🚀 Kubernetes Integration Options

Your Python application is containerized, which is 90% of the battle. To run it on Kubernetes, you need to define your desired state using **YAML manifest files**.

1. The Core Components (The Minimum You Need)

For your simple Python container, you need at least two Kubernetes objects:

K8s Component	Purpose	Analogy
Pod	The **smallest deployable unit** in K8s. It wraps your Docker container.	The apartment unit in a building.
Deployment	Manages the desired state, ensuring a specified number of Pod replicas (copies) are always running. This enables **scaling** and **self-healing**.	The building manager who ensures all apartments are occupied and replaced if damaged.
Service	Provides a **static, internal IP address** and DNS name for the Pods managed by the Deployment. This is how other services (or the outside world) talk to your application without caring which specific Pod they hit.	The receptionist's phone number, which routes calls to any available employee (Pod).

Deployment Options (Where to Run It)

Option	Description	Use Case
Minikube	A tool to run a **single-node K8s cluster** locally on your machine (often integrated into Docker Desktop).	**Learning** and **local development/testing**. The easiest way to start.
Cloud-Managed K8s	Services like **Amazon EKS**, **Google GKE**, or **Azure AKS**.	**Production** environments, offering high availability, automatic upgrades, and integration with cloud services.
K3s / MicroK8s	Lightweight distributions designed for edge computing or **small-scale deployments**.	**Testing on a Raspberry Pi** or running a small personal server.

What Problems Does Kubernetes Solve?

Kubernetes is a **game changer** primarily because it shifts the focus from managing individual servers to managing **application state**.

Problem Solved	How K8s is a Game Changer
Manual Scaling	**Horizontal Pod Autoscaling (HPA)** automatically increases the number of running BankAccount containers (replicas) when CPU usage spikes and scales them down when traffic subsides.
Downtime from Failure	**Self-Healing** ensures that if the server (Pod) running your bank account class crashes, K8s immediately detects it and spins up a brand-new identical Pod to replace it, ensuring continuous availability.
Slow, Risky Updates	**Rolling Updates** allow you to deploy a new version of your mybank.py container one replica at a time. If the new version fails, K8s automatically performs a **Rollback** to the last known good version, minimizing user impact.
Service Discovery	You don't hard-code IP addresses. The **Service** component gives your application a stable name (like bank-api.default.svc.cluster.local), regardless of where or when its Pods are running.
Inconsistent Environments	By using the Docker image as the source of truth, K8s guarantees that the application you tested on your laptop is deployed with the exact same dependencies and configuration everywhere (Dev, Test, Prod).

This automation and reliability allows developers to focus purely on writing code, knowing that the infrastructure will handle stability, scaling, and recovery.

- **We are using Kubeadm to setup Kubernetes through Docker desktop**

You should already have:

1. Your mybank.py and test_mybank.py files.

2. A Dockerfile in the same directory.
3. **Docker Desktop** installed and running (which includes the Docker CLI).

Step 1: Enable Kubernetes and Select the Provisioner

1. Open the **Docker Desktop Dashboard**.
2. Go to **Settings** (or Preferences on Mac).
3. Click the **Kubernetes** tab.
4. Ensure **Enable Kubernetes** is toggled on.
5. Check the option to select kubeadm as the cluster provisioner (if the option is available and not the default).
6. Click **Apply & Restart** (or just **Apply**).
7. Wait for the Kubernetes status light in the bottom corner of Docker Desktop to turn **green**.

Step 2: Create Kubernetes Manifest Files (YAML)

1. Create a file named bank-deployment.yaml – file is in repository
2. Create a file named bank-service.yaml

🧭 Purpose of the bank-service.yaml File

The core function of the Kubernetes Service defined in this file is to act as a **stable, single point of entry** for accessing the application containers (Pods) managed by your mybank-deployment.

1. Stability and Service Discovery

- **Problem:** Kubernetes Pods are designed to be ephemeral (temporary). If a Pod crashes, gets replaced, or scales up/down, its **IP address changes**. Other applications trying to communicate with your bank application would constantly break.
- **Service Solution:** The Service object provides a **permanent internal IP address and DNS name** (e.g., mybank-service.default.svc.cluster.local) that never changes for the lifetime of the Service.
- **Game Changer:** Any other service within the cluster (like a front-end UI or another microservice) can reliably connect to the bank application using this static name, regardless of which Pod is currently running it.

2. Load Balancing

- The Service automatically acts as a **load balancer**. If your bank-deployment.yaml specifies two replicas (replicas: 2), the Service will distribute incoming network requests evenly across those two running Pods. This is critical for scaling.

3. External Exposure (Using NodePort in your example)

- The bank-service.yaml file defined the Service with type: NodePort.
- A **NodePort Service** exposes your application on a specific port on **every node** in the Kubernetes cluster. This is typically used in local environments like Minikube/Docker Desktop to allow you to access the application from your web browser or local machine.
    - *Note: In a production environment, you'd usually use a LoadBalancer or Ingress object instead of NodePort for external access.*

Step 3: Deployment

Use the Standard Kubectl CLI

Once the cluster is running, , using the standard kubectl command line tool. Docker Desktop automatically sets the configuration context to point to its internal cluster.

1. **Verify the context** (it should be docker-desktop):

Bash

kubectl config use-context docker-desktop

2. **Verify the node** (you should see one node named docker-desktop):

Bash

kubectl get nodes

3. **Deploy your app** using the same YAML files:

Bash

kubectl apply -f bank-deployment.yaml

kubectl apply -f bank-service.yaml

4. **Check logs** and verify everything is running:

Bash

kubectl get pods

kubectl logs <your-pod-name>

Two Kubernetes provisioners: The Difference

Kubeadm vs. Minikube:

The choice between the two fundamentally depends on your goal:

Feature	Minikube	Kubeadm (The Tool)	Docker Desktop's Kubeadm Option
Purpose	Local development and quick learning.	Bootstrapping production-grade, multi-node clusters.	Uses kubeadm logic to set up a single-node cluster for local use.
Control	Highly abstracted and simpler setup.	Highly customizable and requires more manual configuration steps.	Provides a standard, simple local cluster setup using kubeadm's production logic.
Focus	Ease of use and feature addons.	Following community best practices for architecture.	Provides a stable, compliant single-node environment.

Note:

To pull images from docker hub - You need to replace mybank-app:v1 with your full Docker Hub image path (e.g., yourdockerhubusername/mybank-app:v1) and change the imagePullPolicy

You have two main choices when pulling from a registry:

- **Always**: Kubernetes will always attempt to download the image from Docker Hub before starting the container. This is **best for development and production** where you want to ensure the latest version is used.
- **IfNotPresent**: Kubernetes will check its local cache first. If the image is already on the node, it uses the local copy; otherwise, it pulls it from Docker Hub. This is **good for saving bandwidth** once the node has the initial image.

- **Never**: (Your original setting) Tells Kubernetes *not* to pull, which is only used when the image is guaranteed to be available locally (like when using Minikube's internal Docker daemon).

```
  spec:
    containers:
    - name: bank-container
      # 1. CHANGE: Use the full Docker Hub image path
      image: yourdockerhubusername/mybank-app:v1

      # 2. CHANGE: Update the imagePullPolicy
      # Option A (Recommended for production): Always pull the latest image
      imagePullPolicy: Always

      # Option B (Good for saving bandwidth): Pull only if not already cached
      # imagePullPolicy: IfNotPresent

# ... (rest of the file remains the same)
```

Important step

Essential Pre-requisite: Pushing to Docker Hub

Before applying the updated YAML, you must ensure your image is actually available on Docker Hub:

1. **Tag the image** with the full Docker Hub path:

Bash

docker tag mybank-app:v1 yourdockerhubusername/mybank-app:v1

2. **Push the image** to Docker Hub:

Bash

docker push yourdockerhubusername/mybank-app:v1

**** Important commands****

kubectl -f apply <.deployment_file.yaml>

kubectl -f apply <.service_file.yaml>

kubectl get nodes

kubectl get pods

kubectl logs <pod-name>

kubectl cluster-info

kubectl describe pod <pod-name>

kubectl delete pod <pod-name>

kubectl delete -f <.deployment.yaml> #deletes resources defined in yaml file

kubectl scale deployment mybank-deployment --replicas=5

@@ To test-out kubernetes replicas and auto-healing feature

We have two main ways to test the replica feature: **Manual Scaling** (direct control) and **Self-Healing** (simulating failure).

---

1. 📈 Test Manual Scaling

Manual scaling is how you increase or decrease the number of application copies (Pods) on demand using the `kubectl scale` command.

A. Scale Up (Increase Replicas)

You will instruct Kubernetes to increase the number of running bank application Pods from 2 (or whatever you set in `bank-deployment.yaml`) to 5.

1. **Execute the Scale Command:**

   Bash

   ```
   kubectl scale deployment mybank-deployment --replicas=5
   ```

2. **Monitor the Change:** Immediately check the status of your Deployment. You'll see Kubernetes creating 3 new Pods.

   Bash

   ```
   kubectl get deployment mybank-deployment
   # Look for READY. It will go from 2/2 -> 5/5

   kubectl get pods
   # You will see 3 new Pods being created (Status: ContainerCreating ->
   Running)
   ```

B. Scale Down (Decrease Replicas)

You will bring the number of running Pods back down to 1. Kubernetes will gracefully terminate the extra 4 Pods.

1. **Execute the Scale Command:**

   Bash

   ```
   kubectl scale deployment mybank-deployment --replicas=1
   ```

2. **Monitor the Change:** Kubernetes will choose 4 of the running Pods and terminate them.

   Bash

   ```
   kubectl get pods
   # You will see 4 Pods transition to Status: Terminating, leaving only 1 running.
   ```

---

2. 🤕 Test Self-Healing (The Game Changer)

Self-healing is a fundamental feature of Kubernetes Deployments. By having multiple replicas, the platform can tolerate and recover from individual Pod failures.

1. **Verify Replicas:** Ensure you are running at least two replicas (scale up to 2 if you scaled down to 1).

   Bash

   ```
   kubectl scale deployment mybank-deployment --replicas=2
   kubectl get pods # Confirm 2 Pods are Running/Ready
   ```

2. **Simulate a Crash (Delete a Pod):** Directly deleting a Pod simulates an unrecoverable crash of the server running that container.
   - Find one of the running Pod names (e.g., `mybank-deployment-abc12345-xyz789`).

   Bash

   ```
   kubectl delete pod <name-of-one-running-pod>
   ```

3. **Monitor the Recovery:**

   Bash

   ```
   kubectl get pods --watch
   ```

   You will see the old Pod immediately go into `Terminating` status. Then, almost instantly, Kubernetes (specifically the Deployment controller) will create a brand new Pod to replace it, maintaining your desired count of **2 replicas**.

This action demonstrates the **"desired state"** principle:

We have told Kubernetes you want 2 copies, and it will automatically heal the cluster to meet that state, regardless of failures.