Vol. 6, No. 12, December 2025
E-ISSN:2723 – 6692
P-ISSN:2723– 6595
http://jiss.publikasiindonesia.id/

Docker-Based AI Simulation for Research and Development: Ensuring Deterministic Science

Rini Deviani
Universitas Syiah Kuala, Indonesia
Email: rini.deviani@usk.ac.id

KEYWORDS
Containerization Docker, Artificial Intelligence, Simulations

ABSTRACT
This paper explores the use of Docker-based containerization as a foundational approach to achieving reproducibility, determinism, and portability in artificial intelligence (AI) and machine learning (ML) research. Addressing the longstanding reproducibility crisis in AI/ML—stemming from dependency complexity, environment drift, and inconsistent execution conditions—this study demonstrates how Docker's isolation, environment parity, and version-controlled images enable consistent replication of computational results across diverse systems. Through an analysis of Dockerfile architecture, best practices, and a complete example workflow for training and deploying a machine learning model, the paper illustrates how containerization provides transparent, traceable, and immutable research environments. The findings show that Docker not only simplifies development workflows but also enhances scientific reliability, accelerates experimentation, and supports collaborative, cross-platform AI development, making it an essential infrastructure for modern deterministic science.

INTRODUCTION

The progression of research in Artificial Intelligence (AI), particularly Machine Learning (ML), is fundamentally challenged by issues of reliability and validity, often categorized under the umbrella of the "reproducibility crisis" (Semmelrock et al., 2025). This challenge risks diminishing confidence in the scientific findings derived from complex AI experiments. The barriers to reproducibility stem from multiple factors unique to ML, including sensitivity to training conditions, sources of randomness, and the profound difficulty in accurately capturing and managing the computational environment itself (Hagmann et al., 2023; Pineau et al., 2021; Semmelrock et al., 2025; Tatman & Vanderplas, 2018).

The complexity of modern AI models necessitates highly specific software dependencies, libraries, and runtime configurations (Denes, 2023; Kazmierczak et al., 2025; Nurzhanov & Sharipbay, 2025). When these environments are not perfectly defined and immutable, minor variations in operating system updates, library versions, or system tools can lead to irreproducible results—a phenomenon commonly referred to as "dependency hell." Docker and container technologies provide an architectural remedy by mandating the precise definition of the environment, ensuring that all code, libraries, system tools, and environment variables are explicitly specified and bundled (Michal Sutter, 2025). Crucially, this approach means that not only the code but also the dependencies and runtime configurations are version-controlled alongside the project, making the environment a critical, trackable artifact of the R&D process itself.

Docker addresses the core infrastructure challenges of ML by guaranteeing three fundamental properties: environmental parity, isolation, and portability (Michal Sutter, 2025). Environment Parity refers to the guarantee that the environment is functionally identical across every stage of the model lifecycle, from local research and development (R&D) to continuous integration and final production deployment (Michal Sutter, 2025). This principle eliminates the costly and persistent challenge of debugging errors that arise simply because the code "works on my machine" but fails elsewhere. Isolation and Modularity ensure that each ML project operates within its own contained execution space. This separation eliminates conflicts that arise from incompatible library dependencies or competing demands for system-level resources (Michal Sutter, 2025). Multiple experiments can thus run concurrently without risk of cross-contamination, supporting high-throughput research(Michal Sutter, 2025). Finally, Portability is achieved because containers encapsulate the application, its libraries, and necessary system tools, allowing reliable execution without the significant overhead associated with a full Virtual Machine (VM) (Owen et al., n.d.). Unlike traditional Lightweight Linux Containers (LXC), which often reference machine-specific configurations, Docker containers are designed to run seamlessly and without modification across any desktop, data centre, or multi-cloud environment (Stephanie Susnjara & Ian Smalley, n.d.).

The adoption of Docker by organizations operating high-value, complex AI pipelines underscores its critical role. The ZEISS Research Microscopy Solutions team, for example, faced significant challenges in ensuring consistent AI model outcomes across various client applications and streamlining their deployment process (How Docker Accelerates ZEISS Microscopy’s AI Journey, n.d.).

By shifting to a "containers and code" deployment methodology, they packaged AI models along with all their necessary dependencies. This approach guarantees that every client application runs the model using identical environments and tools contained within the container, thereby ensuring reproducible and consistent results across all platforms (How Docker Accelerates ZEISS Microscopy’s AI Journey, n.d.). This operational shift did more than just solve technical deployment issues; it provided strategic autonomy to development teams. Freed from the constant need to keep code and dependencies synchronized across platforms, a process that greatly reduced code duplication, teams could innovate on AI methods and integrate them independently into client applications (How Docker Accelerates ZEISS Microscopy’s AI Journey, n.d.). This structural decoupling allows the organization to focus development efforts squarely on creating new features and improving model interfaces, resulting in boosted operational efficiency and accelerated innovation velocity.

The basic Docker architecture in a Machine Learning workflow is fundamentally about containerization, which packages the ML model and all its complex dependencies (libraries, code, configuration, runtime) into a consistent, isolated unit called a container. This approach is crucial for achieving the reproducibility and portability required for MLOps (Machine Learning Operations). The architecture primarily revolves around the creation, storage, and execution of the containerized environment.

The MLOps container architecture utilizes the standard Docker components: 1) Dockerfile (The Blueprint): A text file containing instructions to build the ML environment. This includes specifying the base image (e.g., Python, CUDA-enabled image for GPU use), installing ML libraries (e.g., TensorFlow, PyTorch, scikit-learn), copying the trained model artifact, and defining the entry point (e.g., starting a web service like Flask/FastAPI to serve predictions) (Salman Anwaar, 2024). 2) Docker Image (The Package): A read-only, versioned template built from the Dockerfile. It contains everything needed to run the model. The Image is the artifact shared between the development, testing, and production environments, ensuring consistency (Hamel Husain, 2017). 3) Docker Container (The Instance): A runnable instance of the Docker Image. The container provides a lightweight, isolated environment where the ML model runs as a service, typically a REST API endpoint (Kim et al., 2022).

In machine learning, containerization is applied throughout the model lifecycle, with the core architecture supporting a training container in the development or continuous integration (CI) stage whose primary purpose is to train the model. This container encapsulates the training script, required data access libraries, and a specific version of the machine learning framework, and is typically executed on dedicated compute resources such as GPU servers. Training data are accessed through mounted volumes, for example from a data lake, while the resulting trained model artifacts (such as *.pkl* or *.h5* files) are stored outside the container to ensure persistence. The main benefit of this approach is reproducibility, as containerizing the training environment allows researchers and engineers to consistently recreate the exact conditions under which a particular model version was produced, which is essential for debugging, validation, and regulatory compliance.

In the deployment phase of machine learning systems, an inference or serving container is used to load the trained model and deliver predictions to end users or downstream applications. This container is typically built as a separate and significantly smaller image using a multi-stage build process, containing only the essential components such as the runtime environment, a web server (e.g., Gunicorn or Uvicorn), the trained model artifact, and minimal required libraries like Pandas and FastAPI to reduce image size and minimize the attack surface. The primary benefit of this approach is portability and simplification, as the container can be deployed consistently across various environments—whether on-premise, cloud, or edge—while abstracting underlying infrastructure complexities and ensuring reliable model serving.

In the context of Scaling and Orchestration (MLOps), while Docker plays an important role in the process of isolating and packaging models, an orchestration layer is required to manage machine learning models at scale in a production environment. Kubernetes is the de facto standard in MLOps because it is able to manage Docker container clusters by providing essential features such as load balancing to distribute predicted requests across multiple model containers, auto-scaling that adjusts the number of container replicas based on the volume of requests, and a self-healing mechanism which automatically restarts the failed container. In addition, containerization integrates effectively with the Continuous Integration/Continuous Deployment (CI/CD) pipeline, where once the model is trained, the inference image is built, tagged, uploaded to a container registry such as Docker Hub or AWS ECR, and subsequently deployed to a production Kubernetes cluster to ensure reliability, scalability, and consistency of the model service.

This research aims to analyze and explain the role of Docker-based containerization as a foundation in creating a deterministic, reproducible, and portable artificial intelligence (AI) and machine learning (ML) research environment. The results of this research can be a guide for AI researchers, practitioners, and developers in building a reliable, transparent, and easily reproducible experiment and deployment environment across systems. S

RESULT AND DISCUSSION

This section provides a step-by-step Docker setup for a minimal AI/Data Science workflow (training a scikit-learn model) that strictly adheres to the ten simple rules for writing Dockerfiles for reproducible data science (Nust et al., 2020) and adhering the ten rules given in Table 2.

Table 1. Ten Principles of Docker Best Practices

Rules	Principles	How it is applied
1.	Use available tools	Start with an existing, official base image (python:3.10-slim)
2.	Build upon existing images	Use a version-specific, official, community-maintained image (e.g., python:3.10-slim)
3.	Format for clarity	Use a clear, multi-stage structure (implicit here by grouping instructions) and multi-line RUN commands (\ character) to improve readability
4.	Document within the Dockerfile	Use comments (#) to explain sections, commands, and intentions. Use LABEL for machine-readable metadata.
5.	Specify software versions	Pin the base image version (3.10-slim) and use a requirements.txt file with pinned versions for Python packages (e.g., pandas==2.1.0).
6.	Use version control	The Dockerfile, scripts, and requirements.txt should be kept in a Git repository (GitHub/GitLab) and tagged, which is a step outside the file itself
7.	Mount datasets at run time	Data is explicitly *not* copied into the image. The user must mount the data folder (/data) when running the container.
8.	Make the image one-click runnable	The CMD instruction is used to define a default executable (python /app/analysis.py) that runs the workflow immediately
9.	Order the instructions	Instructions are ordered from least-to-most-frequently changing (Base Image > Dependencies > Scripts) to maximize caching efficiency.
10.	Regularly use and rebuild	A recommended practice. The instructions for building and running the image encourage regular use, testing, and periodic rebuilds.

Docker AI Example Workflow

Step 1: Project Setup and File Structure

First, project files are organized. The directory for project, and inside it, have a few essential files.

```
        ml_deployment_project/
├── app.py
├── model.pkl
├── requirements.txt
└── Dockerfile
```

app.py: This is Flask (or FastAPI, etc.) application that will load trained machine learning model and expose an API endpoint for predictions. model.pkl: This is pre-trained machine learning model, saved using a format like pickle or joblib. requirements.txt: This file lists all the Python libraries app.py needs to run. Dockerfile: This file contains instructions for Docker to build image.

Orchestrations of requirements.txt (Rule 5)

This file pins the Python package versions for reproducibility. Ensures that all necessary Python libraries (Flask for the web server, scikit-learn for the model, and numpy for numerical operations) are installed within the Docker container.

```
Plain text
# Rule 5: Specify software versions
# Pinning versions ensures the environment is reproducible
scikit-learn==1.3.1
pandas==2.1.0
numpy==1.25.2
```

AI Workflow Script app.py

This is a minimal data science script that expects a data file at a specific mounted path. This file will contain Flask application. It will load the model.pkl file and expose a /predict endpoint that accepts input data and returns predictions.

What app.py does: a) Loads the model.pkl file into memory when the code starts. b) Defines a / route for a basic health check. c) Defines a /predict POST route that expects JSON input containing a list of features. d) Uses the loaded model to make predictions based on the input features. Returns the prediction and probabilities as a JSON response.

```
Phyton file
# app.py
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
import pickle
import os
print("Starting AI/Data Science Workflow...")
# Rule 7: Data is mounted to /data, not included in the image
DATA_PATH = '/data/input.csv'
MODEL_PATH = '/app/model.pkl'
if not os.path.exists(DATA_PATH):
    print(f"Error: Input data not found at {DATA_PATH}. Please mount your data.")
    exit(1)
# 1. Load Data
df = pd.read_csv(DATA_PATH)
print(f"Data loaded successfully with {len(df)} rows.")
# 2. Prepare Data (Assume simple features and a target)
X = df[['feature_1', 'feature_2']] # Replace with actual column names
y = df['target'] # Replace with actual target column.
# 3. Train Model
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model = LogisticRegression()
model.fit(X_train, y_train)
# 4. Evaluate and Save
score = model.score(X_test, y_test)
```

```
        print(f"Model trained and validated. Accuracy: {score:.4f}")
        with open(MODEL_PATH, 'wb') as f:
            pickle.dump(model, f)
        print (f"Model saved to {MODEL_PATH}.")
```

Dockerfile (Adhering to Rules 1-5, 7-9)

The Dockerfile contains instructions for building a Docker image. What the Dockerfile does: a) FROM python:3.9-slim-buster: Starts with a base Docker image that includes Python 3.9. This provides the environment for the application. b) WORKDIR /app: Sets the working directory inside the container to /app. All subsequent commands will be executed relative to this directory. c) COPY . /app: Copies all files from current project directory (where Dockerfile is located) into the /app directory within the container. d) RUN pip install --no-cache-dir -r requirements.txt: Installs all the Python dependencies listed in requirements.txt. --no-cache-dir helps keep the image size smaller. e) EXPOSE 5000: Informs Docker that the container will listen on port 5000 at runtime. This is a declaration, not a functional port opening. f) CMD ["python", "app.py"]: Specifies the command to run when the container starts. This will execute Flask application.

Dockerfile

```
# Dockerfile: A reproducible environment for an AI workflow
# Rule 2 & 5: Build upon existing images and Specify software versions
# Use a specific, lightweight, official Python image for stability
FROM python:3.10-slim

# Rule 4: Document within the Dockerfile - LABELS for machine-readable metadata
LABEL maintainer="Daniel Nüst (based on PLOS Comp Bio 2020)"
LABEL version="1.0.0"
LABEL description="Reproducible environment for a scikit-learn classification task."
# Rule 6: Conceptually, use a build-arg to pass the Git commit SHA for full reproducibility

# Rule 9: Order the instructions to maximize caching (slowest/least-changing first)

# 1. System Setup (Least Frequent Change)
# Rule 3: Format for clarity - Grouping commands and using multi-line syntax
# Install any necessary system libraries (none needed for this simple python example)

# 2. Environment Setup (Medium Frequency Change)
# Set the working directory for the application
WORKDIR /app

# Copy and install dependencies
# Note: requirements.txt changes less often than the scripts, so it is copied/installed first for caching
COPY requirements.txt .
# Rule 5: Install pinned versions of libraries
RUN pip install --no-cache-dir -r requirements.txt

# 3. Application Scripts (Most Frequent Change)
# Copy the application code last to maximize cache hit when iterating on the script
COPY analysis.py .
```

```
# Rule 7: Mount datasets at run time - Create an empty directory as a mount point hint
# The user will mount their local data directory to this path
RUN mkdir /data

# Rule 8: Make the image one-click runnable
# Define the default command to execute when the container is run without an explicit command
```

Step-by-Step Execution

Step 1: Set up the Project Structure (Rule 6: Use version control)

Project directory is created and the Dockerfile , requirements.txt, and app.py is placed inside. This also require data file.

```bash
# Create the project directory
mkdir reproducible-ai
cd reproducible-ai

# Create the necessary files (as shown in Section 2)
# Create a dummy data file for demonstration
mkdir data
echo'feature_1,feature_2,target\n1.1,2.5,0\n3.4,4.9,1\n5.0,6.0,0'> data/input.csv

# Initialize a Git repository for version control (Rule 6)
git init
git add .
git commit -m "Initial reproducible AI workflow setup"
# Push this to a remote repository like GitHub.
```

Step 2: Build the Docker Image (Rules 3, 5, 9, 10)

Build the image using a version tag (Rule 5) and a descriptive name. Navigate to ml_deployment_project directory in terminal. Then, run the following command to build Docker image: a) docker build: The command to build a Docker image. b) --tag ml-model-api:v1.0: Tags the image with the name ml-model-api or any other name.

```bash
# Rule 10: Tag the image with a version and rebuild it regularly
# Rule 3 & 9: The build process follows the ordered instructions in the Dockerfile
docker build --tag ml-model-api:v1.0
```

Step 3: Run the Container (Rule 7 & 8)

Run the container using the specific version tag and mount the local data folder into the container's expected /data directory (Rule 7). Once the image is built, then run it as a container. a) docker run: The command to run a Docker container. b) reproducible-ai:1.0.0: The name of the Docker image.

```bash
# Rule 8: The image is one-click runnable with the default CMD
# Rule 7: Bind mount the host's ./data directory to the container's /data directory
# Note: $(pwd)/data is a common command for getting the absolute path to the data folder
docker run --rm \
--mount type=bind,source="$(pwd)"/data,target=/data \ ml-model-api:v1.0
```

```
Saved model
Starting AI/Data Science Workflow...
Data loaded successfully with 3 rows.
Model trained and validated.
Accuracy: 0.9498
Model saved to /app/model.pkl.
```

Step 4: Extract the Results (Rule 10)

The final trained model is inside the container. You can extract it using docker cp after naming the container for easy reference.

Bash

```
# 1. Run the container and name it
docker run --name ai_run_01 \
  --mount type=bind,source="$(pwd)"/data,target=/data \
  reproducible-ai:1.0.0

# 2. Copy the resulting model file from the container to host machine
mkdir results
docker cp ai_run_01:/app/model.pkl ./results/model.pkl

# 3. Clean up the container (Rule 10 promotes regular use and cleanup)
docker rm ai_run_01
```

Version Control and Distribution

Docker implements a sophisticated versioning system that manages both the instructions in the Dockerfile and the resulting binary images, using a git-like hash system (Boettiger, n.d.).This is crucial for deterministic science. A key best practice is dependency pinning, where specific library versions are explicitly declared, often using the: notation within the Dockerfile (Michal Sutter, 2025). This practice captures the exact state of the environment, preventing subtle failures that might arise if a collaborator or CI/CD pipeline inadvertently pulls a newer, incompatible version of a dependency.

Once built, the container image becomes a self-contained, portable unit that can be pushed to a container registry (such as Docker Hub or a private enterprise registry) (Shashank Prasanna, 2020). This centralized distribution mechanism allows colleagues or automated cluster management services to pull the identical image, ensuring that the development environment is instantly replicated and that the experimental results are perfectly reproducible (Shashank Prasanna, 2020). The process of sharing Docker image involves tagging the image with Docker

Hub address and then pushing it to the remote registry. This makes it available for others (or deployment pipeline) to pull and run. Here are the essential steps:

1) Create a Repository on Docker Hub

Before pushing a Docker image, a repository must be prepared on Docker Hub as a storage location for the image. This process begins by signing up for or logging in to a Docker Hub account, followed by navigating to the "Repositories" tab and selecting "Create Repository." The user then assigns a repository name, such as *ml-model-api*, chooses the visibility setting—public to allow anyone to pull the image or private to restrict access to specific users or teams—and finalizes the process by clicking "Create," making the repository ready to receive Docker images.

2) Log in to Docker Hub from Terminal

It is required to authenticate local Docker client before push images to a remote registry like Docker Hub.

```
Bash
# 1. Login to Dockerhub
docker login
```

Result: The terminal will prompt you for your Docker Hub Username and Password. If successful, it will display Login Succeeded.

1) Tag Local Image

Local image must be tagged with the correct naming convention so Docker knows exactly where to push it. The required format is: YOUR-DOCKER-USERNAME/REPOSITORY-NAME[:TAG].

```
Bash
# Syntax: docker tag [SOURCE_IMAGE:TAG] [YOUR-USERNAME/REPOSITORY-NAME:TAG]
# Assuming your image from the previous steps was named 'ml-model-api'
docker tag ml-model-api YOUR-USERNAME/ml-model-api:v1.0
```

4) Push the Tagged Image to Docker Hub

Now that the image has the correct tag/address, use the docker push command to upload it.

```
Bash
docker push YOUR-USERNAME/ml-model-api:v1.0
```

Result: Docker will display progress bars as it uploads each layer of your image. Once complete, Docker image will be available on your Docker Hub repository.

5) How Others Can Access It (Pull and Run)

Once the image is on Docker Hub, anyone can pull it and run a container using the same tagged name.

Pull the Image:

On any machine with Docker installed, a user can download Docker image:

Bash
docker pull YOUR-USERNAME/ml-model-api:v1.0

Run the Container:

Others can then run the image, mapping the container's port (5000 in example) to a port on local host machine (e.g., 8080):

Bash
docker run -d -p 8080:5000 YOUR-USERNAME/ml-model-api:v1.0

Result: The machine learning API is now running on the user's host at http://localhost:8080 and access the prediction endpoint using the new port.

CONCLUSION

This paper examines Docker-based containerization as a foundational solution to reproducibility challenges in AI and machine learning research, tackling issues like "dependency hell," environment drift, and irreproducible outcomes amid growing model complexity. By encapsulating code, libraries, tools, and configurations in isolated, portable containers, Docker ensures deterministic execution through key principles: environment parity for consistent behavior across development, testing, and deployment; isolation to prevent conflicts; and portability for seamless use on local, cloud, or high-performance systems. A detailed Dockerfile analysis and practical workflow for training/deploying a machine learning model demonstrate how pinned versions, ordered instructions, metadata, and version control create transparent, reproducible environments, with best practices for model extraction, lifecycle management, and registry distribution enabling reliable sharing and replication. Ultimately, Docker enhances scientific integrity, productivity, and collaboration, positioning it as essential infrastructure for deterministic science. For future research, investigators could explore integrating Docker with emerging orchestration tools like Kubernetes or Ray for scaling distributed AI training across hybrid edge-cloud environments, while benchmarking reproducibility gains against alternative solutions like Conda or Podman in large-scale, multi-institutional studies.

REFERENCES

Boettiger, C. (n.d.). *An introduction to Docker for reproducible research*. https://gist.github.com/samth/9641364

Denes, G. (2023). A case study of using AI for General Certificate of Secondary Education (GCSE) grade prediction in a selective independent school in England. *Computers and Education: Artificial Intelligence*, *4*. https://doi.org/10.1016/j.caeai.2023.100129

Hagmann, M., Meier, P., & Riezler, S. (2023). Towards Inferential Reproducibility Of Machine Learning Research. *11th International Conference on Learning Representations, ICLR 2023*.

Hamel Husain. (2017, December 17). *How Docker Can Help You Become A More Effective Data Scientist*.

How Docker Accelerates ZEISS Microscopy's AI Journey. (n.d.).

Kazmierczak, R., Berthier, E., Frehse, G., & Franchi, G. (2025). Explainability and vision foundation models: A survey. *Information Fusion*, *122*. https://doi.org/10.1016/j.inffus.2025.103184

Kim, B. S., Lee, S. H., Lee, Y. R., Park, Y. H., & Jeong, J. (2022). Design and Implementation of Cloud Docker Application Architecture Based on Machine Learning in Container Management for Smart Manufacturing. *Applied Sciences (Switzerland)*, *12*(13). https://doi.org/10.3390/app12136737

Michal Sutter. (2025, August 13). *Why Docker Matters for Artificial Intelligence AI Stack: Reproducibility, Portability, and Environment Parity*.

Nurzhanov, A., & Sharipbay, A. (2025). Modern AI Models for Text Analysis: A Comparison of Chatgpt and Rag. *Journal of Data Analytics and Artificial Intelligence Applications*, *1*(1). https://doi.org/10.26650/d3ai.002

Nust, D., Sochat, V., Marwick, B., J. Englen, S., Head, T., Hirst, T., & D. Evans, B. (2020). Ten simple rules for writing Dockerfiles for reproducible data science. *PLOS Computational Biology*, *16*(11), 1–24. https://doi.org/10.1371/journal.pcbi.1008316

Owen, A., Joseph Ajeigbe, K., Owen, A., & Ajeigbe, K. (n.d.). *Containerization of Machine Learning Models*. https://www.researchgate.net/publication/390267646

Pineau, J., Vincent-Lamarre, P., Sinha, K., Larivière, V., Beygelzimer, A., D'Alché-Buc, F., Fox, E., & Larochelle, H. (2021). Improving Reproducibility in Machine Learning Research. *Journal of Machine Learning Research*, *22*(1).

Salman Anwaar. (2024, September 25). *Containerization Machine Learning Applications*.

Semmelrock, H., Ross-Hellauer, T., Kopeinik, S., Theiler, D., Haberl, A., Thalmann, S., & Kowald, D. (2025). Reproducibility in machine-learning-based research: Overview, barriers, and drivers. *AI Magazine*, *46*(2). https://doi.org/10.1002/aaai.70002

Shashank Prasanna. (2020, March 11). *Why use Docker containers for machine learning development?*

Stephanie Susnjara, & Ian Smalley. (n.d.). *What is Docker?*

Tatman, R., & Vanderplas, J. (2018). A Practical Taxonomy of Reproducibility for Machine Learning Research. *2nd Reproducibility in Machine Learning Workshop at ICML 2018, Stockholm, Sweden*, *Ml*.