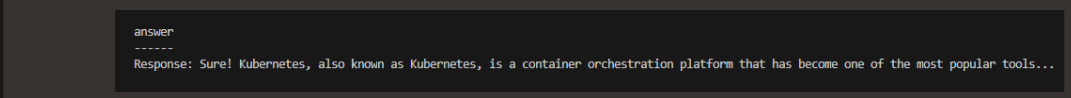
Deploy a RAG API to Kubernetes



Introducing Today's Project!

In this project, I will deploy my containerized RAG API to a local Kubernetes cluster using Minikube. I’m doing this project to learn how to manage, scale, and recover containerized applications beyond a single Docker container. Kubernetes will help me automate deployment, handle failures, scale services, and run my API the same way real production systems do.

Key services and concepts

Key concepts I learnt include Pods as the smallest runnable unit, labels and selectors for discovery, and the reconciliation loop that enforces desired state. Kubernetes provides a control plane that manages scheduling, networking, and self-healing automatically instead of manual container management. Deployments manage desired state by defining replicas and ensuring failed or deleted Pods are recreated. Services route traffic using label selectors and expose stable endpoints through mechanisms like NodePort even as Pods are replaced.

Challenges and wins

This project took me approximately 3 and a half hours. The most challenging part was fixing environment issues on Windows, including Docker Desktop not running, Minikube driver selection problems, and understanding why ErrImageNeverPull happened when the image was not built inside Minikube. I also had to troubleshoot Service reachability, NodePort access, and Ollama connectivity from inside the Pod.

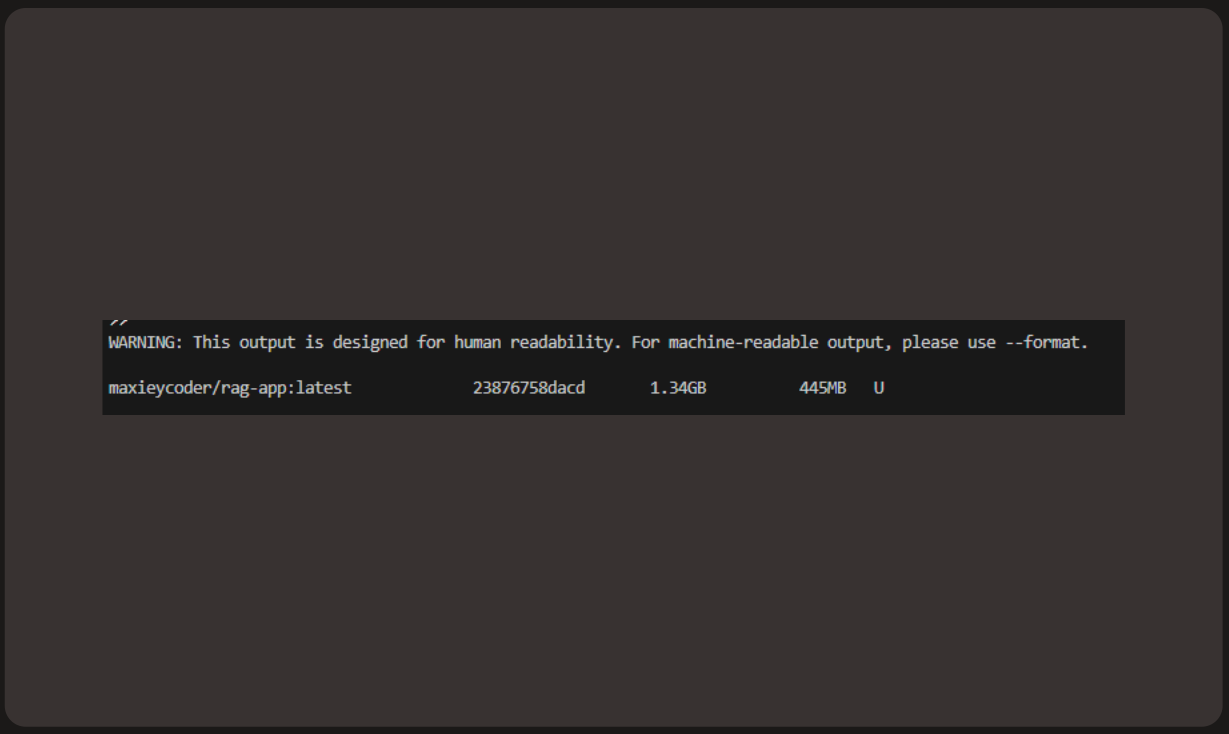
It was most rewarding to finally see the API respond successfully through Kubernetes and to watch Kubernetes automatically recreate a deleted Pod without breaking the Service.

Why I did this project

I did this project because I wanted to learn how Kubernetes actually runs and manages a real application instead of just reading theory. One thing I'll apply from this is using Deployments and Services correctly to get self-healing, stable networking, and separation between infrastructure and application logic in real-world projects.

Setting Up My Docker Image

In this step, I’m setting up my Docker image by packaging my app’s code, dependencies, and configuration into a single portable unit. I need a Docker image because Kubernetes cannot run raw code. It only runs prebuilt images, then handles where they run, how many instances exist, and keeps them alive.



What the Docker image contains

I ran docker images and saw the image maxieycoder/rag-app:latest. The image size was 1.34GB. The IMAGE ID was 23876758dacd.

Installing Kubernetes Tools

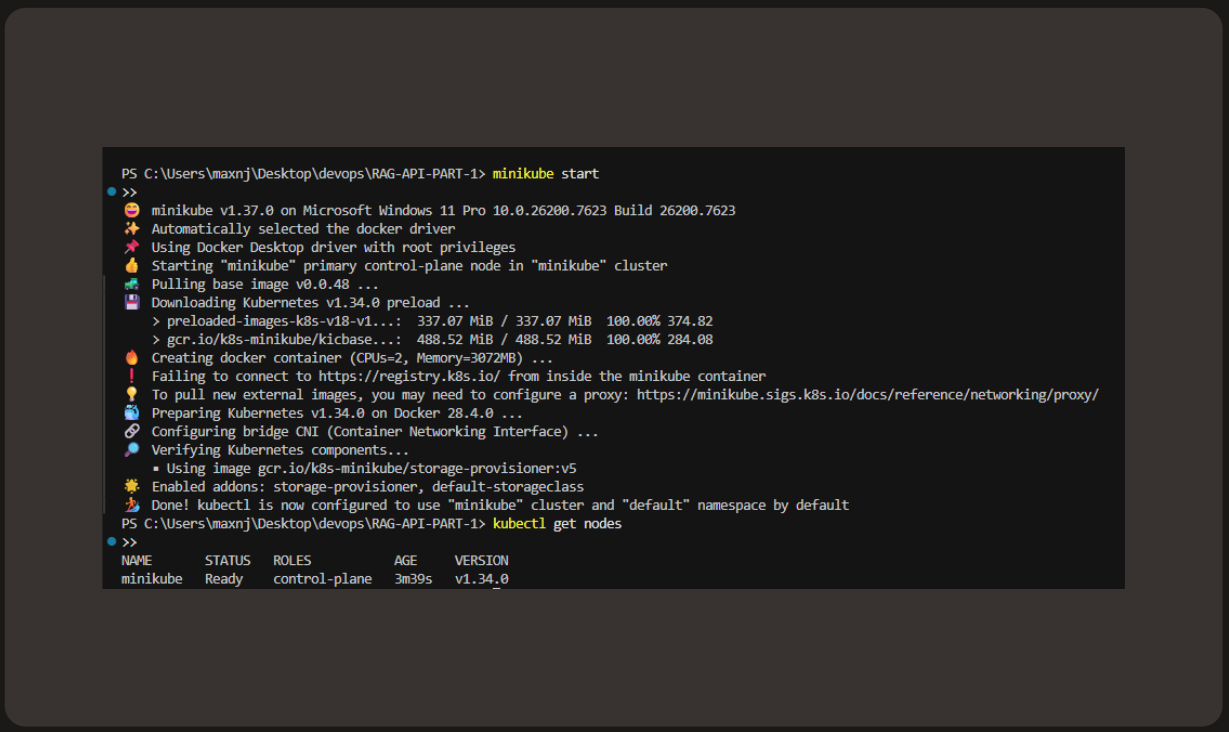
In this step, I'm installing Minikube and kubectl. I need these tools because Minikube runs a local Kubernetes cluster on my machine, and kubectl is how I interact with that cluster to deploy, inspect, and manage my containerized API.

Verifying the tools are installed

I installed Minikube by manually downloading the Windows x64 binary (minikube-windows-amd64.exe), placing it in C:\minikube, renaming it to minikube.exe, and adding C:\minikube to my PATH. I installed kubectl through Minikube, since Minikube bundles and manages kubectl for the cluster, and I verified it using kubectl commands. I could tell both installations were successful because minikube version returned v1.37.0 without errors, and kubectl commands were recognized and able to run, confirming both tools were installed correctly and available in PATH.

Starting My Kubernetes Cluster

In this step, I'm starting a local Kubernetes cluster using Minikube. Minikube will create a real single-node Kubernetes cluster on my machine that can run, manage, and recover my containerized application. I also need to load my Docker image into this cluster so Kubernetes can deploy and orchestrate my RAG API.



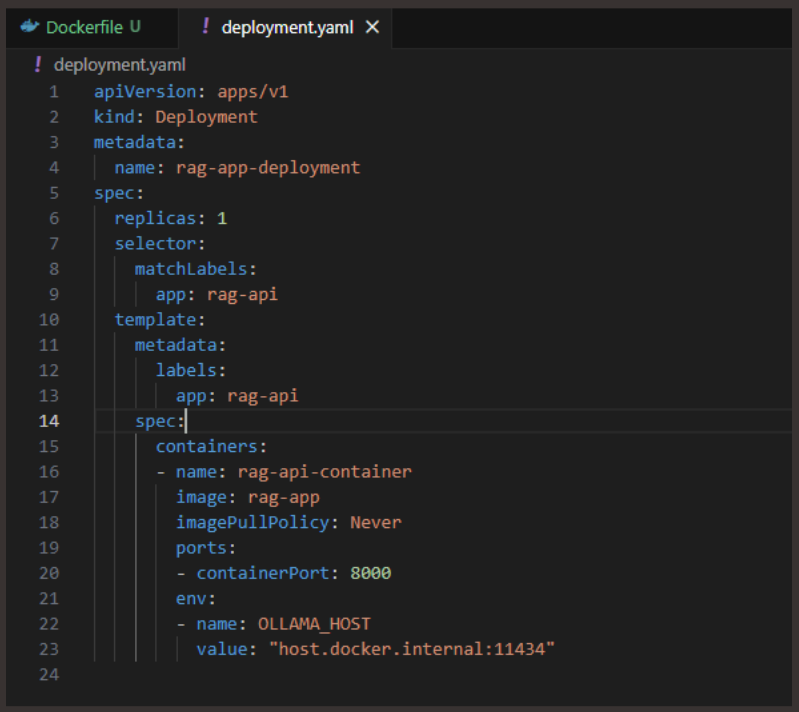
Loading the Docker image into Minikube

I started the cluster by running `minikube start` and saw Minikube create a virtual node, download Kubernetes components, set up networking, and start all core services. Then I ran `kubectl get nodes` to check the cluster. `kubectl get nodes` showed the node named `minikube` with the status `Ready`, the role as `control-plane`, the uptime under `AGE`, and the Kubernetes version under `VERSION`.

This is different from regular Docker because, on your host machine, Docker CLI talks directly to your local Docker daemon and can immediately see images you built but Kubernetes inside Minikube runs in a separate environment and needs the image copied or built there to deploy containers.

Deploying to Kubernetes

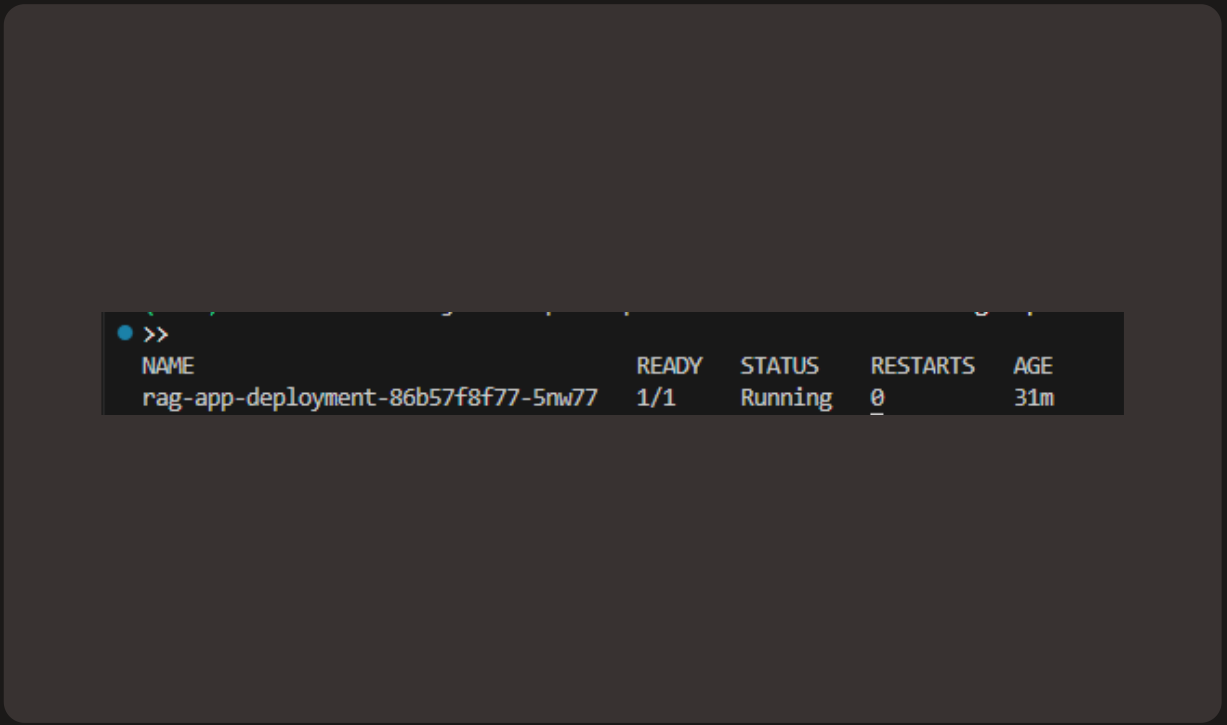
In this step, I'm deploying my `rag-app` container to Kubernetes using a Deployment. I need a Deployment because it acts as a blueprint for Kubernetes, telling it which container image to run, how many copies (replicas) to maintain, and ensuring the application stays running. The Deployment automatically restarts containers if they crash and handles updates without downtime, so my RAG API stays available and stable.



How the Deployment keeps my app running

In this step, I'm creating a `deployment.yaml` file to tell Kubernetes how to run my RAG API.

This file specifies which Docker image to use (`rag-app`), how many copies (replicas) to run, the ports to expose, and the environment variables my app needs. The Deployment ensures that if the Pod crashes, Kubernetes automatically restarts it, and the labels (`app=rag-api`) let Kubernetes manage and connect resources correctly. This file is the blueprint that Kubernetes uses to create and maintain my containerized application.



What did you observe when checking your pods?

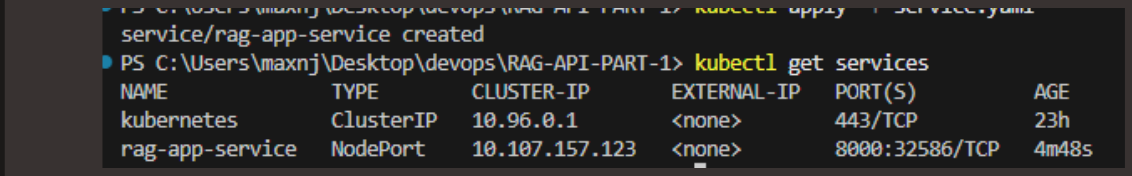
I ran kubectl get pods and saw a Pod created by the Deployment, named something like rag-app-deployment-xxxxxxxx-xxxxx, and its status was Running, which means the container started successfully and is not crashing or waiting on resources. The READY column showed 1/1, which indicates that the single container inside the Pod passed its readiness checks and is fully ready to serve traffic. This confirms the image was found, the container started without errors, and Kubernetes is successfully maintaining the desired state defined in deployment.yaml.

Creating a Service

In this step, I’m creating a Kubernetes Service to expose my RAG API running in a Pod. I need a Service because Pod IP addresses are temporary and change when Pods restart, and a Service provides a stable name and network endpoint that always routes traffic to the correct Pod. By using a NodePort Service, Kubernetes opens a port on the node so I can reliably access my RAG API from my laptop, even if the Pod is recreated or scaled.

What does the service.yaml file do?

The service.yaml file tells Kubernetes how to expose and route traffic to your RAG API in a stable way. The selector finds Pods by matching the label app=rag-api, ensuring traffic is only sent to the Pods created by your Deployment. The port configuration allows the Service to listen on port 8000 and forward requests to port 8000 inside the Pods where the RAG API is actually running. NodePort enables access from outside by opening a high port on the node and routing traffic from that node port to the Service, making the API reachable from your laptop even when Pods restart or change IPs.

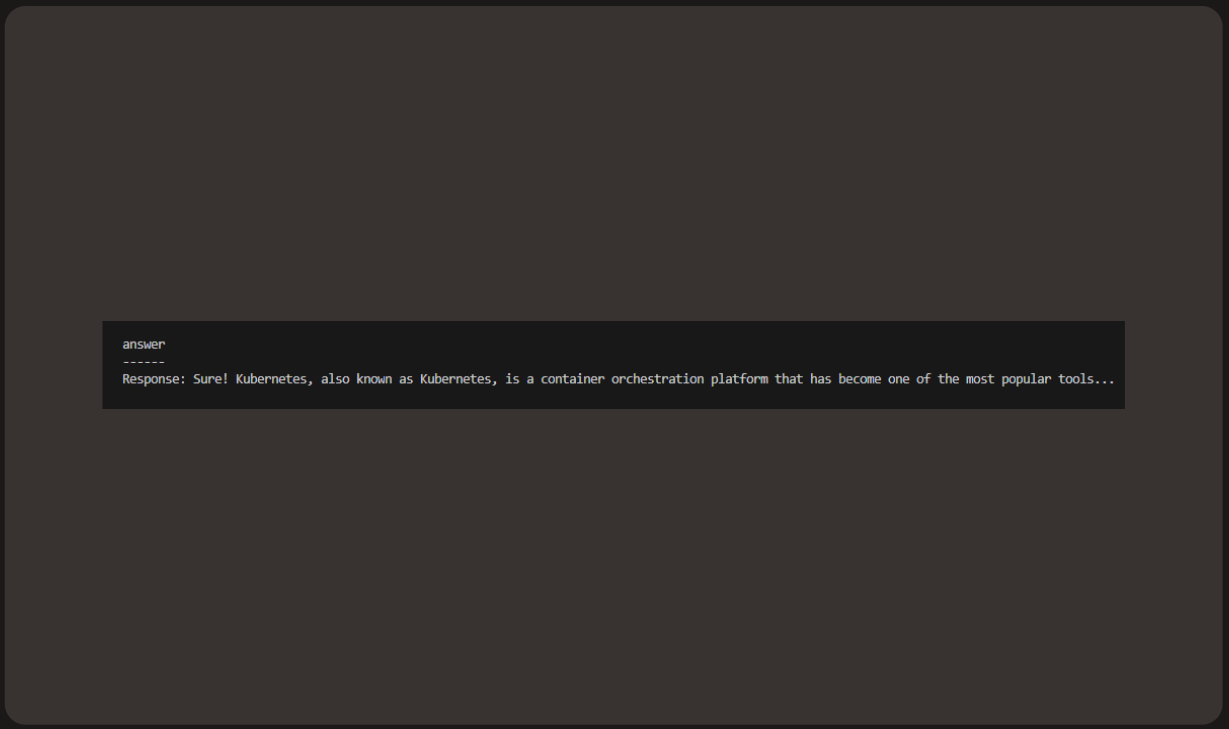


What kubectl commands did you run to create the service?

I applied my Service file by running kubectl apply -f service.yaml. I then verified that the Service was created by running kubectl get services, which showed the rag-app-service with type NodePort and an assigned port.

Accessing My API Through Kubernetes

In this step, I'm testing my RAG API by accessing it through the Kubernetes NodePort to confirm that traffic is correctly routed from my laptop to the Service, then to the Pod, and that the API responds successfully while running inside the Kubernetes cluster.



How I accessed my API

I tested my API by running a POST request using curl or Invoke-RestMethod against the NodePort URL exposed by minikube service rag-app-service --url. The response showed a JSON object containing an AI-generated answer explaining what Kubernetes is, which confirmed that the request reached the API, the knowledge base was queried, Ollama generated a response, and the result was returned correctly. This confirms that the Pod was running, the Service selector matched the Pod, and NodePort networking was functioning as expected.

The main difference between Docker and Kubernetes deployment is that Docker exposes a container directly on a local port, while Kubernetes adds an extra layer where a Service routes traffic to Pods and the cluster manages networking, restarts, and scaling automatically.

Request flow through Kubernetes

The request flow went from my computer to the Minikube node IP on the NodePort, then to the Kubernetes Service on port 8000, then to the Pod with the label app=rag-api, and finally into the container where the RAG API is running. The Service routed traffic by using its selector to match the correct Pod and forward the request internally. NodePort enabled access by opening a port on the Kubernetes node so external traffic from my computer could enter the cluster and reach the Service.

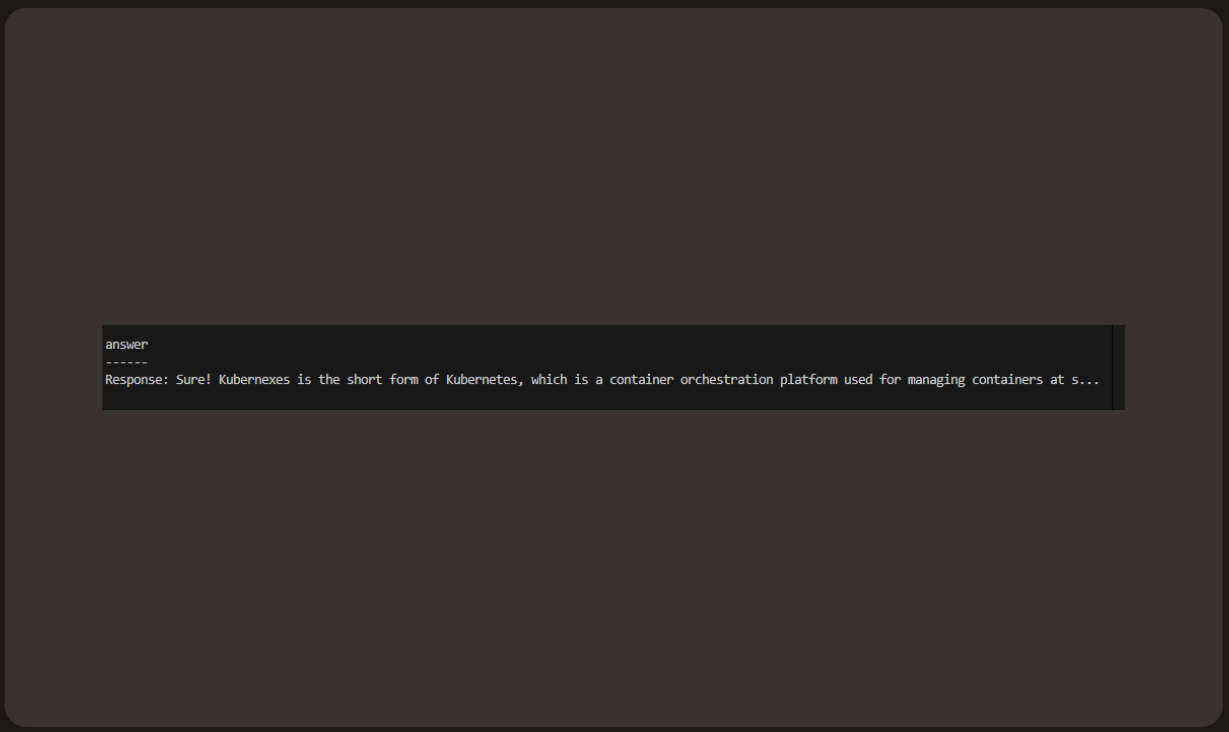
Testing Self-Healing

In this project extension, I'm demonstrating Kubernetes self-healing by manually deleting a running Pod and observing Kubernetes automatically create a new replacement to maintain the Deployment’s desired state. Self-healing is important because it allows applications to recover from crashes or accidental deletions without human intervention, ensuring continuous availability and reliability compared to manually managed containers.

```
>>
NAME                                READY   STATUS              RESTARTS        AGE
rag-app-deployment-86b57f8f77-cmqls   1/1     Running             1 (98m ago)     18h
rag-app-deployment-86b57f8f77-cmqls   1/1     Terminating         1 (106m ago)    18h
rag-app-deployment-86b57f8f77-cmqls   1/1     Terminating         1 (106m ago)    18h
rag-app-deployment-86b57f8f77-tw4wj   0/1     Pending             0               0s
rag-app-deployment-86b57f8f77-tw4wj   0/1     Pending             0               0s
rag-app-deployment-86b57f8f77-tw4wj   0/1     ContainerCreating   0               1s
rag-app-deployment-86b57f8f77-tw4wj   1/1     Running             0               5s
rag-app-deployment-86b57f8f77-cmqls   0/1     Completed           1 (106m ago)    18h
rag-app-deployment-86b57f8f77-cmqls   0/1     Completed           1               18h
rag-app-deployment-86b57f8f77-cmqls   0/1     Completed           1               18h
rag-app-deployment-86b57f8f77-cmqls   0/1     Completed           1               18h
```

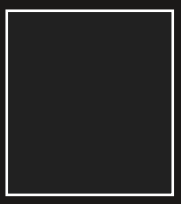
What did you observe when you deleted the pod?

When I deleted the pod, I saw the status change from Running to Terminating, then a new pod appeared in ContainerCreating and finally moved to Running. A new pod was created because the Deployment specifies replicas: 1, and Kubernetes' reconciliation loop automatically created a replacement to match the desired state after the original pod was deleted.



How the Service routed traffic to the new pod

The Service automatically discovered and routed traffic to the new pod because the Deployment created the replacement pod with the same label app=rag-api that the Service selector watches for. Without Kubernetes this would have required manual reconfiguration, updating IP addresses, or restarting services to point traffic to the new container.



Self-healing is critical in production because it ensures applications stay available during crashes, failures, or restarts without human intervention or downtime.