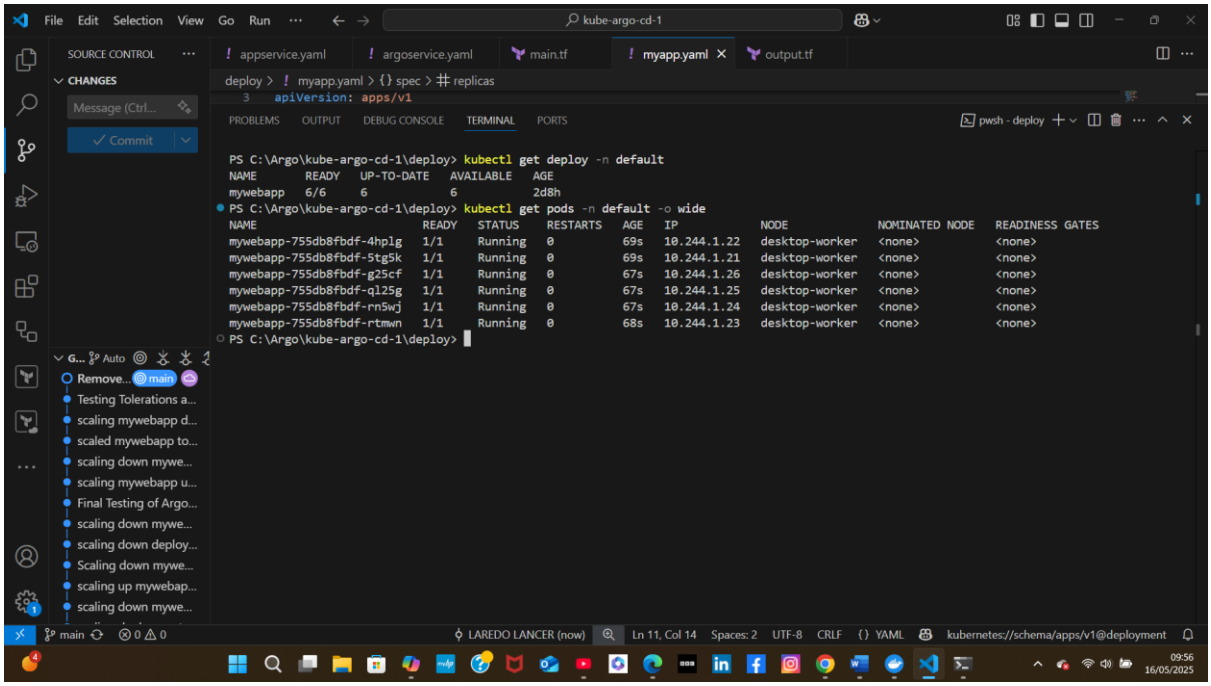
Removing Tolerations and scaling up deploy to 6 replicas.



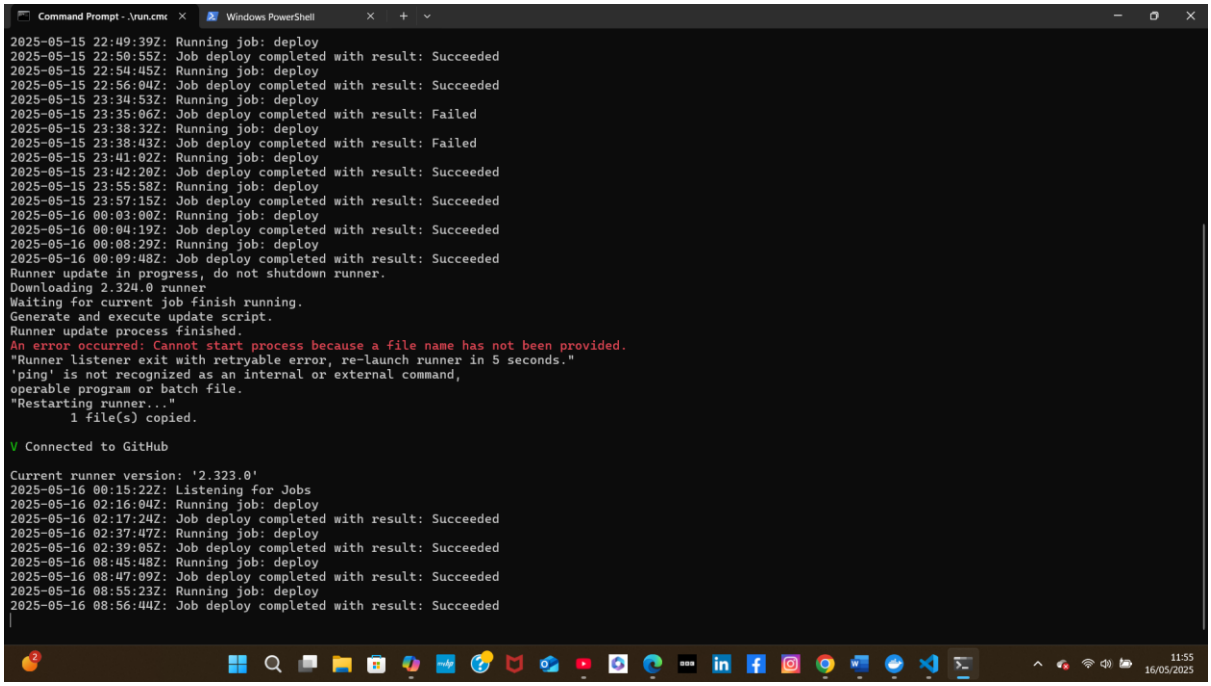
Pipeline success



All six (6) Mywebapp pod with tolerations deployed on the tainted desktop-worker Node.



The Local Self-Hosted Runner Portal



Project Highlights

Managing a Production-Grade Kubernetes Cluster Using Tainting & Tolerations

In a GitOps-driven deployment pipeline for Kubernetes applications—such as the one implemented in this project using ArgoCD and GitHub Actions—ensuring proper node management, workload distribution, and fault tolerance is critical. Therefore, the role of Tainting and Tolerations for was examined to enable a structured approach to node allocation and restrictions of pod scheduling based on predefined constraints to optimize security, high availability, and cluster resilience or fault tolerance thereby emphasizing the need for separation of concerns in microservice architectures.

Tainting Nodes

Taints define scheduling constraints, restricting certain pods from running on tainted nodes unless explicitly permitted. Hence, pods bypasses taint restrictions using tolerations. The format is: kubectl taint nodes <node-name> key=value:<effect>

The above allows the pod to run on nodes tainted with key=value:NoSchedule as tested in this project. The effect can be one of three scenarios as shown:

- **NoSchedule:** Prevents new pods without matching tolerations from scheduling.
- **PreferNoSchedule:** Avoids scheduling new pods but does not enforce strict restrictions.
- **NoExecute:** Evicts existing pods unless they tolerate the taint.

The scaling in or out of the **deployment** while **applying tolerations** to the new deployment achieved several key goals in **workload placement, scheduling control, and resource optimization**. Notably, the project showed how to effectively manage production-grade Kubernetes cluster deployment architectures in a few of many diverse ways such as:

1. Controlled Workload Reduction

By applying **taints to nodes** and **tolerations to pods**, Kubernetes enforces controlled scheduling, ensuring that workloads are distributed efficiently while respecting predefined constraints. This mechanism enables **scaling deployments** to achieve a **balanced redistribution of workloads**, preventing pod overcrowding on nodes and optimizing **cluster resource utilization** for enhanced performance and stability. Therefore, the integrations of tolerations with tainting helps workload isolation from infrastructure services, optimizing resource allocation.

2. Validating Toleration-Based Scheduling

Applying tolerations to the new deployment enables Kubernetes to:

- **Reschedule pods exclusively onto tainted nodes (desktop-worker2)** when their tolerations match the applied taint.
- **Prevent pods from being scheduled on restricted nodes**, ensuring workloads adhere to defined toleration policies and avoiding conflicts with existing constraints.
- **Verify that ArgoCD correctly reconciles deployment updates**, maintaining alignment with the GitOps workflow and enforcing declarative infrastructure management.

3. Ensuring High Availability & Fault Tolerance

With tolerations correctly configured:

- **Pods remain operational on permitted nodes**, ensuring uninterrupted service availability.
- **New pods are scheduled only on designated nodes**, enforcing controlled workload distribution and preventing unintended placement.
- **The cluster maintains resilience**, allowing automatic failover mechanisms to mitigate node failures and ensure stability.

4. Kubernetes Scheduler Testing

This process ensures the **Kubernetes scheduler** effectively manages workload placement by validating:

- **Pod assignment under new toleration rules**, ensuring proper scheduling on tainted nodes.
- **Rescheduling behavior**, confirming that pods are deployed on available nodes while adhering to defined deployment constraints.
- **Reaction to GitOps-driven updates**, verifying that ArgoCD correctly reconciles changes, maintaining consistency between the Git repository and the Kubernetes cluster.

5. Infrastructure Optimization

Refining **toleration rules** has enabled:

- **Efficient resource utilization**, ensuring workloads are scheduled on targeted nodes for optimal performance.
- **Proper workload isolation**, preventing deployment drift and maintaining infrastructure consistency.
- **Seamless integration with GitOps workflows**, enforcing declarative control over infrastructure changes to enhance automation and reliability.

Conclusion

This project established a **GitOps-driven deployment pipeline**, leveraging **Infrastructure-as-Code, ArgoCD, and GitHub Actions** to enhance **automation, security, and reliability**. By maintaining infrastructure declaratively within a **Git repository**, Kubernetes continuously reconciles deployments with the desired state, minimizing manual intervention and configuration drift.

Rigorous testing validated its relevance for **production environments**, focusing on the impact of **taints and tolerations** on workload placement. Scaling deployments with tolerations enabled effective testing of **controlled workload distribution, Kubernetes scheduler behavior, and GitOps-driven deployment integrity**, reinforcing **high availability and fault tolerance**.

Despite its simplicity, this project offers **flexibility for customization**, making it applicable in real-world **production environments**. By streamlining **Kubernetes management**, it optimizes **cluster efficiency, reliability, and automation**, delivering a scalable solution through **ArgoCD and GitHub Actions**.

Next Steps for Optimizing Project

1. **Implement Role-Based Access Control (RBAC) in ArgoCD for Enhanced Security**
   Strengthen **access control** in ArgoCD by defining **RBAC policies**, restricting permissions based on roles such as **admin, developer, and viewer**. This enhances security by preventing unauthorized modifications and ensuring controlled access to deployed applications.
2. **Using ArgoCD ApplicationSets to Deploy Multiple Environments Dynamically**
   Use **ApplicationSets** to automate multi-environment deployments (**dev, staging, production**). Configure **template-based** application definitions to simplify GitOps workflows, enabling **dynamic application provisioning** across clusters with minimal manual intervention.