**GitOps Workflow using ArgoCD on Kubernetes**

**Loom video:** https://www.loom.com/share/f2a737a2a30d4e5c9c54bb504842f11b?sid=745bea95-acda-4902-a8c0-17a46183cf9f

**Objective:**

Implemented a GitOps workflow by syncing Kubernetes deployment states directly from a Git repository using ArgoCD. The goal is to automate application deployment and updates using Git as the single source of truth.

**Tools Used**

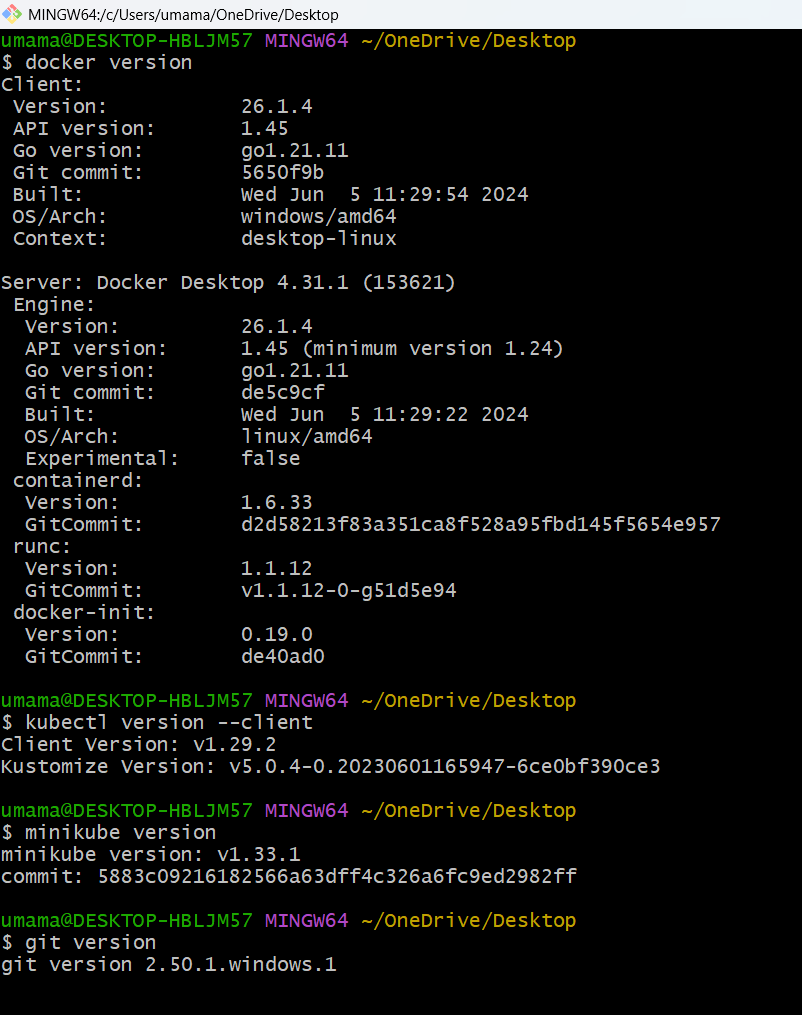
* Kubernetes: Minikube (local cluster)
* ArgoCD: GitOps continuous deployment tool
* GitHub: Source control for deployment manifests
* Docker: Containerization of applications

Prerequisites: Install and confirm these tools are available on our machine

1. Docker installed and running (or another Minikube driver).
2. kubectl installed and on your PATH.
3. minikube installed.
4. Git + GitHub account.

**Verification**

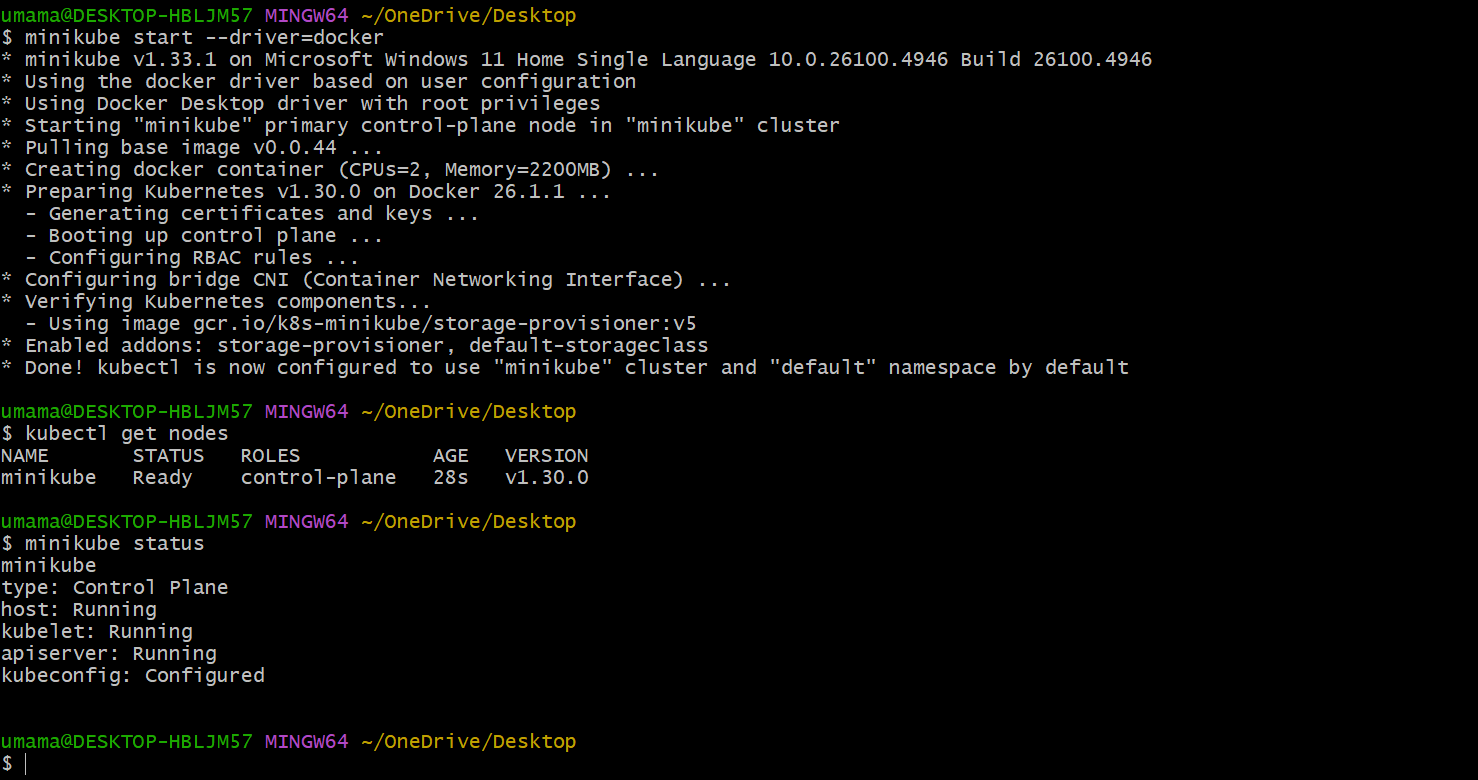
docker –version && kubectl version --client –short && minikube version && git –version



**Starting local Kubernetes cluster (Minikube)**

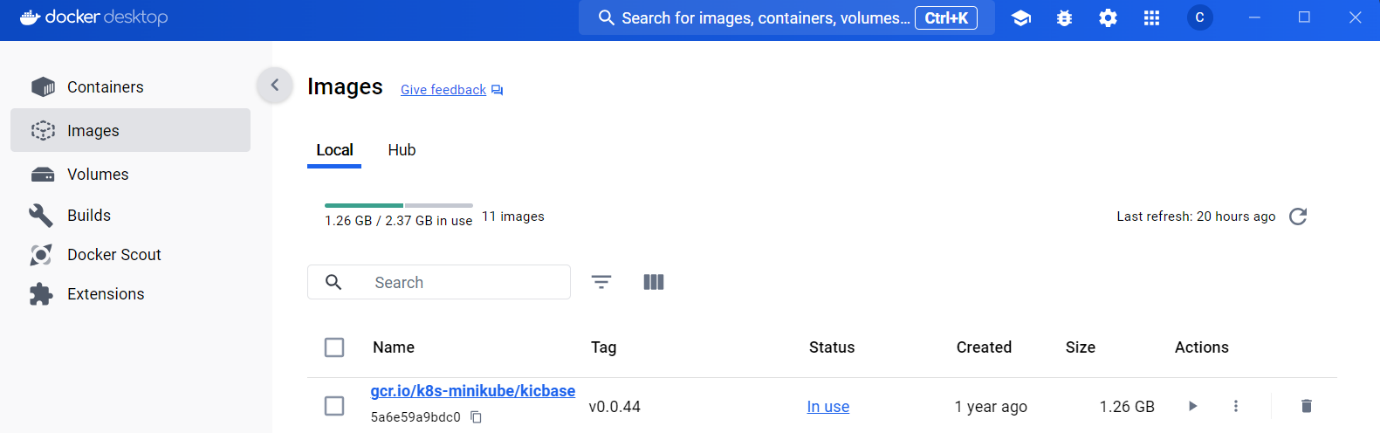
Commands: minikube start --driver=docker && kubectl get nodes

We can see status of minikube by using minikube status

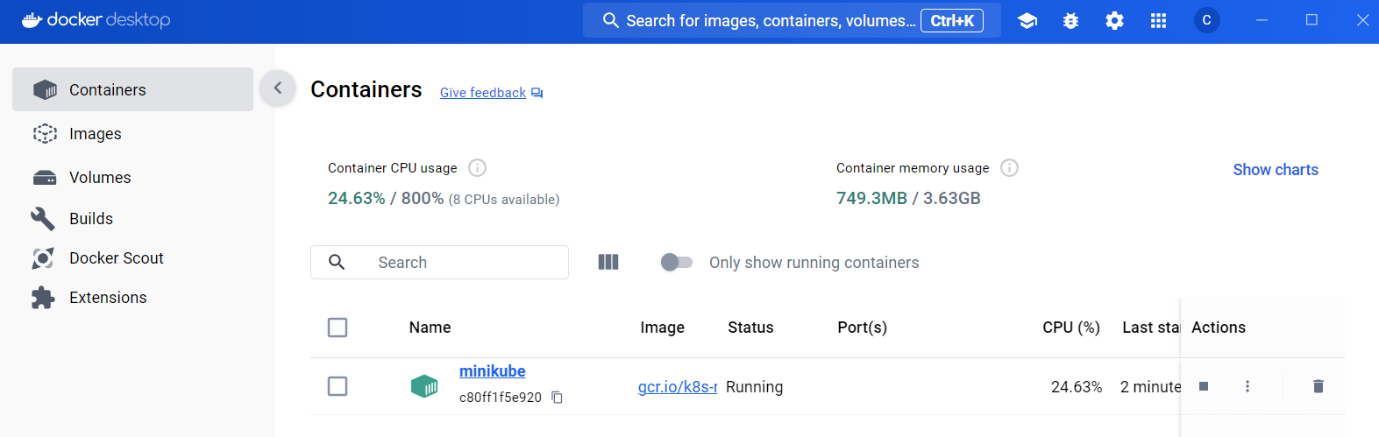


We can see the minikube container is running in docker desktop

Image:



Container:

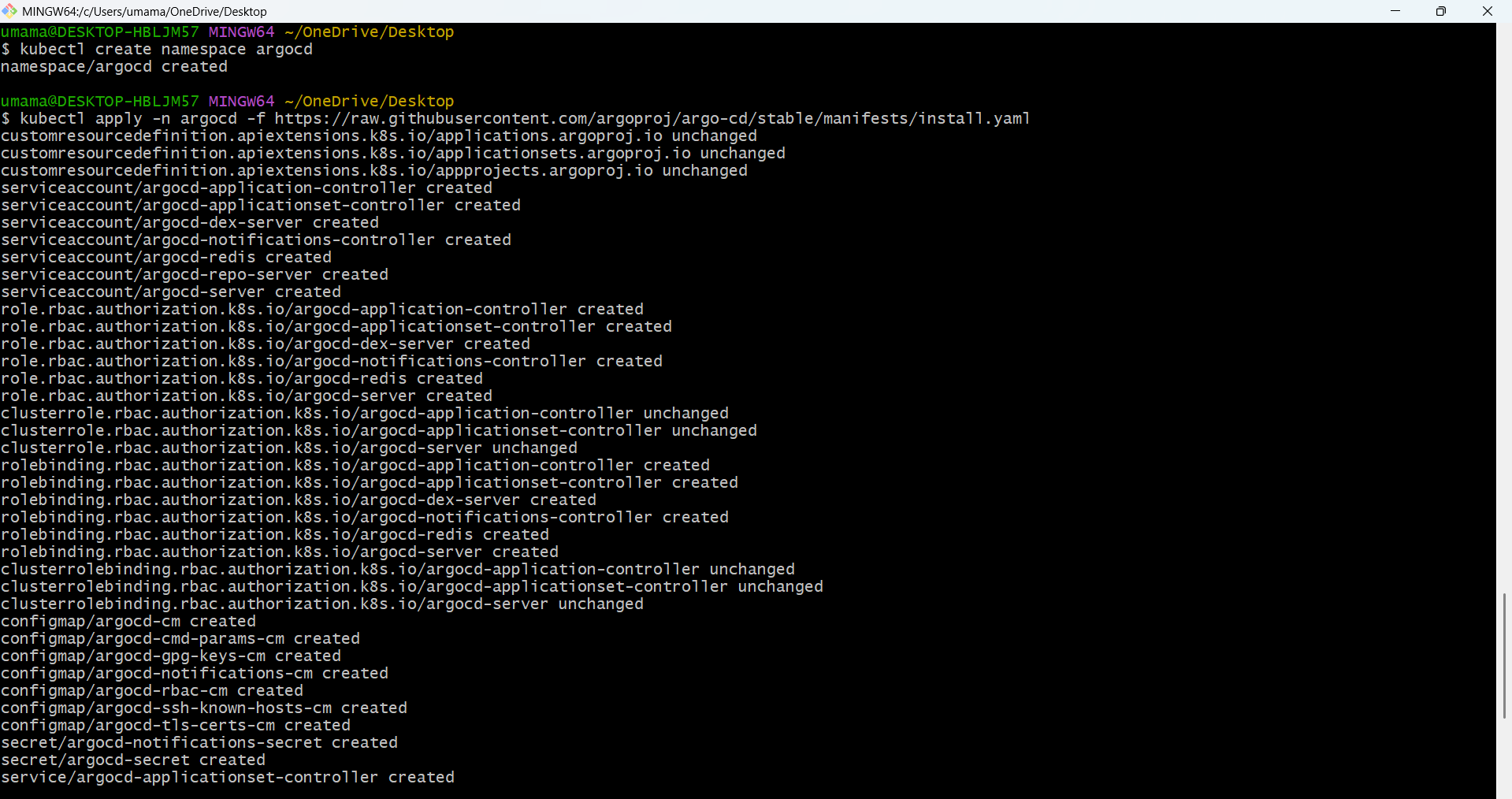


**Install Argo CD into the cluster**

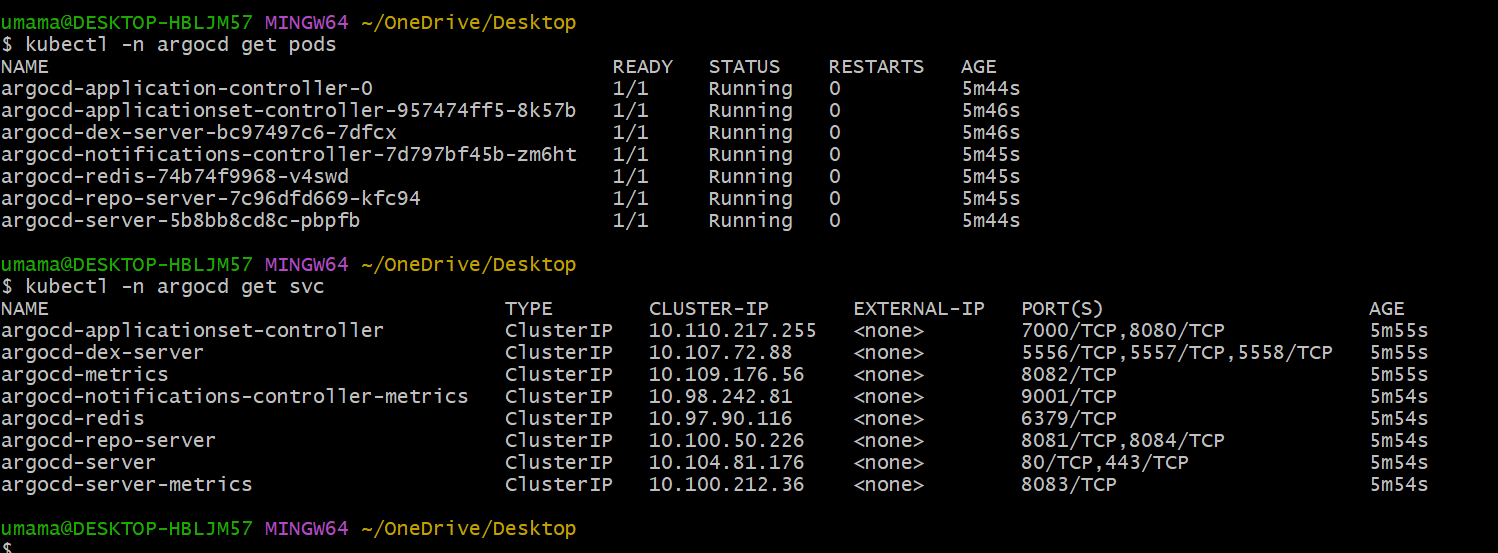
Create separate namespace: kubectl create namespace argocd

We create ArgoCD application by this command:

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

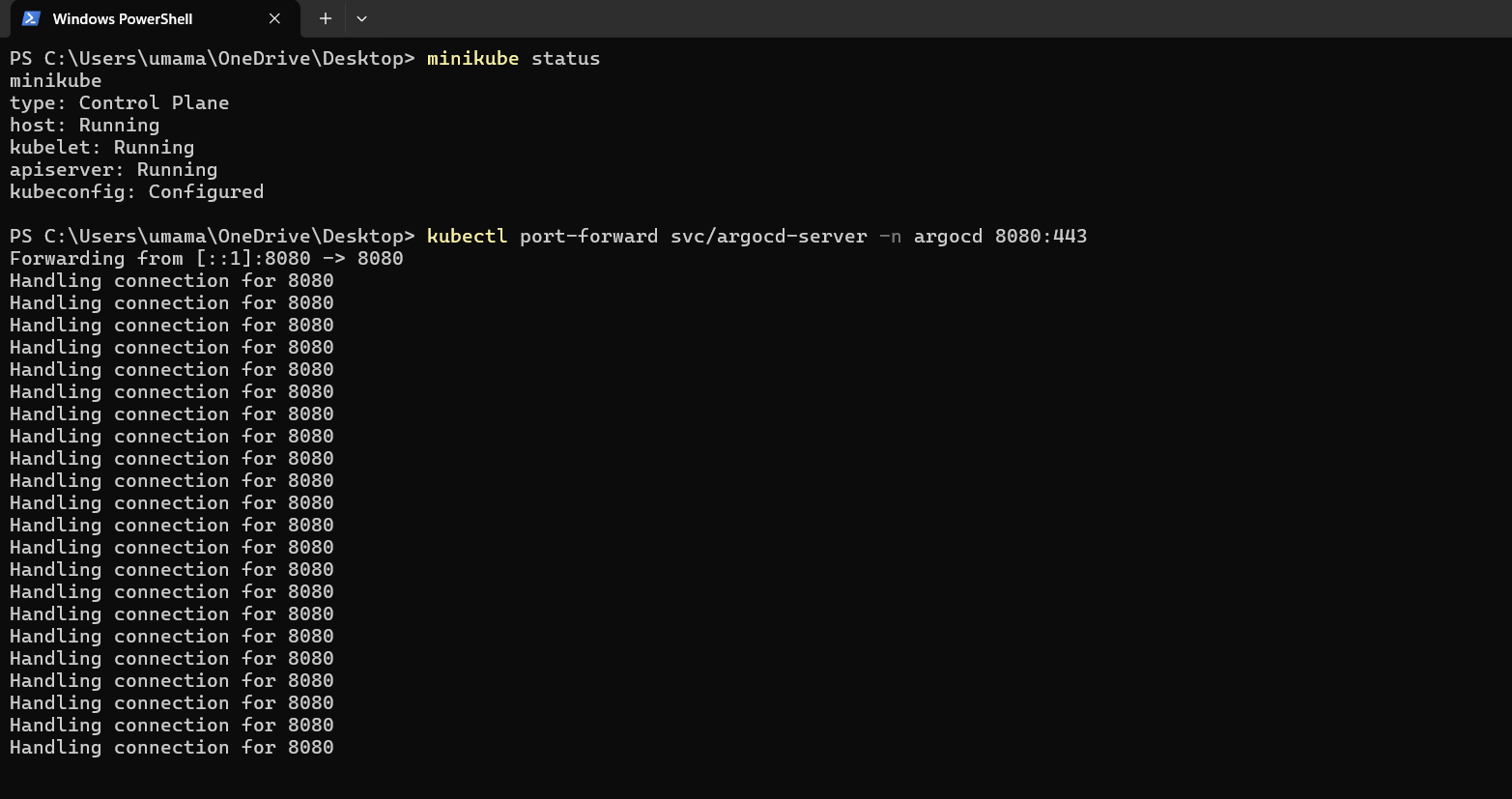


After applying check whether pods and services are running or not



**Open the Argo CD UI (port-forward)**

kubectl port-forward svc/argocd-server -n argocd 8080:443

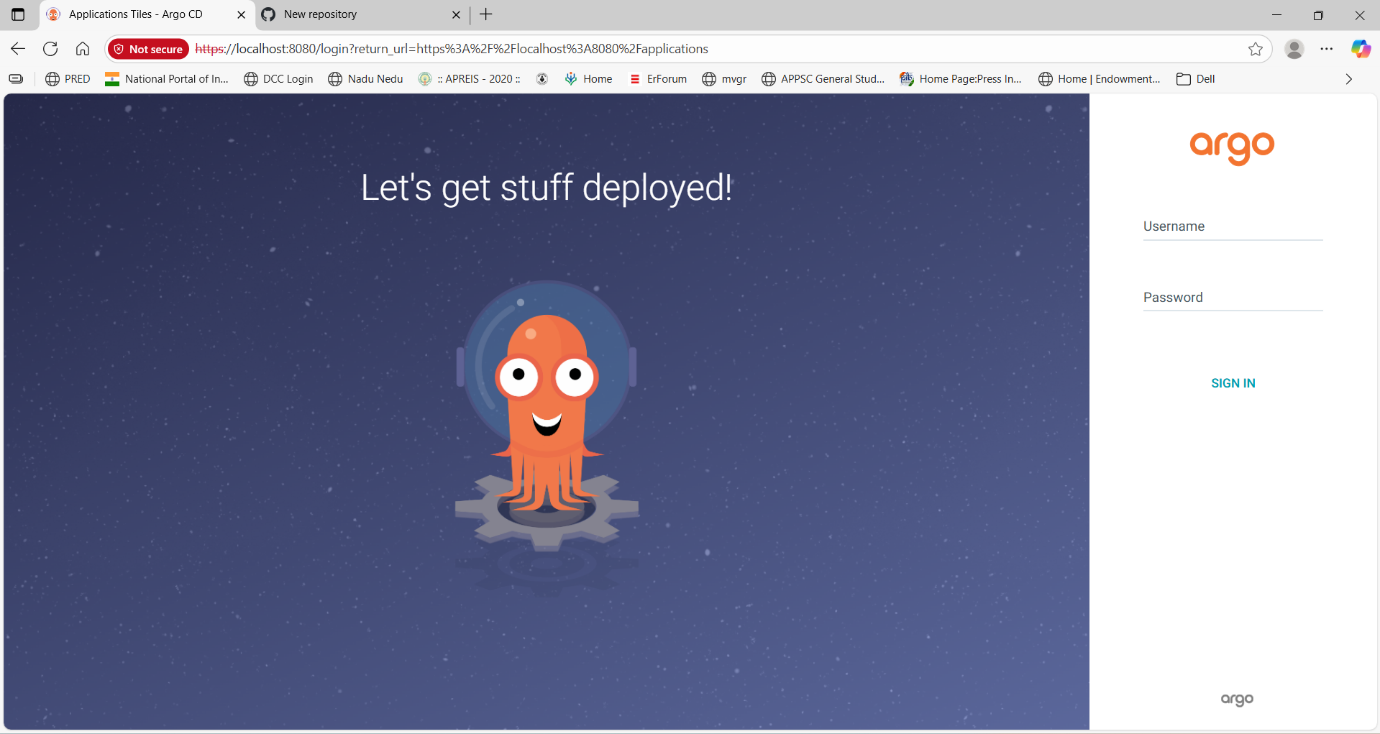


Open browser: <https://localhost:8080>

we need to accept a TLS/self-signed warning — that’s normal.

If port-forward runs successfully, we’ll see it accept connections in the terminal.

In browser you should see the Argo CD login page.

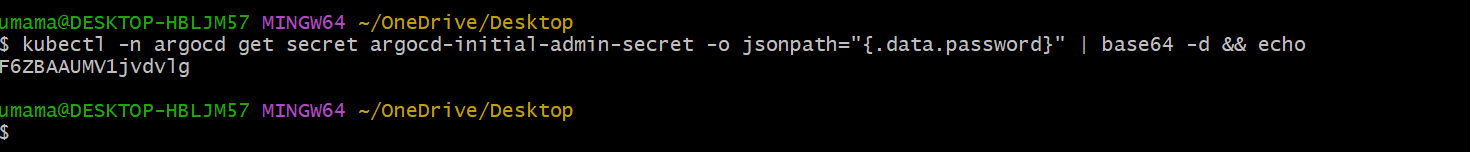


**Login to Argo CD (first time)**

Default username: admin

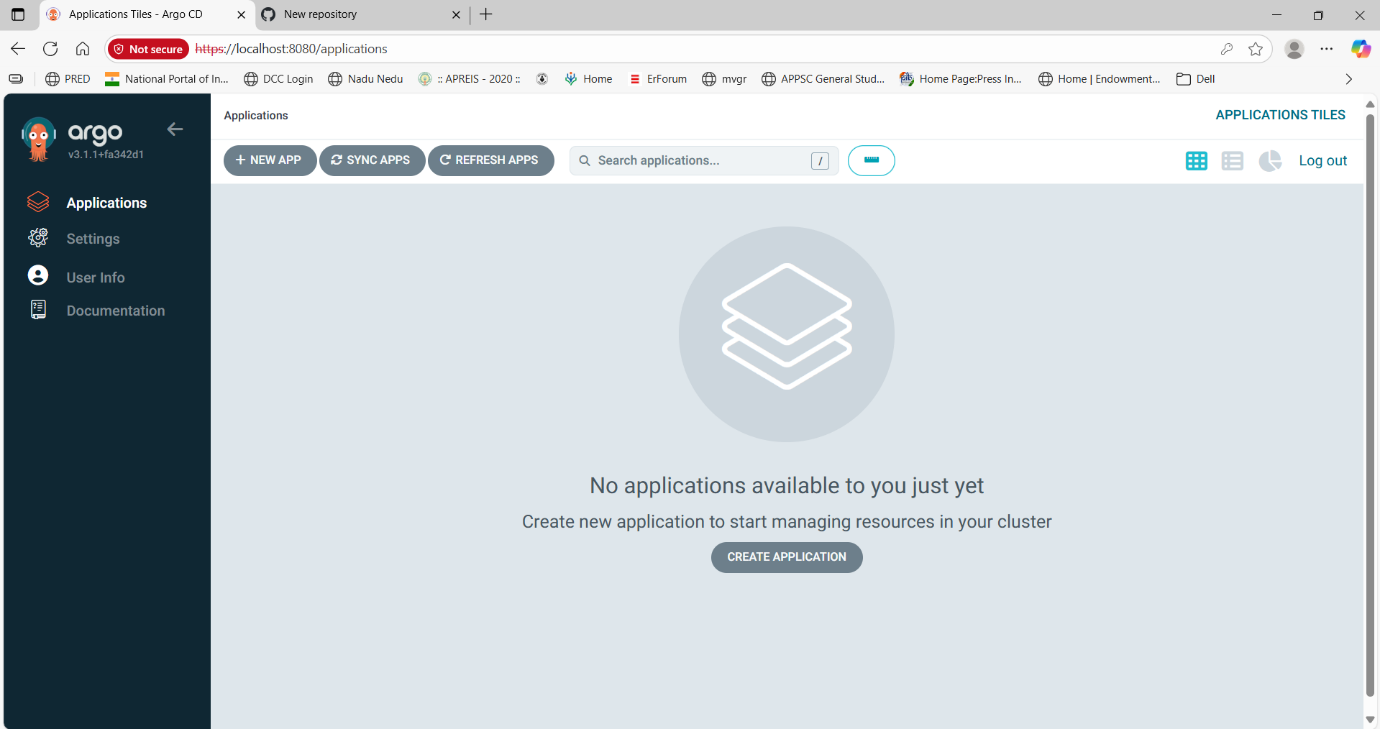
Get the initial password by running this command

kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d && echo



After login we should see the Argo CD dashboard.

In UI top-left, click **Applications** — it should be empty for now.



**Create the GitHub repository and manifest files**

Locally create the folder and files (deployment.yaml and service.yaml)

Deployment.yaml:



The deployment.yaml defines a **Kubernetes Deployment** for running the Nginx application.

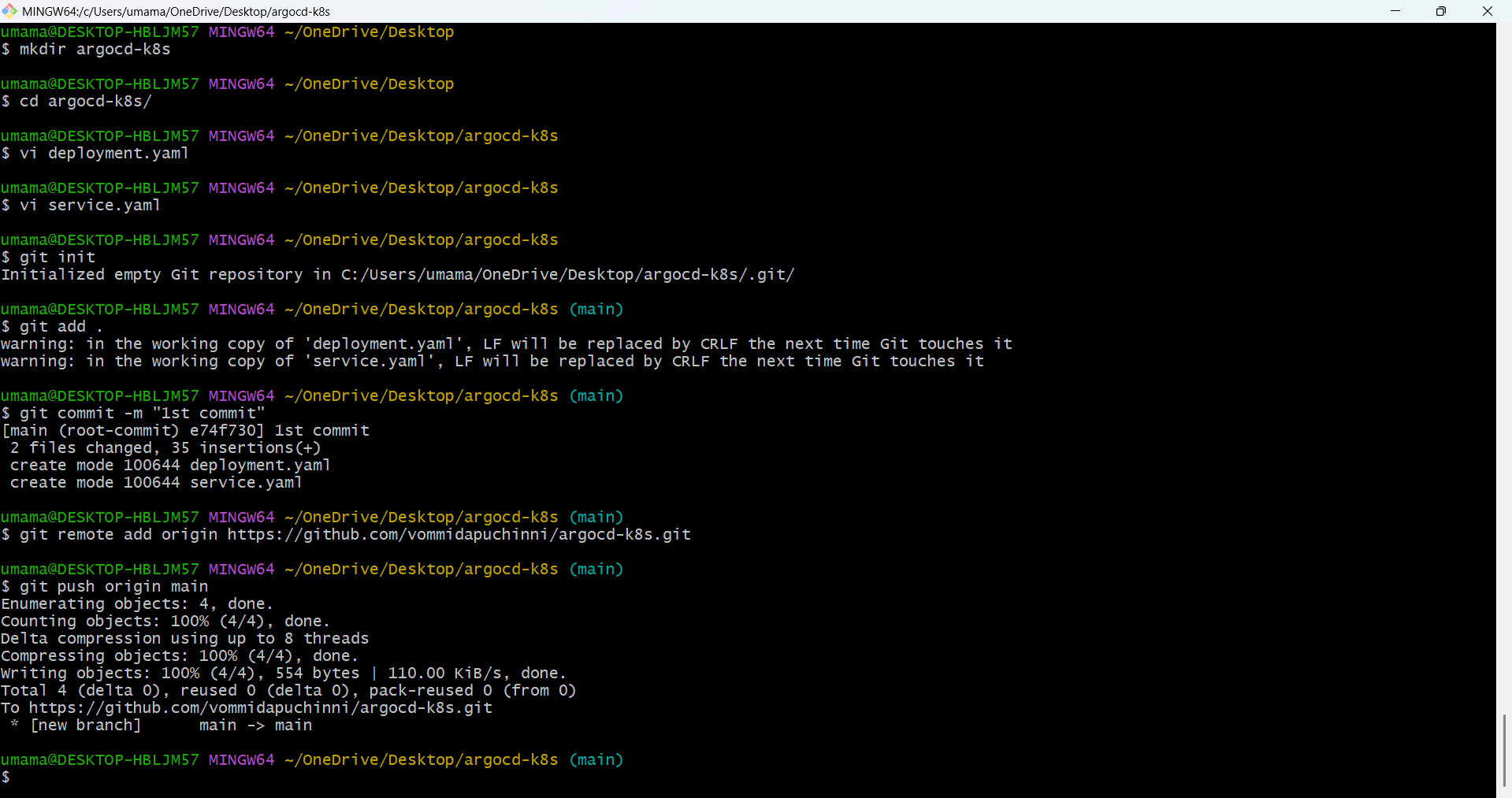
* + **apiVersion, kind, metadata**: Specifies this is a Deployment named nginx-demo.
  + **replicas: 1**: Runs only one Pod (single instance of Nginx).
  + **selector & template labels**: Ensures the Deployment manages Pods with the label app: nginx-demo.
  + **containers**: Defines one container named nginx using the **nginx:stable** image.
  + **ports**: Exposes port **80** inside the container for HTTP traffic.

Service.yaml



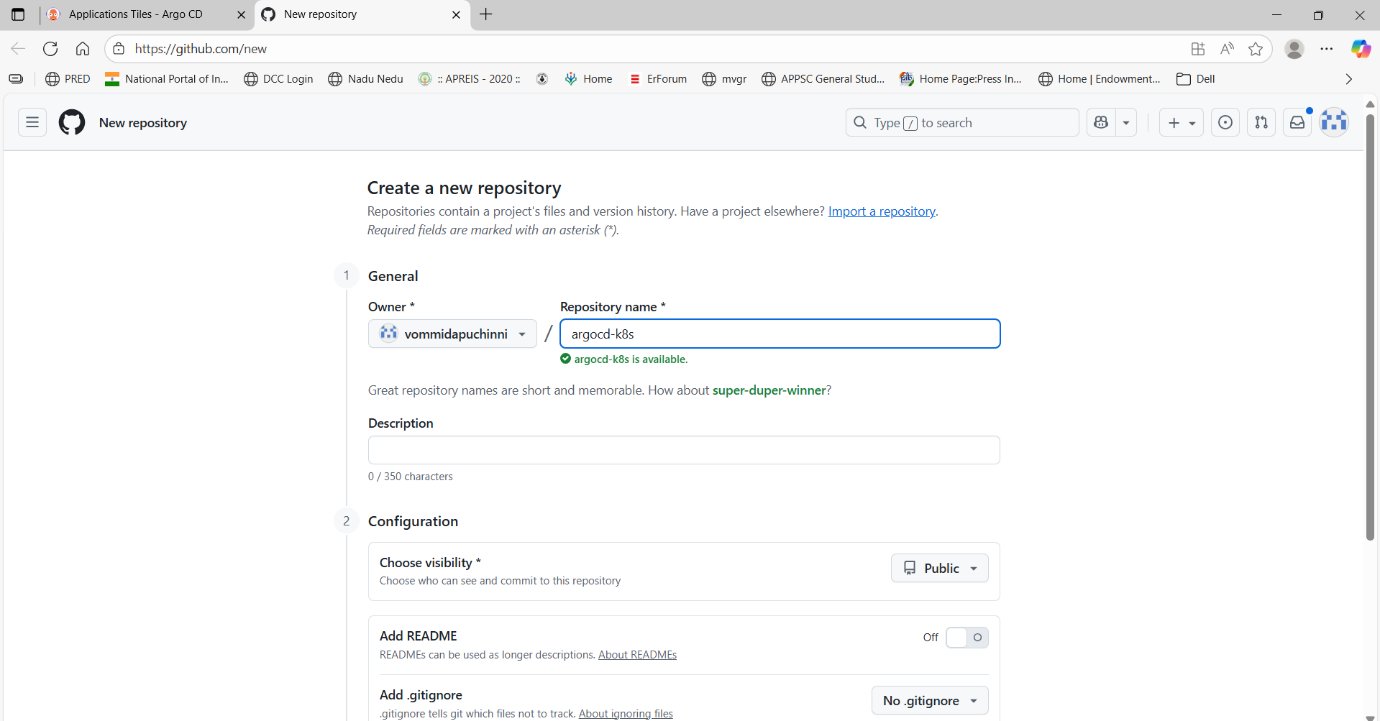
The service.yaml defines a **Kubernetes Service** to expose the Nginx Deployment inside the cluster.

* + **apiVersion, kind, metadata**: Specifies this is a Service named nginx-demo-svc.
  + **selector**: Targets Pods with the label app: nginx-demo (from the Deployment).
  + **ports**: Maps **port 80** of the Service to **port 80** of the container for HTTP traffic.
  + **type: ClusterIP**: Makes the Service accessible **only within the cluster** (not externally).

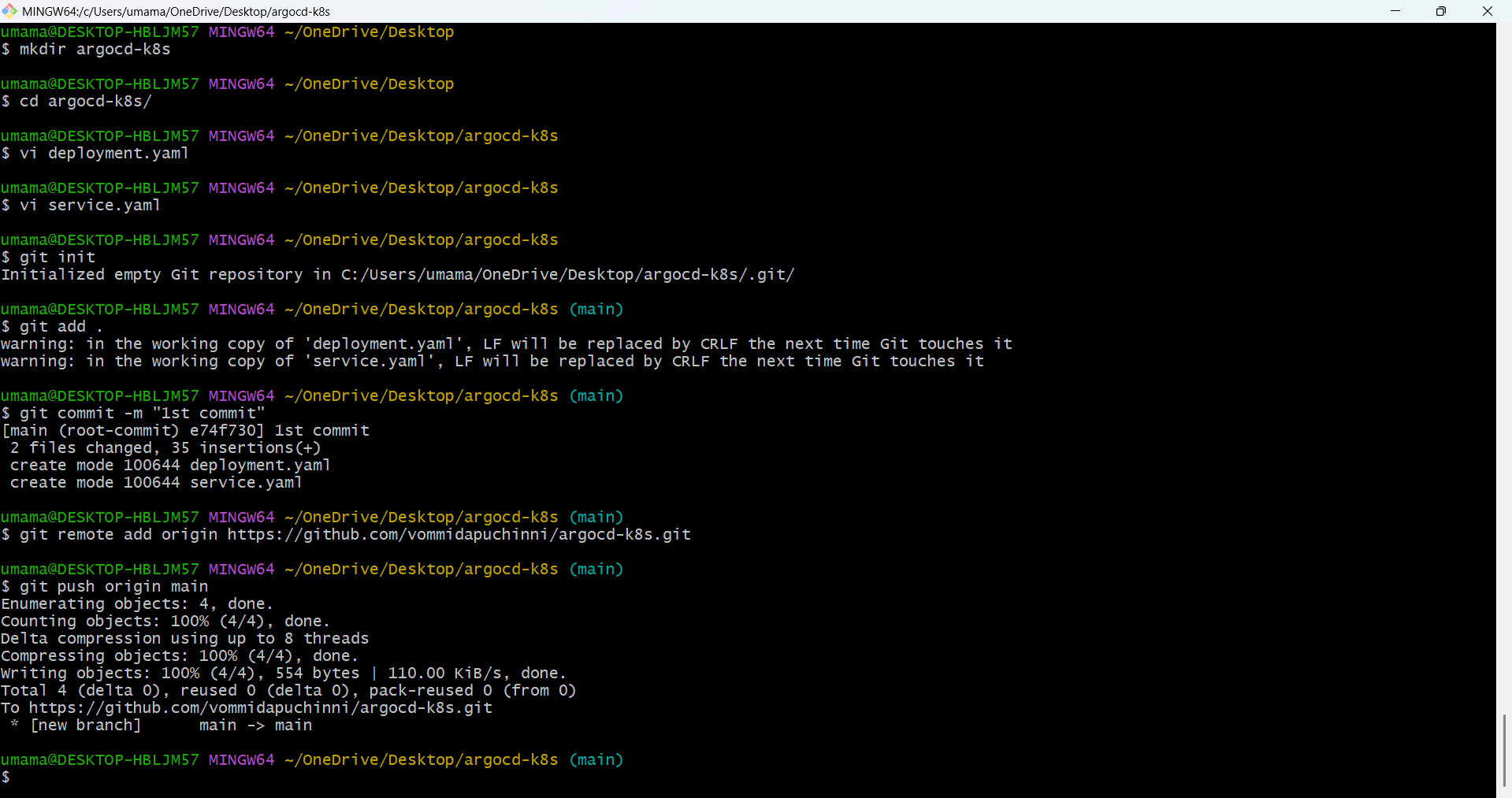


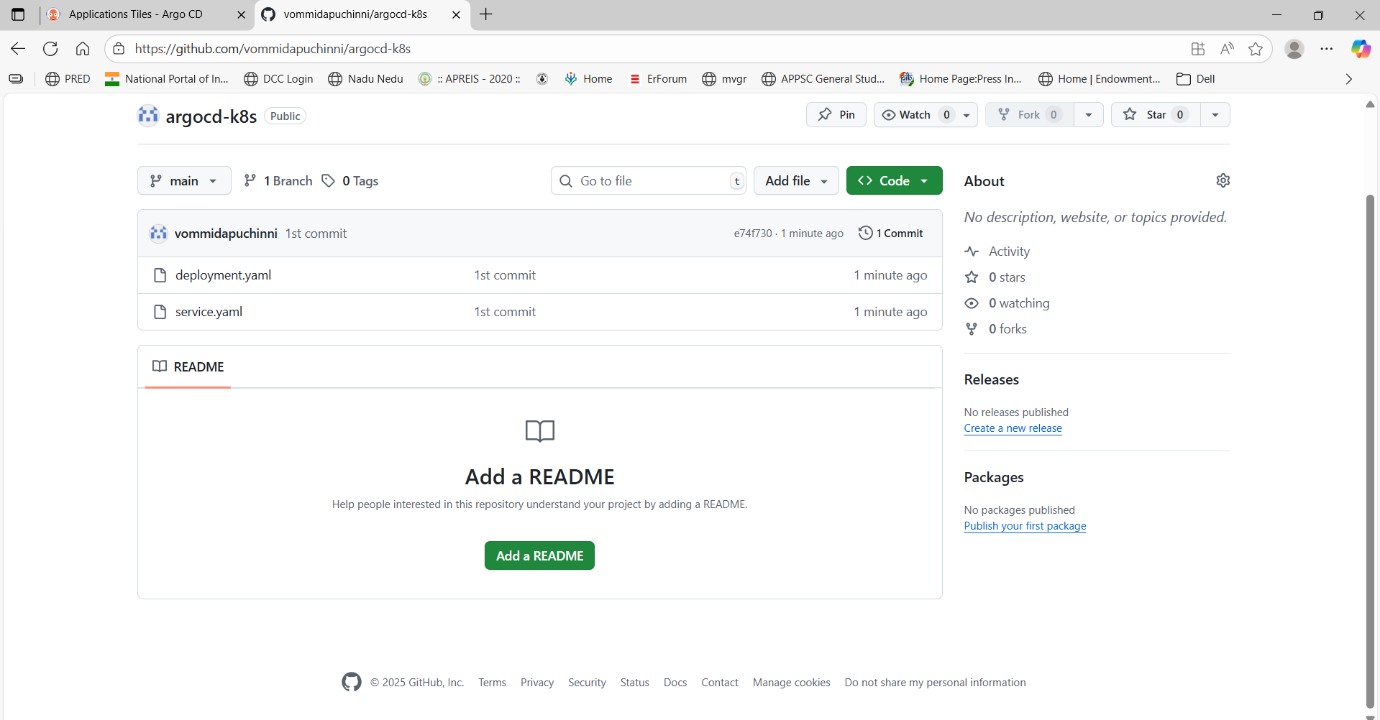
Open github in browser

Right click on + new repository gives name of the repo and give name and keep it public and with initiating readme file click created



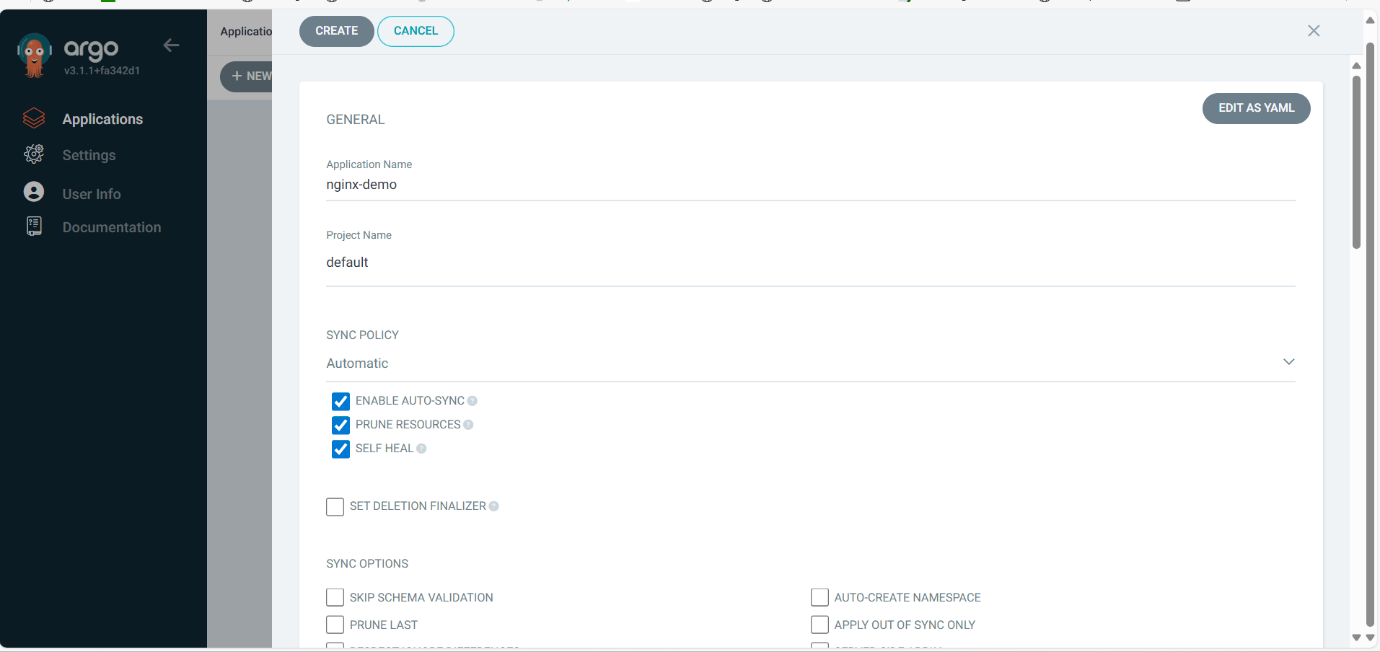
Push files to GitHub



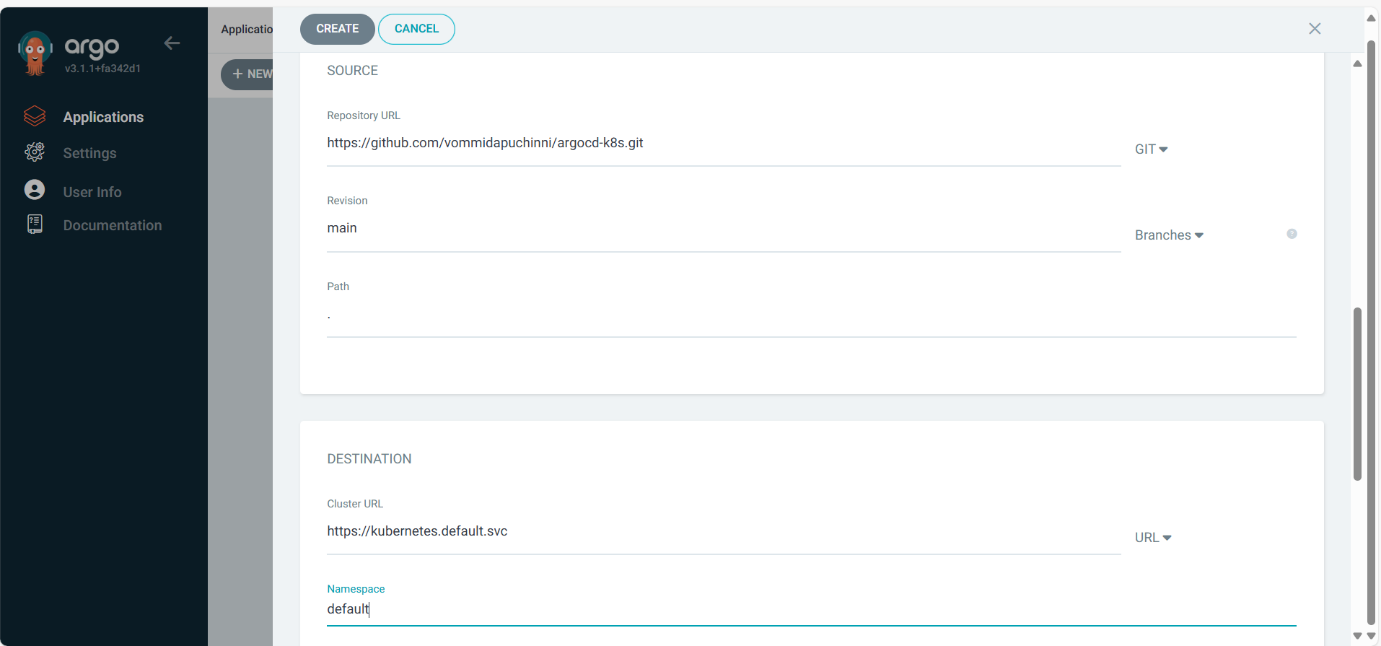


**Create the Argo CD Application**

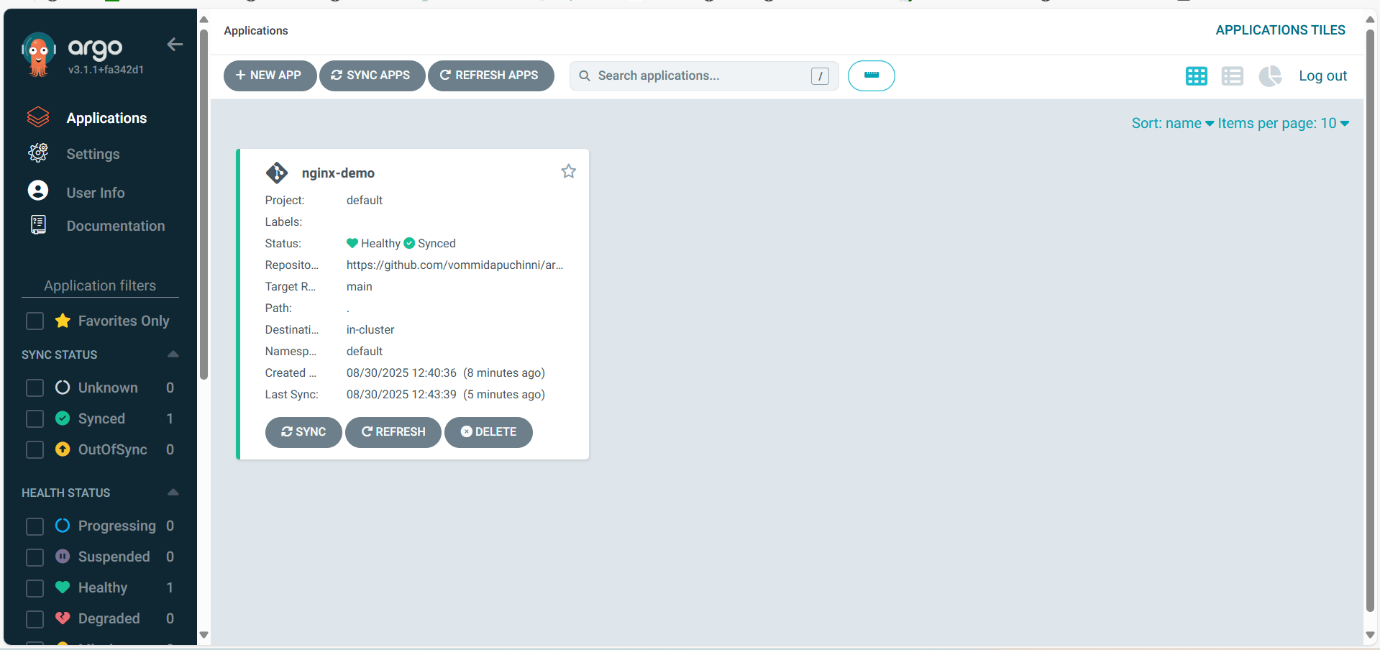
* In Argo CD UI left sidebar click Applications.
* Click NEW APP (top-right).
* Fill the form (exact values):
* Application Name: nginx-demo
* Project: default
* Expand Sync Policy and choose Automatic (tick Prune and Self-Heal if shown).

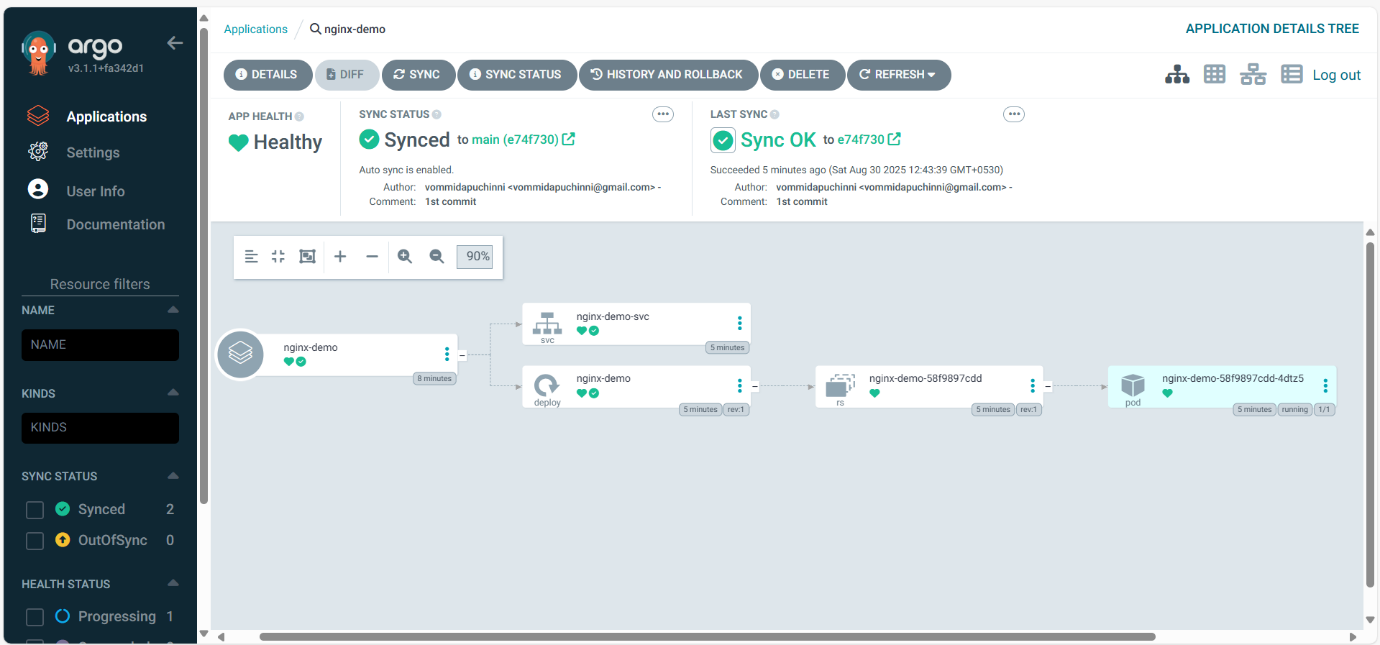


* Repository URL: https://github.com/vommidapuchinni/my-argocd-demo.git
* Revision: main
* Path: .
* Destination Cluster: https://kubernetes.default.svc
* Destination Namespace: default
* Click Create.



* In the Argo CD **Applications** list the nginx-demo app should appear.
* App status should move from OutOfSync → Synced (if automatic) and **Healthy**.
* In UI click the app → resource tree should show the Deployment and Service





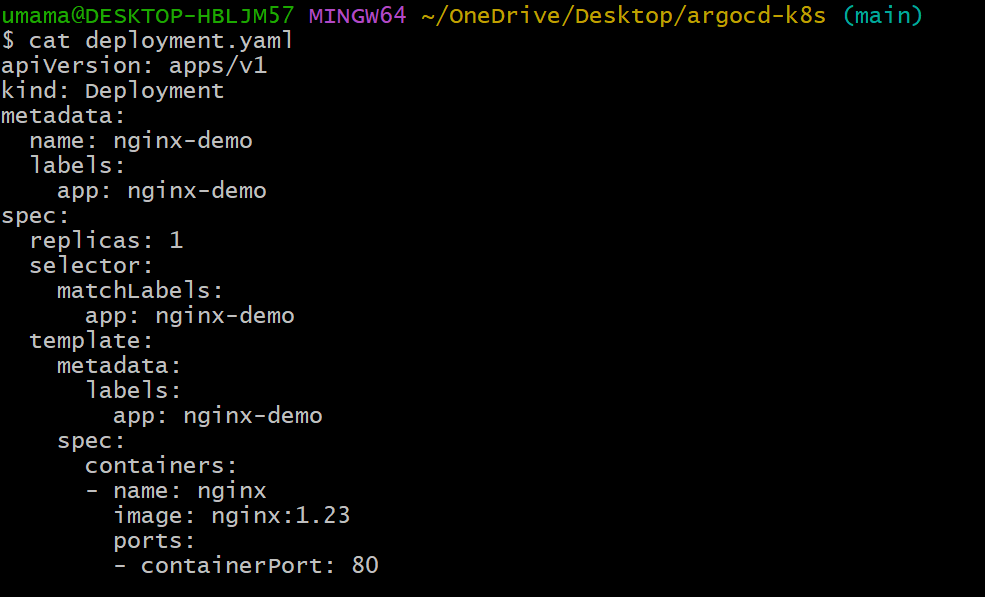
Command-line verification:



**Test the GitOps flow (make a change in yaml files and push to github and watch Argo CD apply it)**

Update the image version to show GitOps flow

* Changed nginx:stable → nginx:1.23

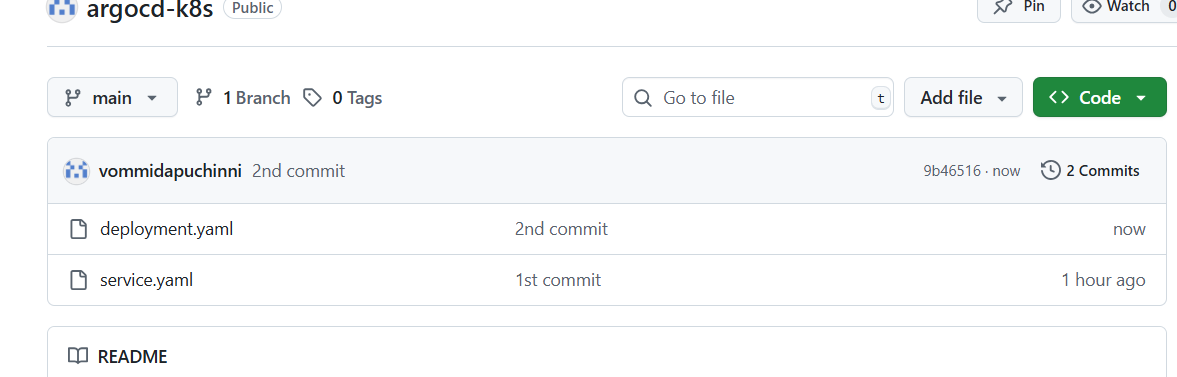


**Commit & push**

git add deployment.yaml

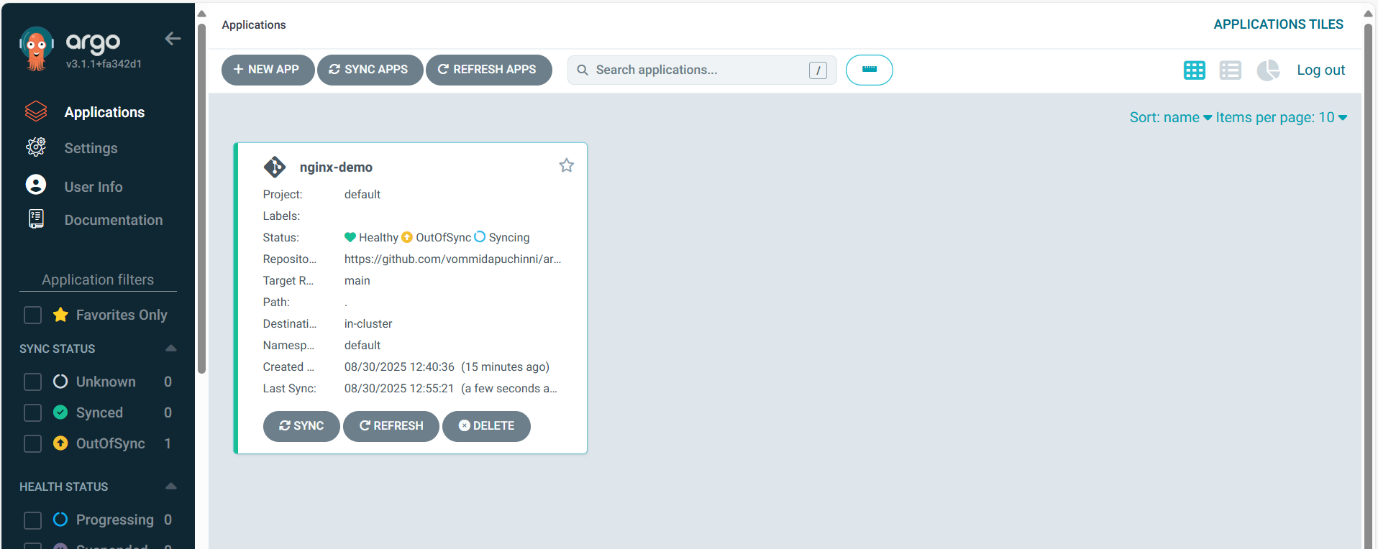
git commit -m "2nd commit"

git push origin main



ArgoCD automatically syncs (since Auto-Sync is enabled)

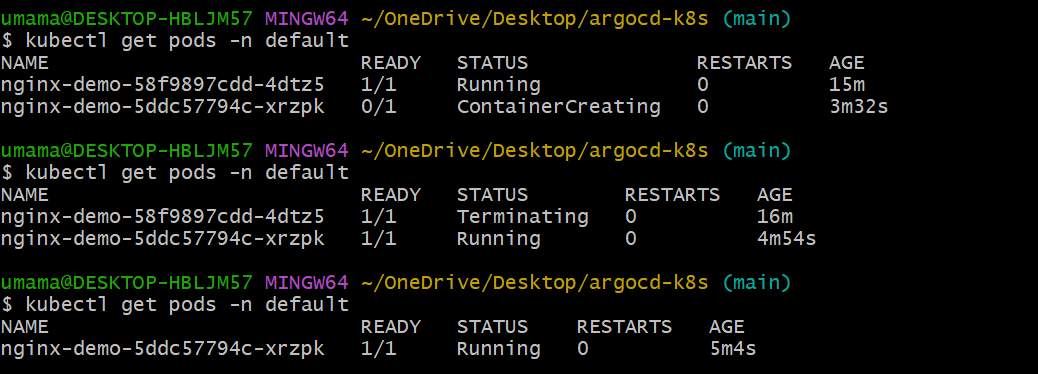
* UI: Shows OutOfSync → syncing → Synced & Healthy



Verify deployment

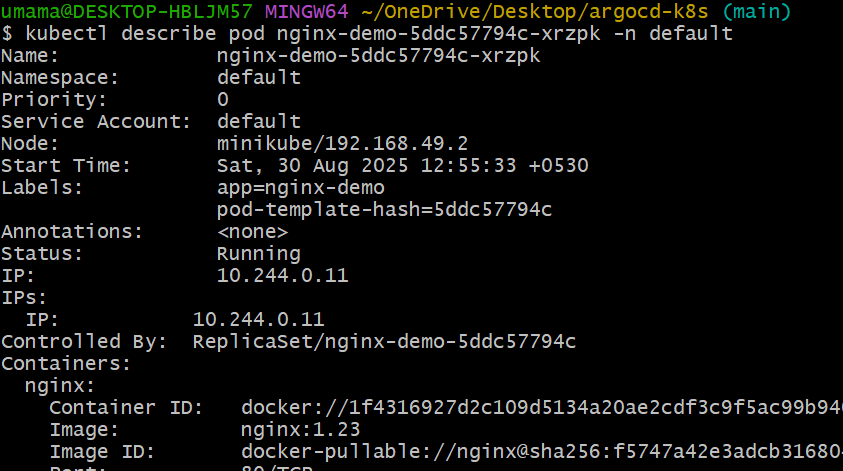
kubectl get pods -n default

kubectl describe pod <pod-name> -n default

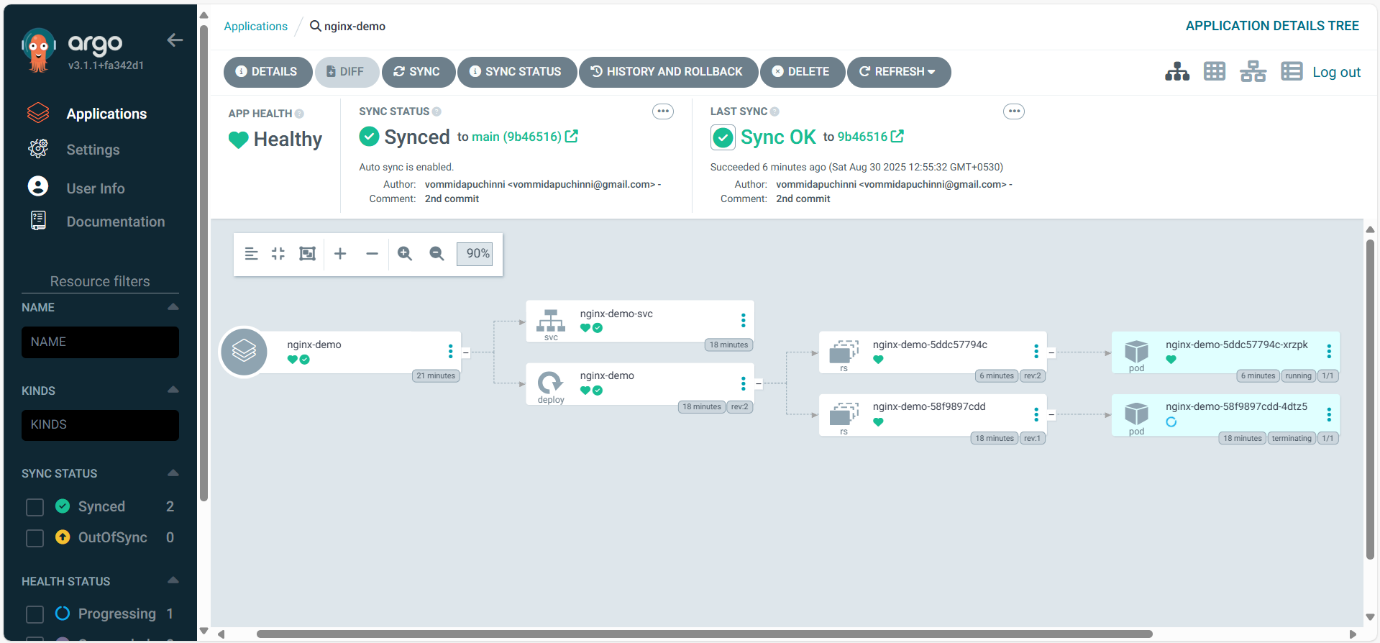


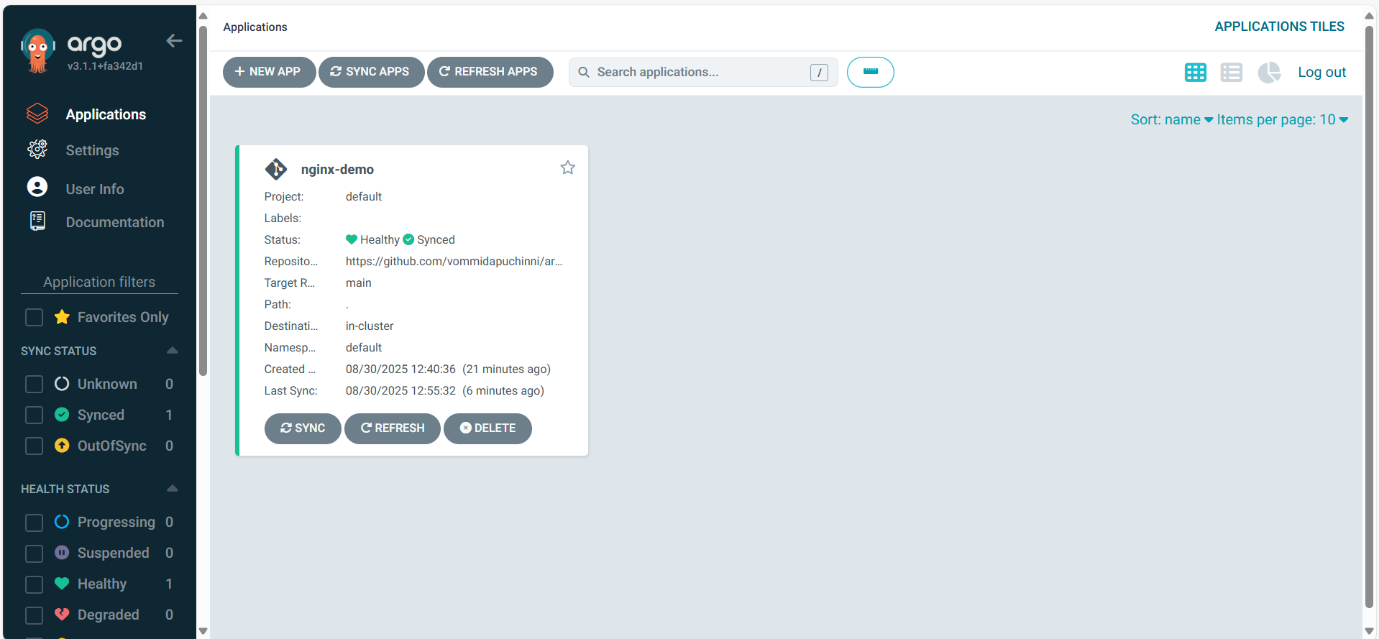
Pod should restart with new image.

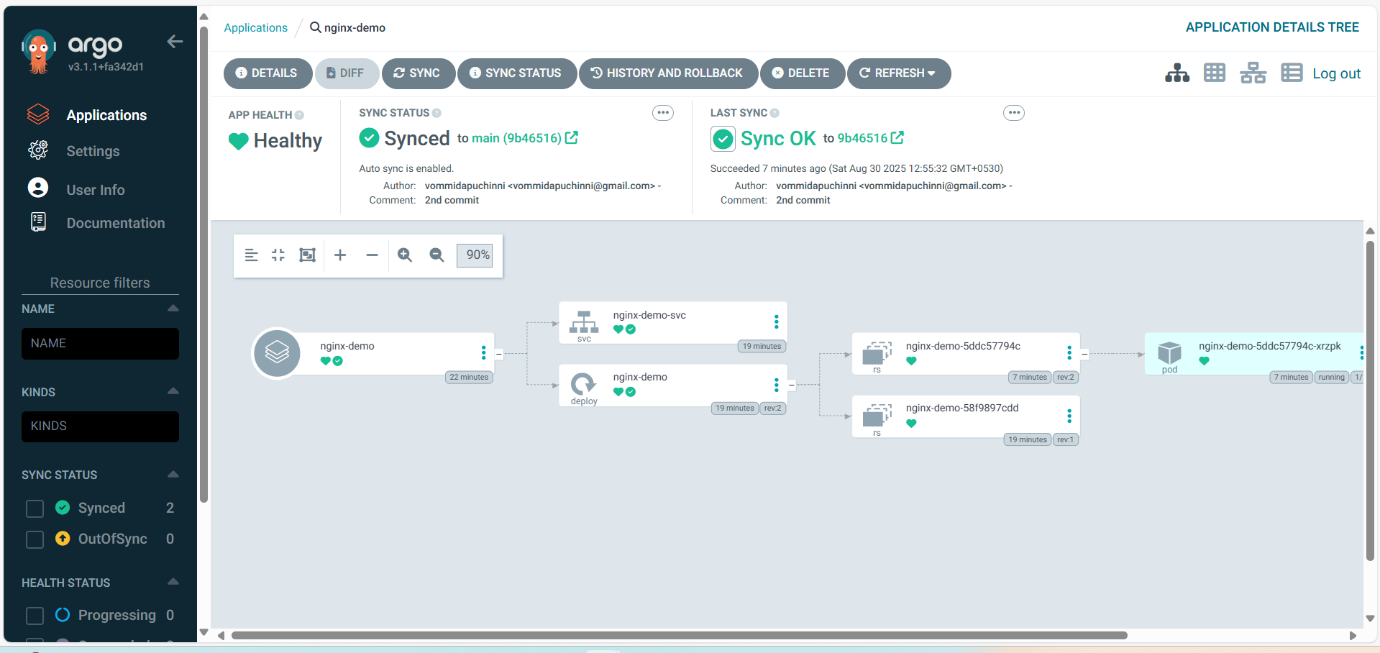
We can see that nginx image we changed and we can see that nginx images changes by kubectl pod describe

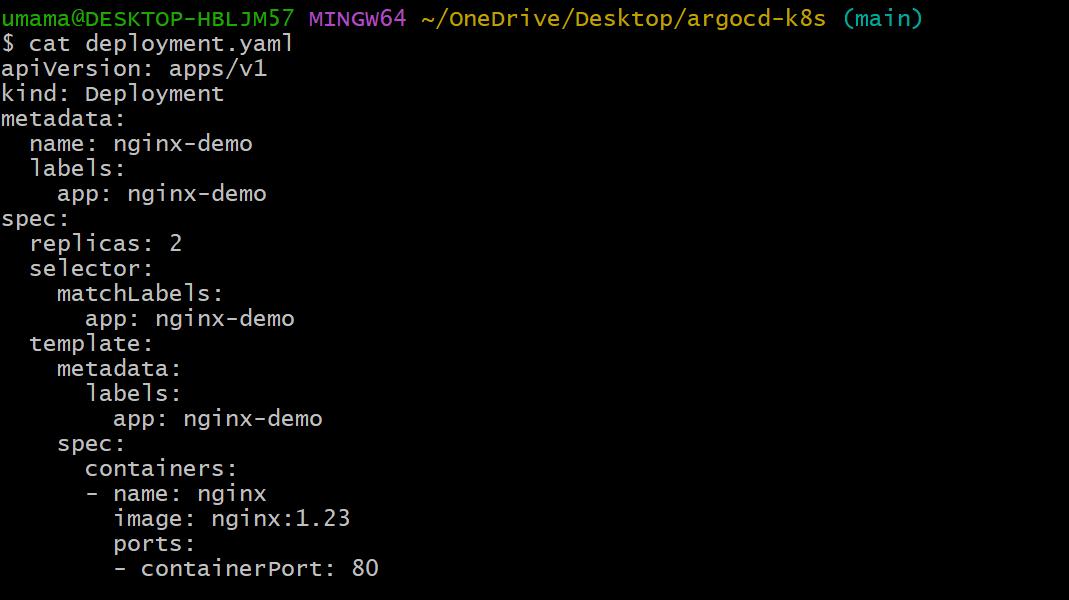
Watch Argo CD UI — with **Auto-Sync** enabled it will detect the change and apply automatically. The app sync history will show a new sync.





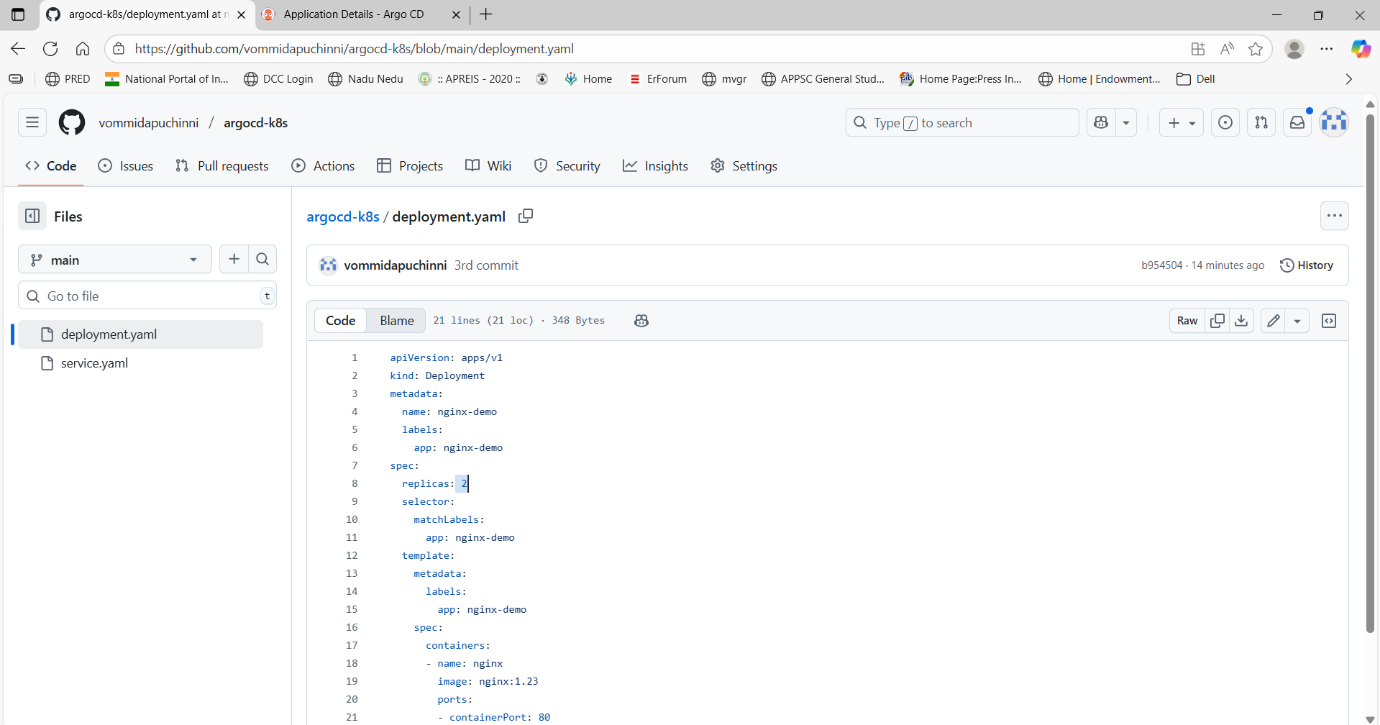


Now I changed the replicas from 1 to 2

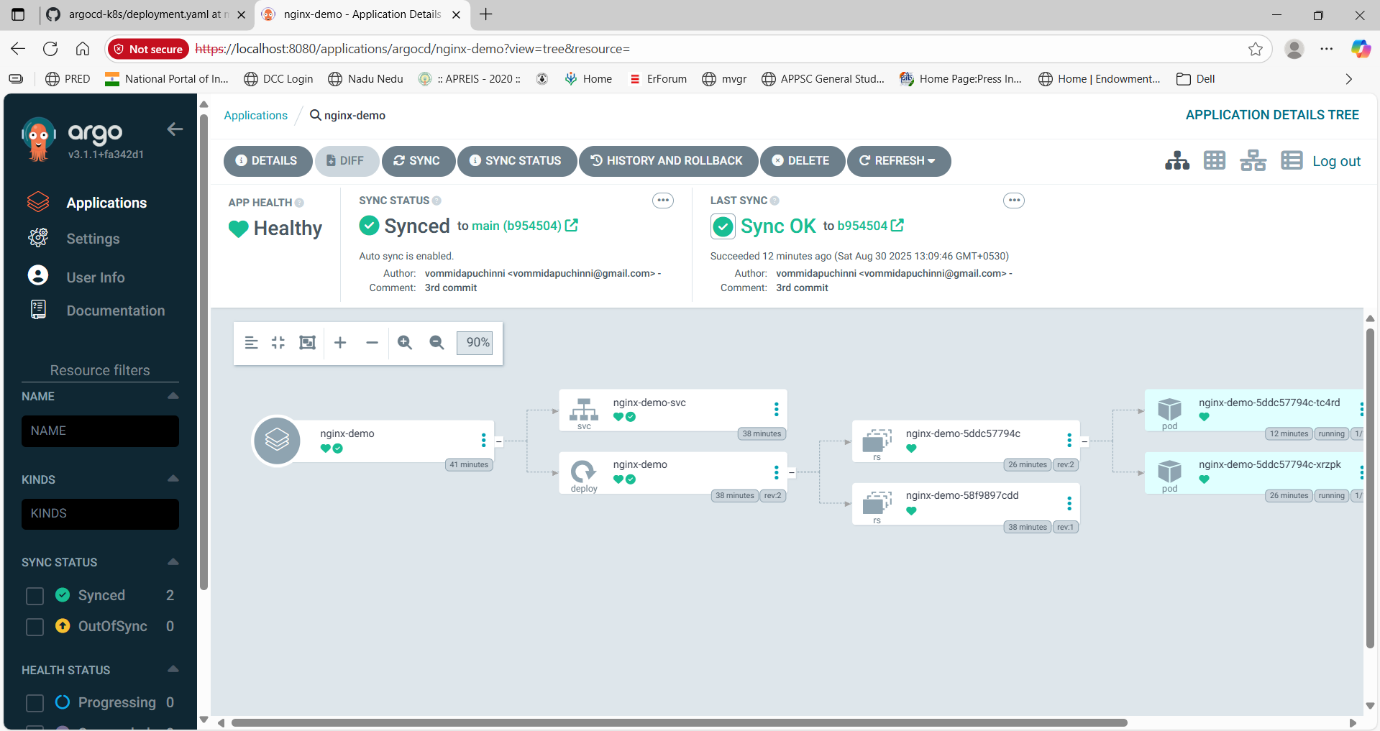


Pushed the edit file to git hub repo

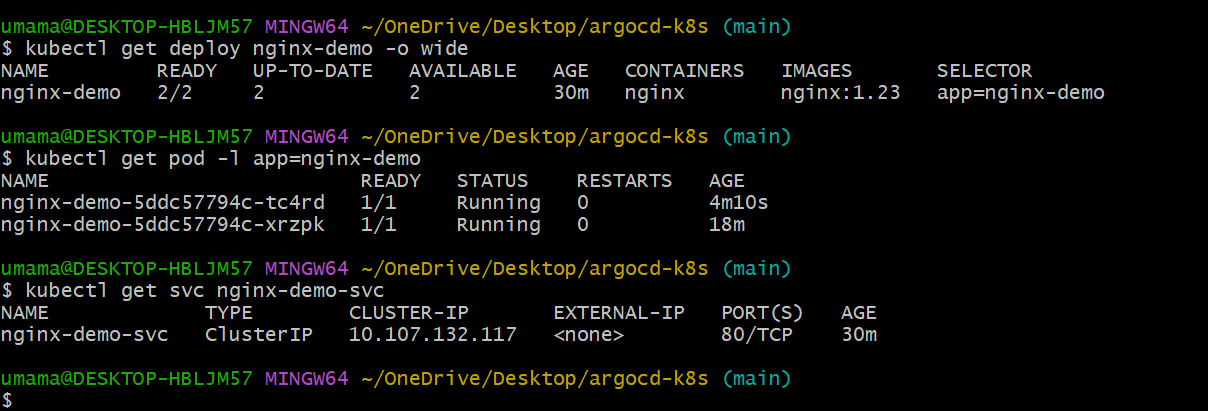




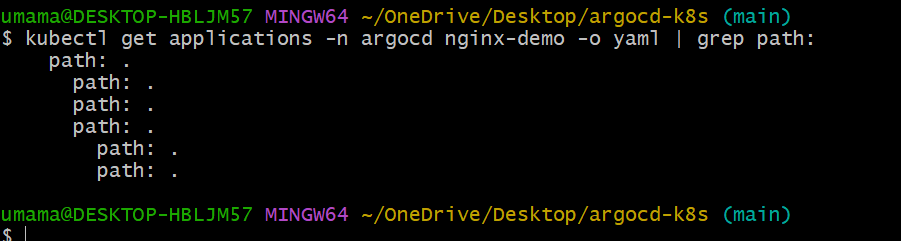
We see auto sync is happened



Local verification of replicas

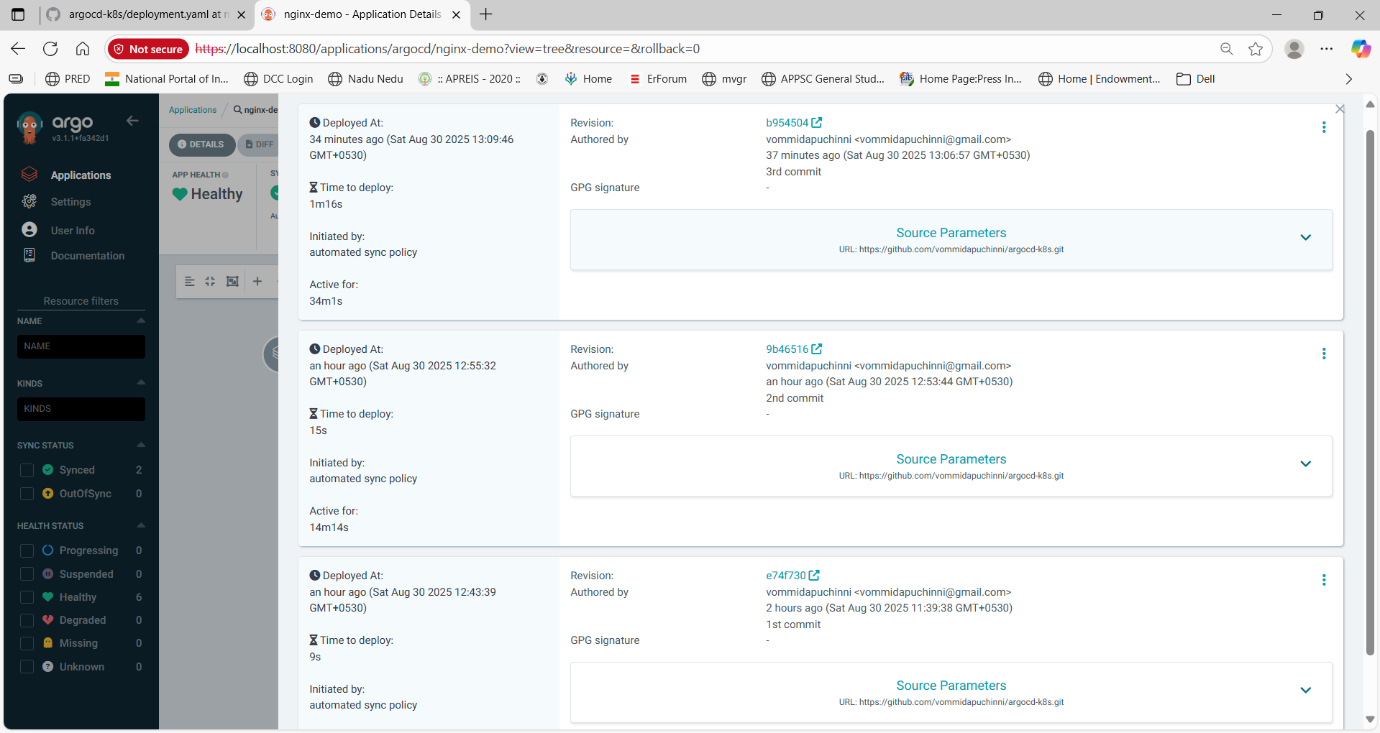


We can see our path local by using this command

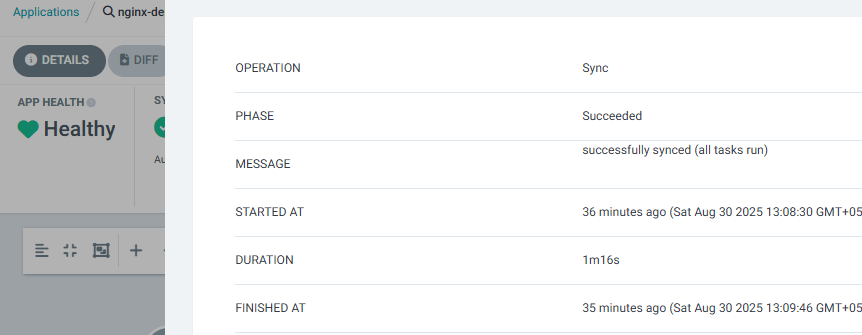


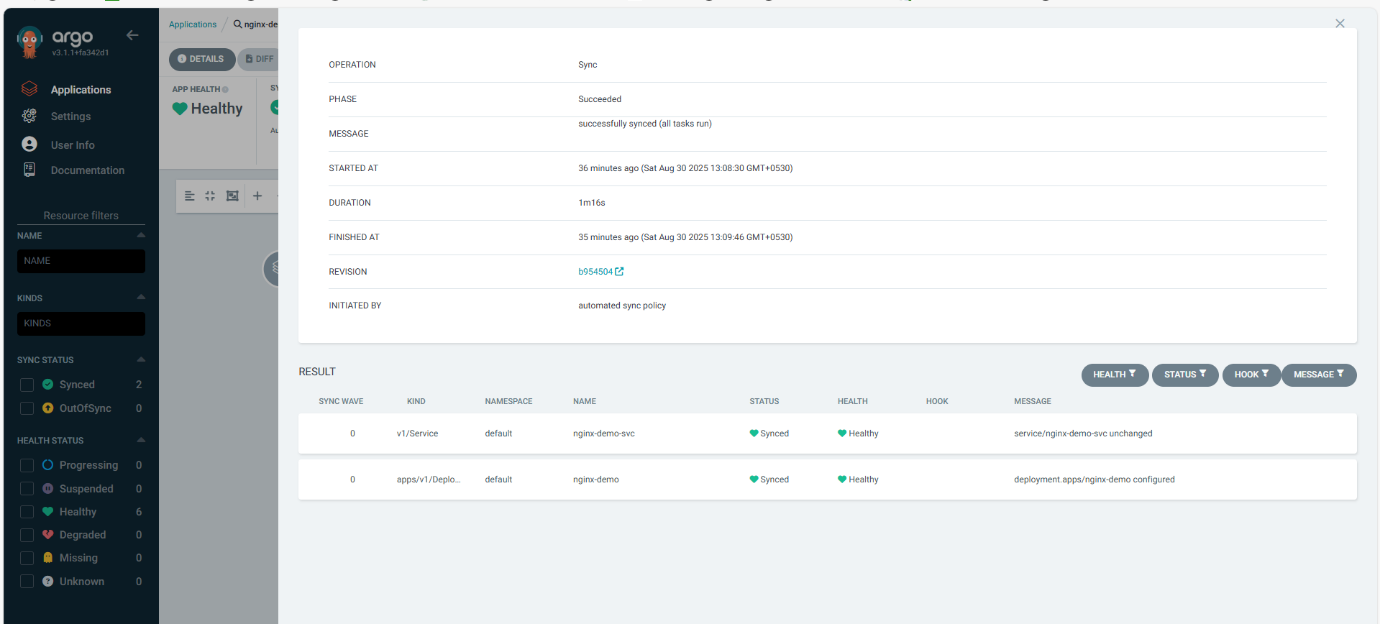
. is given because the files are in root of git repo not in inside other floders

We see history roll back



This is sync status





Our files repo



Optional: create the Argo CD Application declaratively (store Application YAML in Git)

If you want Argo CD itself to be configured from Git (pure GitOps), create an argocd-app.yaml:

apiVersion: argoproj.io/v1alpha1

kind: Application

metadata:

name: nginx-demo

namespace: argocd

spec:

project: default

source:

repoURL: 'https://github.com/<you>/my-argocd-demo.git'

targetRevision: main

path: k8s

destination:

server: https://kubernetes.default.svc

namespace: default

syncPolicy:

automated:

prune: true

selfHeal: true

apply by

kubectl apply -f argocd-app.yaml -n argocd

This is advanced but recommended for infra-as-code

**Troubles Faced**

* ArgoCD UI Access: TLS/self-signed warnings when using port-forward.
* Minikube Issues: Startup failures due to insufficient memory or Docker runtime problems.
* Pod Restarts: Some ArgoCD pods (e.g., repo-server) restarted; needed logs to debug.
* Git Sync Confusion: Changes didn’t reflect until YAMLs were pushed to GitHub and auto-sync enabled.
* Service Access: NodePort optional; ClusterIP and port-forward worked after fixing Minikube and Docker.
* Resolution: Followed GitOps workflow properly—YAMLs in Git → ArgoCD Auto-Sync → changes applied automatically.

**Key Concepts**

* + GitOps: Manage Kubernetes deployments declaratively via Git; Git is the single source of truth.
  + ArgoCD: Continuous delivery tool for Kubernetes that automatically syncs cluster state with Git.
  + Deployment: Defines the desired state of an application (replicas, container image, labels).
  + Service: Exposes a deployment inside the cluster (ClusterIP, NodePort, LoadBalancer).
  + Auto-Sync: ArgoCD feature to automatically apply changes from Git to the cluster.
  + Port-Forwarding: Access cluster services locally for testing UI without exposing NodePorts.
  + ClusterIP vs NodePort: ClusterIP is internal access; NodePort exposes service externally.
  + Pods & ReplicaSets: Pods are running instances of containers; ReplicaSets ensure desired number of replicas.

**Conclusion**

In this project, we implemented a GitOps workflow using ArgoCD and Kubernetes. The deployment manifests for the Nginx application were stored in Git, and ArgoCD automatically synced the desired state to the cluster. This approach ensures version-controlled, automated, and reliable deployments, demonstrating the power of GitOps for managing Kubernetes applications efficiently.

**Infrastructure:**

