Nguyen Cong Phat
I'm a student majoring in Computer Science at UIT – VNUHCM with a data-oriented mindset and a strong enthusiasm for learning

Journey to Master AI/ML with Docker

A Practical Handbook for Docker with 3 Hands-On Projects

Vietnamese Full Name: Nguyen Cong Phat

My portfolio website: https://my-portfolio-website-gamma-lemon.vercel.app/

My Github profile: https://github.com/paht2005

My LinkedIn: https://www.linkedin.com/in/ncphat25/

Image 1.

Table of Contents

A. 🕮 Introduction: Why Docker for AI/ML?

1. What is Docker and Why It Matters in AI/ML

- **Docker** is a lightweight containerization platform that enables you to package code, dependencies, and system configurations into a single, portable unit — a container. In AI/ML, where experiments rely heavily on specific Python versions, library dependencies, and GPU support, Docker provides a reproducible and isolated environment for development and deployment.
- In contrast to traditional "it works on my machine" scenarios, Docker ensures consistent behavior across laptops, development servers, CI/CD pipelines, and cloud environments. This is especially critical in machine learning, where reproducibility is essential for debugging, collaboration, and compliance.
- Docker simplifies the ML lifecycle by enabling:
    - Faster environment setup for Jupyter, MLFlow, FastAPI, etc.
    - Portable packaging of trained models for inference
    - Seamless transition from local development to cloud deployment
    - Scalability across GPU clusters or Kubernetes-based infra

2. Docker vs Virtual Machines in ML Workflows

Feature	Docker	Virtual Machines
Startup Time	⚡ **Fast: Seconds (near instant)**	🐢 **Slow:** Minutes (b**oot OS)**
Resource Use	**Efficient:** Lightweight, shares kernel	**Heavy:** Heavy, full OS per V**M**
Portability	**High**	**Medium**
Isolation	High (via namespaces)	Very High (hardware **emulation)**
Dev Workflow	Smooth, scriptable	Clunky, manual provisioning

Docker eliminates the overhead of full VMs while maintaining strong isolation — making it ideal for iterative ML workflows where fast provisioning and reproducibility are critical.

3. Who Benefits from Using Docker in AI/ML?

- **ML Engineers**: Reproducible training pipelines, scalable model serving.
- **Data Scientists**: Rapid experimentation without dependency hell.
- **DevOps Teams**: Simplified deployment across staging and production.
- **AI Hobbyists**: Run state-of-the-art models locally with minimal setup.

By using containers, teams become 10x more productive. They spend less time setting up environments and more time building models and delivering results.

4. Real-World Adoption: How Industry Leaders Use Docker for AI

Big names rely on containers to make AI reliable

a. Netflix: Orchestrating ML Workflows Using Docker

Netflix utilizes Docker containers to **manage and scale its machine learning workflows** efficiently. By containerizing their ML tasks, Netflix ensures consistent environments across development and production, aiding in tasks like **content recommendation and streaming optimization.**

Source : [Maestro: Data/ML Workflow Orchestrator at Netflix | by Netflix Technology Blog | Netflix TechBlog]

b. Uber: Streamlining ML Pipelines with Docker

Uber employs Docker to **containerize its machine learning workflows, facilitating consistent environments** for development and deployment. This approach enhances the scalability and reproducibility of their ML models, which are integral to services like ETA predictions and dynamic pricing.

Source : [From Predictive to Generative - How Michelangelo Accelerates Uber's AI Journey | Uber Blog]

c. Walmart: Scaling AI Solutions with Docker

Walmart leverages Docker to **containerize its AI applications**, facilitating **scalable and efficient deployment** across its vast retail infrastructure. This strategy supports various use cases, including **inventory management and customer experience** enhancements.

Source : [Machine Learning Platform at Walmart | by Thomas Vengal | Walmart Global Tech Blog | Medium]

d. NASA: Accelerating Data Analysis with Docker

NASA employs Docker containers to **standardize and expedite** its machine learning workflows, particularly in processing vast amounts of satellite data. Containerization aids in maintaining consistent environments, crucial for the reproducibility of scientific analyses.

Source : [How NASA JPL Processes 70 TB of Satellite Data Products a Day Using Amazon EC2 Auto Scaling with Spot Instances | Case Study | AWS]

e. Ingka Group (IKEA): Scalable MLOps with Docker and Kubernetes

Ingka Group, the parent company of IKEA, adopted Docker and Kubernetes to build a robust **MLOps platform**. This setup allows for **dynamic scaling** of AI/ML applications, **improved collaboration** through **uniform development environments**, and **enhanced security**. The containerized approach accelerates prototyping and deployment of new models, aligning with IKEA's commitment to innovation

Source: [Case Study: Ingka and Docker | Docker]

f. ZEISS Microscopy: Cross-Platform AI Model Deployment

ZEISS, a leader in optics and optoelectronics, utilizes **Docker to deploy AI models** across various platforms, including **cloud and local Windows-based clients**. By containerizing their AI solutions, ZEISS ensures **consistent performance** and **simplifies the distribution** of complex models, enhancing their microscopy software's capabilities.

Source: [Case Study: ZEISS and Docker | Docker]

B. 🔧 Docker Essentials for AI/ML

1. Images vs Containers

- **Docker Image**: A static blueprint that defines what is inside the container, including the base OS, Python environment, ML libraries (like `scikit-learn`, `pandas`, `torch`), and your code.
- **Docker Container**: A running instance of an image — like a live, isolated process with its own filesystem, ports, and environment variables.

In ML projects, Docker images capture the exact environment needed to run notebooks, train models, or serve APIs, while containers execute those tasks consistently across platforms.

2. Key Docker Commands for ML Practitioners

- `docker build -t <image-name> .`

→ Build an image from a Dockerfile.

- `docker run -p <host-port>:<container-port> <image-name>`

→ Run a container and map ports for web apps (e.g., Jupyter or FastAPI).

- `docker ps`

→ List running containers.

- `docker logs <container-name>`

→ View logs of a running or stopped container.

- `docker exec -it <container-name> bash`

→ Open an interactive shell inside a container.

These commands are essential for tracking, debugging, and managing containerized ML workloads.

3. Docker Networking, Volumes & Detached Mode

- **Networking (`-p`)**: Exposes container ports to the host system.
  - E.g., MLFlow at `localhost:5555`, Jupyter at `localhost:8888`.
- **Volumes (`-v`)**: Maps host directories to the container for persistent storage — useful for saving notebooks or models.
- **Detached Mode (`-d`)**: Runs the container in the background, ideal for long-running services like APIs or experiment trackers.

These options enable full control and flexibility during ML experimentation and deployment.

4. Container Lifecycle Management

- `docker stop <id>` / `docker start <id>`: Pause/resume containers.
- `docker rm <id>`: Remove containers.
- `docker images` / `docker rmi <image>`: Manage image versions.
- `docker logs -f <name>`: Stream logs live (useful during model training or API serving).
- `docker exec -it <name> sh`: Debug inside the container.

Mastering lifecycle commands is crucial for iterating quickly and maintaining a clean dev environment.

C. ML Workflow & Toolchain with Docker

1. Where Docker Fits into the ML Workflow?

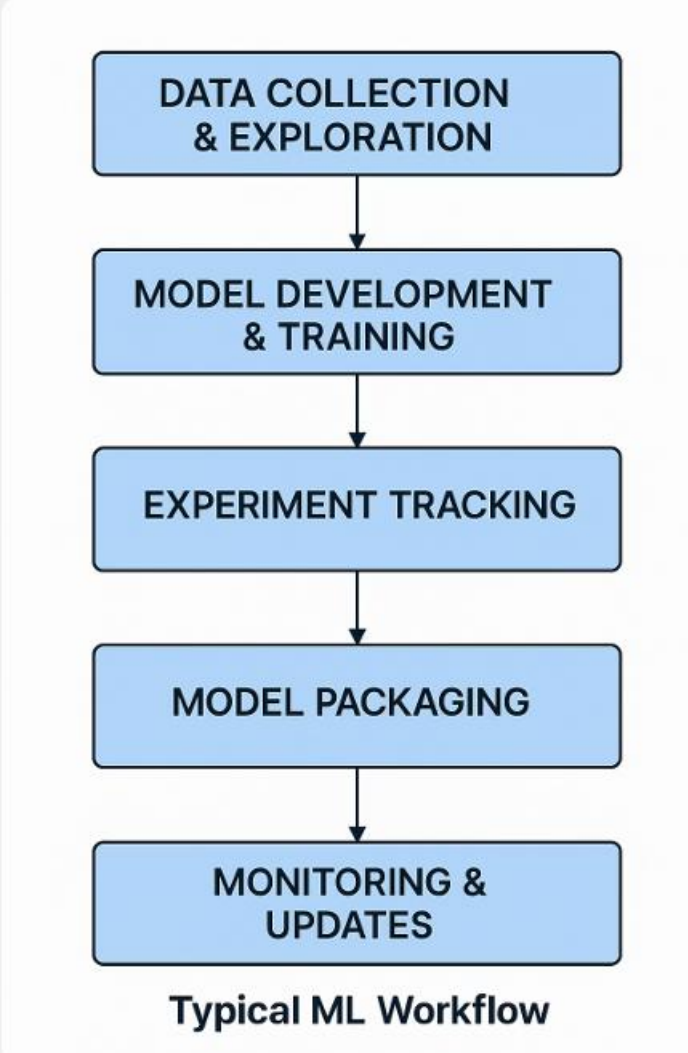

Image 2. Typical ML Workflow

2. Development Environments

- **Jupyter Notebooks**: Easily run notebooks inside prebuilt images like `jupyter/scipy-notebook`, preserving your work with volume mounts.

- **VS Code (Remote - Containers)**: Attach your editor directly to the container environment for seamless ML development.

3. Experiment Tracking

- **MLFlow**: Lightweight experiment manager and model registry — easily run via container on port 5555.
- **DVC (Data Version Control)**: Manages datasets and model versions alongside code.
- **Weights & Biases**: Industry-grade experiment tracking with rich visualizations and hyperparameter tuning support.

Docker enables these tools to be quickly spun up without manual setup.

4. Model Serving & APIs

- **FastAPI / Flask**: Pythonic web frameworks to serve ML models via REST APIs — containerized for portability and deployment ease.

- **TensorFlow Serving / TorchServe**: Production-grade model servers optimized for TensorFlow or PyTorch.

- **Gradio / Streamlit**: For rapid prototyping and interactive demos, often deployed via Docker for reproducibility.

5. Deployment Options

- **Docker Hub**: Hosts and distributes public/private images for reuse.
- **Kubernetes**: Container orchestration system for scalable, distributed ML workloads.
- **Hugging Face Spaces (Docker SDK)**: Fastest way to deploy ML apps with a Dockerfile and GitHub repo — free for most use cases.

Using Docker as the foundation makes these deployment targets interchangeable and production-ready.

D. Docker in the world of LLMs / Agentic AI

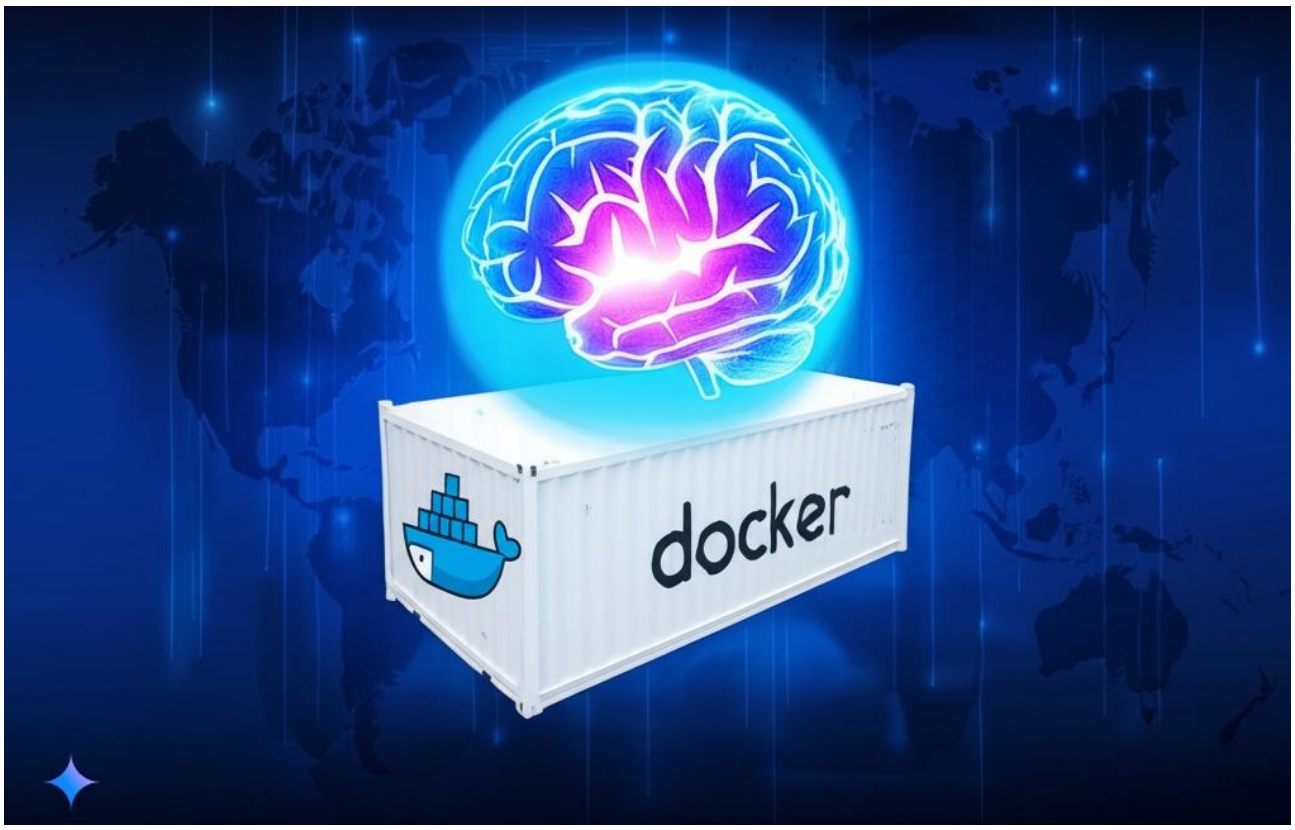

Image 3.

1. Running Models with Docker Model Runner

- **Dockerized abstraction of any trained model:** Package your model with all dependencies
- **Launch a ready-to-serve REST API in seconds:** No need to write custom API code
- **Framework agnostic (TF, Torch, XGBoost):** Works with all major ML frameworks
- **Example:** docker run -p 8080:8080 ghcr.io/mlc-ai/model-runner:latest

2. Docker + MCP Tooling

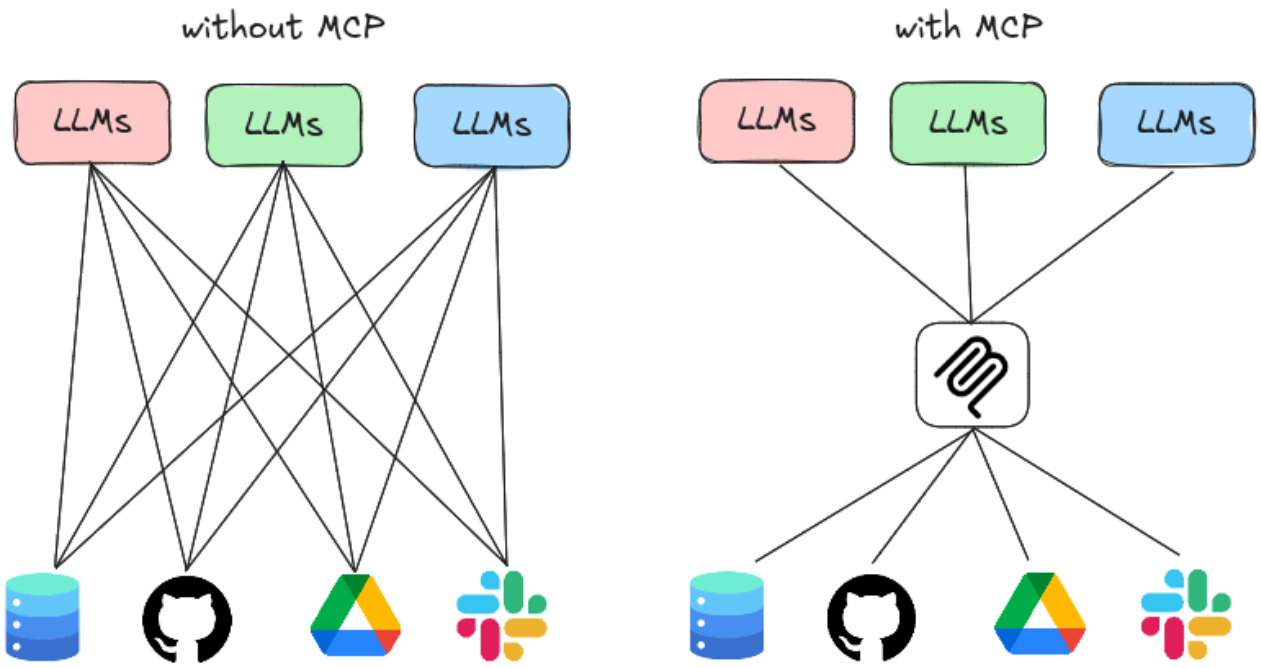

Image 4. Source: https://hoangndst.com/blog/model-context-protocol

- **What's MCP?** Model Context Protocol lets AI models access real-world tools
- **Docker + MCP =**
  - Self-hosted MCP toolkits (Terraform, Kubernetes, CLI agents)
  - Tool-aware autonomous agents
- **Example:** docker run -p 3000:3000 realops/kubernetes-mcp server:latest

3. Deploying NVIDIA NIM with Docker

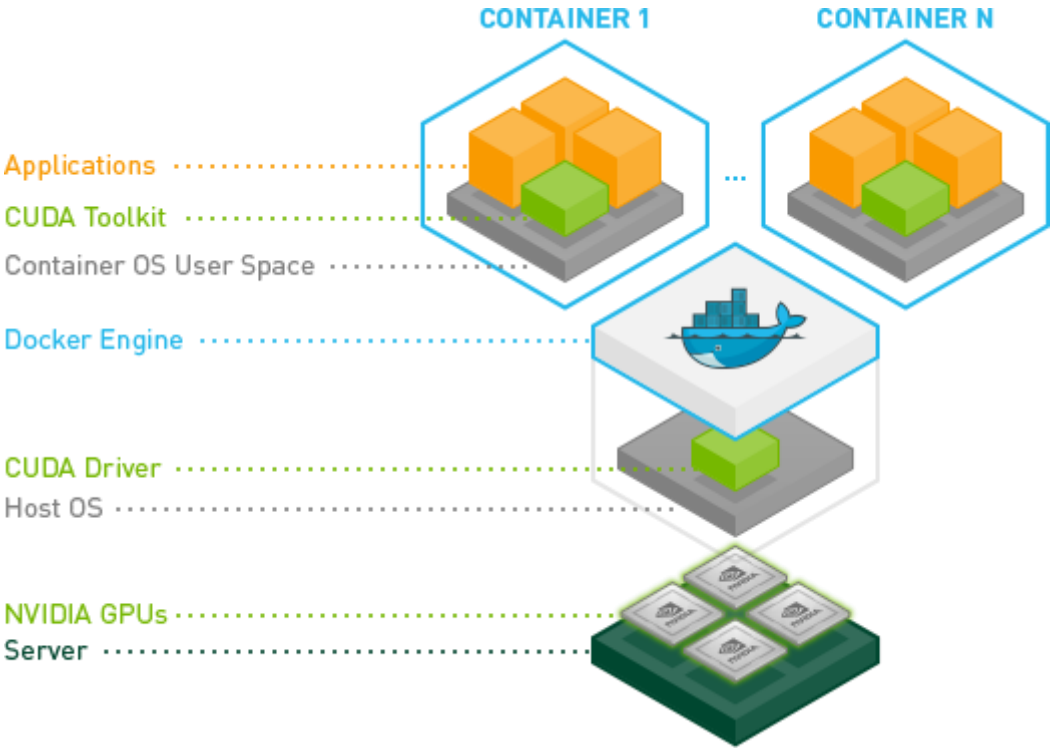

Image 5. Source: https://developer.nvidia.com/blog/nvidia-docker-gpu-server-application-deployment-made-easy/

- **Containerized inference microservices** (Mixtral, LLaMA2, etc.)
- **REST interface for easy integration**
- **GPU-accelerated, secure, scalable**
- **Example:** docker run --gpus all -p 8000:8000 nvcr.io/nim/mixtral:latest

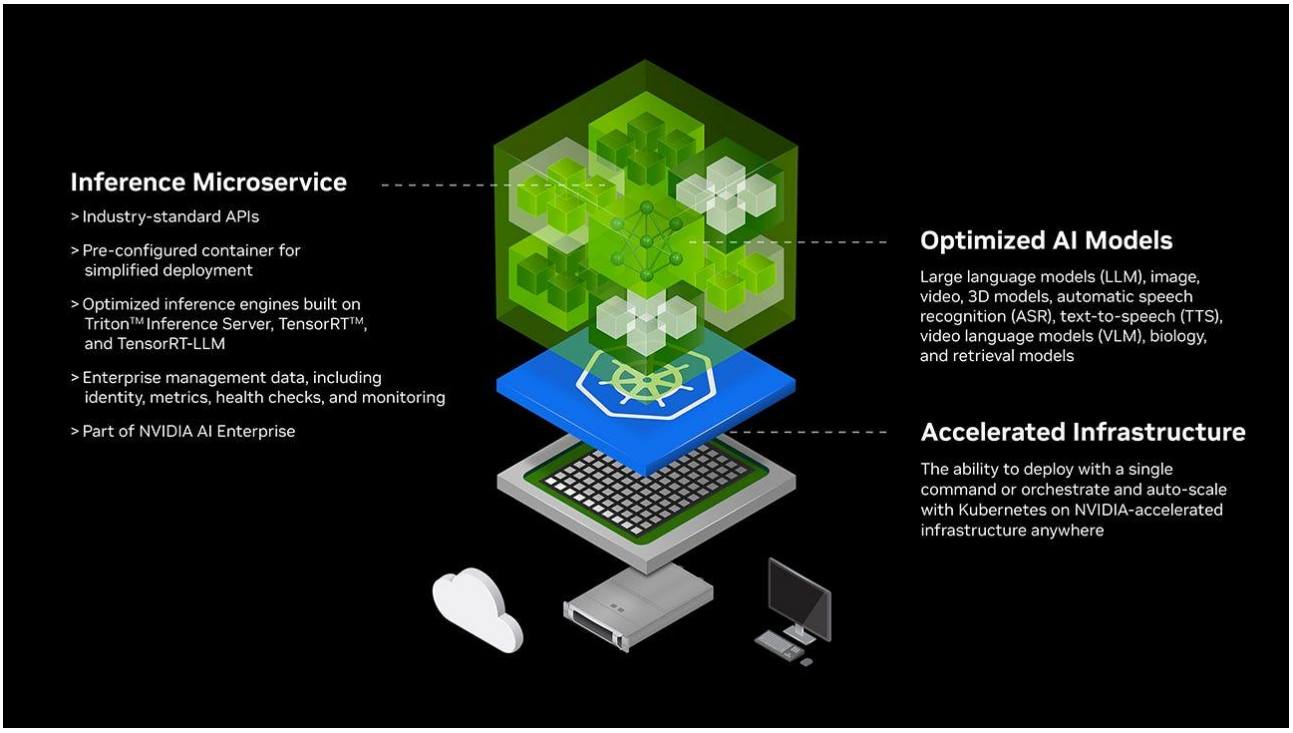

Image 6. Souce: https://pub.towardsai.net/an-introduction-to-using-nvidias-nim-api-da653041b212

4. Agentic AI + Docker

- **Agentic AI** = LLMs + Tools + Memory + Goals

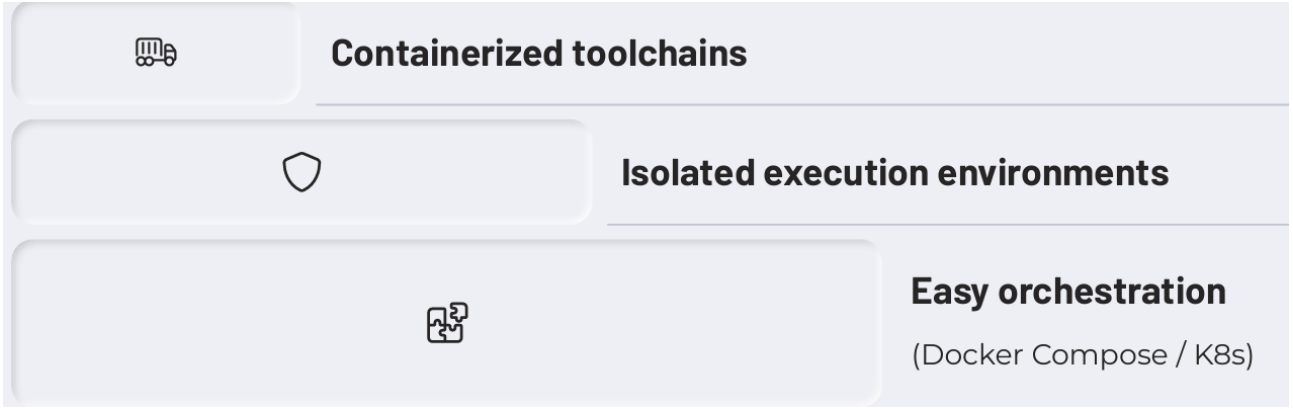

Image 7.

Build and deploy autonomous AI agents with ease

5. Sample Agentic DevOps Setup

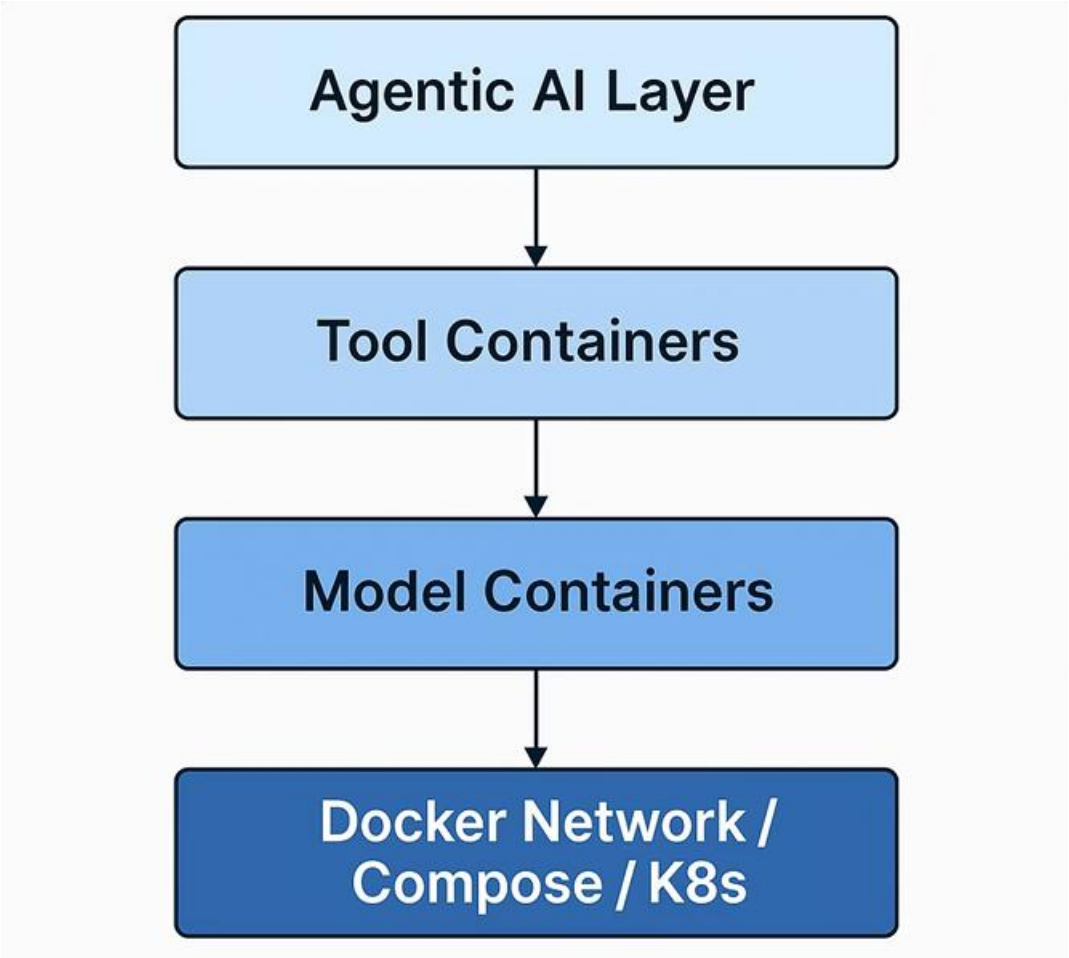

Image 8. Sample Agentic DevOps Setup

Reproducible. Scalable. Composable

E. Project 1: Dockerize Your AI App

1. Goal: Take any Python-based AI app and:

- Package it into a Docker container
- Include models, dependencies, and web interface
- Run it anywhere with a single command

2. Example App: FastAPI Sentiment Classifier

a. Step 1: Create Your App Files

- **File app.py**

```
# file app.py

from fastapi import FastAPI
from pydantic import BaseModel
from transformers import pipeline

app = FastAPI()
classifier = pipeline("sentiment-analysis")

class InputText(BaseModel):
    text: str

@app.post("/predict")
def predict(input: InputText):
    result = classifier(input.text)[0]
    return {
        "label": result["label"],
        "confidence": round(float(result["score"]) * 100, 2)
    }
```

- **File requirements.txt**

```
fastapi
uvicorn
transformers
torch
```

b. Step 2: Create the Dockerfile

```
# Use Python base image
FROM python:3.10-slim

# Set working directory
WORKDIR /app

# Copy files
COPY requirements.txt requirements.txt
COPY app.py app.py

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Expose API port
EXPOSE 8000

# Run the app
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

c. Step 3: Build and Run the Docker Image

```
# Build the Docker image
docker build -t ai-sentiment-app .

# Run the container
docker run -p 8000:8000 ai-sentiment-app
```

- Then open: *http://localhost:8000/docs*
- To test the /predict endpoint with JSON:

```
{
  “text": "This product is amazing!"
}
```

3. Project Summary

- **Input:** Text via FastAPI
- **Backend:** Transformers pipeline
- **Dockerized:** Yes — portable, reproducible, production-ready
- **Use Case:** Deploy AI APIs on local server, cloud, Kubernetes

F. Project 2: Running ML Containers like JupyterLabs and MLFlow with Docker

1. Goal

Use pre-built AI/ML Docker images to run real-world tools like JupyterLabs and MLFlow. Learn how to connect, manage, and persist your work — with a full lifecycle workflow

2. Requirements

- Docker Desktop / Rancher Desktop (installed and running)
- Browser access (for Jupyter)
- Basic terminal skills

3. Part 1: Launch MLFlow using a Container

a. Step 1: Pull the Image

```
docker pull ghcr.io/mlflow/mlflow:latest
```

b. Step 2: Run with Port Mapping

```
docker run -p 5555:5000 ghcr.io/mlflow/mlflow:latest mlflow server --host
0.0.0.0
```

- Open in browser using http://localhost:5555/

- While the MLFlow container is running and you are able to access it, on the console you are stuck attached with the container

[sample outout]

```
docker run -p 5555:5000 ghcr.io/mlflow/mlflow:latest mlflow server --host
0.0.0.0
[2025-05-26 04:23:01 +0000] [13] [INFO] Starting gunicorn 23.0.0
[2025-05-26 04:23:01 +0000] [13] [INFO] Listening at: http://0.0.0.0:5000
(13)
[2025-05-26 04:23:01 +0000] [13] [INFO] Using worker: sync
[2025-05-26 04:23:01 +0000] [14] [INFO] Booting worker with pid: 14
[2025-05-26 04:23:01 +0000] [15] [INFO] Booting worker with pid: 15
[2025-05-26 04:23:01 +0000] [16] [INFO] Booting worker with pid: 16
[2025-05-26 04:23:01 +0000] [17] [INFO] Booting worker with pid: 17
```

- If you want come back to console you have to kill the container using ***ctrl* + *c*.** Try doing that.

- Once exited, you can list it using

```
# this command will list only running containers
# you will not see mlflow running
docker ps

# this command shows you last run container, even if its stopped (exited)
docker ps -l

# more commands you can explore
docker ps -n 2
docker ps -a

#note the container id, which you will use to delete the container
```

- now delete the container as

```
# replace xxxx with actual container id/name noted above
docker rm xxxx

# if you are removing a running container add -f option as
docker rm -f xxxx
```

- You could instead launch the container in detached mode, which is the common way of running the container, which keeps running but in the background as ,

```
docker run -d -p 5555:5000 --name mlflow ghcr.io/mlflow/mlflow:latest
mlflow server --host 0.0.0.0
```

- where, newly added options are
  - `-d`: run container in detached mode
  - `--name mlflow`: sets the name of the container as `mlflow` instead of a auto generated random name
- you can list the containers using

```
docker ps
```

aand connect to it using http://localhost:5555/ (replace localhost with actual hostname / ip if you have set up docker on a remote server/VM)

- Few more container manageemnt commands that you could try

```
# check the logs for mlflow container
docker logs mlflow

# follow the logs. exit with ^c
docker logs -f mlflow

# Get inside container's shell with
docker exec -it mlflow sh

# use ^d to exit
```

4. Part 2: Run Jupyter for ML Notebooks with a local Volume

a. Step 1: Create Local Directory

```
mkdir -p ~/ml-docker/notebooks
```

b. Step 2: Run with Volume Mount

```
docker run -d -p 8888:8888 --name notebook -v ~/ml-
docker/notebooks:/home/
jovyan/work jupyter/scipy-notebook
```

- Check terminal logs for URL with token.

```
docker logs notebook
```

[sample output]

```
....
    To access the server, open this file in a browser:
      file:///home/jovyan/.local/share/jupyter/runtime/jpserver-7-
open.html
    Or copy and paste one of these URLs:

  http://9a95b748605a:8888/lab?token=4390ed0a681b70eaa3b87bd154fc786b833
c712ca6ed24ff

  http://127.0.0.1:8888/lab?token=4390ed0a681b70eaa3b87bd154fc786b833c71
2ca6ed24ff
```

- Open in browser. e.g. http://127.0.0.1:8888/lab
- Create new notebook inside `work/` .
- Create a new notebook and save it. e.g. `work/basic-ml.ipynb`
- Open it from the host directory to check if it is shared via a volume between host and container.

c. Sample Notebook 1: Basic Pandas + Scikit-learn

- Save this as `ml-docker/notebooks/basic-ml.ipynb` or create inside Jupyter

```
# Step 0 / Cell 0
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report, r2_score, mean_squared_error

# Step 1: Load the Iris dataset
print("📥 Loading the Iris dataset...")
data = load_iris()

# Step 2: Explore the dataset structure
print("\n📝 Feature names:", data.feature_names)
print("🎯 Target classes:", data.target_names)
print("📐 Data shape:", data.data.shape)

# Step 3: Create a DataFrame for exploration
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
print("\n🔍 First 5 rows of the dataset:")
print(df.head())

# Step 4: Define features (X) and target (y)
X = df[data.feature_names]
y = df['target']

# Step 5: Train a Logistic Regression model
print("\n⚙️ Training Logistic Regression model...")
model = LogisticRegression(max_iter=200)
model.fit(X, y)

# Step 6: Make predictions
y_pred = model.predict(X)

# Step 7: Evaluate the model
accuracy = accuracy_score(y, y_pred)
print(f"\n📊 Accuracy Score: {accuracy:.2f}")

print("\n📋 Classification Report:")
print(classification_report(y, y_pred, target_names=data.target_names))

# Step 8 : Confusion Matrix Plot
cm = confusion_matrix(y, y_pred)
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=data.target_names,
            yticklabels=data.target_names)
```

```
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("🔍 Confusion Matrix")
plt.show()
```

5. Part 3: Connect Jupyter with MLFlow to log Experiments to

a. Step 1: Install mlflow with pip

Create a new notebook `run_experiment.ipynb` and execute the following command as one of the cells.

```
!pip install mlflow
```

b. Step 2: Add Experiment with MLFlow Tracking

Create a new notebook `run_experiment.ipynb` and add the following code

```
import mlflow
from mlflow.models import infer_signature
from sklearn.linear_model import LinearRegression
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error

# 1. Set tracking URI to local MLflow server
mlflow.set_tracking_uri("http://host.docker.internal:5555")
print("📡 Tracking to:", mlflow.get_tracking_uri())

# 2. Set experiment name (create if not exists)
mlflow.set_experiment("simple-linear-demo")

# 3. Create and log a run
with mlflow.start_run():
    # Generate toy regression data
    X, y = make_regression(n_samples=100, n_features=1, noise=10,
random_state=42)

    # Train model
    model = LinearRegression()
    model.fit(X, y)

    # Predict and evaluate
    y_pred = model.predict(X)
    mse = mean_squared_error(y, y_pred)

    # Infer model signature and input example
    signature = infer_signature(X, y_pred)
    input_example = X[:5]  # A small batch as sample input

    # Log parameters and metrics
    mlflow.log_param("model_type", "LinearRegression")
    mlflow.log_metric("mse", mse)

    # Log model with signature and example
    mlflow.sklearn.log_model(
        model,
        artifact_path="model",
        signature=signature,
        input_example=input_example
    )

    print(f"✅ Run logged with MSE: {mse:.2f}")
```

6. Part 4: Docker Essentials for AI/ML

Concept	Explanation
Image	Blueprint for containers (e.g., `jupyter/scipy-notebook`)
Container	Running instance of an image
Tag	Versioned label for an image (e.g., `latest`, `2.4.1`)
Port Mapping (`-p`)	Connect host port to container port (e.g., -p 8888:8888)
Detached Mode (`-d`)	Run container in background
Interactive Terminal (`-it`)	Keeps terminal attached for CLI-based containers
Volume Mount (`-v`)	Bind-mount host dir into container

7. Part 5: Container Lifecycle Management Commands

```
docker ps      # Show running containers

docker ps -a       # Show all containers

docker stop <container_id>       # Stop container

docker start <container_id>  # Restart a stopped container

docker rm <container_id>   # Remove container

docker logs  <container_id>      # View logs

docker exec -it bash    # Open interactive shell
```

8. Project Outcome

You now know how to:

- Pull and run ML-ready containers
- Mount notebooks for persistence
- Interact with Python-based ML frameworks inside Docker
- Manage containers like a pro