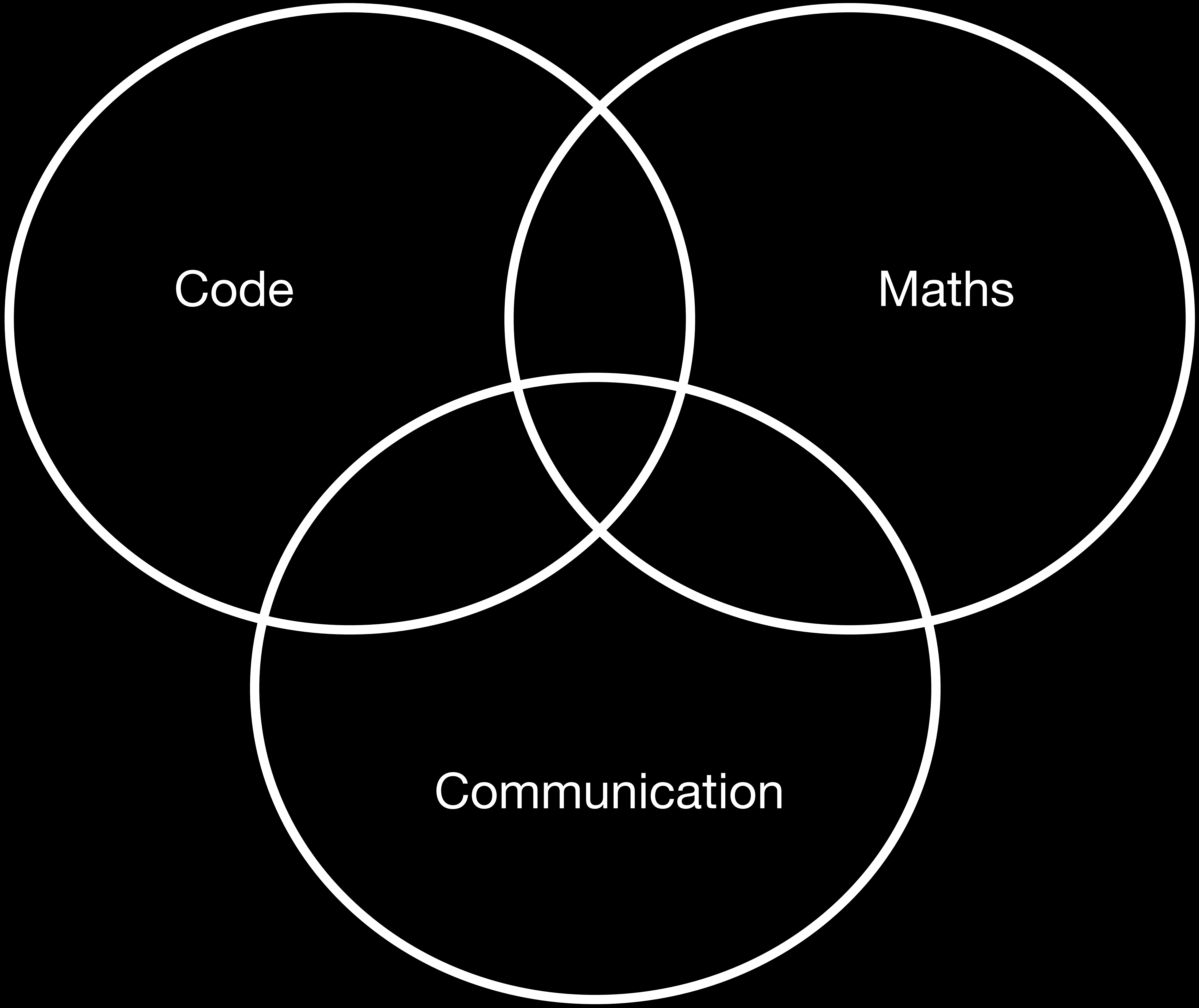
MADS - Deployment 1

It works on my machine

Raoul Grouls, 10 November 2025



Code

Maths

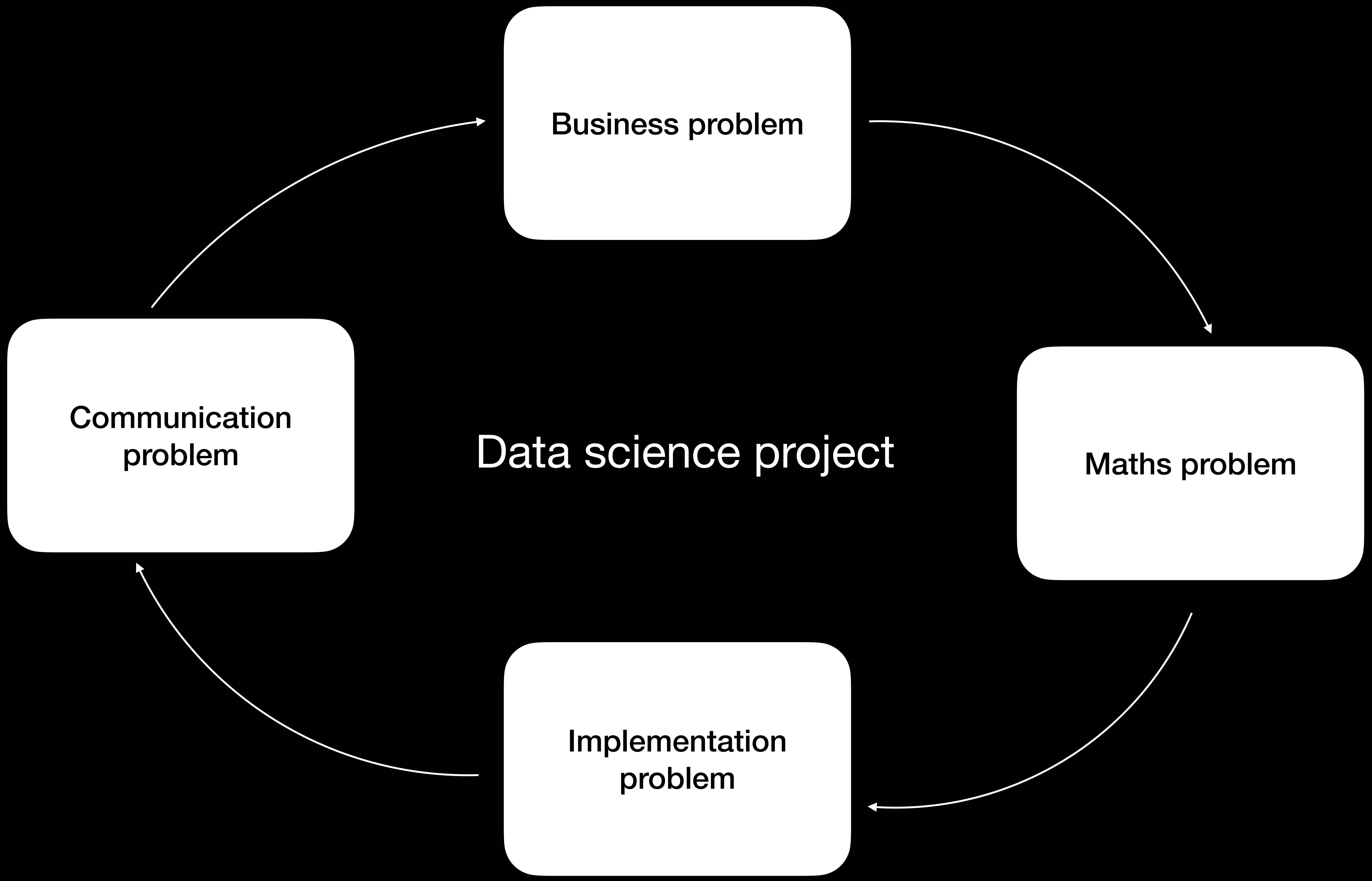
Communication

Different types of data scientists

“I understand the algorithms, let someone else worry about implementation”

Different types of data scientists

- Writing python code that is acceptable for deployment
- Make your algorithm work on other machines beside your own laptop
- Collaborate with colleagues that build a frontend
- Use code that is published by others as a docker container



Data science project

Business problem

Maths problem

Implementation problem

Communication problem

Docker for Data Scientists

Why containerization matters

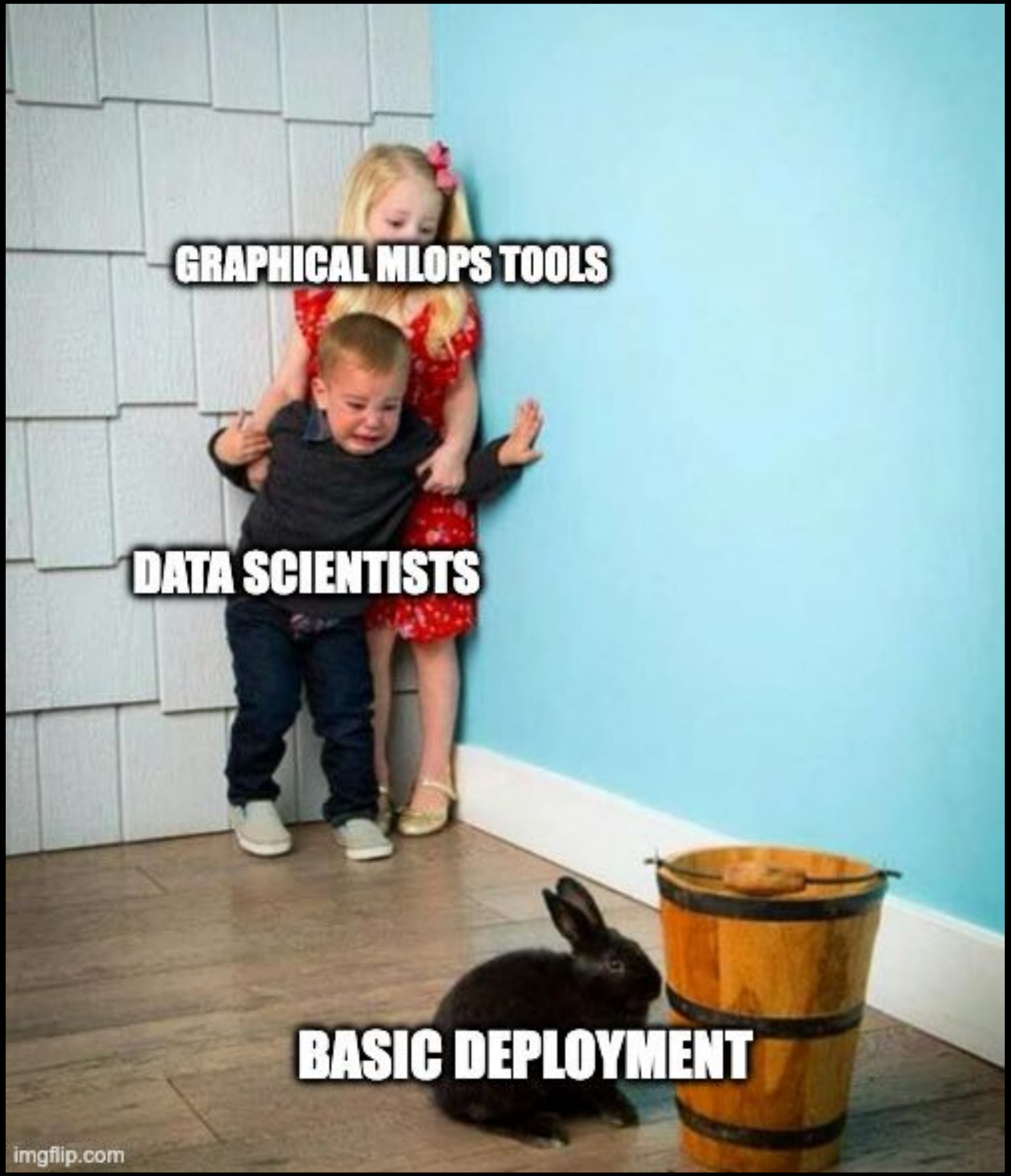
Science is about reproducibility

Docker for Data Scientists: Getting Started

Why containerization matters for ML/AI projects

Promise of reproducibility across different machines

- Same environment
- Same dependencies
- Same behavior
- Isolation of dependencies from your personal workflow



Docker for Data Scientists: Getting Started

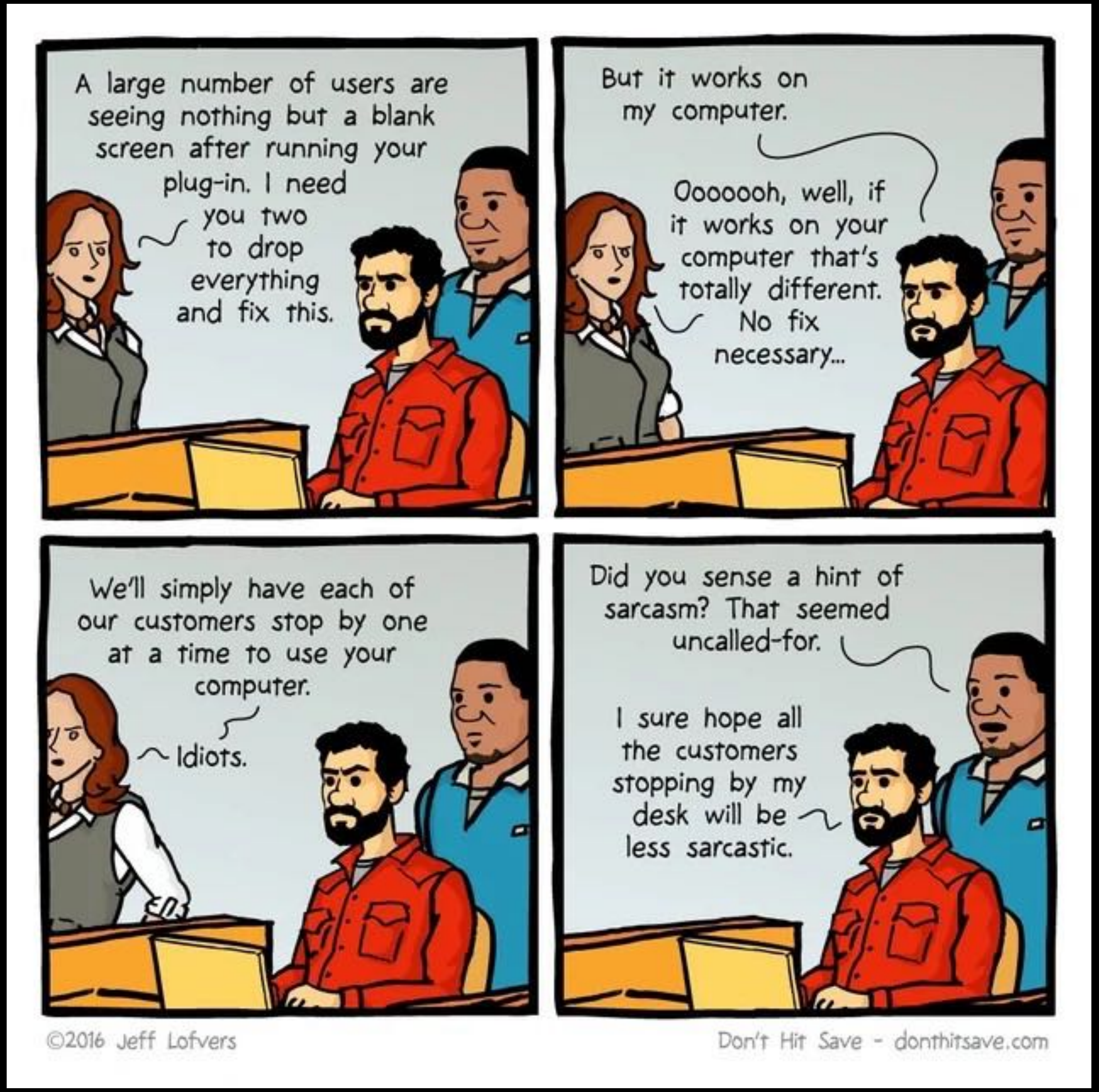
Why containerization matters for ML/AI projects

Easy sharing of complete project setups

- Less "it works on my machine”
- Fast setup on clean VMs
- Quick onboarding of new team members (if they have enough RAM for Docker…)

Simplified deployment process

- From local to production with minimal changes


A large number of users are seeing nothing but a blank screen after running your plug-in. I need you two to drop everything and fix this.
But it works on my computer.
Oooooh, well, if it works on your computer that's totally different. No fix necessary...
We'll simply have each of our customers stop by one at a time to use your computer.
Idiots.
Did you sense a hint of sarcasm? That seemed uncalled-for.
I sure hope all the customers stopping by my desk will be less sarcastic.

©2016 Jeff Lofvers

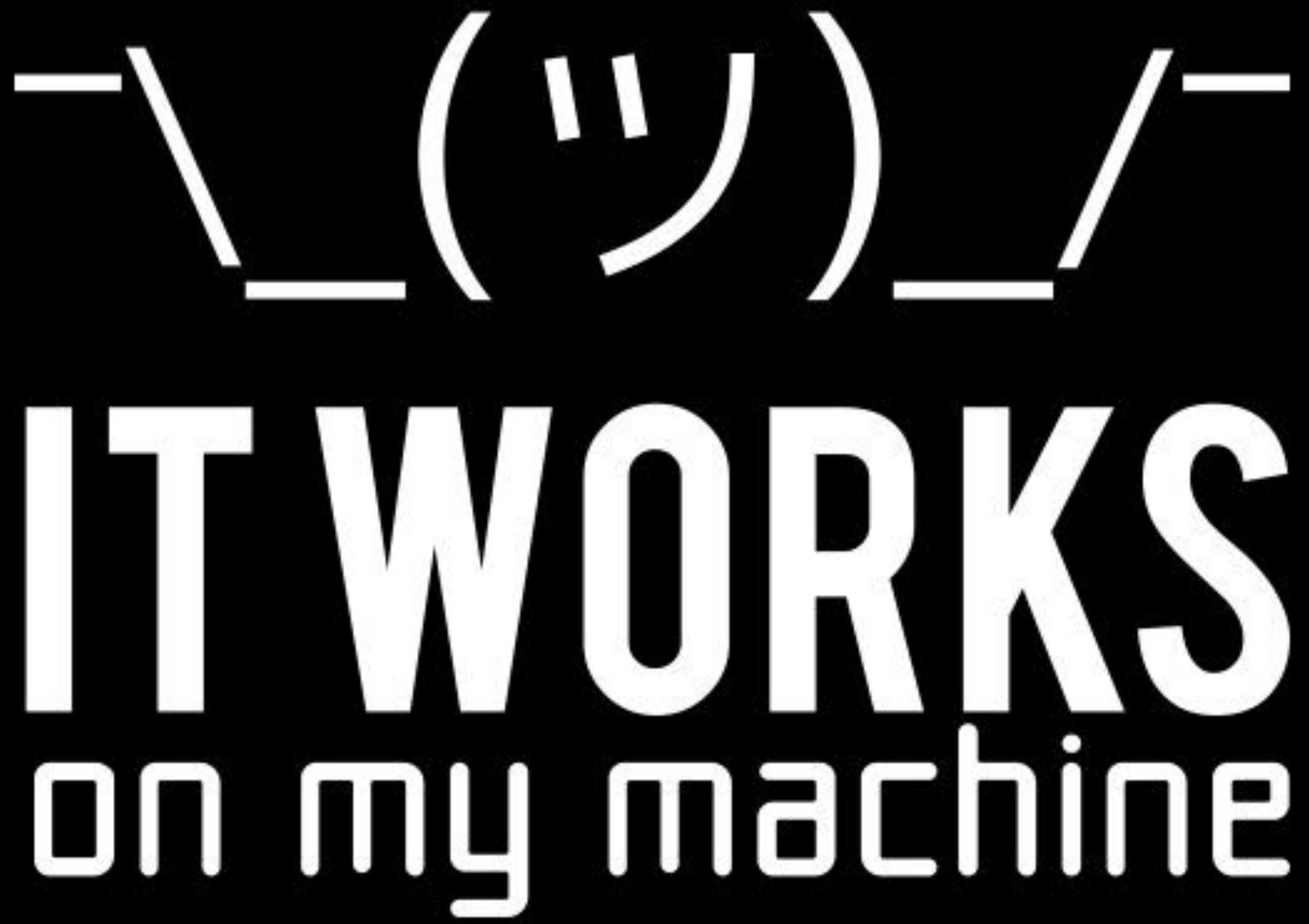
Don't Hit Save - donthitsave.com

The "Works on My Machine" Paradox

Docker doesn't completely solve architecture differences:

- CPU Architecture Challenges
  - x86_64 (Intel/AMD) vs ARM64 (M1/M2 Macs)
  - Some packages aren't available for all architectures
  - Performance can vary significantly

- Common Issues
  - TensorFlow/PyTorch optimizations differ
  - C++ extensions might not compile
  - Binary dependencies might be architecture-specific
- Possible Solutions
  - Use `--platform` flag when building
  - Build multi-architecture images
  - Document architecture requirements clearly



Cross-Platform Docker: Using Platform Flags

Force AMD64 build

- `docker build --platform linux/amd64 -t mymodel:latest .`

Force ARM64 build

- `docker build --platform linux/arm64 -t mymodel:latest .`

Run for a speccific platform

- `docker run --platform linux/amd64 my-image`

Docker Fundamentals: Key Concepts

Core Components:

- **Dockerfile :** The build instructions for creating an **Image**. Similar to a recipe with step-by-step instructions

- **Image :** The blueprint/template for **Containers**. Like a snapshot that can create identical containers

- **Container** : A running instance with your application and dependencies. Think of it as a lightweight VM that shares the host OS kernel

- **Registry :** A repository for storing and sharing images. Can be public (Docker Hub) or private

Docker Fundamentals: Key Concepts

Dockerfile -> builds -> **Image** -> runs -> **Container**

Understanding the Dockerfile

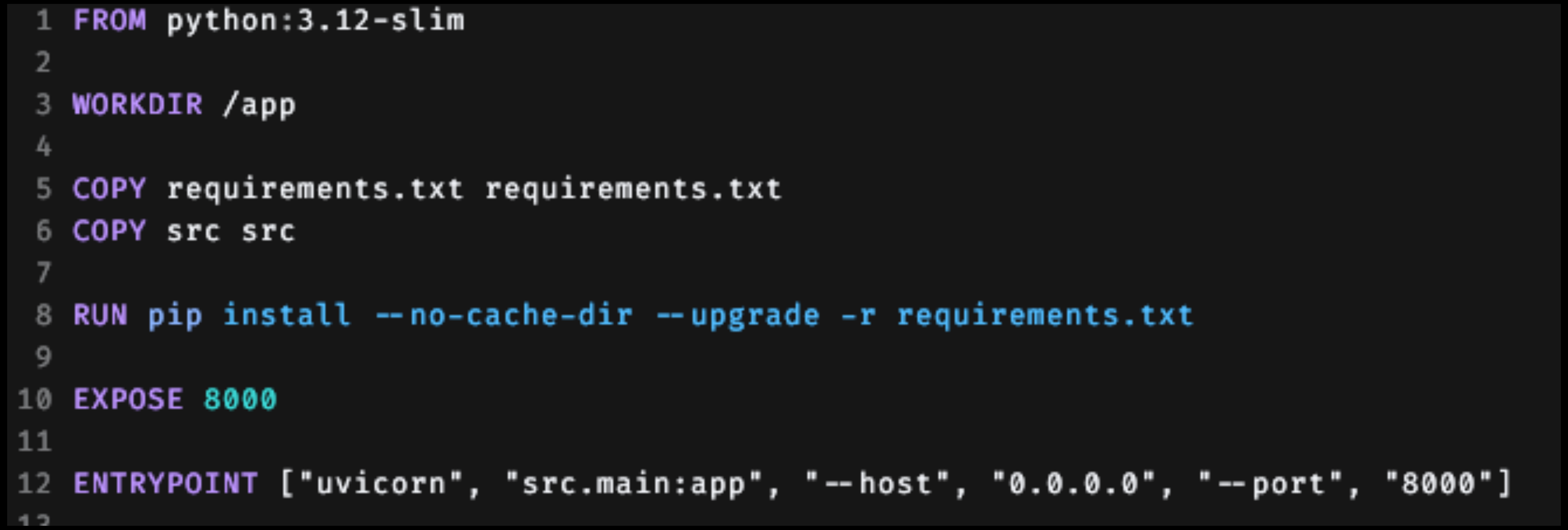
FROM

The command FROM:
Docker checks your local machine:

- Looks in local image cache
- If found, uses cached image
- If not found, needs to download

If not found locally:

- Connects to registry (default: Docker Hub)
- Downloads the image in layers
- Saves in local cache for future use



```dockerfile
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt requirements.txt
COPY src src

RUN pip install --no-cache-dir --upgrade -r requirements.txt

EXPOSE 8000

ENTRYPOINT ["uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Understanding the Dockerfile

FROM python:3.12-slim

There are multiple python images, you can find them on https://hub.docker.com/_/python/tags

- `-alpine`
  - Ultra minimal (50MB)
  - Based on alpine linux
  - Security focused
  - Harder to debug
- `-slim`
  - Minimal image (130MB)
  - Only essentials
  - Good for simple webapps like fastapi when size matters but alpine is too minimal
- `Python:3.x`
  - Full image (920MB)
  - Includes common system libraries
  - Good for complex dependencies
  - Larger deployment size
- `-slim-bullseye / -slim-bookworm`
  - Bullseye = Debian 11
  - Bookworn = Debian 12

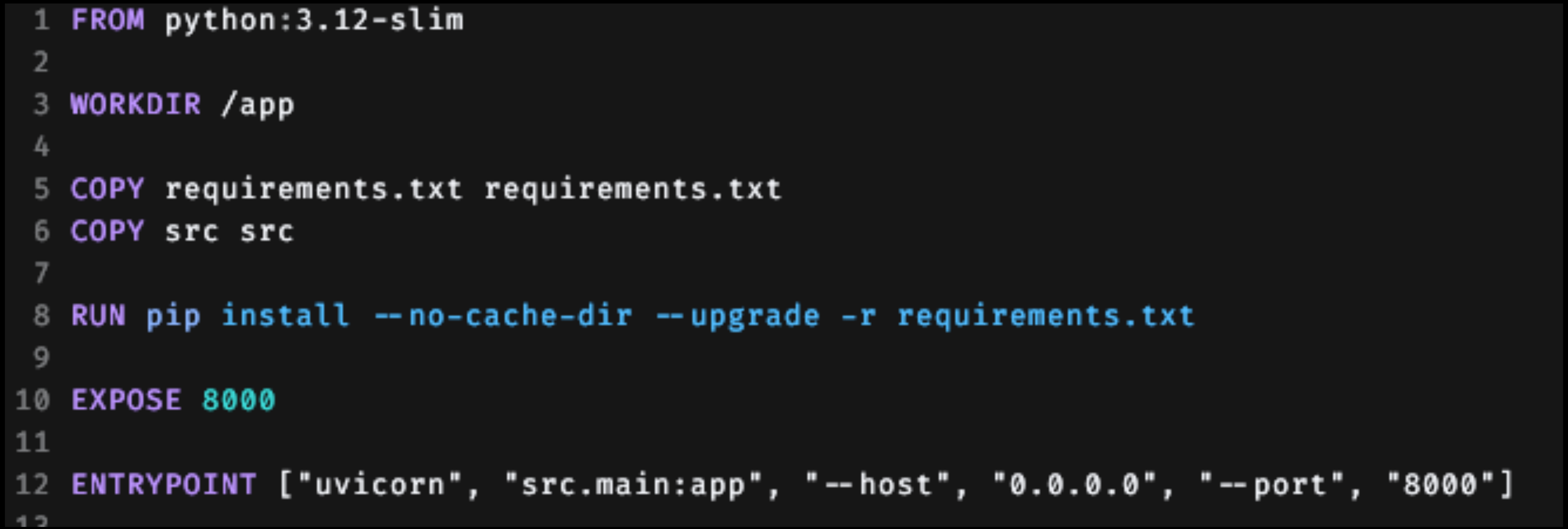
Prebuilt baseimages

- With the script found here https://github.com/raoulg/minimal-torch-docker
- I publish minimal docker containers on hub.docker
- https://hub.docker.com/repository/docker/raoulgrouls/torch-python-slim/general
- You can find small prebuilt containers for arm64/amd64, python-slim 3.11/3.12, PyTorch 2.3-2.7 and with/without uv
- `FROM raoulgrouls/torch-python-slim:py3.12-torch2.7.1-arm64-uv0.8.13`

Understanding the Dockerfile

WORKDIR /app

- Sets the working directory
- All subsequent commands run here
- Good practice for organization

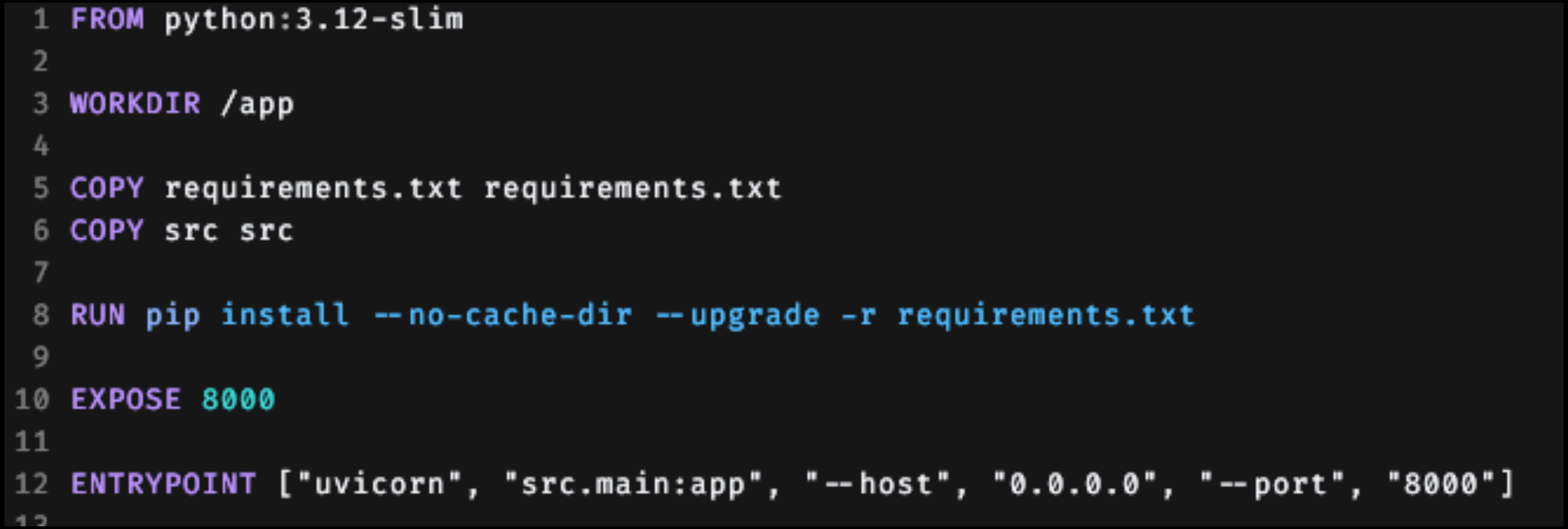


```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt requirements.txt
COPY src src

RUN pip install --no-cache-dir --upgrade -r requirements.txt

EXPOSE 8000

ENTRYPOINT ["uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Understanding the Dockerfile

COPY

- Copies files from host to container
- Best practice: Copy requirements first
- Helps with layer caching



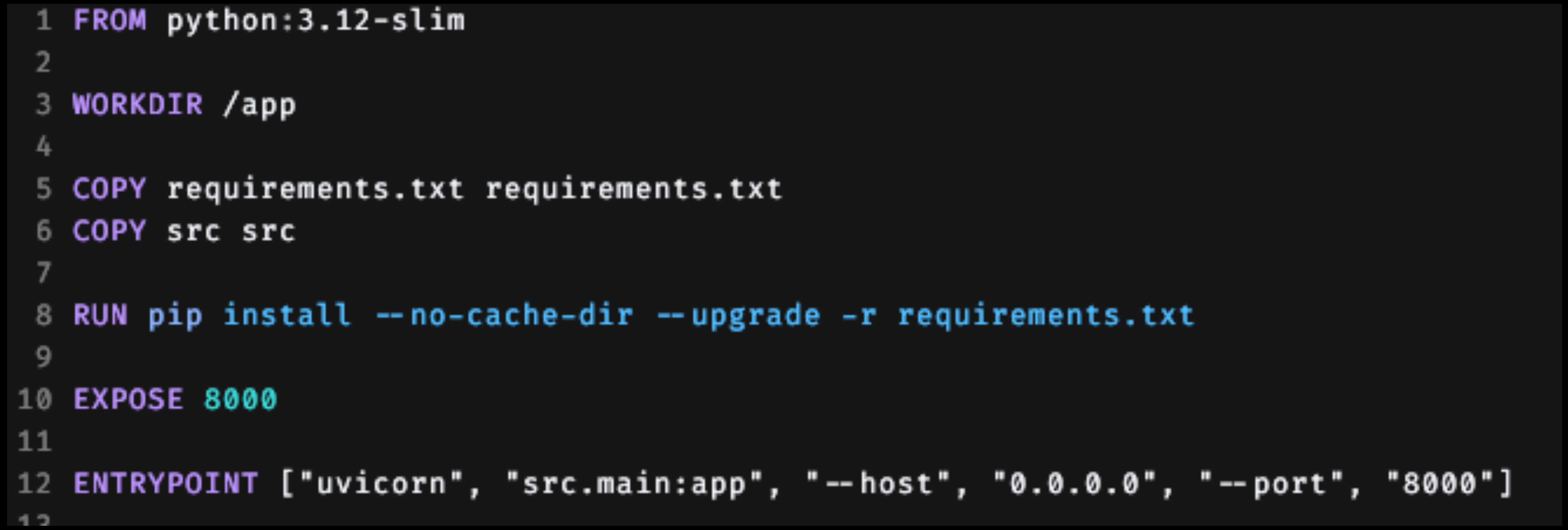
```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt requirements.txt
COPY src src

RUN pip install --no-cache-dir --upgrade -r requirements.txt

EXPOSE 8000

ENTRYPOINT ["uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Understanding the Dockerfile

RUN

```
RUN pip install -r requirements.txt
```

- Executes commands during build
- Creates a new layer in the image
- Use && for multiple commands to reduce layers

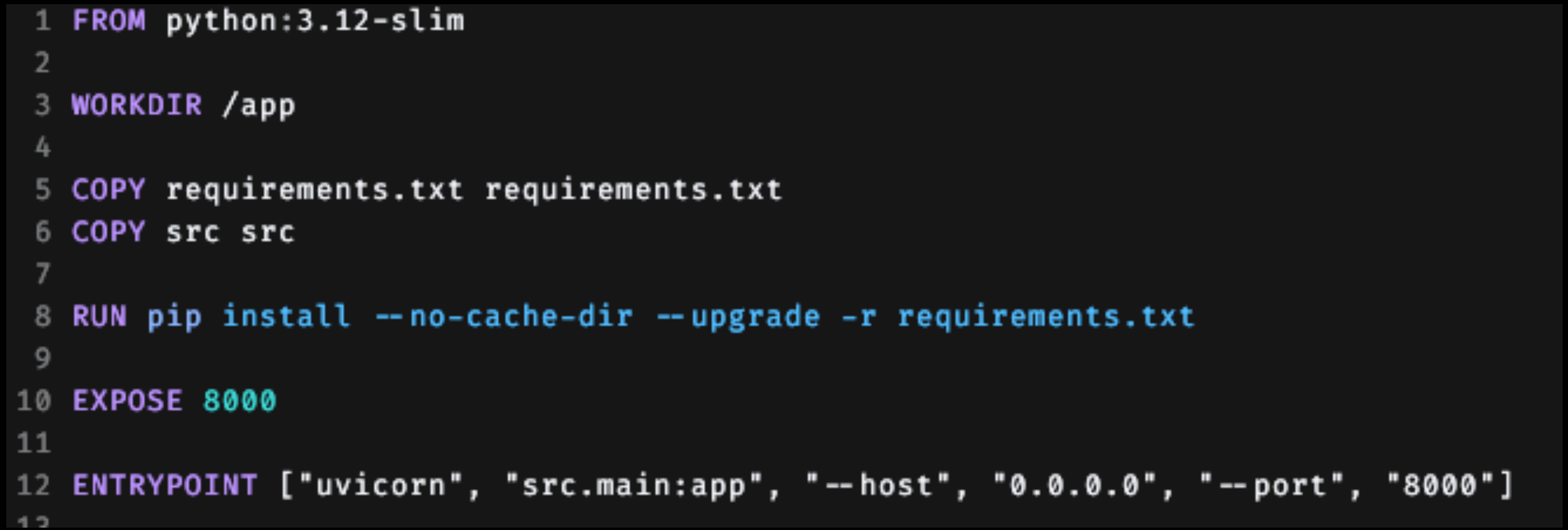


```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt requirements.txt
COPY src src

RUN pip install --no-cache-dir --upgrade -r requirements.txt

EXPOSE 8000

ENTRYPOINT ["uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Understanding the Dockerfile

Layers

- Docker caches layers, such that they dont need to rebuild every time you run `build`
- Once docker detects that a layer has been changed, it will rebuild that layer, and all layers that come next
- In this example, once you change the requirements file, all subsequent layers will rebuild
- How would you improve this example?



```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt requirements.txt
COPY src src

RUN pip install --no-cache-dir --upgrade -r requirements.txt

EXPOSE 8000

ENTRYPOINT ["uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

CMD vs ENTRYPOINT

Making Your Container Executable

CMD:

- Can be easily overridden from command line
- If you provide runtime arguments, entire CMD is replaced
- Good for default behavior that might change

ENTRYPOINT:

- Runtime arguments are added to ENTRYPOINT arguments
- Harder to override (needs --entrypoint flag)
- Good for containers that always run the same command

CMD vs ENTRYPOINT

Making Your Container Executable

```
# Dockerfile

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



# Runtime - completely replaces CMD

docker run myimage echo "hello"  # runs: echo "hello"
```

CMD vs ENTRYPOINT

Making Your Container Executable

```
# Dockerfile
ENTRYPOINT ["python"]


# Runtime - adds to ENTRYPOINT
docker run myimage app.py  # runs: python app.py
docker run myimage -V      # runs: python -V
```

CMD vs ENTRYPOINT

Making Your Container Executable

Best Practices:

- Use `ENTRYPOINT` for "main executable” that should always run
- Use `CMD` for arguments that might change
- Always use JSON array format ["command", "arg"]

```
ENTRYPOINT ["uvicorn"]

CMD ["main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Interactive Docker

Great for debugging

```
# Only overrides the CMD
docker run -it my-image /bin/bash

# for Alpine-based images
docker run -it my-image /bin/sh

# Ignores both ENTRYPOINT and CMD
docker run -it --entrypoint /bin/bash my-image
```

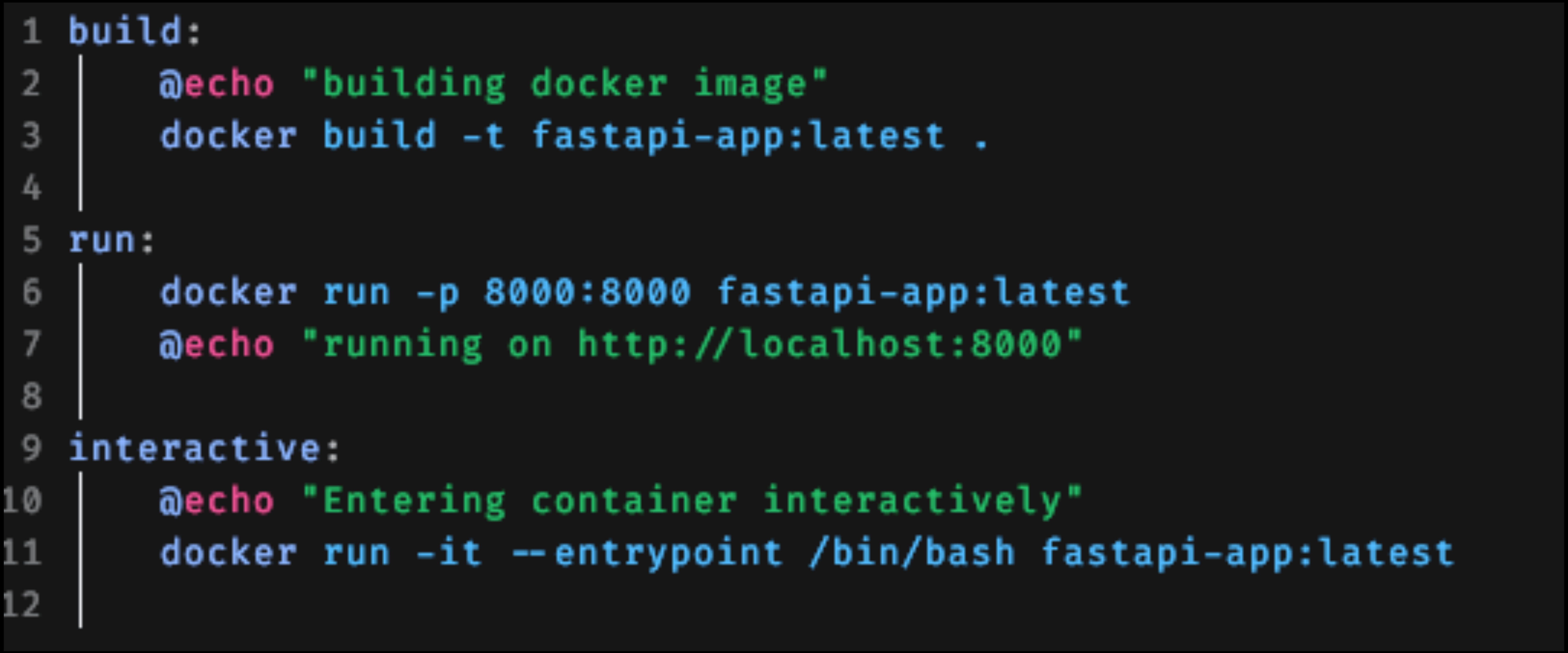
Interactive Docker

Great for debugging

```
# Check installed Python packages
pip list

# Test imports
python
>>> import numpy

# Check environment variables
env

# Check file system
ls -la
cat requirements.txt
```

Makefile

Using Makefile to automate the docker commands



```makefile
build:
	@echo "building docker image"
	docker build -t fastapi-app:latest .

run:
	docker run -p 8000:8000 fastapi-app:latest
	@echo "running on http://localhost:8000"

interactive:
	@echo "Entering container interactively"
	docker run -it --entrypoint /bin/bash fastapi-app:latest

```