My Journey Using Docker as a Development Tool:

From Zero to Hero

by Haseeb Majid

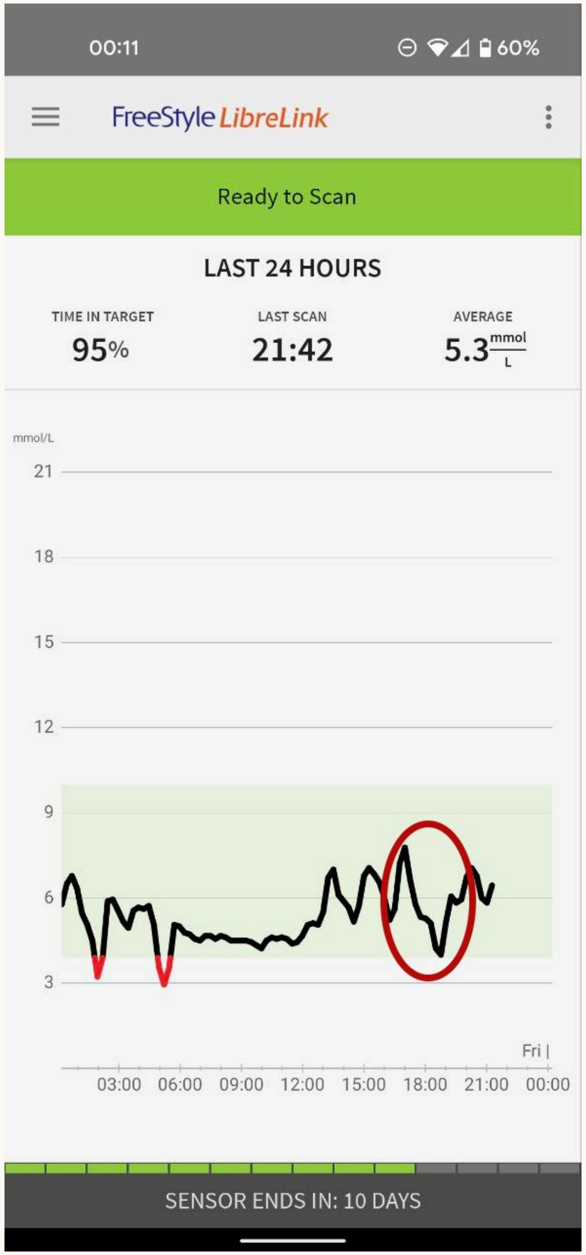
About Me

- Haseeb Majid
  - A software engineer
  - https://haseebmajid.dev
- Loves cats 🐱
- Avid cricketer 🏏 #BazBall

ZOE

- I work for ZOE 🥑
  - https://joinzoe.com
  - Personalised nutrition product
  - Health study

My Blood Sugar Levels


00:11 60%

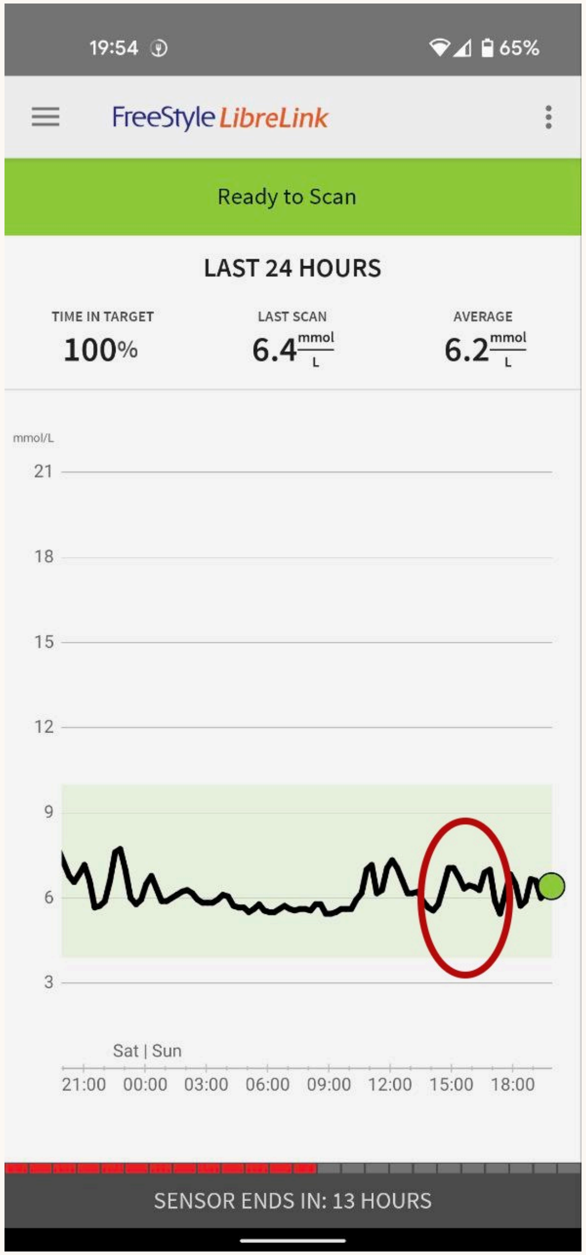
FreeStyle LibreLink

Ready to Scan

LAST 24 HOURS

TIME IN TARGET	LAST SCAN	AVERAGE
95%	21:42	5.3 mmol/L

SENSOR ENDS IN: 10 DAYS


19:54 65%

FreeStyle LibreLink

Ready to Scan

LAST 24 HOURS

TIME IN TARGET	LAST SCAN	AVERAGE
100%	6.4 mmol/L	6.2 mmol/L

SENSOR ENDS IN: 13 HOURS

Who Is This Talk For?

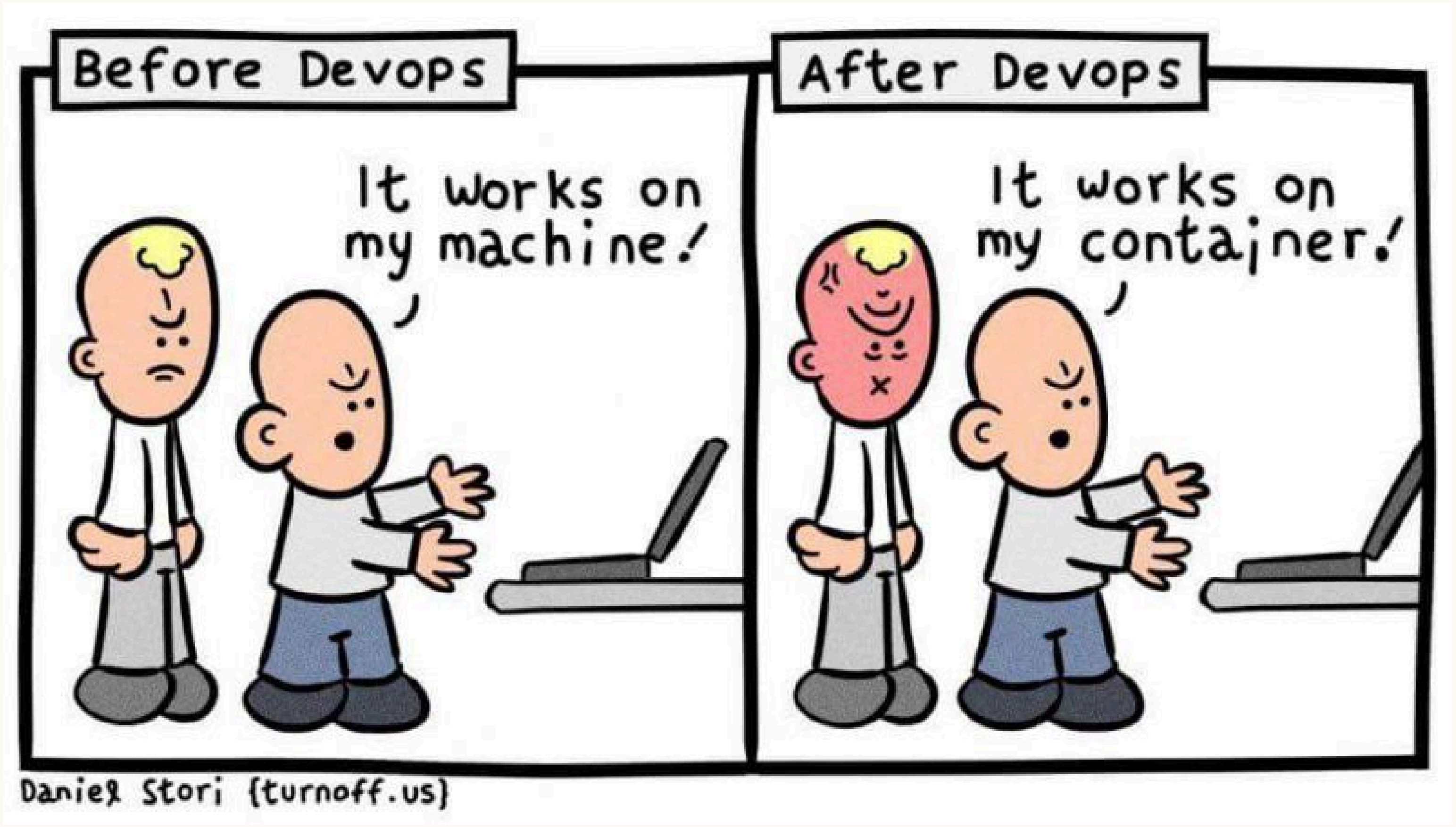
- Have used Docker
    - But not an expert
- Know basic CLI commands
- Want to use Docker in CI

Example Code

- Simple FastAPI web-service
  - Interacts with DB
- It allows us to get and add new users
- Poetry for dependency management

Why Docker?

- Reproducible builds
    - Easy setup for developers
    - OS independent


Before Devops
It works on my machine!
After Devops
It works on my container!
Daniel Stori {turnoff.us}

My First Image

```
# Dockerfile

FROM python:3.9.8

ENV PYTHONUNBUFFERED=1 \
  PYTHONDONTWRITEBYTECODE=1 \
  PYTHONPATH="/app" \
  PIP_NO_CACHE_DIR=off \
  PIP_DISABLE_PIP_VERSION_CHECK=on \
  PIP_DEFAULT_TIMEOUT=100 \
  \
  POETRY_VERSION=1.1.11 \
```

Let’s Run It

```
docker build --tag app .
docker run --publish 80:80 app

# Access app on http://localhost
```

One container is not enough
We need to go deeper
imgflip.com

App Dependencies

- App depends on a database
  - Dockerise it

Without Docker

```
sudo apt update
sudo apt install postgresql postgresql-contrib
sudo systemctl start postgresql.service

sudo -u postgres createuser --interactive
sudo -u postgres createdb test
```

With Docker

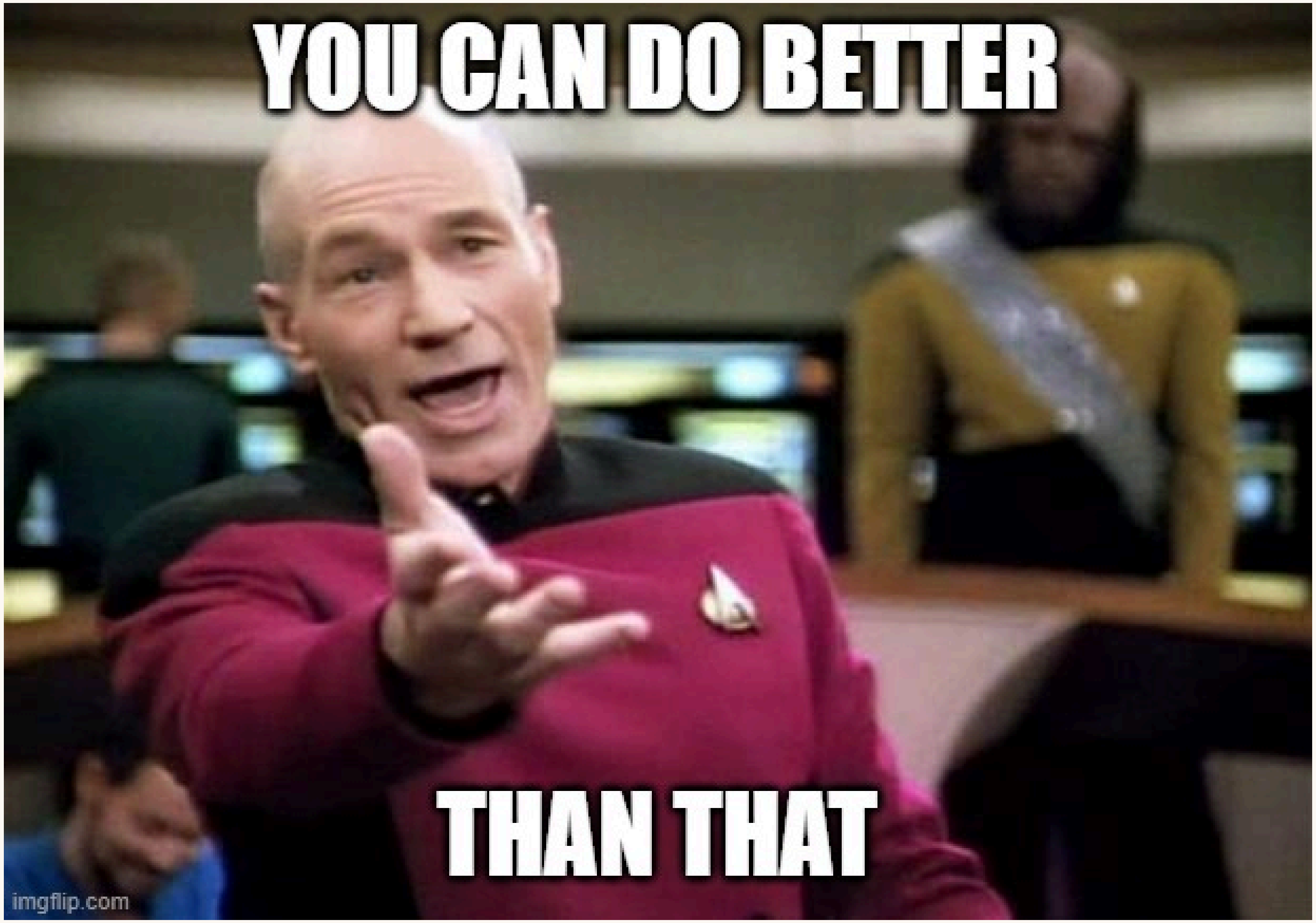
```
docker run --volume "postgres_data:/var/lib/postgre
--environment "POSTGRES_DATABASE=postgres" \
--environment "POSTGRES_PASSWORD=postgres" \
--publish "5432:5432" \
  postgres:13.4
```

```
# Start Commands:
docker network create --driver bridge workspace_ne
docker volume create  postgres_data
docker build -t app .
docker run --environment "POSTGRES_USER=postgres"
  --environment "POSTGRES_HOST=postgres" \
  --environment "POSTGRES_DATABASE=postgres" \
  --environment "POSTGRES_PASSWORD=postgres" \
  --environment "POSTGRES_PORT=5432" \
  --volume "./:/app" --publish "80:8080" \
  --network workspace_network --name workspace_app
  --detach app
docker run --volume "postgres_data:/var/lib/postgr
--environment "POSTGRES_DATABASE=postgres" \
--environment "POSTGRES_PASSWORD=postgres" \
--publish "5432:5432" --network workspace_network
--name workspace_postgres --detach postgres:13.4

```

```
  --environment "POSTGRES_DATABASE=postgres" \
  --environment "POSTGRES_PASSWORD=postgres" \
  --environment "POSTGRES_PORT=5432" \
  --volume "./:/app" --publish "80:8080" \
  --network workspace_network --name workspace_app
  --detach app
docker run --volume "postgres_data:/var/lib/postgr
--environment "POSTGRES_DATABASE=postgres" \
--environment "POSTGRES_PASSWORD=postgres" \
--publish "5432:5432" --network workspace_network
--name workspace_postgres --detach postgres:13.4

# Delete Commands:
docker stop workspace_app
docker rm workspace_app
docker stop workspace_postgres
docker rm workspace_postgres
docker network rm workspace_network
```


YOU CAN DO BETTER
THAN THAT
imgflip.com

Docker Compose

- Manage multiple Docker containers
- Existing tool docker-compose
  - V2 called docker compose
- Use docker compose today

```yaml
# docker-compose.yml

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    command: bash /app/start.sh --reload
    volumes:
      - ./:/app
    environment:
```

Run It!!!

```
docker compose up --build
```

```
docker compose down
```

Summary

- Dockerise your app
- Dockerise dependencies (DB)
- Use docker compose
  - Manage multiple containers

WHAT IF I TOLD YOU
WE CAN DO BETTER
imgflip.com

Running Tests

- Run tests in Docker
  - pytest runner
- Consistent environment

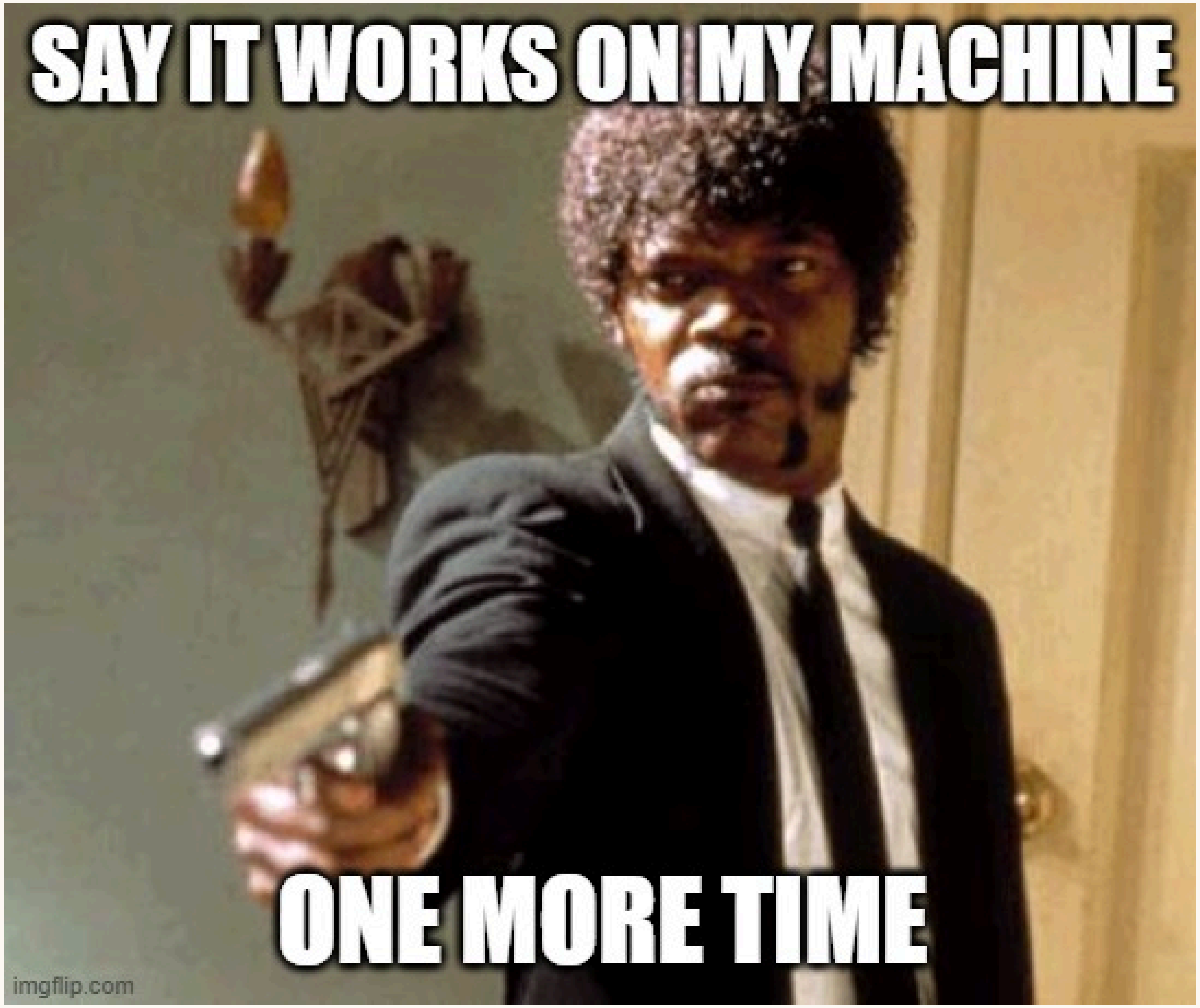
```
docker compose run app pytest
```

```yaml
# docker-compose.yml

services:
 app:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - postgres
    # ...
```

CI Pipeline

- Docker running locally
- Can we use Docker in CI?


SAY IT WORKS ON MY MACHINE
ONE MORE TIME
imgflip.com

Before

```yaml
# .github/workflows/branch.yml

name: Check changes on branch

on:
  push:
    branches:
      - "*"
      - "!main"

jobs:
```

After

```yaml
# .github/workflows/branch.yml

name: Check changes on branch

#...

jobs:
  test:
    runs-on: ubuntu-latest
    timeout-minutes: 5
    steps:
```

Summary

- Dockerise development tasks
  - Tests
  - Linting
  - DB migrations
- Use Docker on CI
  - Local environment = CI enviromnent

NOPE
WE CAN DO BETTER
imgflip.com

Smaller Image

- Remove redundant dependencies
  - Fewer security vectors
- Less storage

```dockerfile
# Dockerfile

FROM python:3.9.8-slim

# ...

WORKDIR $PYSETUP_PATH
COPY pyproject.toml poetry.lock ./

RUN pip install poetry==$POETRY_VERSION && \
  poetry install
```

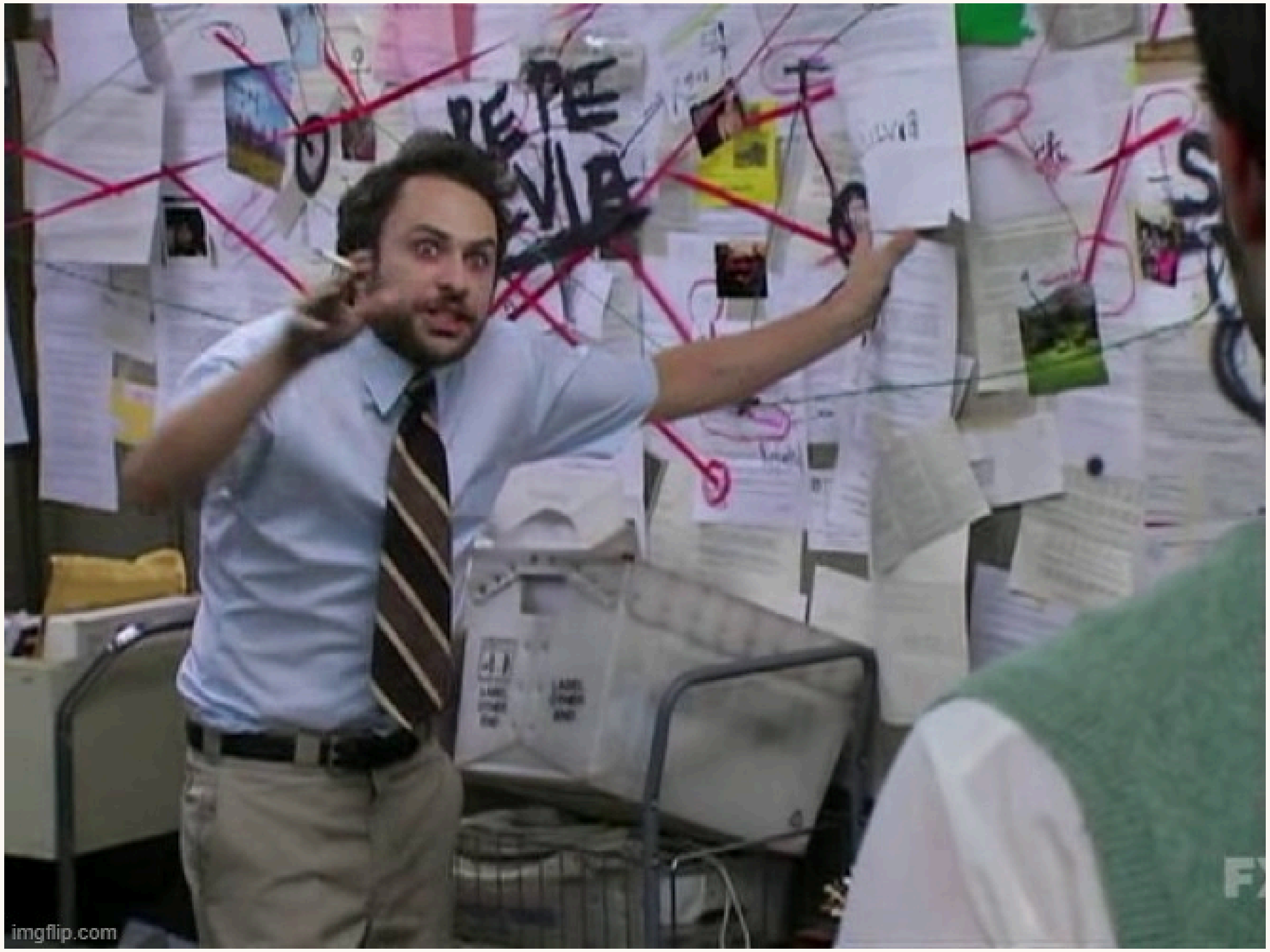
Comparison

	python:3.9.8	**python:3.9.8-slim**
Size	1 GB	280 MB
Build[1]	75 sec	30 sec
CI Pipeline Job	2 min 40 sec	1 min 57 sec

[1] No Cache

Summary

- Aim to use smaller base images
- Remove unnecessary dependencies
- Reduce build time


PEPE
SILVIA
imgflip.com
FX

Dependencies

- Dev dependencies in Docker image
    - Don’t need pytest in prod

Multistage Builds

1 Build Stage

```
FROM python:3.9.8 as builder
RUN make build-app
```

Stage Alias

Build Artefacts

2 Main Stage

```
FROM python:3.9.8-slim as main
COPY --from=builder /app/build /app/build
```

Stage Alias

```
# Dockerfile

FROM python:3.9.8-slim as base

ARG PYSETUP_PATH
ENV PYTHONPATH="/app"
ENV PIP_NO_CACHE_DIR=off \
  PIP_DISABLE_PIP_VERSION_CHECK=on \
  PIP_DEFAULT_TIMEOUT=100 \
  \
  POETRY_VERSION=1.1.11 \
```

```yaml
# docker-compose.yml

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
      target: development
    command: bash /app/start.sh --reload
    depends_on:
      - postgres
```

Comparison

	python:3.9.8-slim	Multistage[2]
Size	280 MB	200 MB
Build[1]	30 Seconds	35 seconds

[1] No Cache

[2] Building for production target

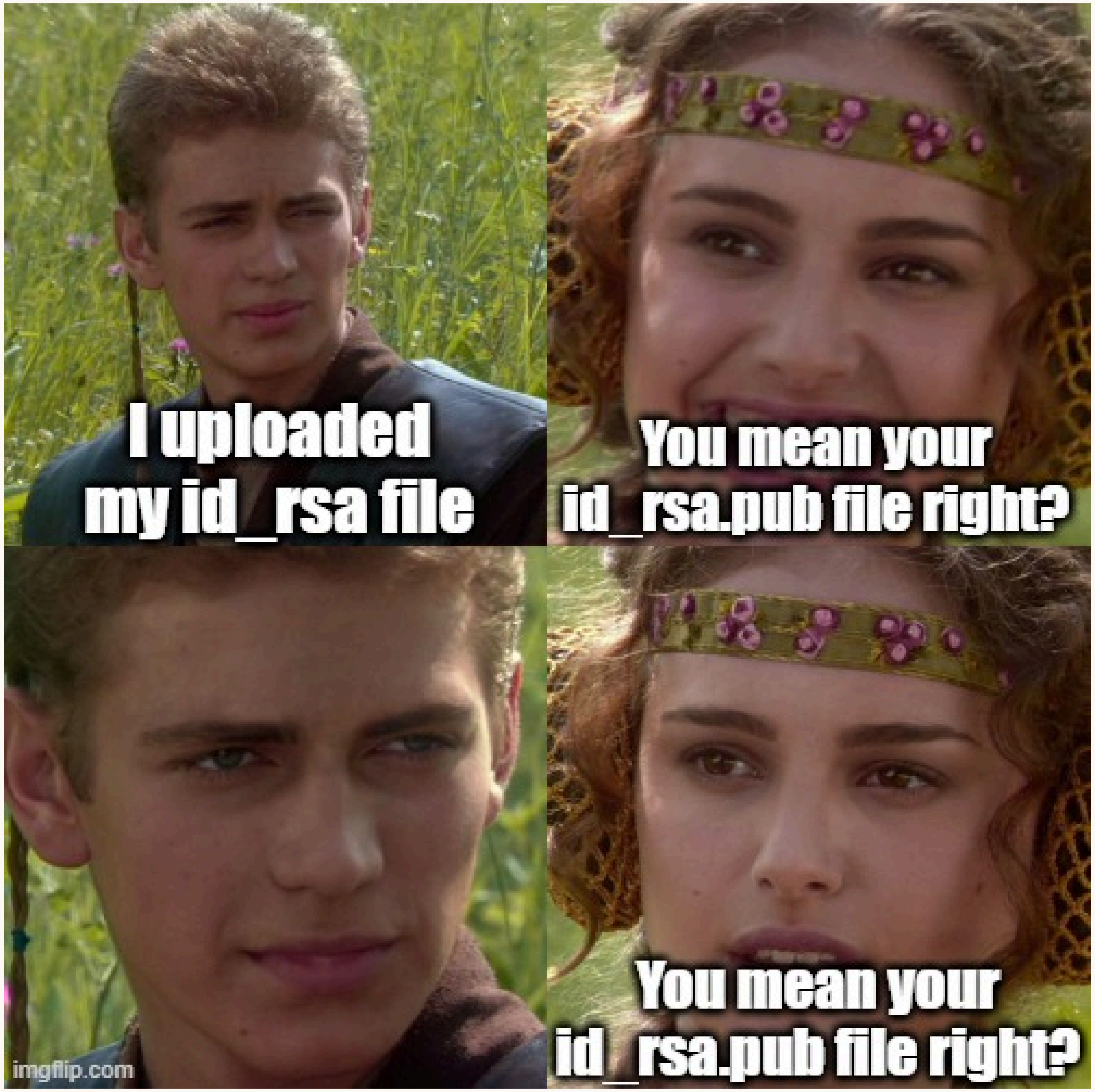
Cache From

```
# docker-compose.yml

services:
  app:
    build:
      context: .
      target: development
      cache_from:
        - registry.gitlab.com/haseeb-slides/developin
    command: bash /app/start.sh --reload
    # ....
```

Private Deps

- Private git repository
- Inject an SSH key
  - At build time


I uploaded my id_rsa file
You mean your id_rsa.pub file right?
You mean your id_rsa.pub file right?
imgflip.com

```
poetry add git+ssh@github.com:zoe/pubsub.git
```

```
  [tool.poetry.dependencies]
  python = "^3.9"
  fastapi = "^0.70.0"
  pubsub = { git = "ssh://git@github.com:zoe/pubsub.g
              rev = "0.2.5" }
  psycopg2-binary = "^2.9.3"
  SQLAlchemy = "^1.4.36"
  uvicorn = "^0.17.6"
```

```
FROM base as builder

RUN apt-get update && \
    apt-get install openssh-client git -y && \
    mkdir -p -m 0600 \
    ~/.ssh && ssh-keyscan github.com >> ~/.ssh/known_
    pip install poetry==$POETRY_VERSION

WORKDIR $PYSETUP_PATH
COPY poetry.lock pyproject.toml ./
```

First add our ssh key

```
ssh-add ~/.ssh/id_rsa
```

Then we can do

```
docker compose build --ssh default
```

CI Changes

```yaml
# .github/workflows/branch.yml

jobs:
  # ...
  test:
    # ...
    steps:
      - uses: actions/checkout@v3
      - uses: webfactory/ssh-agent@v0.5.4
        with:
          ssh-private-key: ${{ secrets.PRIVATE_SSH
      - name: Build Image
        run: docker compose build --ssh default
      - name: Run Tests
        run: docker compose run app pytest
```

CI Changes

```yaml
# .github/workflows/branch.yml

jobs:
  # ...
  test:
    # ...
    steps:
      - uses: actions/checkout@v3
      - uses: webfactory/ssh-agent@v0.5.4
        with:
          ssh-private-key: ${{ secrets.PRIVATE_SSH
      - name: Build Image
        run: docker compose build --ssh default
      - name: Run Tests
        run: docker compose run app pytest
```

Comparison

	python:3.9.8-slim[2]	Multistage[3]
Size	400 MB	200 MB
Build[1]	39 Seconds	46 seconds

[1] No Cache

[2] Assuming there was no multistage build

[3] Building for production target

Summary

- Use multistage builds
  - Slimmer production images
- Leverage SSH injection
  - During build time

What Did We Do?

- Dockerised app/deps
- Used docker compose
- Used Docker for dev tasks
- Multistage builds

Even Better

- Common base image
- Makefile
- Devcontainer in VSCode
- Docker Python interpreter in Pycharm

Any Questions?

- Code: https://gitlab.com/hmajid2301/talks/docker-as-a-dev-tool
- Slides: https://docker-as-a-dev-tool.haseebmajid.dev/

Extra Reading

- Breaking Down Docker by Nawaz Siddiqui
- Announcing Compose V2 General Availability
- Caching Docker layers on serverless build hosts with multi-stage builds
- Using Alpine can make Python Docker builds 50× slower

Useful Tools

- Dive
- Anchore image scan

Appendix