Improve Your Inner-loop development Workflow and Productivity using Docker

Your Instructor

Ajeet Singh Raina

- Developer Advocate at Docker
- Former Docker Captain
- Docker Community Leader
- Distinguished ARM Ambassador
- Worked @ Dell EMC, VMware, Redis & CGI

Agenda

Lab 1 : Inner-Loop Dev Workflow (45 min)

- Overview
- Docker Developer Workflow
- Image Building Best Practices
- Speed: Docker Init
- Speed: Compose File Watch

Lab 2: Container-first Dev Workflow (35 min)

- Overview
- Building a sample app using React, Node and Mongo
- Integrate app with the real-cloud Service

Lab 3: Container-supported Dev Workflow (40 min)

- Overview
- Building a sample app tech stack using emulated-cloud service
- Bringing up the services
- Validating the results
- Wrapping up

Before we get started

https://www.docker.com/products/docker-desktop/

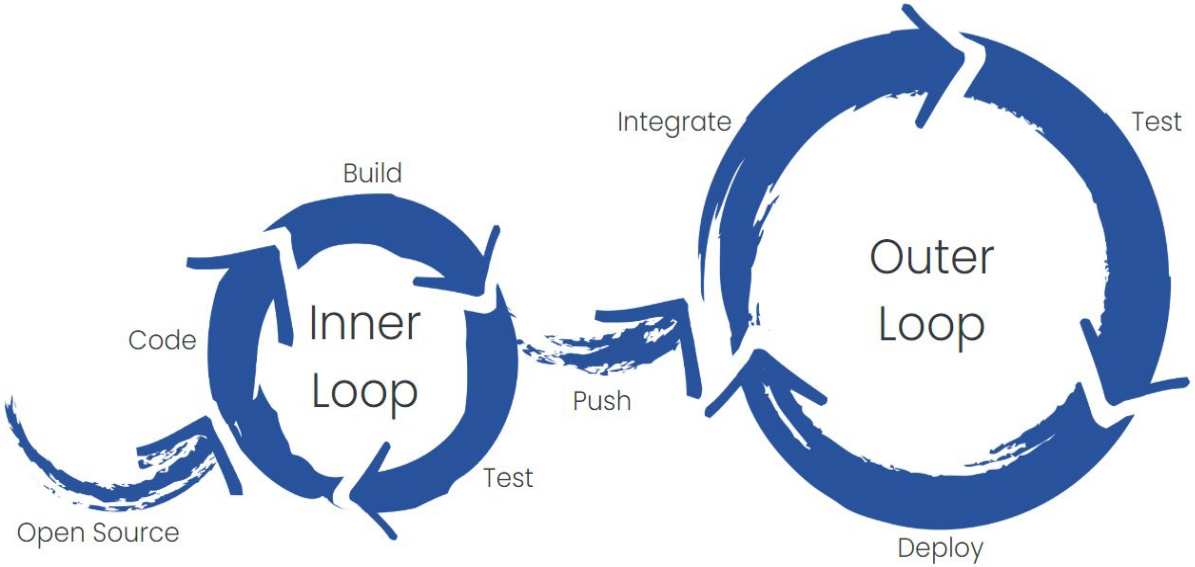
Access the Workshop

```
$ docker extension install  \

 dockerdevrel/wad-docker-extension:2.0
```

Access the Workshop

https://wad2024-workshop.vercel.app/

Docker is Uniquely Focused on Developer Success



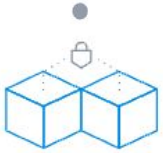
Trusted Images | Docker Desktop | **Docker Ecosystem** | Delivery Platforms

20M+ Active Developers | 450+ Trusted Partners

Built for Developers, by Developers

Speed

- Docker init
- Compose File Watch
- VirtioFS Support
- VPNKit => gVisor
- Docker Build Cloud



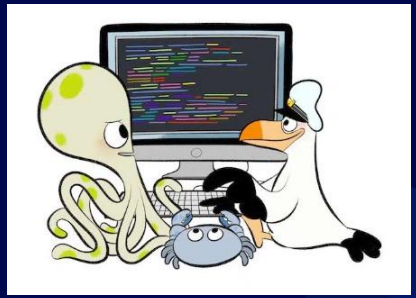
Security

- Docker Scout
- Attestations

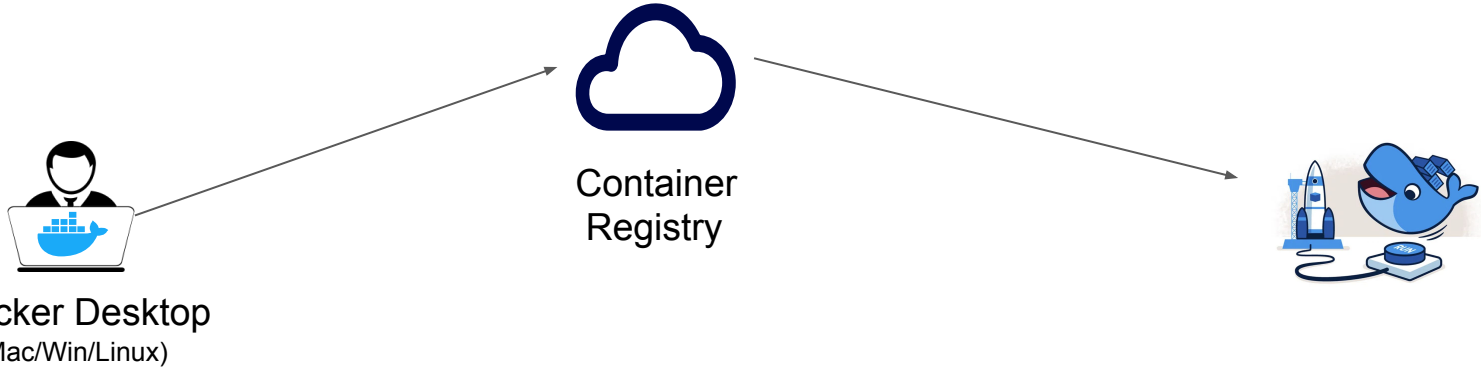
Choice

- Docker Extensions
- Docker Sponsored Open Source Projects
- Rosetta 2
- WebAssembly

Speed for developers



A 30,000 ft View



BUILD

- Package applications as portable container images
- Create Multi-container apps using Docker Compose

```
$ docker build
```

SHARE

- Collaborate and distribute via Registry
- Shareable application with clear interface for operators

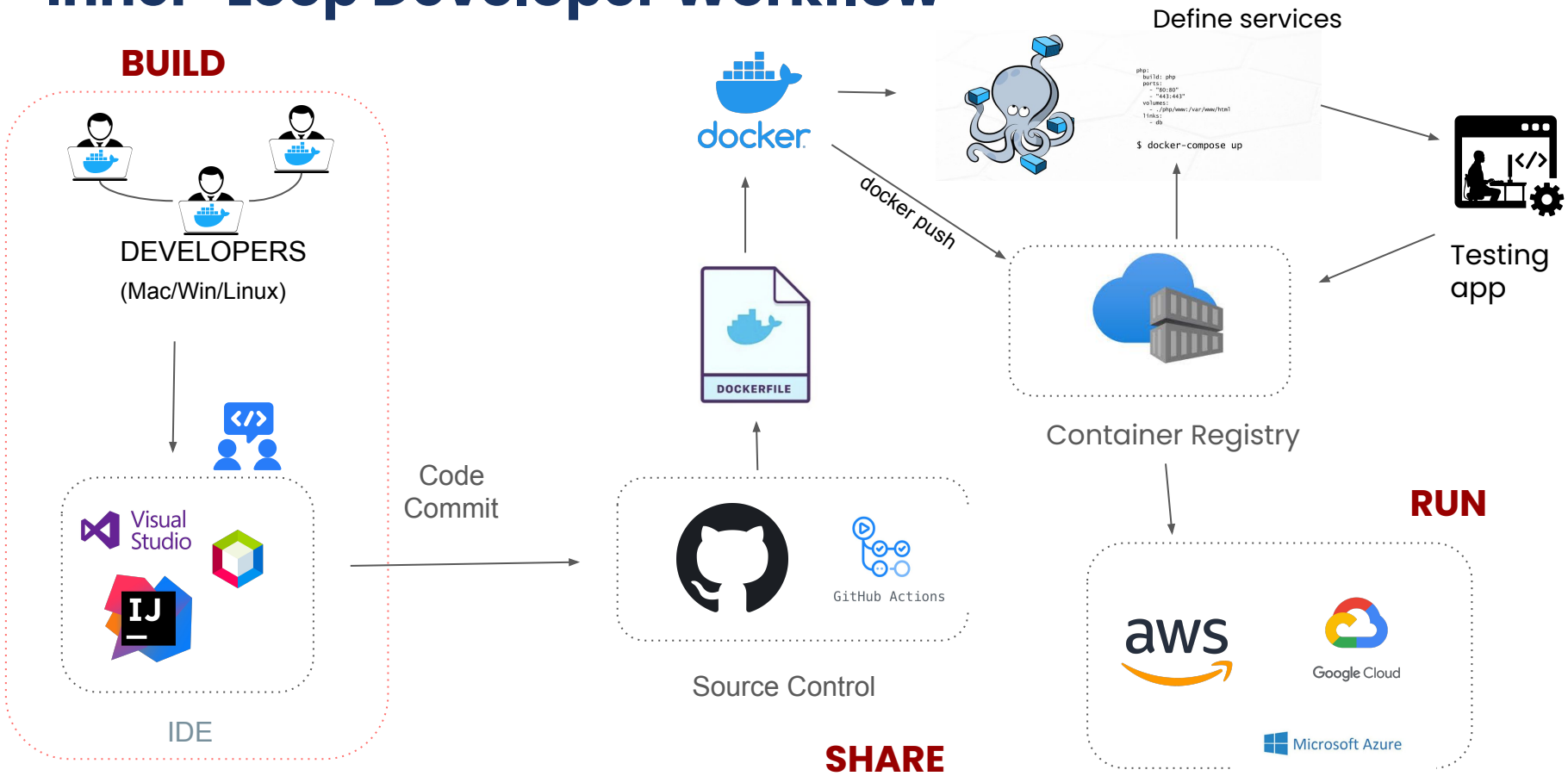
```
$ docker push
```

RUN

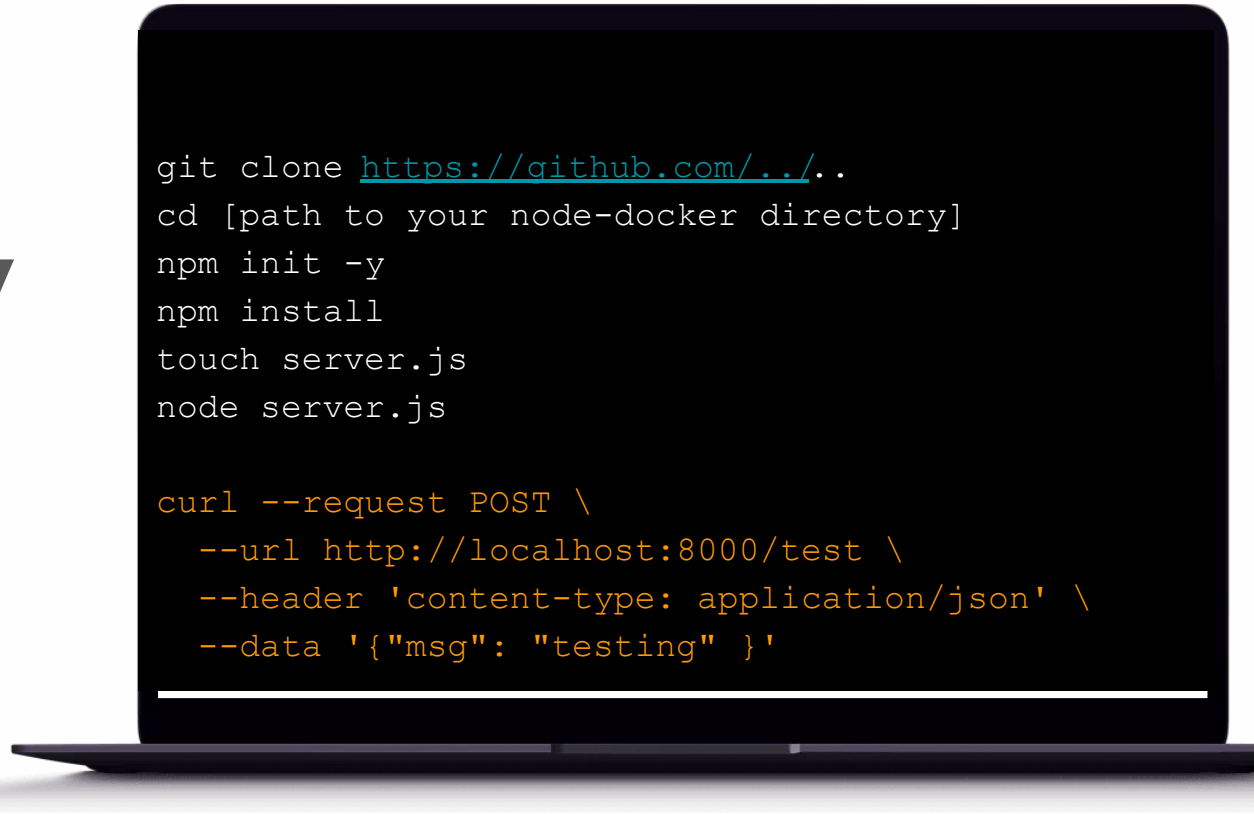
- Run multiple versions of the same application and manage pre-environment settings
- Launch your applications locally and on the cloud with AWS ECS and Azure ACI.

```
$ docker run
```

Inner-Loop Developer Workflow



Build and Deploy a Simple Node Application



Application

```
git clone https://github.com/../..
cd [path to your node-docker directory]
npm init -y
npm install
touch server.js
node server.js

curl --request POST \
  --url http://localhost:8000/test \
  --header 'content-type: application/json'
\
  --data '{"msg": "testing" }'
```

Dockerfile

```
FROM node

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```

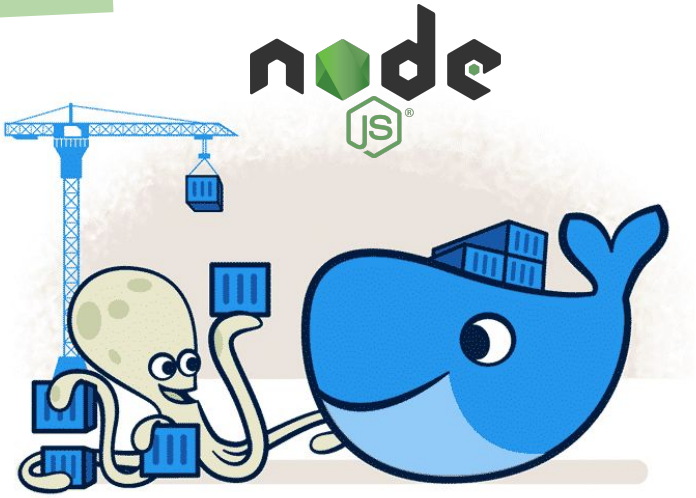
```
$ docker build -t node-docker .
$ docker run -d -p 8080:800 node-docker
```

Do's

- node:<tag>
- USER node
- MEM LIMIT
- HEALTHCHECK
- COPY
- SCANNING
- Multi-Stage
- .dockerignore
- buildx –platform

Don't

- node:latest
- ADD
- USER root
- EXPOSE db_port
- node_modules
- SIGTERM



9 Tips for Containerizing a NodeJS Application

1. **Use a specific base image tag instead of latest**
   The "latest" Docker image tag is unpredictable, hence try to use specific image tags whenever possible.

2. **Use Multi-Stage Build**
   Multi-Stage build makes the final image more secure and smaller in size.

3. **Fix security vulnerabilities in your Node image**
   Always run security scans on the images to detect any known vulnerabilities.

4. **Leverage HEALTHCHECKS**
   Test a container and confirm that it's still working.

5. **Use .dockerignore**
   To increase build performance, it is recommended to use .dockerignore file in the same directory as your Dockerfile.

6. **Run container as non-root user**
   Running app with user privileges is safe since it helps mitigates risks.

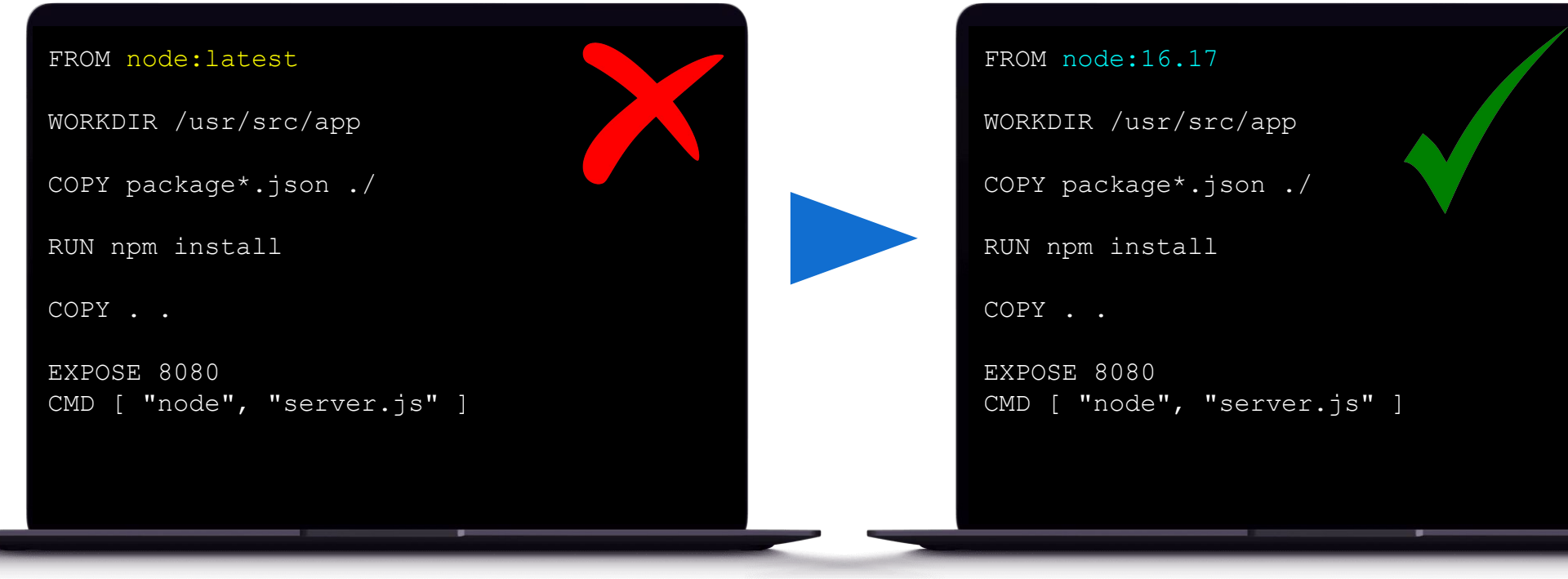
7. **Favor Multi-arch Docker Image**
   Always build the Docker image with Multi-Architecture Platform Support.

8. **Explore graceful shutdown options for Node**
   Ensure that your app is handling the ongoing requests and cleaning up resources in a timely fashion.

9. **Measure Node Performance**
   Use the OpenTelemetry API to ensure that apps are faster and more performant.

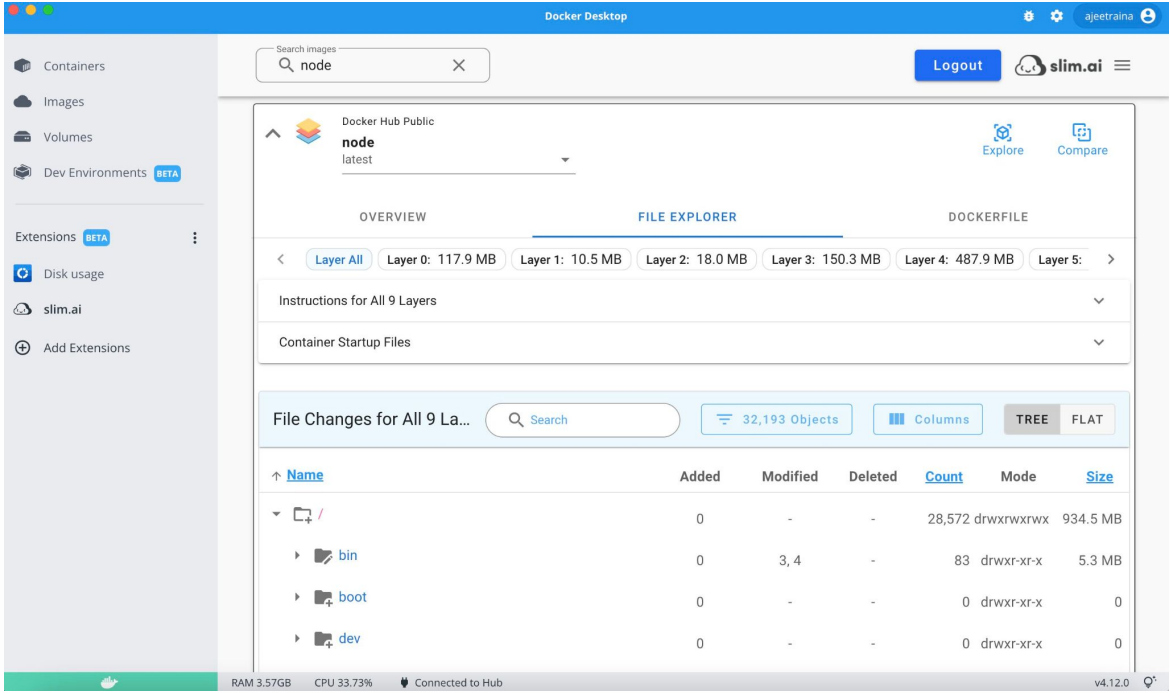
docker | node

#1 Use explicit Base Image reference instead of latest

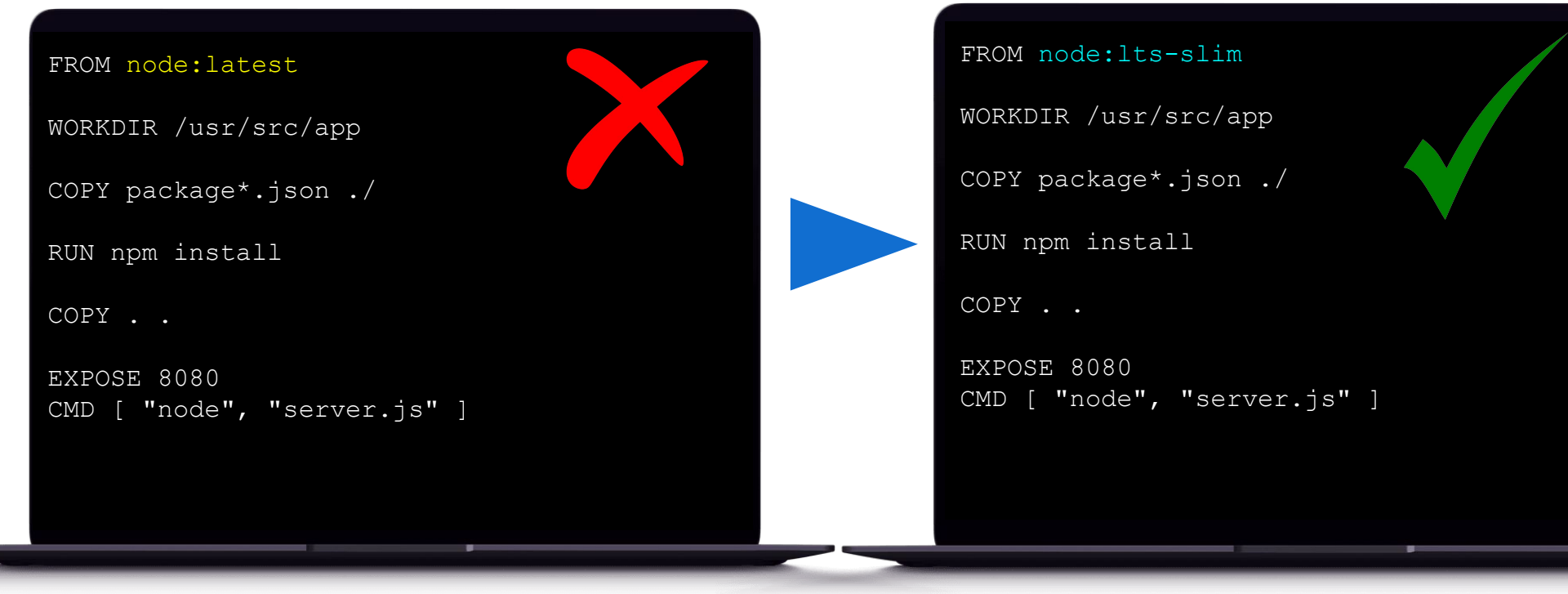


Downsides of latest Images

- Inconsistent Docker Image Builds
- Non-deterministic Behavior

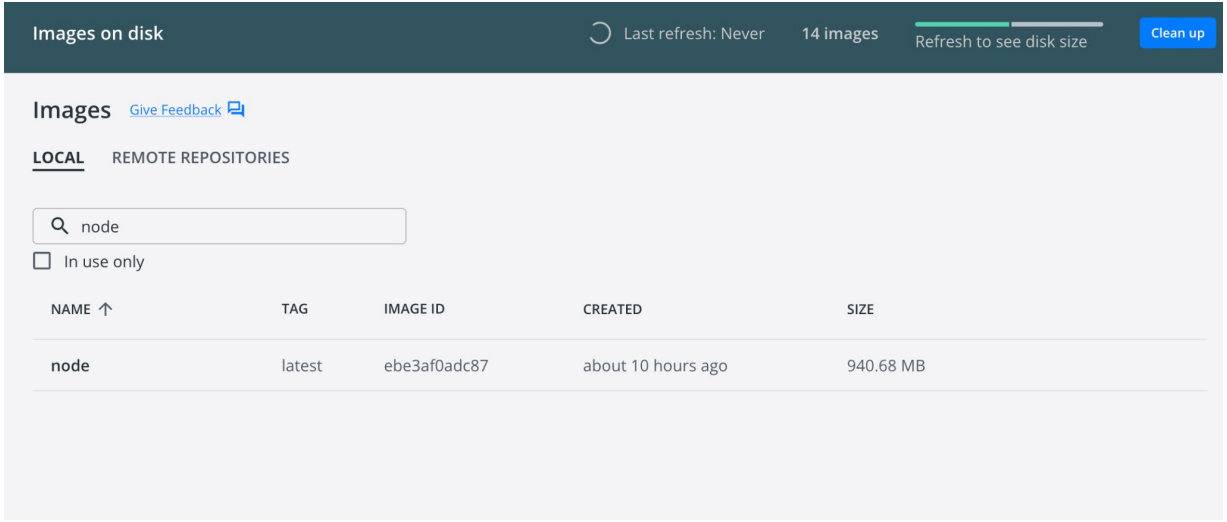


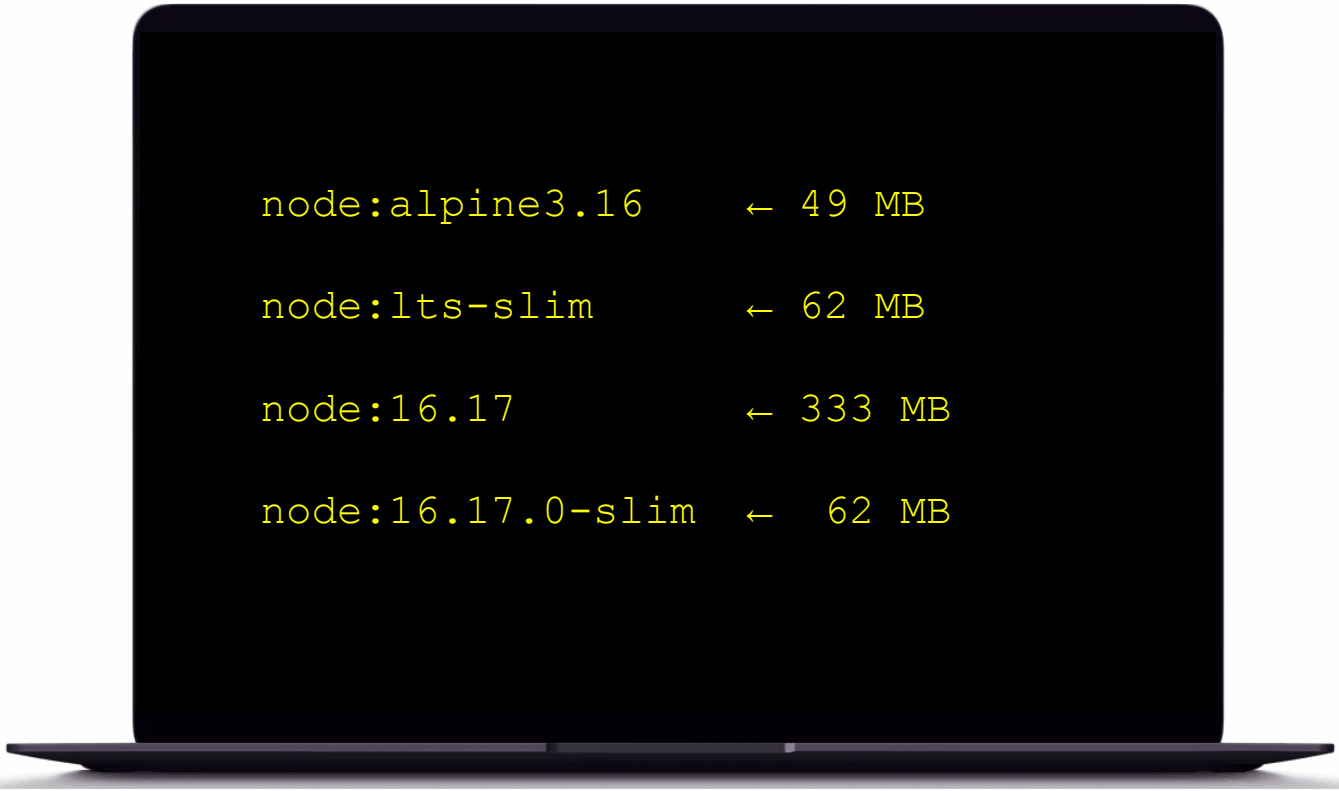
#2 Prefer Leaner Docker Images



Downsides of Bigger Image

- Larger Download Size
- Higher Exposure to Vulnerabilities
- Increases Resource Consumption

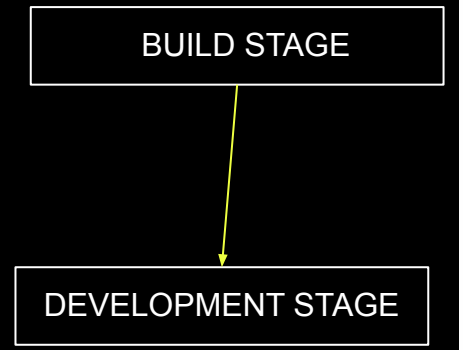


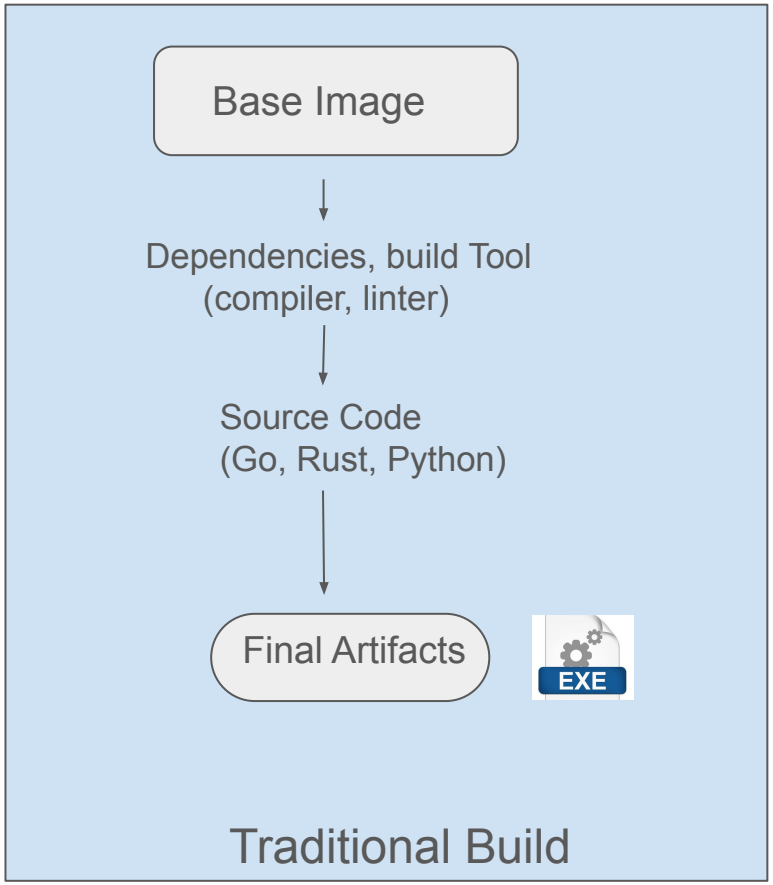

node:alpine3.16 ← 49 MB
node:lts-slim ← 62 MB
node:16.17 ← 333 MB
node:16.17.0-slim ← 62 MB

#3

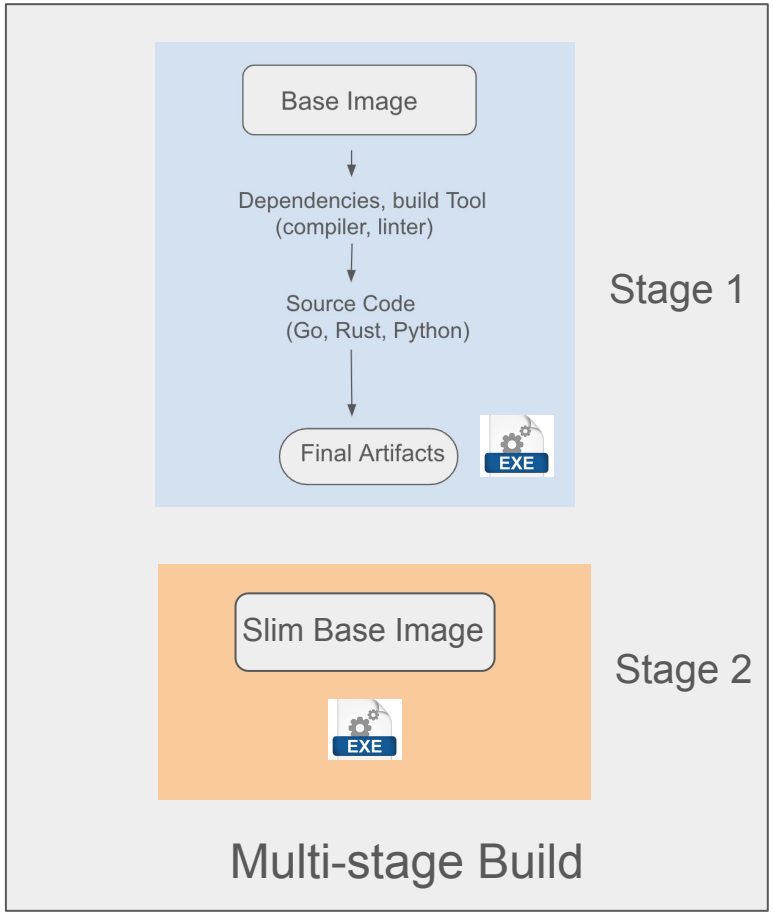
Use Multi-Stage Build

```
FROM node:lts-buster-slim AS builder
WORKDIR /usr/src/app

COPY package.json /usr/src/app/package.json
COPY package-lock.json /usr/src/app/package-lock.json
RUN npm ci

COPY . /usr/src/app

EXPOSE 3000

CMD [ "npm", "run", "dev" ]

FROM builder as dev-envs
RUN <<EOF
apt-get update
apt-get install -y --no-install-recommends git
EOF


# install Docker tools (cli, buildx, compose)
COPY --from=gloursdocker/docker / /
```




Base Image
Dependencies, build Tool
(compiler, linter)
Source Code
(Go, Rust, Python)
Final Artifacts
EXE
Traditional Build

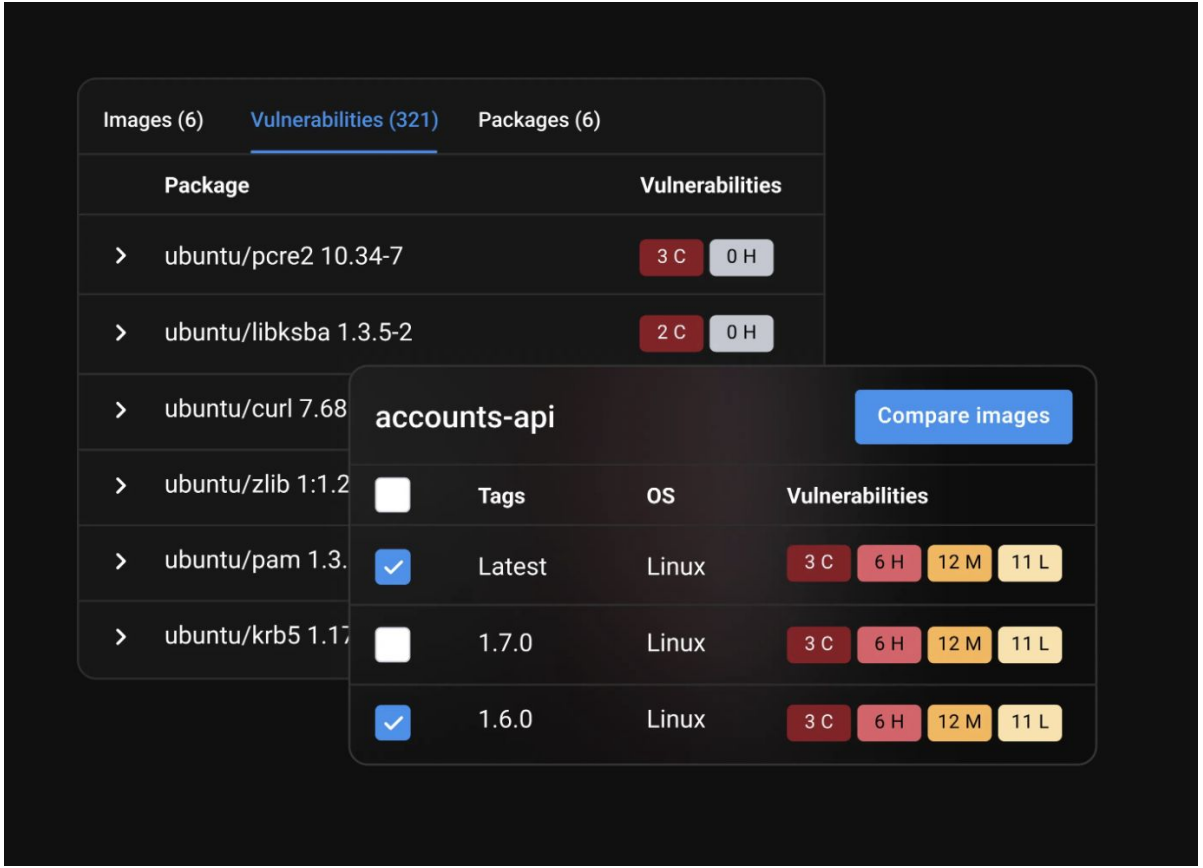
Gigs


Base Image
Dependencies, build Tool
(compiler, linter)
Source Code
(Go, Rust, Python)
Final Artifacts
EXE
Stage 1
Slim Base Image
EXE
Stage 2
Multi-stage Build

MBs

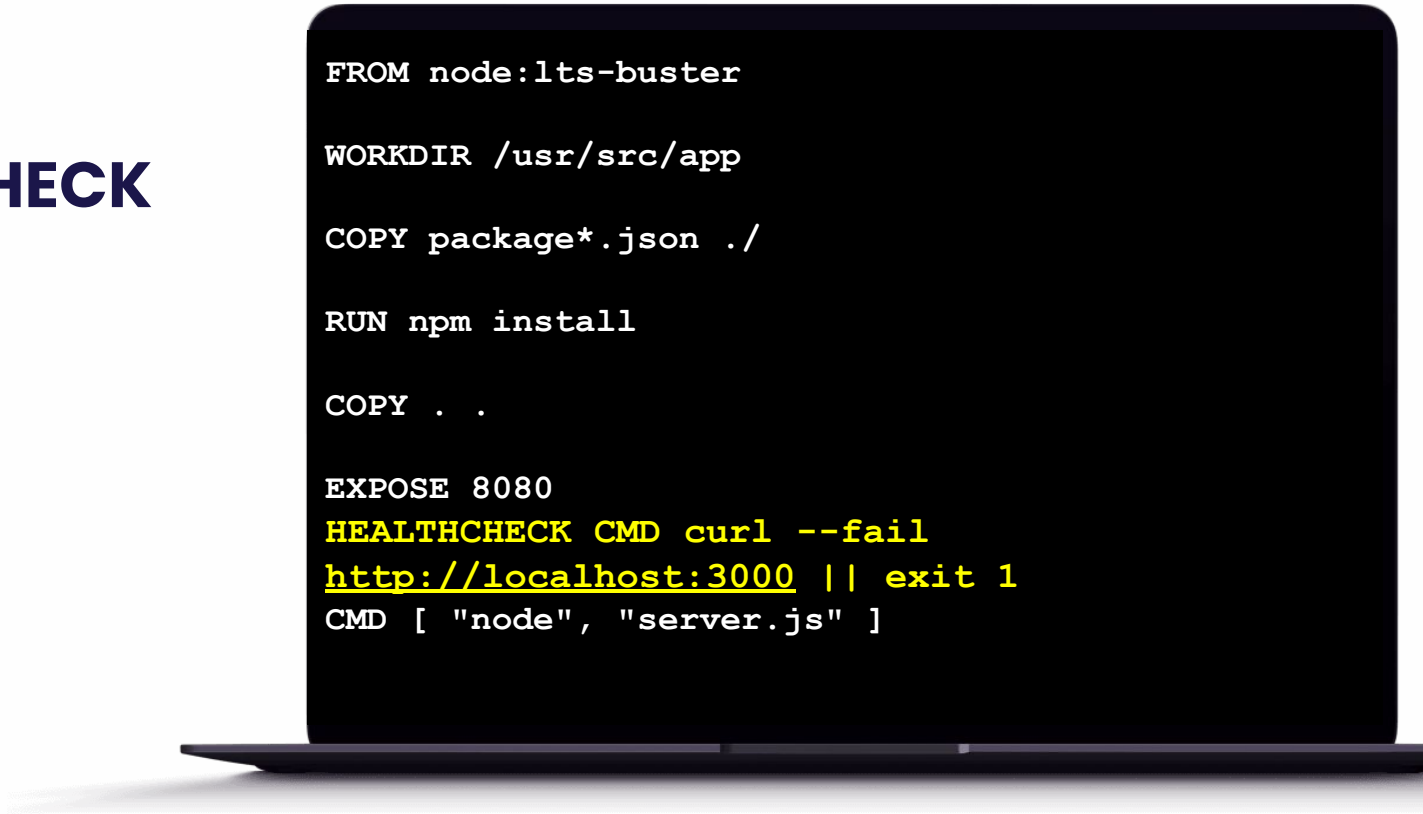
#4

Quickly identify and fix vulnerabilities during the build time using Scout



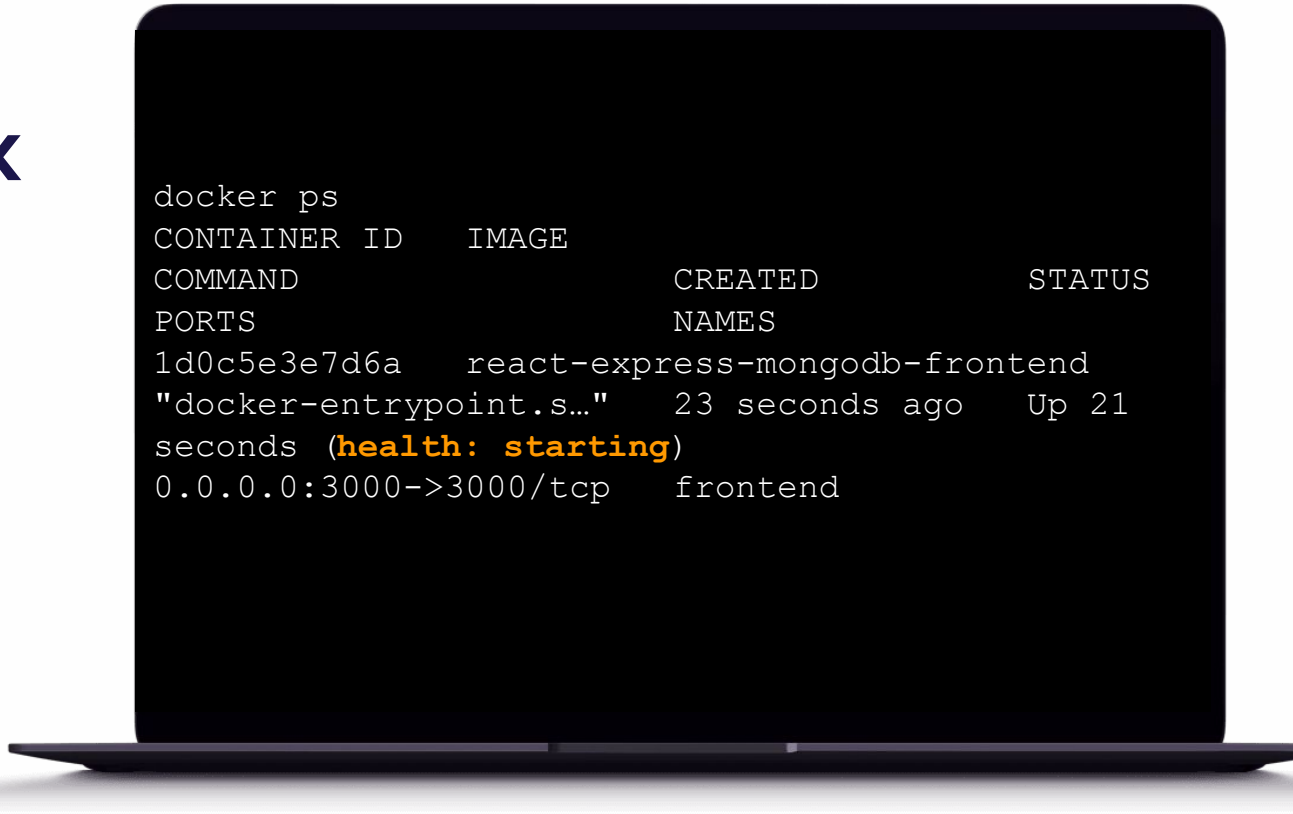
#5

Add HEALTHCHECK



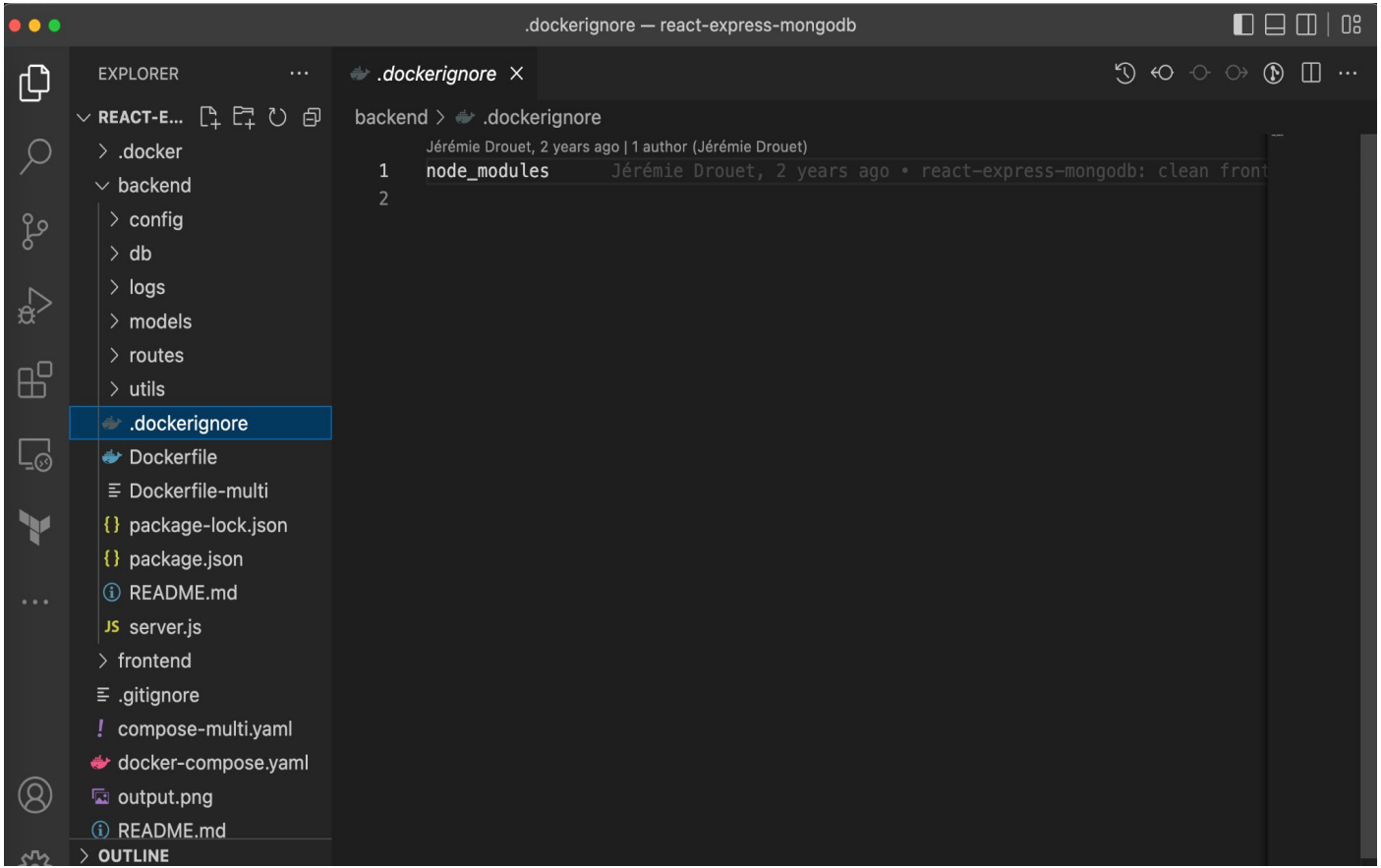
#5

Add HEALTHCHECK



#6

Use .dockerignore



#7

Run as non-root user for security purpose

```
FROM node:lts-buster AS development
WORKDIR /usr/src/app
COPY package.json /usr/src/app
COPY package-lock.json /usr/src/app
RUN npm ci
COPY . /usr/src/app
EXPOSE 3000
CMD ["npm", "start"]

FROM development as dev-envs
RUN <<EOF
apt-get update
apt-get install -y --no-install-recommends git
EOF

RUN <<EOF
useradd -s /bin/bash -m vscode
groupadd docker
usermod -aG docker vscode
EOF
# install Docker tools (cli, buildx, compose)
COPY --from=gloursdocker/docker / /
```

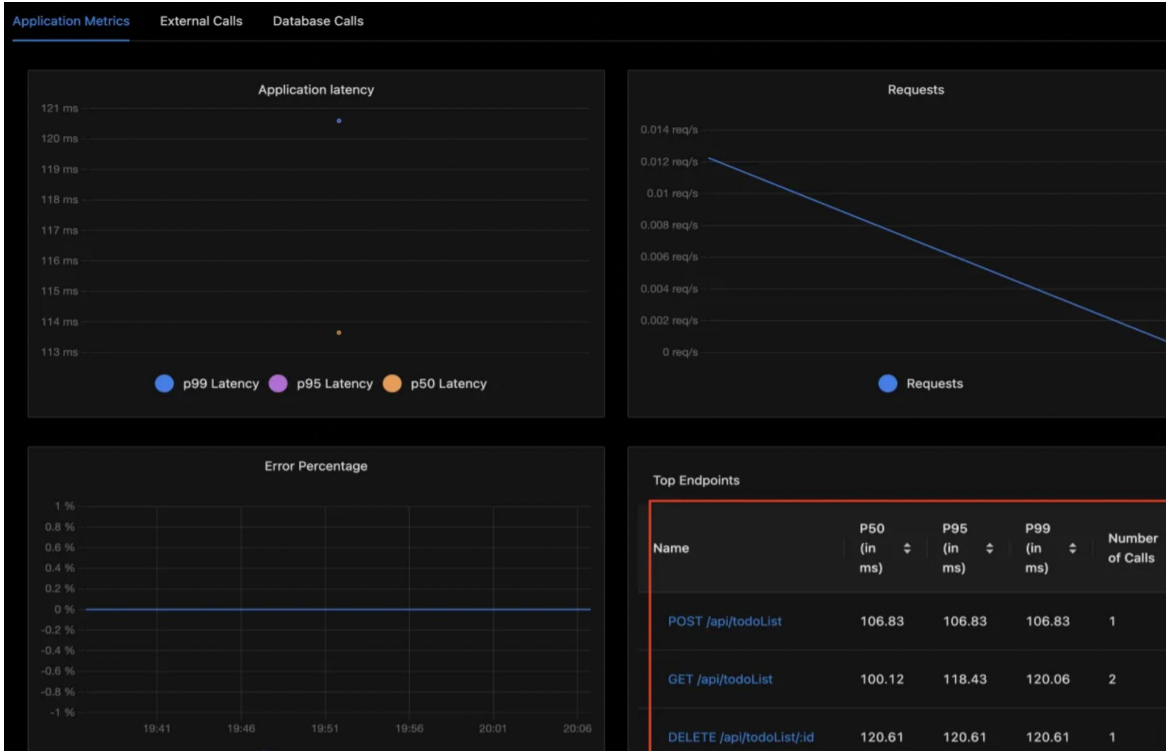
#8

Favour Multi-Architecture Docker Images

```
docker run --rm mplatform/mquery node:lts-buster
Unable to find image 'mplatform/mquery:latest'
locally
d0989420b6f0: Download complete
af74e063fc6e: Download complete
3441ed415baf: Download complete
a0c6ee298a93: Download complete
894bcacb16df: Downloading
[==========================================>
]  3.146MB/3.452MB
Image: node:lts-buster (digest:
sha256:a5d9200d3b8c17f0f3d7717034a9c215015b7aae70cb2
a9d5e5dae7ff8aa6ca8)
 * Manifest List: Yes (Image type:
application/vnd.docker.distribution.manifest.list.v2
+json)
 * Supported platforms:
   - linux/amd64
   - linux/arm/v7
   - linux/arm64/v8
```

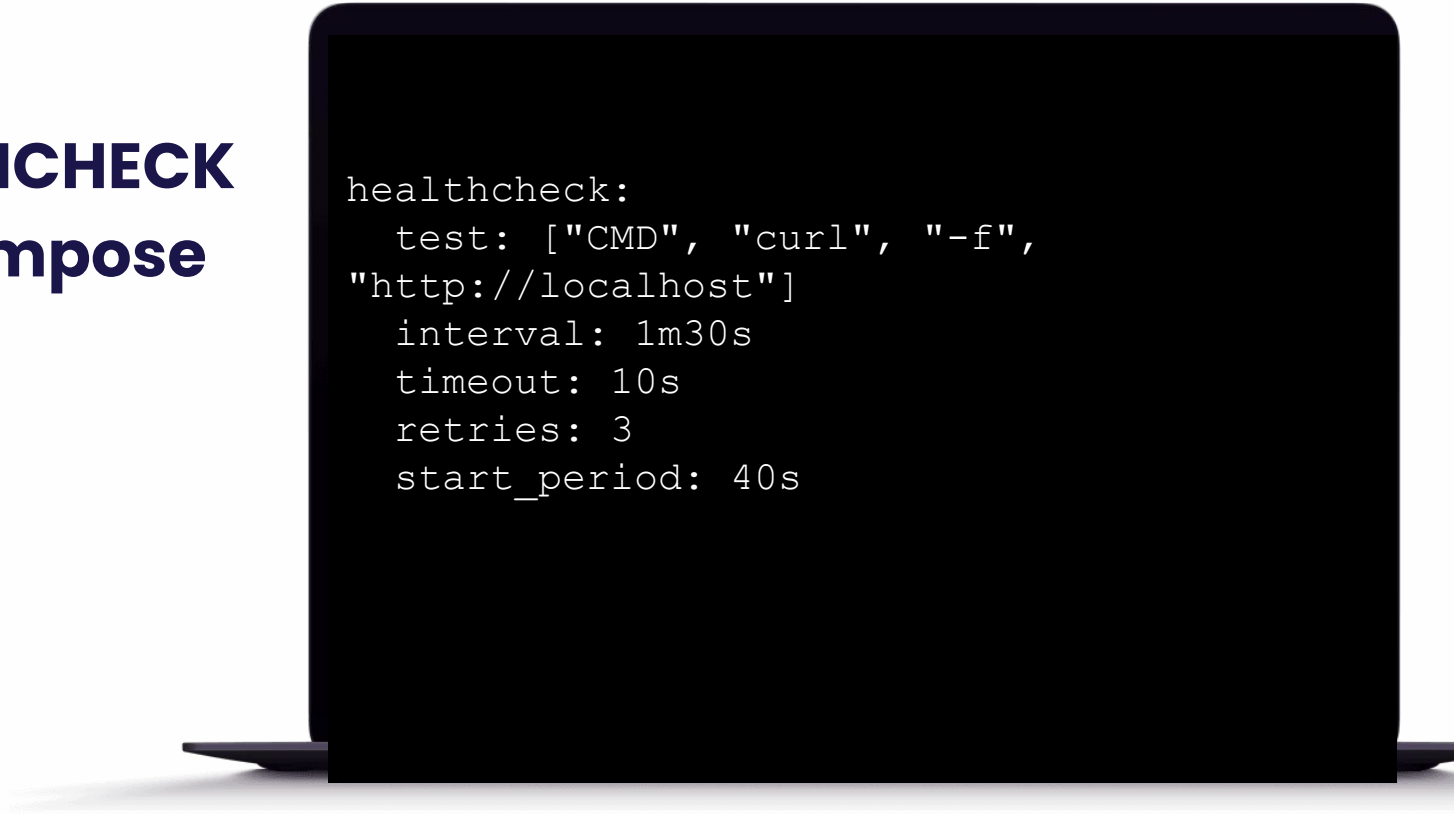
#9 Use Open Telemetry API to measure App performance

- Metrics(what)
- Logs(why)
- Trace(how)

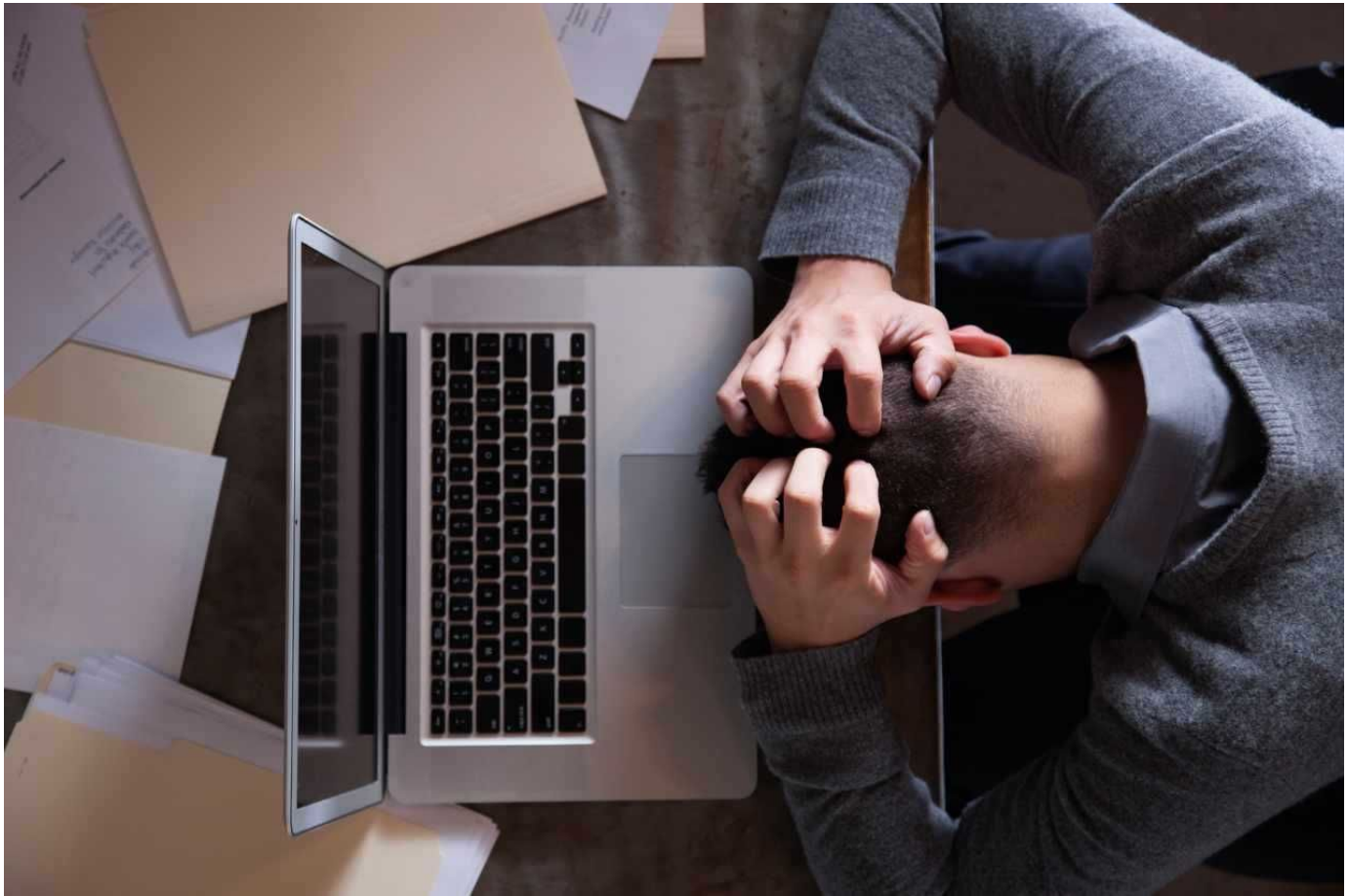


#10

Using HEALTHCHECK in Docker Compose



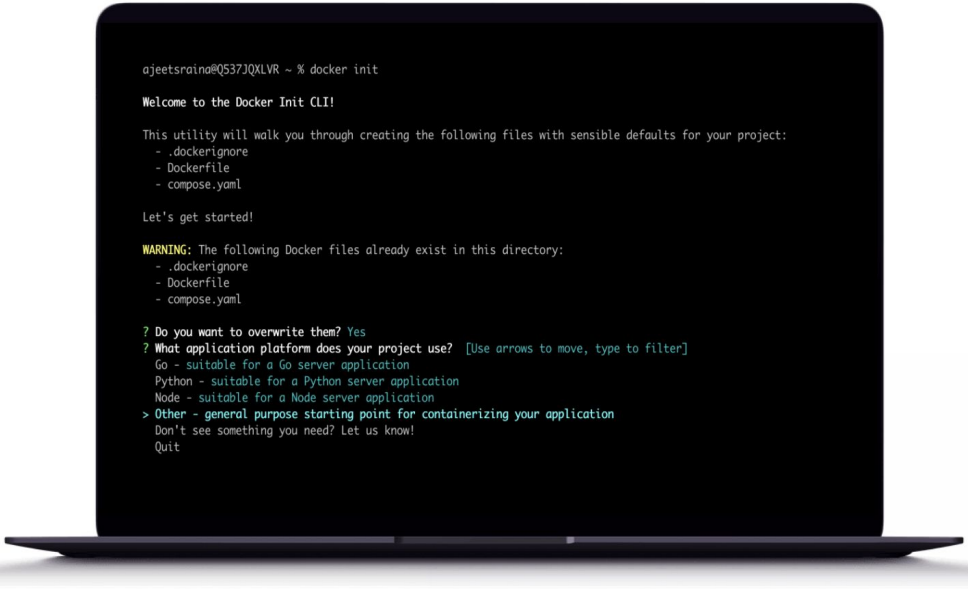
Best Practices – A Developers' Love-Hate Relationship



Introducing

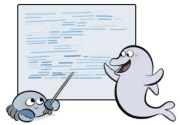
$ docker init

- Introduced for the first time in [Docker Desktop 4.18]
- Generates Docker assets for projects
- Allows you to choose application platform
- Makes it easier to create Docker images and containers

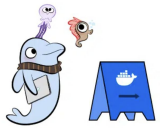


$ docker init

Simplified Docker Assets Creation

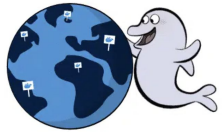


Saves Time and Effort

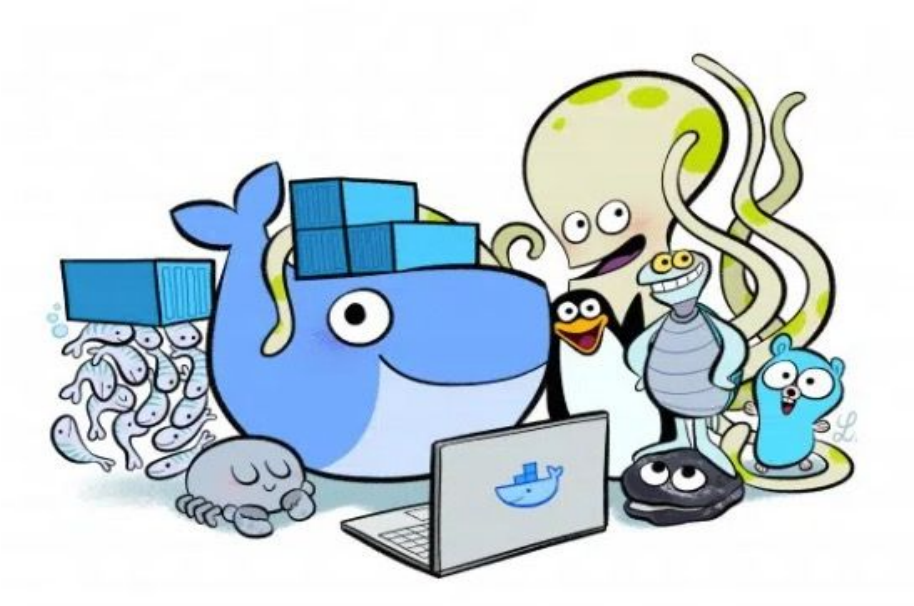


Better Project Organization

Enhanced Portability



It’s Demo Time



https://github.com/dockersamples/docker-init-demos

Compose Watch

Compose Watch

- Automatically updates your compose service containers while you work
- Blazing-fast file synchronization supporting live update

```
services:
  web:
    build: .
    command: npm start
    x-develop:
      watch:
        - action: sync
          path: ./web
          target: /src/web
          ignore:
            - node_modules/
        - action: rebuild
          path: package.json
```

How it works?

- Automatically builds a new image with BuildKit and replaces the running service container
- Add a develop section to your services in the compose.yaml file
- Configure it with a list of paths to watch and actions to take
- Watch rules allow ignoring specific files or entire directories within the watched tree.

Compose Watch Actions

Sync

Specifies a path to watch for changes in the host file system, and a corresponding target path inside the container to synchronize changes to.

Rebuild

The "rebuild" action specifies a path to watch for changes in the host file system, and triggers a rebuild of the container when changes are detected.

sync+restart

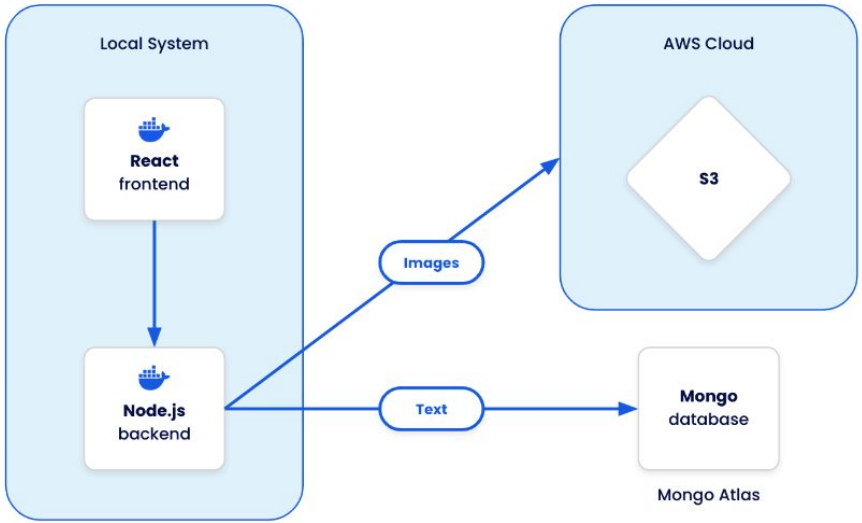
The "sync+restart" action specifies a path to watch for changes in the host file system, and a corresponding target path inside a container to first synchronize changes to and then restart the container

https://wad2024-workshop.vercel.app/

Container-First Development Workflow

Container-first Development Workflow

Using containers for every aspect of software development, including the application runtime itself.

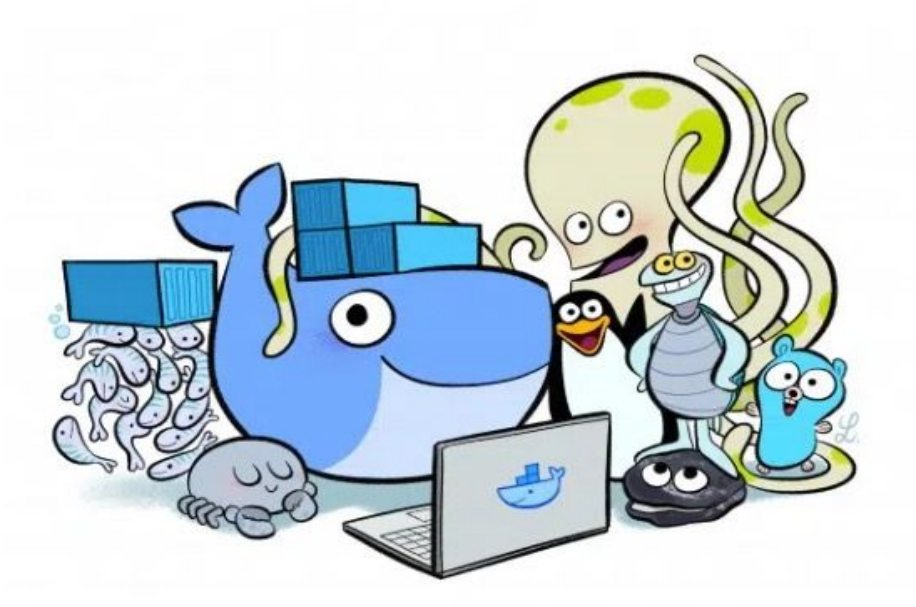


Container-first Development Workflow

Benefits for Developers:

- **Extreme Portability:** Developers only need a container engine and an IDE to work on the project. This ensures identical development environments regardless of the underlying operating system or pre-installed software.
- **Faster Setup:** No time wasted installing the application runtime or fiddling with local configurations. Developers can start coding as soon as the containerized environment is up.
- **Improved Isolation:** Each project runs within its own isolated container, preventing conflicts between projects or dependencies from interfering with other applications on the developer's machine.
- **Simplified Collaboration:** Team members can easily share and reproduce development environments using the same container images.

It’s Demo Time

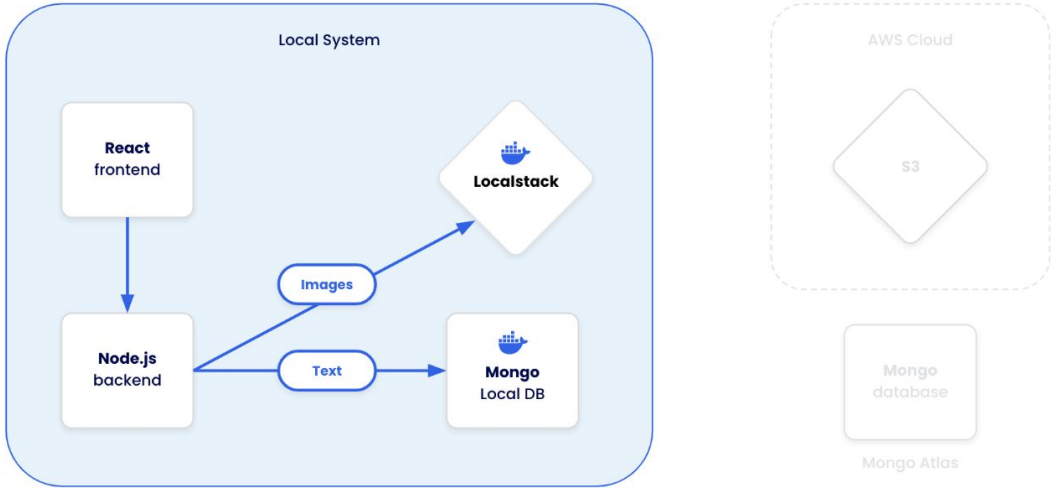


https://wad2024-workshop.vercel.app/lab2/services/

Container-supported Development Workflow

Container-first Development Workflow

Using containers to support and enhance development without touching the main application runtime itself.

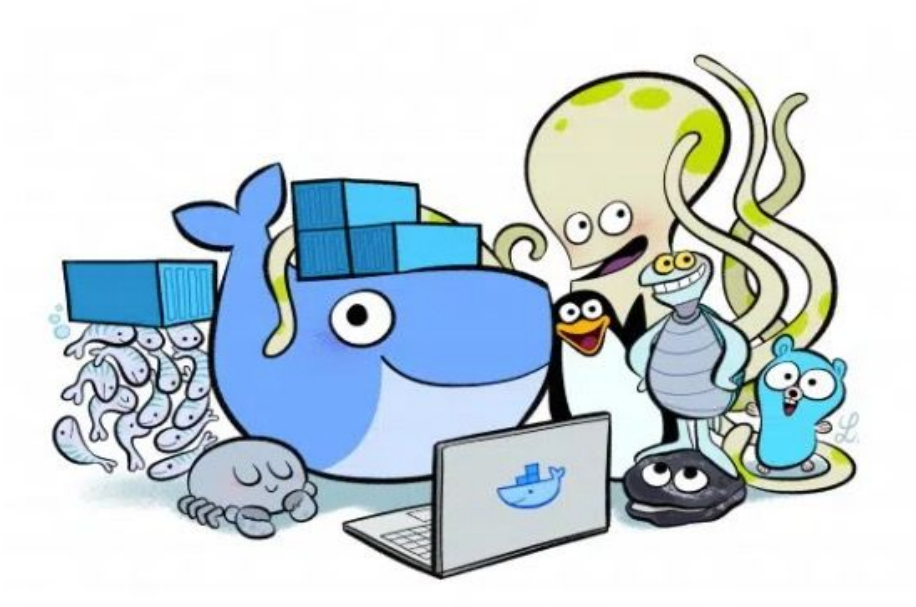


Container-supported Development Workflow

Benefits for Developers:

- **Separation of Concerns**: Developers focus on the core application logic using their familiar runtime (JVM, Node, Python etc.), while external dependencies are isolated in containers.
- **Improved Efficiency**:
- Easier setup: Containers simplify dependency management, reducing time spent configuring environments. Version consistency: All developers use the same container image, ensuring consistent dependencies across the team.
- Enhanced Capabilities: Docker allows running local simulations of cloud services and real services within tests, providing a more realistic development environment.
- **Flexibility in Implementation**: There's no single approach. Teams can use:
- Wrapper scripts for simple container execution.
- IDE plugins for launching development environments defined in Docker Compose files.
- Programming interactions with libraries like Testcontainers.
- **Leveraging Shared Resources**: Teams can benefit from pre-built container images from public repositories like Docker Official Images (DOI) and Docker Verified Publishers (DVP).

It’s Demo Time



https://wad2024-workshop.vercel.app/lab3/services/

Wrapping up

docker

Thank you

See you again soon.