Optimizing Cache Usage in Docker Builds

BY IJEOMA ETI

About Me:

- Software Engineer.
- Passionate about sharing knowledge through writing.
- Active contributor to the community through open-source projects.

Find me on:

- LinkedIn : https://www.linkedin.com/in/ijeoma-eti
- X (formerly Twitter): https://x.com/EtiIjeoma
- GitHub: http://github.com/Aijeyomah

Introduction

Why Do Docker Builds Feel Slow?

➥ Even minor code changes can trigger full rebuilds, significantly increasing build times.

➥ Without optimization, unnecessary steps are repeated, leading to wasteful compute usage.

➥ In CI/CD environments, long build times delay testing, deployment, and impact overall developer productivity.

Importance of Build Optimization

➥ Speeds up development: Faster builds lead to shorter feedback loops, improving productivity.

➥ Reduces compute costs: Avoiding redundant processing conserves resources, reducing infrastructure expenses.

➥ Improves CI/CD performance: Optimized caching ensures that pipelines run efficiently, enabling faster releases.

Understanding Docker Builds

How Docker Builds Work

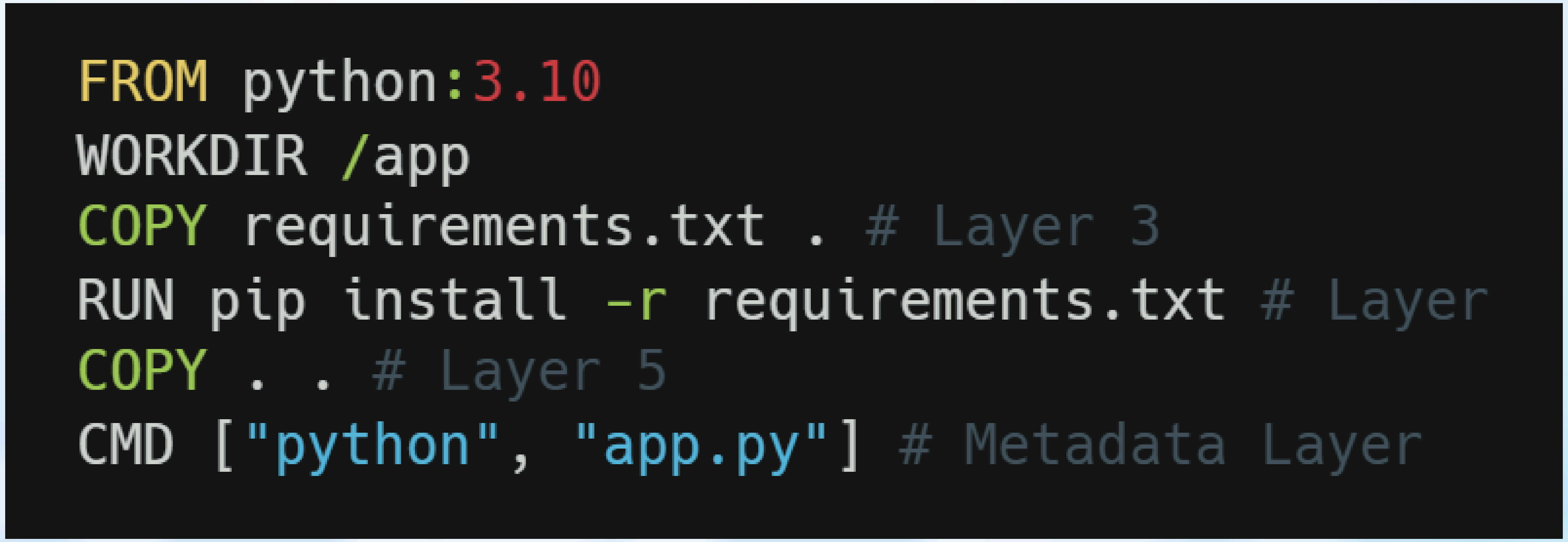
➢ Loads Build Context (All files in the directory)

➢Parses the Dockerfile, executing each instruction

➢Creates Immutable Layers for each step

➢Uses Caching to speed up rebuilds

Example Dockerfile Layers:



```
FROM python:3.10
WORKDIR /app
COPY requirements.txt . # Layer 3
RUN pip install -r requirements.txt # Layer
COPY . . # Layer 5
CMD ["python", "app.py"] # Metadata Layer
```

How Docker Caching Works

What is Docker Cache?

- Saves previously built image layers
- Speeds up builds by **reusing unchanged steps**

Key Concept: Layered Caching

Step	Instruction	Cacheable?	Explanation
1	FROM python:3.10	Yes	Base image is cached
2	WORKDIR /app	Yes	Doesn’t change often
3	COPY requirements.txt .	Yes	Cached if unchanged
4	RUN pip install -r requirements.txt	Yes	Cached if dependencies don’t change
5	COPY . .	No	Breaks cache if any file changes

The Problem – Why Docker Builds Become Inefficient

❶ **Common Reasons for Slow Builds**

➋ **Unnecessary Cache Busting**

➌ **Poor Dockerfile Structure**

➍ **Changing Dependencies Too Often**

➎ **Inefficient Use of COPY and ADD**

➏ **Ignoring Build Context Best Practices**

➐ **Large Image Sizes**

How do we fix these? → Let’s analyze each.

Common Pitfalls That Break Docker Caching

Mistake	Why It’s Bad?	Fix
Wrong Order of Instructions	Changes in early steps invalidate cache	Move frequently changing steps to the end
Using COPY . .	Copies unnecessary files, breaking cache	Use .dockerignore and copy files explicitly
Running apt update Without Pinning Versions	Fetches new package lists, breaking cache	Pin package versions and remove unnecessary files
Using ADD Instead of COPY	Unnecessary file extraction causes cache invalidation	Use COPY unless extracting archives
Using Wildcards (*)	Any change in directory invalidates cache	Be explicit about copied files

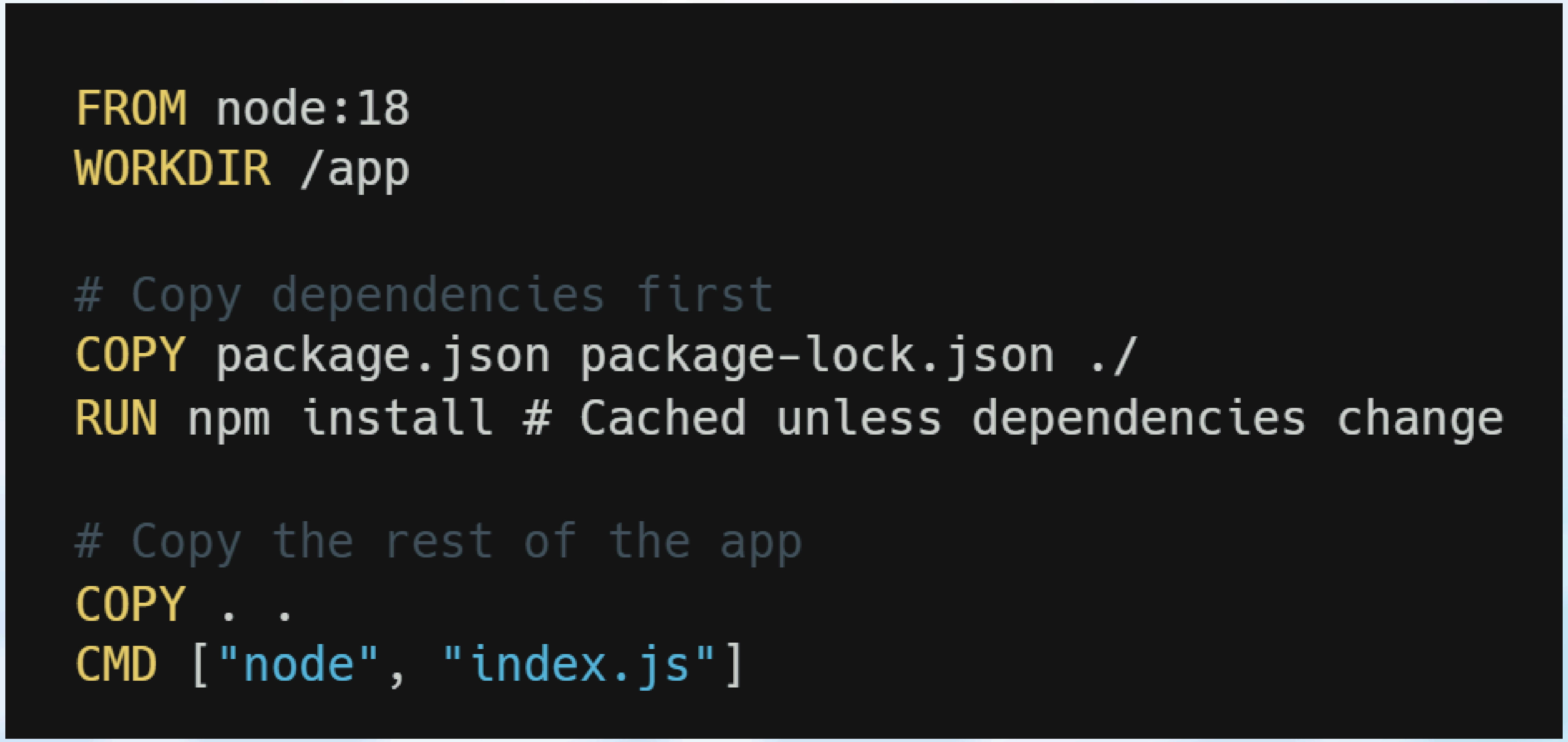
Best Practices for Optimizing Docker Builds Using Cache

- ➤ Structuring Dockerfiles for Maximum Cache Reuse
- ➤ Placing Stable Instructions Before Frequently Changing Ones
- ➤ Using Multi-Stage Builds to Reduce Final Image Size
- ➤ Leveraging .dockerignore to Reduce Build Context Size

Best Practices for Optimizing Docker Builds Using Cache

- **Structuring Dockerfiles for Maximum Cache Reuse**
- **Placing Stable Instructions Before Frequently Changing Ones**
- **Using Multi-Stage Builds to Reduce Final Image Size**
- **Leveraging .dockerignore to Reduce Build Context Size**
- **Using Arguments (ARG) vs. Environment Variables (ENV) in Docker Builds**
- **Selecting the Right Base Image to Improve Build Performance**

Example **Optimized Dockerfile:**



```
FROM node:18
WORKDIR /app

# Copy dependencies first
COPY package.json package-lock.json ./
RUN npm install # Cached unless dependencies change

# Copy the rest of the app
COPY . .
CMD ["node", "index.js"]
```

Advanced Docker Caching Techniques

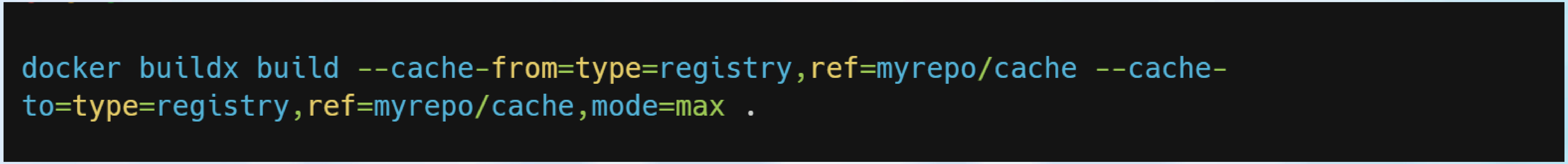
Using Mounts for Build Caching

- Bind Mounts vs. Volume Mounts for Caching
- --mount=type=cache for BuildKit

Leveraging External Cache Sources

- **Remote Cache in CI/CD Pipelines**
- **Using Docker Buildx for Distributed Caching**

Example: **Persistent Caching in CI/CD**



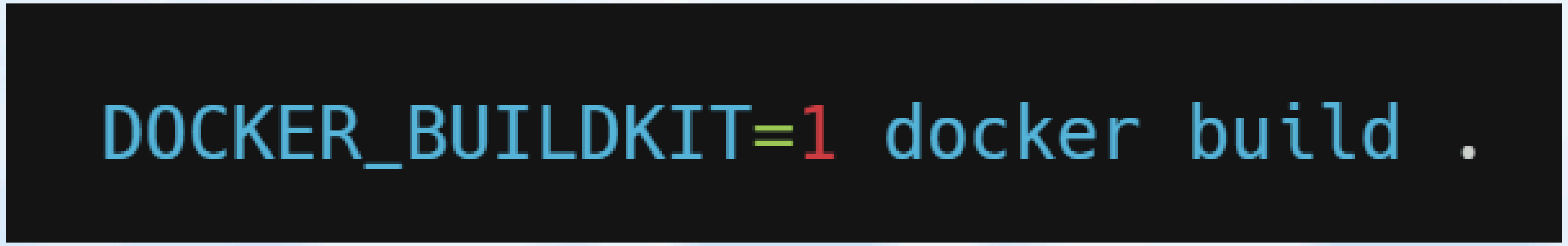
```
docker buildx build --cache-from=type=registry,ref=myrepo/cache --cache-
to=type=registry,ref=myrepo/cache,mode=max .
```

Using BuildKit to Supercharge Docker Builds

What is BuildKit?

- Faster parallel builds
- Automatic cache optimization
- More efficient file handling

Enabling BuildKit:



```
DOCKER_BUILDKIT=1 docker build .
```

Measuring and Debugging Build Performance

How to Check if Cache is Working?

- docker build --progress=plain
- docker history myapp

Tools to Analyze Docker Images:

Tool	Purpose
docker history	Shows image layers
dive	Analyzes image size
time docker build	Measures build time

Conclusion

- ➢ Understand how Docker caching works
- ➢ Optimize Dockerfile structure for caching
- ➢ Avoid cache-breaking mistakes
- ➢ Use advanced caching techniques in CI/CD

Next Steps:

- ➢ Apply these best practices to your projects
- ➢ Experiment with Docker BuildKit for better caching
- ➢ Optimize CI/CD pipelines for efficient builds