docker compose down

docker system prune -a --volumes -f 🡪 clear all images

docker-compose build --no-cache 🡪 build everything from the beginning

docker-compose up -d 🡪

 MLflow → <http://localhost:5000>

 FastAPI → <http://localhost:8000/docs>

 Flower → <http://localhost:5555>

 pgAdmin → <http://localhost:5050>

Docker

docker-compose down -v

docker-compose build

docker-compose up -d

rabbitmq

docker stop rabbitmq

docker rm rabbit mq 🡪 remove rabbimq

docker run -d --hostname rabbit --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management

Celery

celery -A tasks worker --loglevel=info --pool=eventlet -E

mlflow 🡪 <http://127.0.0.1:5000>

Swagger UI

uvicorn main:app --reload --port 8000

🡪 <http://127.0.0.1:8000/docs>

logistic\_regression

celery logs 🡪 docker logs -f celery\_worker

flower 🡪 <http://127.0.0.1:5555>

rabbitmq 🡪 <http://127.0.0.1:15672>

postgresql 🡪 <http://localhost:5050>

* 

Name: MLflow DB (anything you like)

 Under the **Connection** tab:

* Host name/address: postgres
* Port: 5432
* Username: mlflow
* Password: mlflow

docker exec -it postgres psql -U mlflow -d mlflow

\dt

SELECT experiment\_id, name FROM experiments;

SELECT run\_uuid, experiment\_id, status, start\_time, end\_time

FROM runs

ORDER BY start\_time DESC

LIMIT 5;END

* **FastAPI** serves as the main backend API.  
  When a user calls the /train\_model endpoint, FastAPI doesn’t train the model directly — it sends the job to **Celery**, a distributed task queue, so the API remains responsive.
* **RabbitMQ** acts as the message broker between FastAPI and the Celery worker.  
  It handles the communication of background training tasks.
* The **Celery worker** container picks up those tasks and actually trains the model.  
  During training, it logs all parameters, metrics, and artifacts to **MLflow**.
* **MLflow** runs in its own container with a **PostgreSQL** backend that stores experiment metadata and models.  
  The artifacts themselves are stored inside a shared Docker volume, so everything is persistent even if the containers restart.
* I also included **Flower**, a lightweight dashboard to monitor Celery task execution in real time, and **pgAdmin**, a web UI to inspect the PostgreSQL database.

When the training completes, the results automatically appear in the MLflow UI, where we can compare experiments, check metrics, and register models for deployment.

This setup is fully containerized, scalable, and follows production best practices — it separates responsibilities between services and makes it easy to reproduce, extend, or deploy on any environment.”

git init

git add README.md

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/zioviris/Demo\_Project.git

git push -u origin main

**If he asks: “What are the high-level steps to apply an ML model in Azure?”**

You answer something like this 👇

“At a high level, the lifecycle of applying a machine learning model in Azure involves five main steps:

1️⃣ **Data Preparation:**  
We store and manage data in Azure Blob Storage or Data Lake. Then, we preprocess and clean it using Azure Databricks or Azure ML pipelines.

2️⃣ **Model Training:**  
We use Azure Machine Learning Studio or the Azure ML SDK to run experiments — either locally or on cloud compute clusters.  
Training scripts are version-controlled, and the trained model is registered in the **Azure ML Model Registry**.

3️⃣ **Model Evaluation & Optimization:**  
We track metrics (accuracy, ROC-AUC, PSI, etc.) and can automate hyperparameter tuning with Azure ML’s *HyperDrive*.

4️⃣ **Model Deployment:**  
Once validated, the model is deployed as a **real-time endpoint** using an Azure Container Instance (ACI) for testing, or **Azure Kubernetes Service (AKS)** for production scalability.

5️⃣ **Monitoring & Maintenance:**  
Logs and metrics are captured via **Application Insights**, and we monitor data drift or model performance through Azure ML’s monitoring tools.

This workflow ensures traceability, scalability, and compliance — key for fintech environments like Snappi.”

That’s clean, complete, and senior-level.

**🧠 Possible CTO Questions & How to Answer Them**

**1️⃣ “How would you deploy your Snappi model on Azure?”**

“I’d containerize it with Docker, push the image to **Azure Container Registry (ACR)**, and deploy it to **AKS** using Kubernetes manifests or Helm.  
For simpler cases, I could deploy directly from Azure ML Studio as an endpoint — that automatically creates the scoring container.”

**2️⃣ “How would you handle model versioning?”**

“Azure ML Model Registry automatically tracks model versions, metadata, and lineage.  
Each new training run produces a new version that can be deployed or rolled back safely.”

**3️⃣ “How do you ensure scalability and resilience in production?”**

“By deploying to **AKS** — it allows autoscaling, rolling updates, and load balancing.  
Combined with Azure Application Gateway and health probes, it ensures zero-downtime model serving.”

**4️⃣ “How do you monitor your ML models after deployment?”**

“Through **Application Insights** for logs and metrics, and **Azure ML Data Drift Monitor** for detecting performance degradation.  
I’d also integrate alerts to trigger retraining if drift or low accuracy is detected.”

**5️⃣ “What tools would you use for feature engineering and training?”**

“For scalable feature engineering: **Azure Databricks** or **Azure Synapse**.  
For training: **Azure ML Pipelines**, which orchestrate steps like data prep, training, evaluation, and registration.”

**6️⃣ “How would you manage secrets or credentials?”**

“I’d use **Azure Key Vault** to store sensitive info like database passwords or API keys, and access them securely through managed identities.”

**7️⃣ “What’s the difference between deploying on ACI vs AKS?”**

“ACI (Azure Container Instances) is best for development and quick testing — simple but not scalable.  
AKS (Azure Kubernetes Service) is for production — scalable, supports auto-healing, and integrates with CI/CD pipelines.”

**8️⃣ “How do you ensure compliance and security?”**

“By isolating resources in private VNets, using managed identities, and integrating **role-based access control (RBAC)**.  
All model artifacts and data remain encrypted at rest (via Azure Storage encryption) and in transit (TLS).”

**9️⃣ “If Snappi wanted to automate retraining, how would you set that up?”**

“I’d define an Azure ML Pipeline that’s triggered by an Event Grid or Azure Function whenever new data arrives in Blob Storage.  
That pipeline retrains, evaluates, and if performance is better, promotes the new model version to production.”

**🔟 “Why Azure instead of AWS or GCP?”**

“Azure has tight integration between ML, data, and DevOps tools — especially valuable in financial environments that already use Microsoft infrastructure.  
Also, Azure ML’s managed endpoints and strong governance make it easier to build auditable, production-ready ML systems.”

**💬 How to Close Strong**

At the end, you can add something like:

“I’m especially interested in how Snappi uses Azure for deploying ML in a regulated fintech setup.  
I’ve already containerized similar models, and I’m comfortable setting up pipelines and monitoring in Azure ML and AKS.  
I’d love to contribute to optimizing that workflow.”