**lLEVEL 1:**

# Docker and Kubernetes Workshop for Beginners: Hands-On with a Simple Web App

\*\*Workshop Prerequisites:\*\*

- A computer with Python installed (version 3.8+).

- Install Docker Desktop (free from docker.com—download and run the installer).

- Install Minikube (a simple way to run Kubernetes locally; download from kubernetes.io/docs/tasks/tools/).

- A code editor like VS Code or Notepad.

- Basic terminal/command line access (we'll use simple commands).

\*\*Time Estimate:\*\* 90-120 minutes, with breaks for trying things yourself.

\*\*Key Concepts We'll Cover:\*\*

- \*\*Docker:\*\* Think of it as a way to package your app into a "box" (container) that runs the same everywhere, without worrying about different computers.

- \*\*Kubernetes (K8s):\*\* Like a manager for multiple Docker boxes—it handles starting, stopping, and scaling them.

Let's dive in!

## Section 1: How to Generate the Dummy Project Using LLM

In a real workshop, we'd code everything live together. But to make it fun and interactive, we'll use an AI tool (like Grok, ChatGPT, or any LLM) to generate the basic code for our dummy project. This saves time and shows how AI can help developers.

\*\*Why use an LLM?\*\* It helps create clean, working code quickly, so we can focus on Docker and Kubernetes. You'll input a prompt, and the LLM will output the files—copy them into your editor.

\*\*Exact Prompt to Use:\*\*

Copy-paste this into an LLM (e.g., chat.x.ai or chatgpt.com):

"Create a simple beginner-level Python Flask web application that serves a 'Hello World' message at the root URL ('/') and a simple JSON API at '/api' that returns {'message': 'Welcome to the workshop!'}. Include only the necessary files: app.py (the main app code) and requirements.txt (for dependencies). Add comments in the code to explain each part. Keep it under 50 lines total, and make sure it's runnable with 'python app.py'. Do not include any Docker or Kubernetes code—just the app itself."

\*\*What to Expect from the LLM:\*\*

- It will output two files: `app.py` and `requirements.txt`.

- Save them in a new folder on your computer, e.g., `C:\workshop\hello-app` (Windows) or `~/workshop/hello-app` (Mac/Linux).

- If the output isn't perfect, tweak the prompt (e.g., add "fix any errors").

Now, let's assume you've generated it—here's what the code should look like (for reference; generate it live in class!):

### app.py

```python

# Import Flask: This is the main library for building the web app.

from flask import Flask, jsonify

# Create the app instance: This is like the "heart" of your web server.

app = Flask(\_\_name\_\_)

# Define the root route ('/'): This handles visits to the main page.

@app.route('/')

def hello\_world():

return 'Hello World from our workshop!' # Simple text response.

# Define the API route ('/api'): This returns JSON data.

@app.route('/api')

def api():

return jsonify({'message': 'Welcome to the workshop!'}) # JSON response.

# Run the app: This starts the server when you run the file.

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True, host='0.0.0.0', port=5000) # Debug mode for easy testing; listens on all interfaces.

```

### requirements.txt

```

Flask==3.0.3 # The web framework we need.

```

\*\*Hands-On Tip:\*\* In class, let's all input the prompt now and compare outputs. If yours differs slightly, that's okay—as long as it runs!

## Section 2: Creating the Project Step by Step

Now that we have the code from the LLM, let's set it up and run it locally (without Docker yet). This helps you understand the app before containerizing it.

\*\*Why do this?\*\* It shows the app works on its own, so you can see how Docker "wraps" it without changing the code.

1. \*\*Create a Project Folder:\*\*

- Open your terminal (Command Prompt on Windows, Terminal on Mac/Linux).

- Run: `mkdir hello-app` (creates a folder).

- `cd hello-app` (go into it).

2. \*\*Save the Generated Files:\*\*

- Copy `app.py` and `requirements.txt` into this folder using your editor.

3. \*\*Install Dependencies:\*\*

- Run: `pip install -r requirements.txt`

- Why? This installs Flask so your app can run.

4. \*\*Run the App Locally:\*\*

- Run: `python app.py`

- Why? This starts the web server.

- Open a browser and go to `http://localhost:5000`—you should see "Hello World from our workshop!"

- Try `http://localhost:5000/api`—it shows JSON: {"message": "Welcome to the workshop!"}

- Stop it with Ctrl+C in the terminal.

\*\*Quick Concept Explanation:\*\*

- Your app is like a tiny web server. Without Docker, it only runs on your machine if Python and Flask are installed. Docker will make it portable.

\*\*Hands-On Pause:\*\* Everyone try running it now. If it doesn't work, check for typos in the code!

## Section 3: Introduction to Docker – Containerizing Your App

Docker lets you package your app, code, and dependencies into an "image" (like a blueprint), then run it as a "container" (the live version).

\*\*Simple Diagram (ASCII Art):\*\*

```

+---------------+ Build +---------------+ Run +---------------+

| Your Code | ----------> | Docker Image | ----------> | Container |

| (app.py, etc.)| | (Blueprint) | | (Running App) |

+---------------+ +---------------+ +---------------+

```

- \*\*Why Docker?\*\* It solves "it works on my machine" problems—runs the same on any computer.

### Step 3.1: Write the Dockerfile

Create a file called `Dockerfile` (no extension) in your `hello-app` folder.

\*\*Dockerfile Content:\*\*

```dockerfile

# Start with a base image: This is like a mini-OS with Python pre-installed.

FROM python:3.9-slim

# Set the working directory: Where your app code will live inside the container.

WORKDIR /app

# Copy requirements first: This optimizes builds by caching dependencies.

COPY requirements.txt .

# Install dependencies: Runs 'pip install' inside the container.

RUN pip install -r requirements.txt

# Copy the app code: Adds your app.py to the container.

COPY app.py .

# Expose the port: Tells Docker the app listens on port 5000.

EXPOSE 5000

# Command to run the app: Starts the server when the container runs.

CMD ["python", "app.py"]

```

\*\*Why each line?\*\*

- FROM: Bases it on Python to avoid installing it manually.

- WORKDIR: Keeps things organized.

- COPY and RUN: Ensures dependencies are installed.

- EXPOSE: Not mandatory, but good practice for documentation.

- CMD: The "start" command.

### Step 3.2: Build the Docker Image

1. In your terminal (in `hello-app` folder):

- Run: `docker build -t hello-app:latest .`

- Why? Builds the image named "hello-app" (tag: latest). The "." means use the current folder.

- Output: You'll see layers being built—takes 1-2 minutes first time.

2. Check it: `docker images`—see "hello-app" listed.

### Step 3.3: Run the Container Locally

1. Run: `docker run -p 5000:5000 hello-app:latest`

- Why? Starts the container and maps port 5000 (outside) to 5000 (inside).

- Visit `http://localhost:5000` in your browser—same as before!

2. Stop it: Ctrl+C, or in another terminal: `docker ps` (list running), then `docker stop <container-id>`.

\*\*Hands-On Pause:\*\* Build and run your container now. Fun fact: Try running it on a friend's machine—they just need Docker!

## Section 4: Introduction to Kubernetes – Orchestrating Your Containers

Kubernetes (K8s) manages multiple containers—like a boss telling them when to start, scale, or restart.

\*\*Simple Diagram (ASCII Art):\*\*

```

+---------------+ Deploys +---------------+ Exposes +---------------+

| Docker Image | ----------> | Pod (Container| ----------> | Service |

| | | in K8s) | | (Access Point)|

+---------------+ +---------------+ +---------------+

```

- \*\*Pod:\*\* Smallest unit—a running container.

- \*\*Deployment:\*\* Manages pods (e.g., restarts if one crashes).

- \*\*Service:\*\* Gives a stable way to access pods (like a front door).

- \*\*Why K8s?\*\* For apps that need to scale or be reliable beyond one container.

\*\*Setup Minikube:\*\*

1. Start Minikube: `minikube start` (creates a local K8s cluster—takes 2-5 minutes).

2. Check: `kubectl get nodes` (should show one node ready).

### Step 4.1: Push Image to a Registry (Optional but Recommended)

For K8s to use your image, push it to Docker Hub (free account needed).

1. Login: `docker login` (use your credentials).

2. Tag: `docker tag hello-app:latest yourusername/hello-app:latest`

3. Push: `docker push yourusername/hello-app:latest`

- Why? Minikube can use local images, but this teaches real-world sharing. If skipping, use `minikube image load hello-app:latest`.

### Step 4.2: Write Kubernetes Manifests

Create two YAML files in your folder.

#### deployment.yaml

```yaml

apiVersion: apps/v1 # Version of the K8s API.

kind: Deployment # Type: A Deployment manages pods.

metadata:

name: hello-deployment # Name for this deployment.

spec:

replicas: 1 # Number of pods to run (start with 1).

selector:

matchLabels:

app: hello-app # Labels to identify pods.

template: # Pod template.

metadata:

labels:

app: hello-app # Label for the pod.

spec:

containers: # List of containers in the pod.

- name: hello-container # Container name.

image: yourusername/hello-app:latest # Your Docker image (or local: hello-app:latest).

ports:

- containerPort: 5000 # Port the app uses inside.

```

\*\*Why?\*\* This tells K8s to run 1 copy of your container.

#### service.yaml

```yaml

apiVersion: v1 # API version.

kind: Service # Type: A Service exposes the app.

metadata:

name: hello-service # Name for the service.

spec:

selector:

app: hello-app # Matches the deployment's labels.

ports:

- protocol: TCP # Protocol used.

port: 80 # External port (browser uses this).

targetPort: 5000 # Internal pod port.

type: LoadBalancer # Type: Makes it accessible outside Minikube.

```

\*\*Why?\*\* Without this, you can't access the pod from your browser.

### Step 4.3: Apply Manifests and Access the App

1. Apply Deployment: `kubectl apply -f deployment.yaml`

- Why? Creates the pod.

2. Check: `kubectl get pods`—see it running.

3. Apply Service: `kubectl apply -f service.yaml`

4. Get Access URL: `minikube service hello-service --url`

- This gives a URL like `http://192.168.49.2:XXXX`—open in browser.

- Visit `/` and `/api`—your app is now in K8s!

\*\*Hands-On Pause:\*\* Apply and access it. Scale up: Edit replicas to 2 in deployment.yaml, re-apply, and check `kubectl get pods`.

## Section 5: Troubleshooting Tips and FAQs

\*\*Common Issues:\*\*

- \*\*Docker Build Fails:\*\* Check for typos in Dockerfile. Run `docker build ...` with `--no-cache` to rebuild fresh.

- \*\*Container Won't Start:\*\* Use `docker logs <container-id>` to see errors (e.g., missing dependencies).

- \*\*K8s Pod Crashing:\*\* `kubectl logs <pod-name>`—fix code or image.

- \*\*Minikube Not Starting:\*\* Ensure Virtualization is enabled in BIOS; restart Docker.

\*\*FAQs:\*\*

- Q: What's the difference between Docker and K8s? A: Docker packages; K8s manages many packages.

- Q: Why YAML? A: It's human-readable for configs.

- Q: Is this production-ready? A: No—this is local; real K8s uses clouds like AWS.

## Section 6: Exercises and Challenges

Great job! Now practice:

1. \*\*Basic:\*\* Change the "Hello World" message in app.py, rebuild Docker image, update K8s deployment, and see the change.

2. \*\*Intermediate:\*\* Add a new route in app.py (e.g., '/health' returning {'status': 'ok'}), rebuild, and test in K8s.

3. \*\*Challenge:\*\* Scale to 3 replicas in deployment.yaml. Use `kubectl get pods` to verify, then stress-test by refreshing the browser rapidly—K8s load-balances!

4. \*\*Advanced:\*\* Add environment variables to Dockerfile (e.g., ENV MESSAGE="Custom Hello"), use in app.py, and pass via K8s deployment.

Cleanup: `kubectl delete -f service.yaml -f deployment.yaml`; `minikube stop`; `docker rmi hello-app:latest`.

Thanks for joining! You've just containerized and orchestrated your first app. Practice more, and soon you'll be deploying real projects. Questions? Let's discuss!

**LEVEL 2**

# Level 2: Intermediate Docker and Kubernetes Workshop – Scaling and Deploying Your Web App

Hello class! Welcome back to Level 2 of our Docker and Kubernetes series. In Level 1, we built a simple "Hello World" web app using Python Flask (generated via LLM), containerized it with Docker, and deployed it locally with Minikube. Now, we'll take that same project, enhance it with more features, and dive into intermediate Kubernetes concepts like scalability, autoscaling, and cloud deployment. We'll use free or open-source tools so you can try this at home or in class without costs.

This workshop builds directly on Level 1—keep your `hello-app` folder handy. We'll add endpoints for time and metrics, simulate delays to mimic real loads, and deploy to a lightweight local cluster (K3d) or free cloud platforms (Render.com or Railway.app, which have free tiers as of September 2025). These platforms let you expose your app on the internet quickly, making it feel "real."

\*\*Why Level 2?\*\* To show how Kubernetes shines in handling growth—like scaling from 1 user to 100 without crashing. We'll use analogies: Think of Kubernetes as a restaurant manager scaling waiters (pods) during rush hour.

\*\*Prerequisites:\*\*

- Level 1 completed (or basic Docker/K8s knowledge).

- Python 3.8+, Docker Desktop, kubectl (Kubernetes CLI).

- Free accounts: Docker Hub (or GitHub), Render.com, Railway.app.

- Install K3d: `curl -s https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | bash` (free, open-source).

- Time: 120-150 minutes, with hands-on breaks.

\*\*Key Concepts Recap/Advance:\*\*

- \*\*Deployment:\*\* A blueprint for creating pods (running containers).

- \*\*ReplicaSet:\*\* Ensures the right number of pods are running (part of Deployment).

- \*\*Service:\*\* A stable "front door" for pods.

- \*\*Ingress:\*\* Routes external traffic to Services (like a traffic cop).

- \*\*HPA (Horizontal Pod Autoscaler):\*\* Automatically adds/removes pods based on load (e.g., CPU > 50%).

- \*\*Probes:\*\* Health checks to ensure pods are ready/alive.

Let's enhance our project!

## Section 1: How to Generate the Enhanced Dummy Project Using LLM

We'll use an LLM again to generate the updated code—it's quick and lets us focus on deployment. Input this exact prompt into an LLM (like Grok or ChatGPT):

"Act as a code generator and create an enhanced web application using Python Flask. The app should:

1. Serve a 'Hello, World!' message at the root endpoint (/).

2. Provide another endpoint (/time) that returns the current server time in JSON format.

3. Include an endpoint (/metrics) that exposes simple metrics like request count.

4. Simulate latency by adding optional random delays to requests.

5. Include proper logging and comments explaining each part.

6. Be structured in a way that is easy for beginners to follow.

7. Include a requirements.txt or package.json with dependencies.

8. Provide instructions on how to run it locally, containerize it with Docker, and push the image to Docker Hub or another free container registry.

Format the folder structure at the top and include comments in the code."

\*\*Expected Output:\*\* The LLM will give you `app.py` and `requirements.txt`. Here's a sample (generate it live in class!):

\*\*Folder Structure:\*\*

```

hello-app/

├── app.py

├── requirements.txt

```

### app.py

```python

# Import necessary modules: Flask for web app, datetime for time, random for latency simulation, logging for logs.

from flask import Flask, jsonify

import datetime

import random

import time

import logging

# Set up logging: This helps debug by printing info to console.

logging.basicConfig(level=logging.INFO)

# Create the app instance.

app = Flask(\_\_name\_\_)

# Global counter for requests: Simple metric to track total requests.

request\_count = 0

# Function to simulate latency: Adds a random delay (0-2 seconds) to mimic real-world slowdowns.

def simulate\_latency():

delay = random.uniform(0, 2) # Random delay in seconds.

time.sleep(delay)

logging.info(f"Simulated latency: {delay} seconds")

# Root endpoint: Serves a simple message.

@app.route('/')

def hello\_world():

global request\_count

request\_count += 1

simulate\_latency() # Add delay for realism.

logging.info("Handled / request")

return 'Hello World from our enhanced workshop!'

# /time endpoint: Returns current server time in JSON.

@app.route('/time')

def get\_time():

global request\_count

request\_count += 1

simulate\_latency()

logging.info("Handled /time request")

current\_time = datetime.datetime.now().isoformat()

return jsonify({'time': current\_time})

# /metrics endpoint: Exposes basic stats like request count.

@app.route('/metrics')

def metrics():

global request\_count

logging.info("Handled /metrics request")

return jsonify({'request\_count': request\_count})

# Run the app: Starts the server.

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True, host='0.0.0.0', port=5000)

```

### requirements.txt

```

Flask==3.0.3

```

\*\*Local Run Instructions (from LLM):\*\*

- `pip install -r requirements.txt`

- `python app.py`

- Visit `http://localhost:5000/`, `/time`, `/metrics`.

\*\*Hands-On:\*\* Generate and run it now. Hit `/metrics` after a few requests—see the count grow!

## Section 2: Containerizing the Updated Project

\*\*Why?\*\* Docker packages everything (code, deps, latency sim) into a portable image.

1. Update/Create `Dockerfile` in `hello-app`:

```dockerfile

# Base image with Python.

FROM python:3.9-slim

# Working dir.

WORKDIR /app

# Copy and install deps.

COPY requirements.txt .

RUN pip install -r requirements.txt

# Copy code.

COPY app.py .

# Expose port.

EXPOSE 5000

# Run command.

CMD ["python", "app.py"]

```

2. Build: `docker build -t enhanced-app:latest .`

3. Run Locally: `docker run -p 5000:5000 enhanced-app:latest`

- Test endpoints.

4. Push to Free Registry (e.g., Docker Hub):

- Create free Docker Hub account.

- `docker login`

- `docker tag enhanced-app:latest yourusername/enhanced-app:latest`

- `docker push yourusername/enhanced-app:latest`

- Alternatives: GitHub Container Registry (ghcr.io—free with GitHub account) or Quay.io (free open-source).

\*\*Why Push?\*\* For Kubernetes to pull the image from anywhere.

## Section 3: Setting Up a Kubernetes Environment and Introducing Concepts

We'll use K3d (free, lightweight local K8s) for local testing, then Render.com (free tier for web services) for cloud exposure.

\*\*Install K3d:\*\* `k3d cluster create mycluster --agents 2` (creates a cluster with 2 worker nodes).

- Why K3d? It's faster than Minikube, free, and simulates multi-node scaling.

\*\*Kubernetes Concepts with Analogies:\*\*

- \*\*Deployment & ReplicaSet:\*\* Deployment is a recipe; ReplicaSet ensures X copies (replicas) of pods follow it. Why? If one "waiter" (pod) quits, it hires another.

- \*\*HPA:\*\* Like auto-hiring more waiters when the restaurant gets busy (high CPU).

- \*\*Service:\*\* A VIP entrance that routes customers to any available waiter.

- \*\*Ingress:\*\* The restaurant's signboard, directing street traffic inside.

- \*\*Probes:\*\* Health checks—liveness (is the pod alive?) restarts if sick; readiness (is it ready to serve?) waits before sending traffic.

\*\*Diagram: Traffic Flow (ASCII):\*\*

```

Internet/User --> LoadBalancer/Ingress --> Service --> Pods (Replicas)

(Routes external) (Balances) (Running Apps)

```

\*\*Autoscaling Diagram:\*\*

```

Low Load: 1 Pod

High Load (CPU > 50%): HPA --> Add Pods (up to max)

```

### Step 3.1: Create Enhanced Manifests

Update from Level 1.

#### deployment.yaml

```yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: enhanced-deployment

spec:

replicas: 2 # Start with 2 pods for redundancy.

selector:

matchLabels:

app: enhanced-app

template:

metadata:

labels:

app: enhanced-app

spec:

containers:

- name: enhanced-container

image: yourusername/enhanced-app:latest # Your pushed image.

ports:

- containerPort: 5000

livenessProbe: # Checks if alive.

httpGet:

path: / # Hit root to check.

port: 5000

initialDelaySeconds: 5

periodSeconds: 10

readinessProbe: # Checks if ready.

httpGet:

path: /metrics # Use metrics as it's quick.

port: 5000

initialDelaySeconds: 5

periodSeconds: 5

resources: # For HPA.

requests:

cpu: "100m" # Min CPU.

limits:

cpu: "500m" # Max.

```

\*\*Why Probes?\*\* Prevents sending traffic to unhealthy pods.

#### service.yaml

```yaml

apiVersion: v1

kind: Service

metadata:

name: enhanced-service

spec:

selector:

app: enhanced-app

ports:

- protocol: TCP

port: 80

targetPort: 5000

type: LoadBalancer # Exposes externally.

```

#### hpa.yaml (for Autoscaling)

```yaml

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: enhanced-hpa

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: enhanced-deployment

minReplicas: 1

maxReplicas: 5 # Scale up to 5.

metrics:

- type: Resource

resource:

name: cpu

target:

type: Utilization

averageUtilization: 50 # Scale if CPU > 50%.

```

### Step 3.2: Apply and Test Locally with K3d

1. `kubectl apply -f deployment.yaml -f service.yaml -f hpa.yaml`

2. Get URL: `kubectl get svc enhanced-service` (use external IP/port).

3. Test: Browser to the URL—hit endpoints, see load balancing (refresh to hit different pods).

\*\*For Ingress (Optional):\*\* Install Ingress controller in K3d: `kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/kind/deploy.yaml` (then add ingress.yaml for domain routing).

## Section 4: Cloud Deployment with Free Services

\*\*Why Cloud?\*\* To see real internet access and scaling without local limits.

### Option 1: Render.com (Free Tier: 512MB RAM services, always-on)

1. Sign up (free).

2. New → Web Service → Docker → Paste your Docker Hub image URL.

3. Set env vars (if needed, e.g., for secrets).

4. Deploy—Render handles building/pulling.

5. Access: Gets a free subdomain (e.g., yourapp.onrender.com).

6. For K8s-like: Render is PaaS, but add autoscaling in settings (free tier limited).

\*\*Configure LoadBalancer:\*\* Render auto-provides; no manual Ingress needed.

### Option 2: Railway.app (Free: $1 credit/month for small apps)

1. Sign up.

2. New Project → Docker Image → Your image.

3. Deploy.

4. Access: Free subdomain.

5. Scaling: Set replicas in UI; limited autoscaling on free.

\*\*Hands-On:\*\* Deploy to one now—share your live URL in class!

## Section 5: Simulate Real-World Usage

\*\*Why?\*\* To see Kubernetes react to "traffic jams."

1. \*\*Install Load Tool:\*\* `go install github.com/rakyll/hey@latest` (free; or use `ab -n 1000 -c 10 http://your-url/`).

2. \*\*Generate Load:\*\* `hey -n 1000 -c 50 http://your-url/time` (1000 requests, 50 concurrent).

3. \*\*Observe Scaling:\*\* `kubectl get hpa --watch`—watch replicas increase.

4. \*\*Troubleshoot:\*\* If not scaling, check metrics: `kubectl top pods` (needs metrics-server: `kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml`).

\*\*Analogy:\*\* Like cars on a highway—HPA adds lanes (pods) when jammed.

## Section 6: Best Practices and Security

- \*\*Env Vars/ConfigMaps/Secrets:\*\* Use ConfigMap for non-sensitive (e.g., app config): Create configmap.yaml with data, mount in deployment. Secrets for API keys: `kubectl create secret generic my-secret --from-literal=key=value`.

- \*\*Why?\*\* Keeps configs separate from code; secure (secrets encrypted).

- Production Tip: Use resource limits to prevent one pod hogging CPU.

## Section 7: Diagrams and Explanations

\*\*Traffic Flow:\*\*

```

User --> Internet --> Cloud LoadBalancer --> Ingress --> Service --> Pod1, Pod2... (via ReplicaSet)

```

\*\*Autoscaling:\*\*

- Monitor: CPU high? HPA signals Deployment to create more ReplicaSets/pods.

- Distributes: Service round-robins requests.

## Section 8: Exercises and Challenges

1. \*\*Basic:\*\* Deploy to Fly.io (usage-based free for tiny apps) instead—compare ease.

2. \*\*Intermediate:\*\* Simulate spikes: Run load tests with varying concurrency; graph CPU with `kubectl top`.

3. \*\*Challenge:\*\* Add Prometheus (free open-source): Install via Helm (`helm install prom prometheus-community/prometheus`), monitor metrics.

4. \*\*Advanced:\*\* Add a database (free Postgres on Render), connect via env vars/Secrets.

## Section 9: Troubleshooting and FAQs

- \*\*Pods Not Scaling:\*\* Ensure metrics-server installed; check `kubectl describe hpa` for errors.

- \*\*Logs/Metrics:\*\* `kubectl logs <pod-name>`; `kubectl top pods`.

- \*\*Networking Issues:\*\* On free services, check firewall/ports; for local, `kubectl port-forward svc/enhanced-service 5000:80`.

- Q: Why no free forever on Fly.io? A: Usage-based; monitor credits.

- Q: App not accessible? A: Verify Service type (LoadBalancer) and external IP.

Cleanup: `k3d cluster delete mycluster`; delete cloud deployments.

Awesome work! You've scaled a real app. Share your live links—next level: CI/CD. Questions?