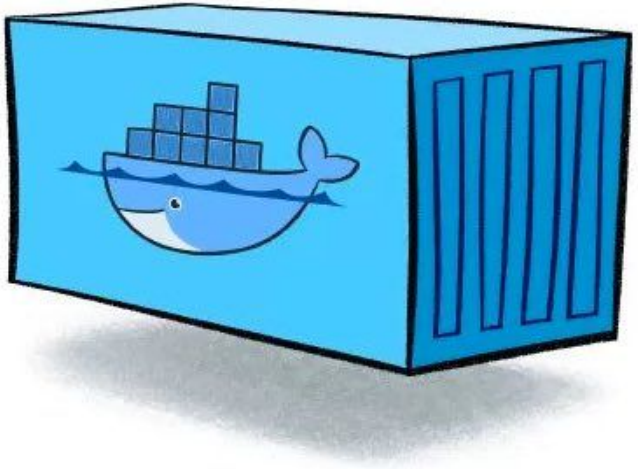
Module Five

Image Deep Dive

Image recap

Images = shipping containers



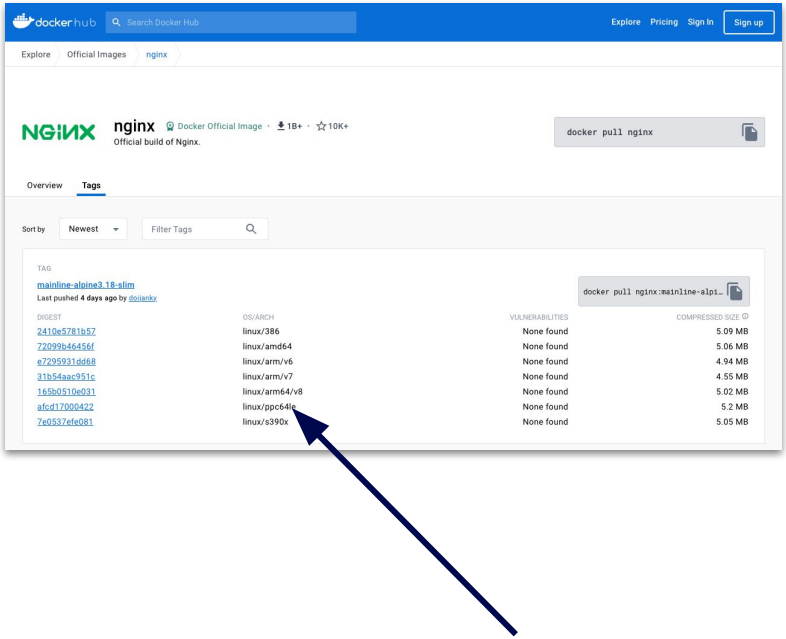
The container specifications

- Docker created the Open Container Initiative in June 2015
- Currently owned by the Linux Foundation
- Currently defines three specifications
  - `image-spec` - defines image structures and manifests
  - `runtime-spec` - defines how to run OCI images
  - `distribution-spec` - defines the API protocol to push, pull, and discovery content

OPEN CONTAINER INITIATIVE

Image architectures

- Binaries are compiled to their underlying architectures
- Multi-arch images make it possible to run on multiple architectures
- The container runtime will automatically pull the appropriate architecture

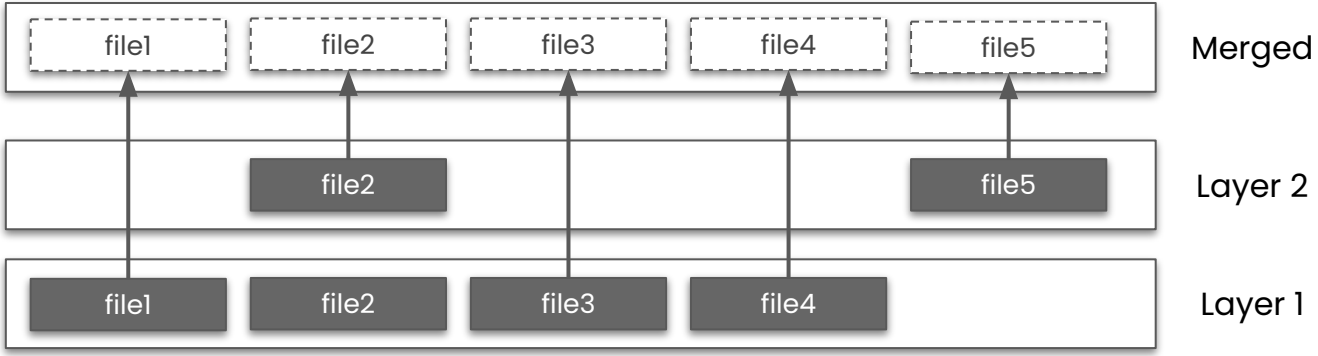


docker

Understanding unioned filesystems

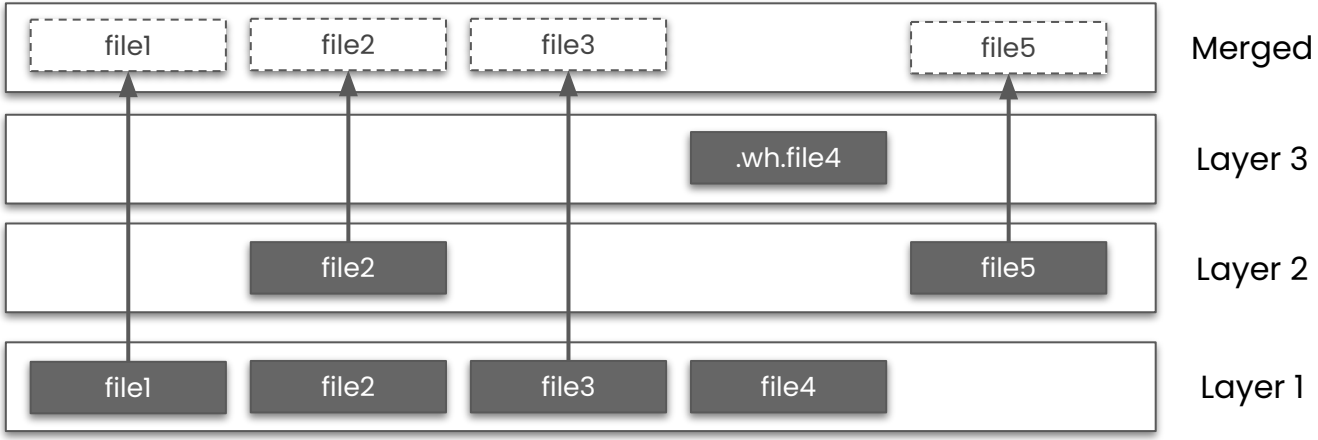
Image layering

- Unioned filesystems give the ability of combining multiple directories to make a single, unioned filesystem
  - Each layer can add files as needed
  - Files in “higher” layers replace those from “lower” layers

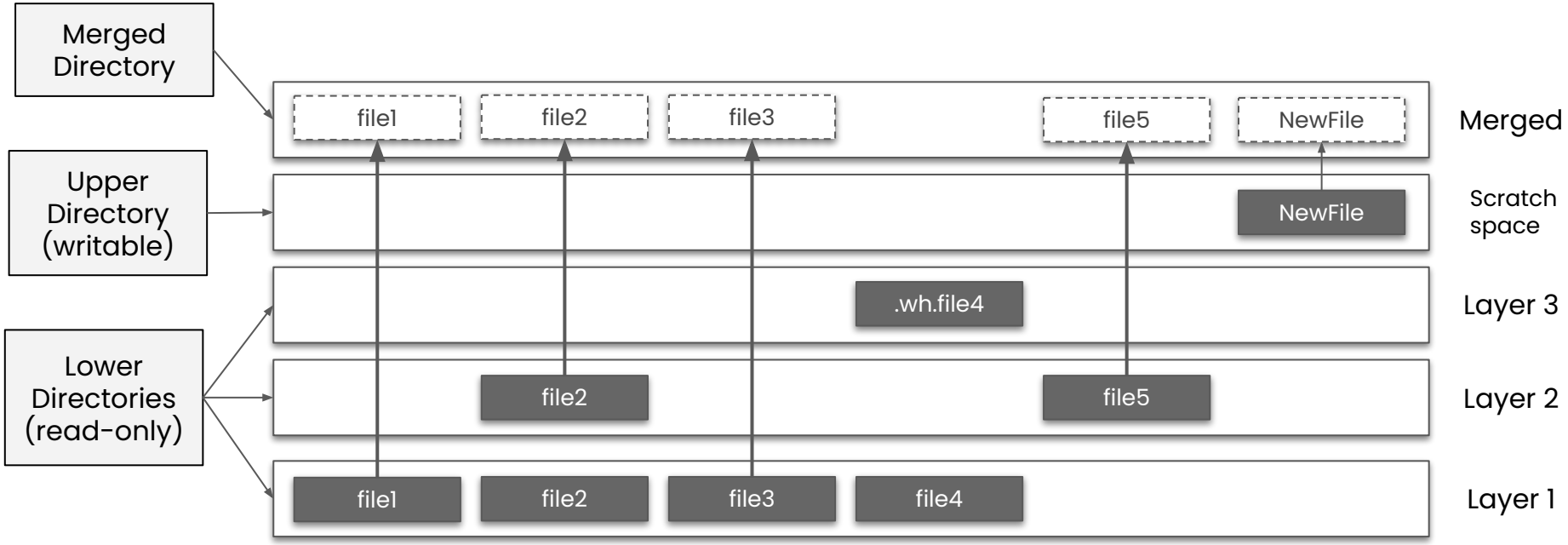


Deleted files

- If a layer wants to delete a file, it creates a “whiteout” file
  - Whiteout files are only used by the filesystem driver and not visible in the merged filesystem



Union filesystem terminology



All of this is viewable using `docker inspect <container>`

Creating an image manually

- Images can be manually created using these steps:
  - `docker run` - start a new container
  - `docker exec` and `docker cp` to make changes
  - `docker commit` to save the container's filesystem as an image

Or use a Dockerfile!

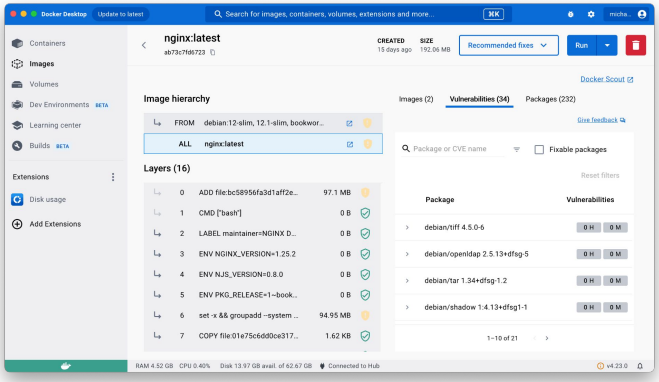
- Most often included with source code repositories
- Text-based file with instructions on how to build an image
  - **FROM** - the base to start from
  - **WORKDIR** - set the working directory in the image
  - **COPY** - copy files from host into the image
  - **RUN** - run a command
  - **CMD** - set the default command
- Build it with **`docker build`**

```
FROM ubuntu
WORKDIR /usr/local/app
RUN apt update && apt install -y nodejs
COPY index.js .
CMD ["node", "index.js"]
```

Visualizing the layers

- `docker image history` - view details about each layer
- Image analysis in Docker Desktop
- Other open-source tools
  - github.com/wagoodman/dive - dive into an image and see changes

```
❯ docker image history nginx
IMAGE          CREATED       CREATED BY                                      SIZE      COMMENT
ab73c7fd6723   2 weeks ago   /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon…   0B
<missing>      2 weeks ago   /bin/sh -c #(nop)  STOPSIGNAL SIGQUIT           0B
<missing>      2 weeks ago   /bin/sh -c #(nop)  EXPOSE 80                    0B
<missing>      2 weeks ago   /bin/sh -c #(nop)  ENTRYPOINT ["/docker-entr…   0B
<missing>      2 weeks ago   /bin/sh -c #(nop) COPY file:9e3b2b63db9f8fc7…   4.62kB
<missing>      2 weeks ago   /bin/sh -c #(nop) COPY file:57846632accc8975…   3.02kB
<missing>      2 weeks ago   /bin/sh -c #(nop) COPY file:3b1b9915b7dd898a…   298B
<missing>      2 weeks ago   /bin/sh -c #(nop) COPY file:caec368f5a54f70a…   2.12kB
<missing>      2 weeks ago   /bin/sh -c #(nop) COPY file:01e75c6dd0ce317d…   1.62kB
<missing>      2 weeks ago   /bin/sh -c set -x     && groupadd --system -…   94.9MB
<missing>      2 weeks ago   /bin/sh -c #(nop)  ENV PKG_RELEASE=1~bookworm   0B
<missing>      2 weeks ago   /bin/sh -c #(nop)  ENV NJS_VERSION=0.8.0        0B
<missing>      2 weeks ago   /bin/sh -c #(nop)  ENV NGINX_VERSION=1.25.2     0B
<missing>      2 weeks ago   /bin/sh -c #(nop)  LABEL maintainer=NGINX Do…   0B
<missing>      2 weeks ago   /bin/sh -c #(nop)  CMD ["bash"]                 0B
<missing>      2 weeks ago   /bin/sh -c #(nop) ADD file:bc58956fa3d1aff2e…   97.1MB
```



docker

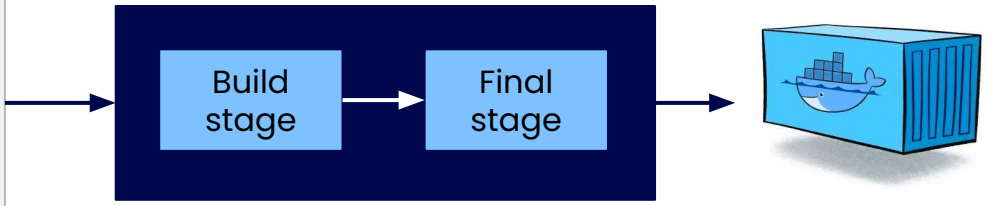
Overview of builders and terminology

Legacy Builder

- The original builder used when running `docker build`
- Reads a `Dockerfile` and produces an image
- Very basic support for multi-stage builds
  - Runs all instructions in serial order
  - Builds all stages up to the target stage
- Not able to build multi-architecture images

```
FROM node:lts-alpine AS build
WORKDIR /usr/local/app
COPY package.json yarn.lock ./
RUN yarn install
COPY public ./public
COPY src ./src
RUN yarn build

FROM nginx:alpine AS final
COPY --from=build /usr/local/app/build /usr/sh…
```

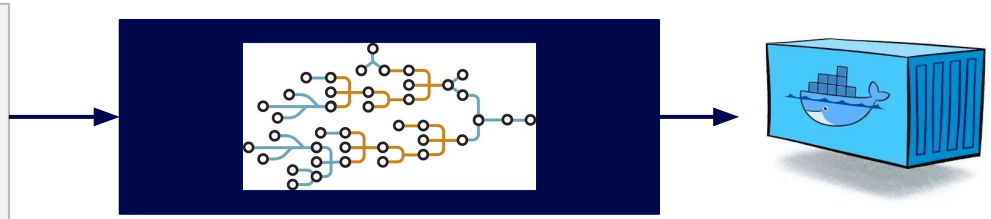


BuildKit

- Open-source builder started in Nov 2017
  - Replaced legacy builder in Docker Engine 23.0 (DD 4.19)
- Provides support for caching, distributed workers, new frontends, and much more
- Provides very low-level build definition format
  - You'll most likely never interact with BuildKit directly, but tools that use it

```
FROM node:lts-alpine AS build
WORKDIR /usr/local/app
COPY package.json yarn.lock ./
RUN yarn install
COPY public ./public
COPY src ./src
RUN yarn build

FROM nginx:alpine AS final
COPY --from=build /usr/local/app/build /usr/sh…
```



docker buildx

- A new Docker CLI command that exposes BuildKit features
  - Manages the BuildKit builders
  - Multi-architecture builds
  - Advanced cache management
  - SBOM and provenance creation

```
❯ docker buildx --help

Usage:  docker buildx [OPTIONS] COMMAND

Extended build capabilities with BuildKit

Options:
      --builder string   Override the configured builder instance

Management Commands:
  imagetools  Commands to work on images in registry

Commands:
  bake        Build from a file
  build       Start a build
  create      Create a new builder instance
  du          Disk usage
  inspect     Inspect current builder instance
  ls          List builder instances
  prune       Remove build cache
  rm          Remove a builder instance
  stop        Stop builder instance
  use         Set the current builder instance
  version     Show buildx version information

Run 'docker buildx COMMAND --help' for more information on a command.
```

Using buildx

Built-in BuildKit

- Docker Desktop includes a BuildKit daemon
- `docker build` has been aliased to `docker buildx build` since DD 4.19
- It has limitations though…
  - Not able to produce multi-architecture images
  - The move to use containerd for image storage should fix this (currently an experimental feature)

Creating and using builders

- A builder = a BuildKit daemon
- `docker buildx create --name my-builder`
  - Creates a builder named my-builder that will run in a container
- `docker buildx create --platform linux/amd64,linux/arm64`
  - Creates a multi-arch builder that will be given a generated name
- `docker buildx ls`
  - List all of the builders
- `docker buildx use <builder-name>`
  - Make the specified builder the default

Build drivers

- Drivers provide different ways for how the BuildKit backend runs
- They include:
  - `docker` - uses the BuildKit library bundled into the Docker daemon
  - `docker-container` - creates a dedicated BuildKit container
  - `kubernetes` - creates BuildKit pods in a Kubernetes cluster
  - `remote` - connects directly to a manually managed BuildKit daemon

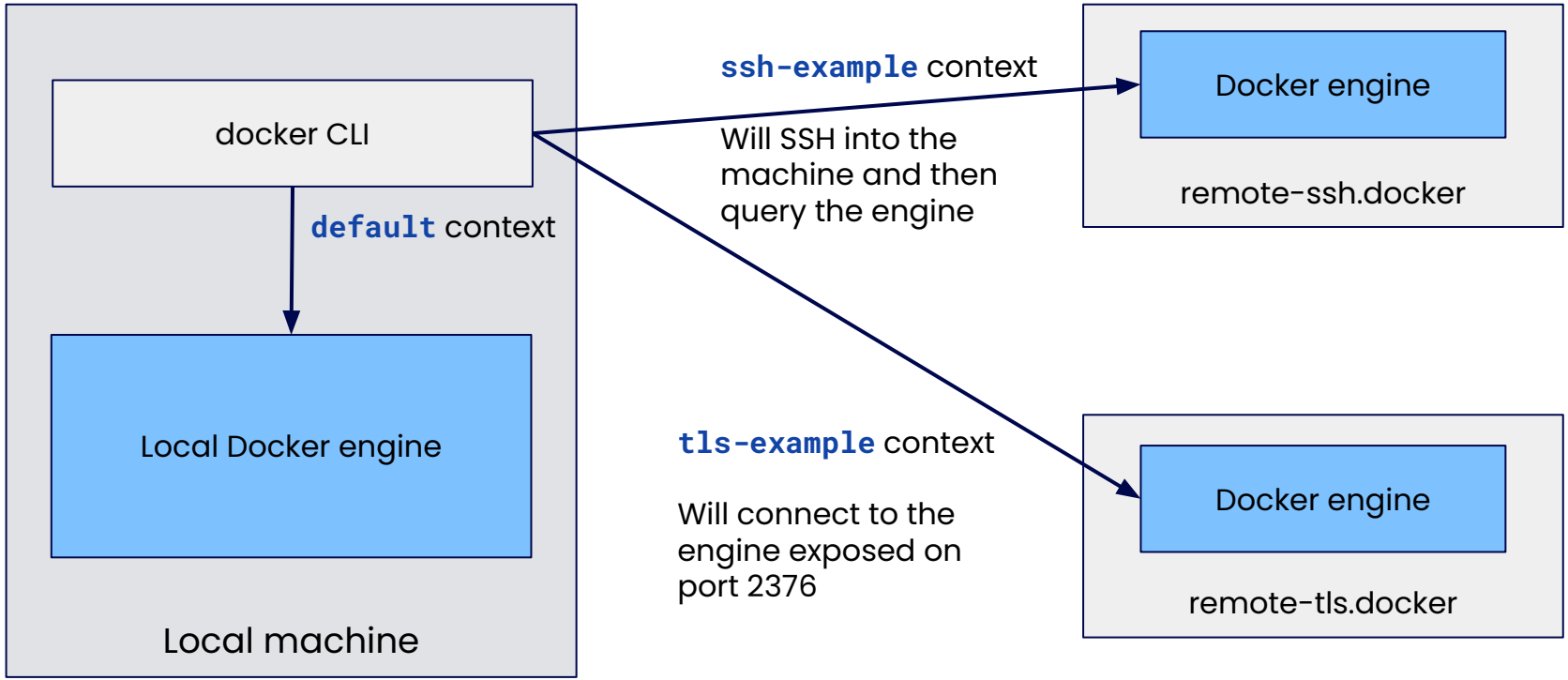
Feature	`docker`	`docker-container`	`kubernetes`	`remote`
Automatically load image	✅			
Cache export	Inline only	✅	✅	✅
Tarball output		✅	✅	✅
Multi-arch images		✅	✅	✅
BuildKit configuration		✅	✅	Managed externally

Docker contexts 101

- Contexts allow your local CLI to work with other Docker engines
- The default context works with the engine on your machine
- Common commands:
  - `docker context create` - create a new context
  - `docker context ls` - list current contexts
  - `docker context use <name>` - make the specified context the default

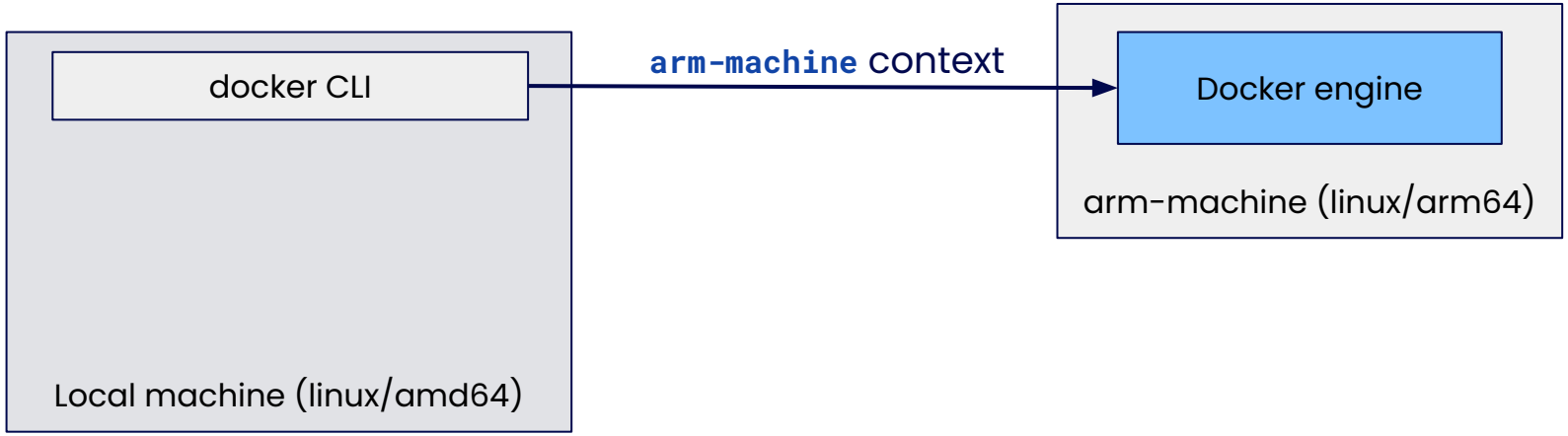
```
docker context create ssh-example --docker "host=ssh://user@remote-ssh.docker"
docker context create tls-example \
    --docker "host=tcp://remote-tls.docker:2376,ca=ca.pem,cert=cert.pem,key=key.pem"
```

Docker contexts visually



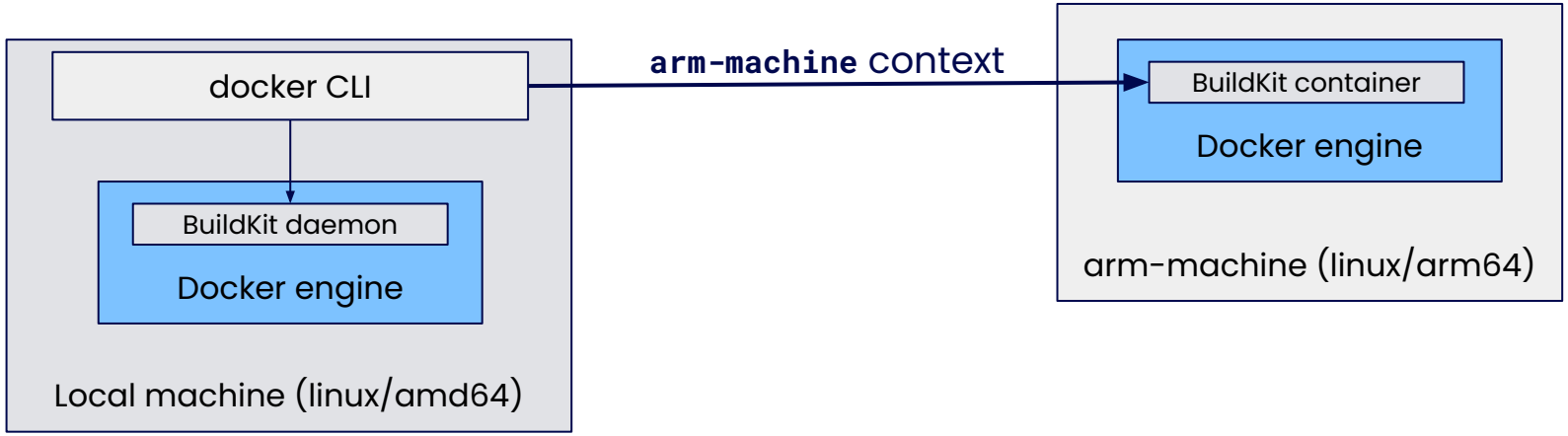
Native multi-arch builders

1. First, create a Docker context for the remote nodes
   - `docker context create arm-machine --docker "host=ssh://ubuntu@arm-machine"`
2. Create a builder, specifying the platform and context to use
   - `docker buildx create --platform linux/arm64 --name native-builds arm-machine`



Native multi-arch builders

3. Append another node (our local builder) for amd64 builds
   - `docker buildx create --append --name native-builds --platform linux/amd64 default`
4. Use the builder to create and push the image
   - `docker buildx build --builder native-builds --platform linux/amd64,linux/arm64 \`
     `-t my-custom-image --push .`



Cache management

- Make the use of various cache backends
  - Great for ephemeral environments or builders (like CI/CD)
- The backends…
  - `inline` - embed the cache into the image itself
  - `registry` - push the cache to a registry
  - `local` - writes the cache to a local directory on the filesystem
  - `gha` - push the build cache to GitHub Actions cache
  - `s3` - uploads the build cache to AWS S3
  - `azblob` - uploads the build cache to Azure Blob Storage
- Cache modes…
  - `min` = only final exported layers are cached (the default)
  - `max` = all intermediate steps are also cached

Buildx cache examples

- Build and push an image, but store max cache in a registry
  - ```
    docker buildx build --push -t my-repo/my-image \
        --cache-to type=registry,ref=my-repo/my-cache,mode=max \
        --cache-from type=registry,ref=my-repo/my-cache
    ```
- Do the same build, but store the cache in a S3 bucket
  - ```
    docker buildx build --push -t my-repo/my-image \
        --cache-to type=s3,region=us-east-1,bucket=my-cache,prefix=buildx/cache \
        --cache-from type=s3,region=us-east-1,bucket=my-cache,prefix=buildx/cache
    ```

GitHub Action example

- This action will checkout the code, setup Buildx, login to Hub, build and push the image, and use the GitHub Actions for the build cache

```yaml
jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3
      - name: Login to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build and push
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: user/app:latest
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

Frontends/Syntax

- Buildx supports multiple “frontends” for an image build
  - The frontend converts the file into steps
  - Provides support to test new features
- The frontend is specified using a comment at the top of the file

```
# syntax=docker/dockerfile:1-labs
FROM ubuntu
RUN --security=insecure cat /proc/self/status | grep CapEff
```

Exporters

- Provide the ability to adjust the output type of a build
- Supported exporters include:
  - `image` - exports to a container image
  - `registry` - exports to a container image and pushes to a registry
  - `local` - exports the build's root filesystem into a local directory
  - `tar` - exports the build's root filesystem into a local tarball
  - `oci` - exports to local filesystem in OCI image layout format
  - `docker` - exports to local filesystem in Docker Image Spec v1.2.0 format
  - `cacheonly` - doesn't export a build, but runs the build and creates cache

docker

Advanced builds using bake

Docker Bake

- Higher level build orchestration
- Codify your build commands
- Publish different variants of your images or build several linked projects in parallel
- Uses orchestration file
  - HashiCorp Configuration Language (HCL)
  - JSON
  - YAML (Compose file)

Complex Docker buildx command without Bake

```
docker buildx build \
   --push \
   --cache-from "type=registry,ref=foo/myapp" \
   --cache-to "type=inline" \
   --platform "linux/amd64,linux/arm/v6,linux/arm/v7,linux/arm64" \
   --label "org.opencontainers.image.title=myapp" \
   --label "org.opencontainers.image.source=https://github.com/foo/myapp" \
   --label "org.opencontainers.image.version=1.0.0" \
   --label "org.opencontainers.image.licenses=Apache-2.0" \
   --tag "foo/myapp:v1.0.0" \
   --tag "foo/myapp:latest" \
   --file "./main.Dockerfile" \
   .
```

Complex Docker buildx command with Bake

```
{
  target: {
    "docker-metadata-action": {
      output = ["type=registry"]
      cache-from = ["type=registry,ref=foo/myapp"]
      cache-to = ["type=inline"]
      dockerfile = "./main.Dockerfile"
      platforms = [“linux/amd64”,”linux/arm/v6”,”linux/arm/v7”,”linux/arm64”]
      labels: {
           "org.opencontainers.image.title=myapp",
           "org.opencontainers.image.source=https://github.com/foo/myapp",
           "org.opencontainers.image.version=1.0.0",
           "org.opencontainers.image.licenses=Apache-2.0”
        },
      tags: [
           "foo/myapp:v1.0.0",
           "foo/myapp:latest"
        ],
    }
  }
}
```

Docker Bake - parallel execution

```
group "default" {
  targets = ["app", "db", "cron"]
}

target "app" {
  dockerfile = "Dockerfile.app"
  platforms = ["linux/amd64", "linux/arm64"]
  tags = ["repo/app:test"]
}

target "db" {
  dockerfile = "Dockerfile.db"
  platforms = ["linux/amd64", "linux/arm64"]
  tags = ["repo/db:test"]
}

target "cron" {
  dockerfile = "Dockerfile.cron"
  platforms = ["linux/amd64", "linux/arm64"]
  tags = ["repo/cron:test"]
}
```

Docker Bake - Inherit attributes

```
target "app-dev" {
  args = {
    GO_VERSION = "1.20"
    BUILDX_EXPERIMENTAL = 1
  }
  tags = ["docker.io/username/myapp"]
  dockerfile = "app.Dockerfile"
  labels = {
    "org.opencontainers.image.source" = "https://github.com/username/myapp"
  }
}

target "_release" {
  args = {
    BUILDKIT_CONTEXT_KEEP_GIT_DIR = 1
    BUILDX_EXPERIMENTAL = 0
  }
}

target "app-release" {
  inherits = ["app-dev", "_release"]
  platforms = ["linux/amd64", "linux/arm64"]
}
```

More Docker Bake Goodness

- Use targets as base images with automatic image dependency resolution
- Variable Block or command line variables
- Call directly from CI/CD tools

docker

Best Practices for Images and Builds

“Do”

node:<tag>

USER node

MEM LIMIT

HEALTHCHECK

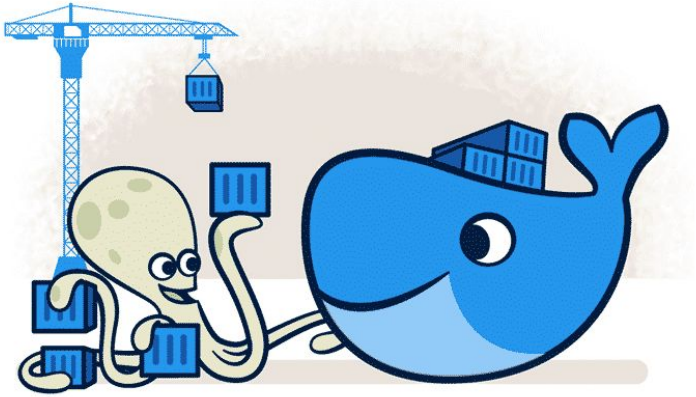
SCANNING

COPY

Multi-Stage

.dockerignore

buildx –platform



“Don’t”

node:latest

ADD

USER root

EXPOSE db_port

node_modules

SIGTERM

“Do”

“Don’t”

Don’t mid-air refuel

Use a vault for secrets

General Tips

- Each container should do one thing and do it well
  - Avoid creating “general purpose” images
- Don’t ship dev tooling into production
  - Multi-stage image builds help tremendously here!
- Use or create trusted base images
  - Can leverage our Docker Official Images or build your own
- Pin your images (don’t use the default `:latest` tag)
  - `FROM node` -> `FROM node:20.5-alpine3.17`
  - Can pin to specific SHA sums too! `FROM node@sha256:0b889cbf7…`

Check out the new [“Build with Docker”]() guide for more build tips!

Separate app setup from app code

- Install dependencies separate from the app code
- Each of the dependency layers can then be reused

```
FROM node
COPY . .
RUN npm install
EXPOSE 3000
CMD ["node", "src/index.js"]
```

→

```
FROM node
COPY package.json package-lock.json .
RUN npm install
COPY src ./src
EXPOSE 3000
CMD ["node", "src/index.js"]
```

Advanced tip - using `--link`

- Indicates a layer is completely independent of other layers
  - Cache busting of previous layers can still allow reuse
- Only works for **`COPY`** and **`ADD`** commands

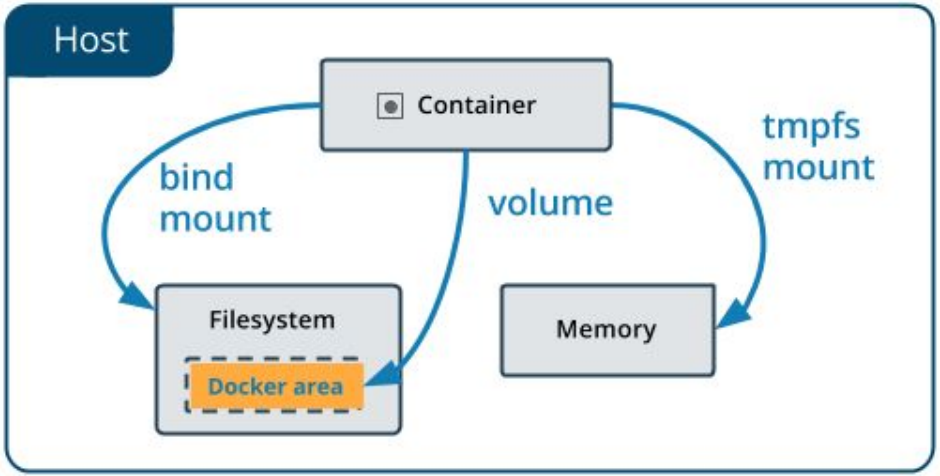
```
FROM node
COPY package.json package-lock.json .
RUN npm install
COPY src ./src
EXPOSE 3000
CMD ["node", "src/index.js"]
```

→

```
FROM node
COPY package.json package-lock.json .
RUN npm install
COPY --link src ./src
EXPOSE 3000
CMD ["node", "src/index.js"]
```

Volumes

Volumes help store data outside running container. This helps your image stay small and avoids bringing unnecessary or risky data on its journey to production.



Identifying problems

- Make a small change to your app and watch how many image layers are rebuilt
- In this example, a single file was changed. How many layers are reused?
  - Every code change reships dependencies

```
⌥⌘2                    root@ca55e6f6a1ec: /

❯ docker build -t node-demo .
[+] Building 1.6s (9/9) FINISHED                    docker:desktop-linux
 => [internal] load build definition from Dockerfile                0.0s
 => => transferring dockerfile: 128B                                0.0s
 => [internal] load .dockerignore                                   0.0s
 => => transferring context: 2B                                     0.0s
 => [internal] load metadata for docker.io/library/node:alpine      0.5s
 => [1/4] FROM docker.io/library/node:alpine@sha256:77516e190b361   0.0s
 => [internal] load build context                                   0.0s
 => => transferring context: 28.22kB                                0.0s
 => CACHED [2/4] WORKDIR /usr/local/app                             0.0s
 => [3/4] COPY . .                                                  0.1s
 => [4/4] RUN npm install                                           0.9s
 => exporting to image                                              0.0s
 => => exporting layers                                             0.0s
 => => writing image sha256:5762d7ddb66912676329cf0ed5af416f5bd0c   0.0s
 => => naming to docker.io/library/node-demo                        0.0s
```

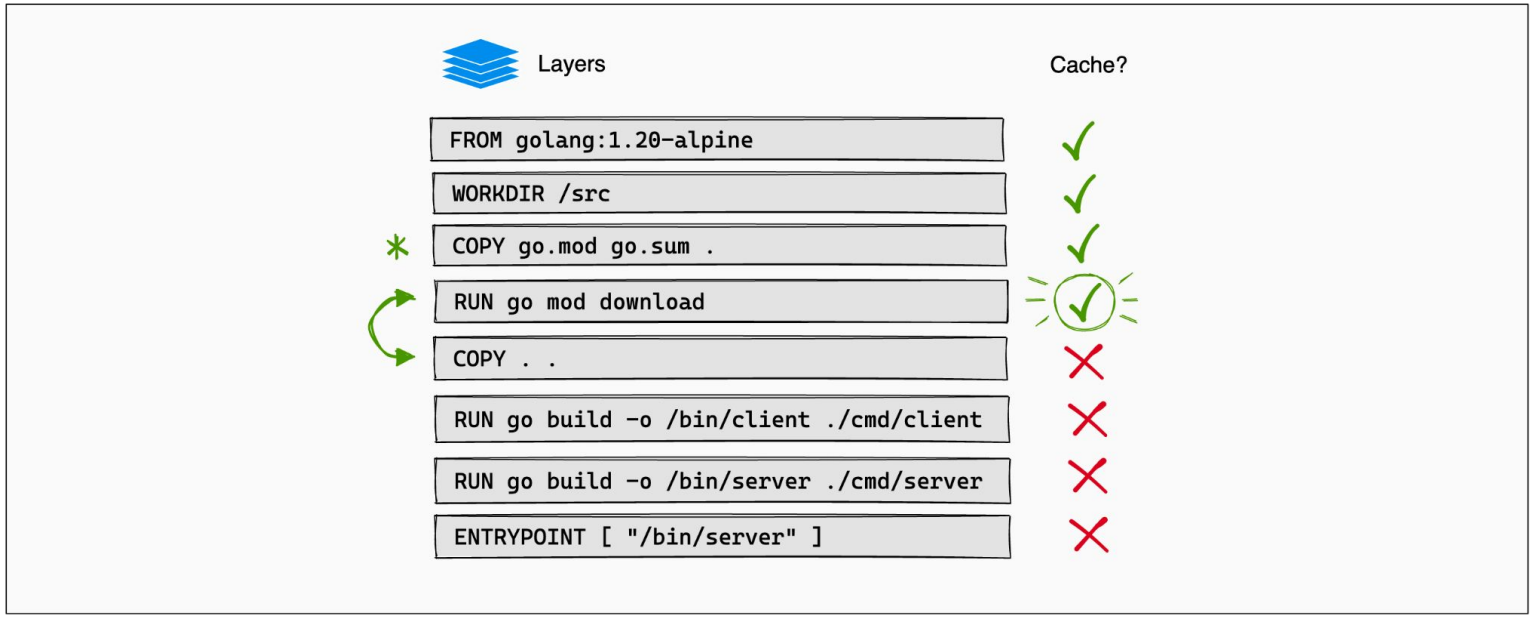
Layers

Using layers properly means you're leveraging the build cache.

Layers	Cache?
`FROM golang:1.20-alpine`	✓
`WORKDIR /src`	✓
`COPY . .`	✗
`RUN go mod download`	✗
`RUN go build -o /bin/client ./cmd/client`	✗
`RUN go build -o /bin/server ./cmd/server`	✗
`ENTRYPOINT [ "/bin/server" ]`	✗

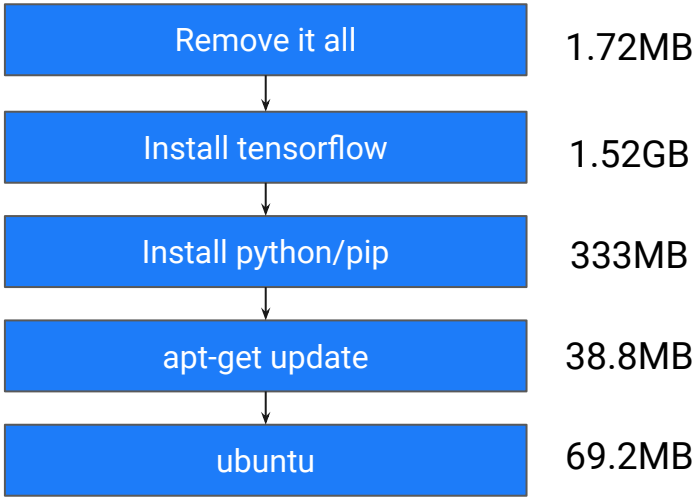
Layers

Re-order your commands to better leverage cache.



Looking at a Dockerfile

```
FROM ubuntu
RUN apt update
RUN apt install -y python3 python3-pip
RUN pip install tensorflow
RUN apt autoremove --purge -y python3-pip
```



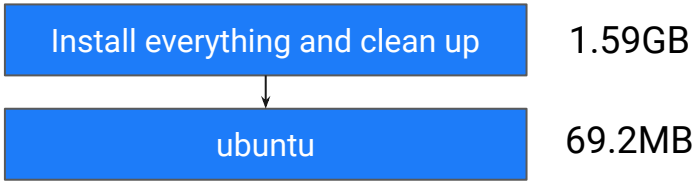
When we ship this image, we are sending the layer that contains the apt repo cache, the full pip install, and the marker files to indicate we deleted those things.

Tip: deleting files does NOT affect previous layers, but only what a running container will see

Cleaning up as we go

- Chain commands into a single **RUN** to combine file changes
- Clean up repo caches after updating/installing

```
FROM ubuntu
RUN apt update && \
    apt install -y python3 python3-pip && \
    pip install tensorflow && \
    apt autoremove --purge -y python3-pip
```



Net image reduction of 1.96GB to 1.66GB (15% savings)

The build cache

- When building an image, Docker tries to reuse layers
- Cache rules...
  - `ADD`/`COPY` instructions - a checksum of the contents/metadata is created. If the checksum has changed, the cache is invalidated
  - If a `RUN` command changes, the cache is invalidated
  - Any new/removed instructions will invalidate caches
- If the cache is invalidated, that layer is rebuilt, as well as all child layers

Multi-stage builds

- Multi-stage allows you to create a pipeline within a Dockerfile
- Provides the ability to separate build-time and run-time images
- Use `COPY --from=<stage>` to copy from previous stages
- The last stage is the default target (can be overridden)

```
FROM <image> AS stage1
COPY ...
RUN ...
...

FROM <image2> AS stage2
COPY --from=stage1 /path/in/stage1 /path/in/stage2
```

A React example

- Use Node to build the React code
- Copy the HTML/CSS/JS into a static web server

```
FROM node:lts AS build
WORKDIR /usr/local/app
COPY package.json package-lock.json ./
RUN npm install
COPY public ./public
COPY src ./src
RUN npm run build


FROM nginx:alpine
COPY nginx.conf /etc/nginx/nginx.conf
COPY --from=build /usr/local/app/build /usr/share/nginx/html
```

Using multi-stage in dev

- Create additional stages for dev and target them in Compose
- Stages can start/extend from other stages
- No more need for separate Dockerfiles for dev vs prod!

```dockerfile
FROM node:lts AS base
WORKDIR /usr/local/app

FROM base AS dev
CMD ["npm", "run"]

FROM base AS build
COPY package.json package-lock.json ./
RUN npm install
COPY public ./public
COPY src ./src
RUN npm run build


FROM nginx:alpine
COPY nginx.conf /etc/nginx/nginx.conf
COPY --from=build /usr/local/app/build /usr/share/nginx/html
```

```yaml
services:
  client:
    build:
      context: .
      target: dev
    volumes:
      - ./:/usr/local/app
    ports:
      - 3000:3000
```

docker

Questions and Answers