BuildKit: A Modern Builder Toolkit on Top of containerd

Tõnis Tiigi, Docker
Akihiro Suda, NTT
About us

Tonis Tiigi
@tonistiigi
Software engineer
Docker Inc.
Maintainer of BuildKit and Moby/Docker

Akihiro Suda
@_AkihiroSuda_
Software engineer
NTT Corporation
Maintainer of BuildKit, containerd, Moby...
What is BuildKit?
How are container images built?

Dockerfile

> docker build .

Bundled into Docker daemon
What’s the issue with old builder?

- Old design/codebase
- Tightly modeled after Dockerfile instructions
- Hard to add new (Dockerfile) features
- Suboptimal performance
- Leaks state to other Docker APIs
- Not usable for other projects
BuildKit solves these problems

- Dozens on new features and bugfixes
- Much faster
- Language agnostic
- Componentized
- Toolkit for building opinionated builders
Built on containerd

containerd - An open and reliable container runtime

- Snapshotters
- Distribution
- Blobs storage
- GC
Embraces OCI standards

OCI - Open Container Initiative

- Process execution with OCI Runtime specification
- Build results can be exported with OCI Image specification (including manifest lists)
Part 2
BuildKit Innovations
Problems of legacy docker build

- The legacy docker build does not compute dependencies across Dockerfile instructions correctly

- Modifying line $N$ always invalidates the cache for line $(N+1)$

```bash
FROM debian
EXPOSE 80
RUN apt update && apt install -y HEAVY-PACKAGES
```
Problems of legacy docker build

FROM golang AS stage0
...
RUN go build -o /foo ...

FROM clang AS stage1
...
RUN clang -o /bar ...

FROM debian AS stage2
EXPOSE 80
RUN apt ...
COPY --from=stage0 /foo /
COPY --from=stage1 /bar /

Expected schedule

Actual
● LLB is to Dockerfile what LLVM IR is to C

● Accurate dependency expression with graph structure
  ○ Efficient caching
  ○ Concurrent execution

● Encoded in protobuf; typically compiled from Dockerfile
  ○ Other “frontends” are also available:
    Buildpacks, Mockerfile, Gockerfile, Docker Assemble
FROM golang AS stage0
...
RUN go build -o /foo ...

FROM clang AS stage1
...
RUN clang -o /bar ...

FROM debian AS stage2
EXPOSE 80
RUN apt ...
COPY --from=stage0 /foo /
COPY --from=stage1 /bar /

Note: No “ExposeOp”
BuildKit

Performance example

Based on github.com/moby/moby Dockerfile, master branch. Smaller is better.

Time for full build from empty state

<table>
<thead>
<tr>
<th>Version</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>v18.03</td>
<td>0:05:42</td>
</tr>
<tr>
<td>BuildKit</td>
<td>0:02:50</td>
</tr>
</tbody>
</table>

2.0x faster

Measured on DO 4vcpu droplet
Extensible syntax

- “LLB frontend” container can be specified in the first line of Dockerfile (# syntax = ...)
- You can also create your own LLB frontend container i.e. you can define your own syntax

```bash
# syntax = docker/dockerfile:1.1-experimental
FROM ...
RUN ...
```
RUN --mount=type=cache

- Allows preserving caches of compilers and package managers

```bash
# syntax = docker/dockerfile:1.1-experimental
...
RUN --mount=type=cache,target=/root/.cache go build
...```
Dockerfile syntax directive

Example: RUN --mount

moby/buildkit Dockerfile: time to binary rebuild after code change

- v18.03: 139.3 sec
- v18.03 + commented out all unused commands: 41.4 sec
- BuildKit: 31.8 sec
- BuildKit + RUN --mount: 3.9 sec

**33.1x slower**

**10.6x slower**

**8.1x slower**
RUN --mount=type=secret

- Allows accessing private assets without leaking credential in the image

```
# syntax = docker/dockerfile:1.1-experimental
...
RUN --mount=type=secret,id=aws,target=/root/.aws/credentials \
    aws s3 cp s3://... ... 

$ buildctl build --secret id=aws,src=~/.aws/credentials ...
```
Note: DON’T do this!

COPY my_aws_credentials /root/.aws/credentials
RUN aws s3 cp s3://... ...
RUN rm -f /root/.aws/credentials
...
RUN --mount=type=secret

- Note: DON’T do this either!

$ docker build \n   --build-arg \n   MY_AWS_CREDENTIALS=$(cat ~/.aws/credentials)
Part 3
Using BuildKit
Many ways to use BuildKit

- Docker, docker buildx
- img
- Tekton
- Rio

With or without daemon, in container, in k8s, with containerd daemon, without root privileges, etc.
Docker

- Integrated into "docker build" v18.09+
- Opt-in:

  ```
  export DOCKER_BUILDKIT=1
  ```
Docker

```bash
# docker build .
```
Docker Buildx

- Next generation Build command from Docker
- Familiar Docker UI + full BuildKit
- Manages instances of Builders and Build nodes
- With container driver, works with any version of Docker engine
Buildx: Full BuildKit

- Remote caching (eg. for CI)
- Multi-platform images support
  - `--platform=linux/amd64,linux/arm64`
  - QEMU, distributed among nodes, or cross-compilation in multi-stage Dockerfile
Buildx: Multi-platform images

- **Webassembly**: wasi/wasm
  
  https://github.com/tonistiigi/wasm-cli-plugin

- **Initial RISC-V support**: linux/riscv64
  
  https://tinyurl.com/docker-riscv
Part 4
Deploying BuildKit on Kubernetes
Why build images on Kube?

1. CI/CD

Some webhook

Some Pod

BK Pod

BK Pod

BK Pod
Why build images on Kube?

1. CI/CD
   - BK Pod
   - BK Pod
   - BK Pod

2. Developer Experience

Some webhook

poor CPU, RAM, Wi-Fi, battery
The common pattern was to run `docker Pod` with `/var/run/docker.sock hostPath`.

Or run `docker:dind Pod` with `securityContext.privileged`.

Both are insecure.
Rootless mode

- BuildKit can be executed as a non-root user so as to protect the host from potential BuildKit vulns

- No extra `securityContext` configuration needed (but seccomp and AppArmor need to be disabled)
Rootless BuildKit vs Kaniko

- Kaniko runs as the root user but “unprivileged”
  - No need to disable seccomp and AppArmor
- Kaniko might be able to mitigate some vuln that Rootless BuildKit cannot mitigate - and vice versa
  - Rootless BuildKit might be weak against kernel vulns
  - Kaniko might be weak against runc vulns
Deployment strategy

Deployment?

DaemonSet?

StatefulSet?

Job?
Deployment strategy

- **Deployment**
  - Most typical deployment

- **DaemonSet**
  - Optimal load-balancing but non-optimal caching

- **StatefulSet**
  - Good for Consistent Hashing (discussed later)

- **Job** (client and ephemeral daemon in a single container)
  - No need to manage the life cycles of the daemons
Caching

Load-balancing component
(Can be just headless svc with DNSRR)

Client -> Service

registry

buildkitd
buildkitd
buildkitd

Image

Cache
Caching

- Remote cache might be slow compared to the daemon-local cache

- Example:
  - No cache: 2m50s
  - Remote cache: 36s
  - Daemon-local cache: 0.5s
Caching

- Consistent hashing allows sticking a build request to a specific Pod in StatefulSet

- Always hits the cache, but non-optimal load balancing
Recap

- BuildKit is a **modern** container Build toolkit
- **Significant advantages** over previous tools
- Usable with **Docker**, **K8s** and many other tools
- **Open** platform for collaboration around build
Extra slides
Tekton

- CRD for building images
- Successor of Knative Build
apiVersion: tekton.dev/v1alpha1
kind: TaskRun
metadata:
  name: foobar
spec:
  taskRef:
    name: buildkit
  serviceAccount: someServiceAccount

The interface is same as other image builders (Buildah, Kaniko, and Makisu)

Credentials are loaded from the Secret associated with the ServiceAccount
inputs:

resources:
- name: source

resourceSpec:
  type: git

params:
- name: url
  value: git@github.com:foo/bar.git

outputs:

resources:
- name: image

resourceSpec:
  type: image

params:
- name: url
  value: registry.example.com/foo/bar
Rancher Rio

- k8s/k3s-based micro PaaS
- “rio run https://github.com/...” builds and deploy app in one-line
- Internally using BuildKit, but users don’t need to care about BuildKit