Building images efficiently and securely on Kubernetes with BuildKit

Akihiro Suda, NTT

Raise your hand if you have heard of BuildKit?

Raise your hand if you are already using BuildKit?

Raise your hand if you are already running BuildKit on Kubernetes? ☸

Part 1
Introduction to BuildKit

- Concurrent multi-stage build

- Efficient caching

- Secure access to private assets

- Flexible syntax for build definition

- Does not require root privileges

BuildKit: next-generation docker build

- BuildKit is included in Docker since v18.06

```
$ export DOCKER_BUILDKIT=1
$ docker build ...
```

- But this talk will focus on the standalone version of BuildKit (`buildkitd` & `buildctl`)
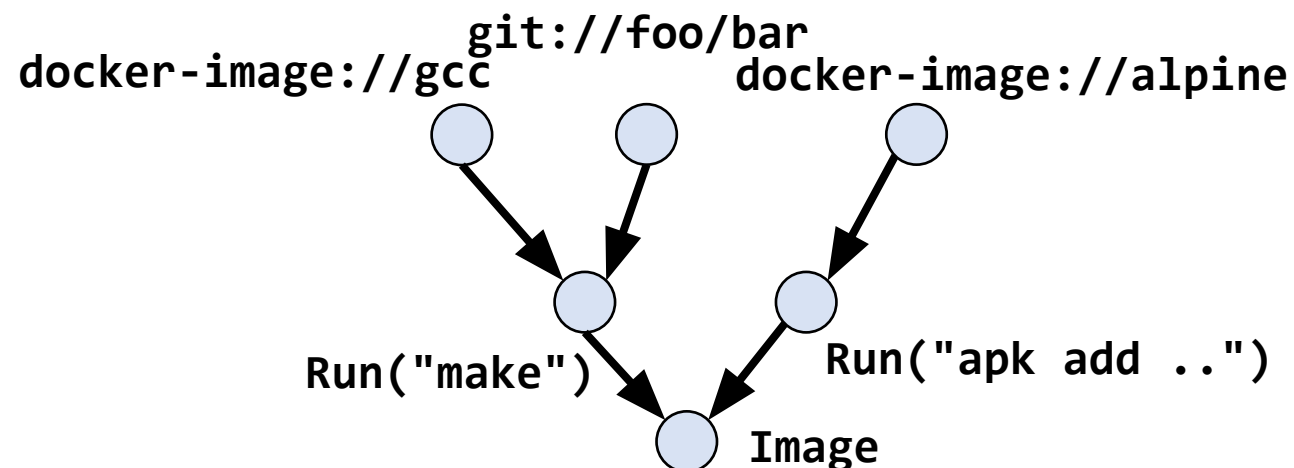 - No dependency on Docker

LLB DAG

- LLB is to Dockerfile what LLVM IR is to C

- Typically compiled from Dockerfile

- Accurate dependency expression with DAG structure
  - Efficient caching
  - Concurrent execution

```
                              git:///foo/bar
docker-image://gcc              docker-image://alpine

        O          O                    O

         Run("make")      Run("apk add ..")

                    Image
```

```
FROM golang AS stage0
...
RUN go build –o /foo ...

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

FROM debian AS stage2
COPY --from=stage0 /foo /usr/local/bin/foo
COPY --from=stage1 /bar /usr/local/bin/bar
```
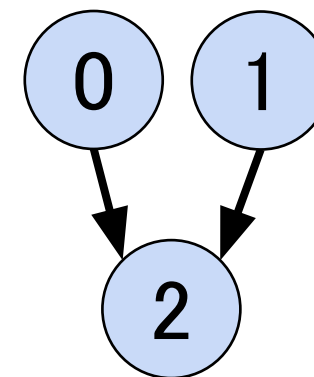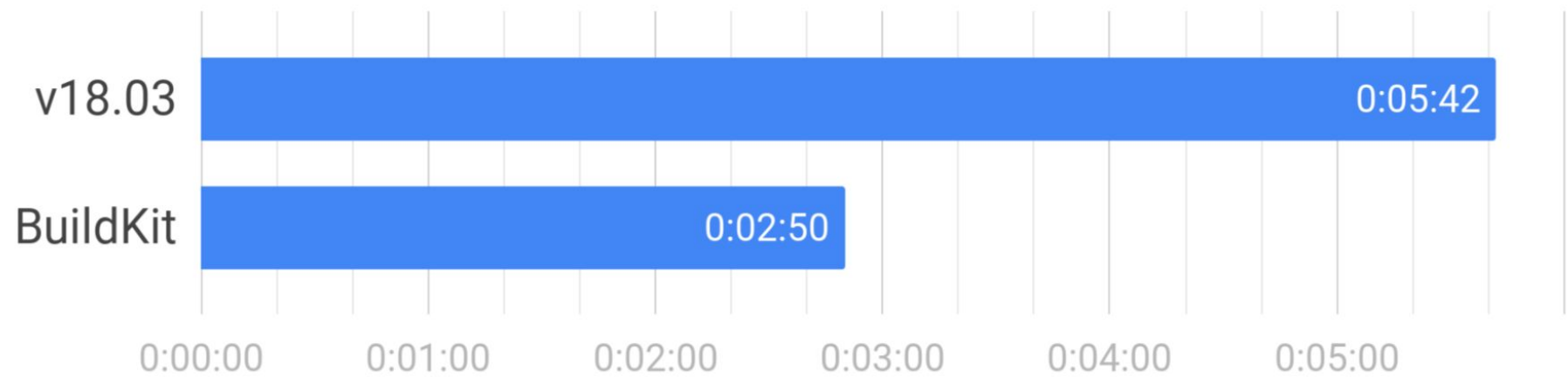
```
 0     1
    \   /
      2
```

BuildKit
Performance example

Based on **github.com/moby/moby** Dockerfile, master branch. **Smaller** is better.
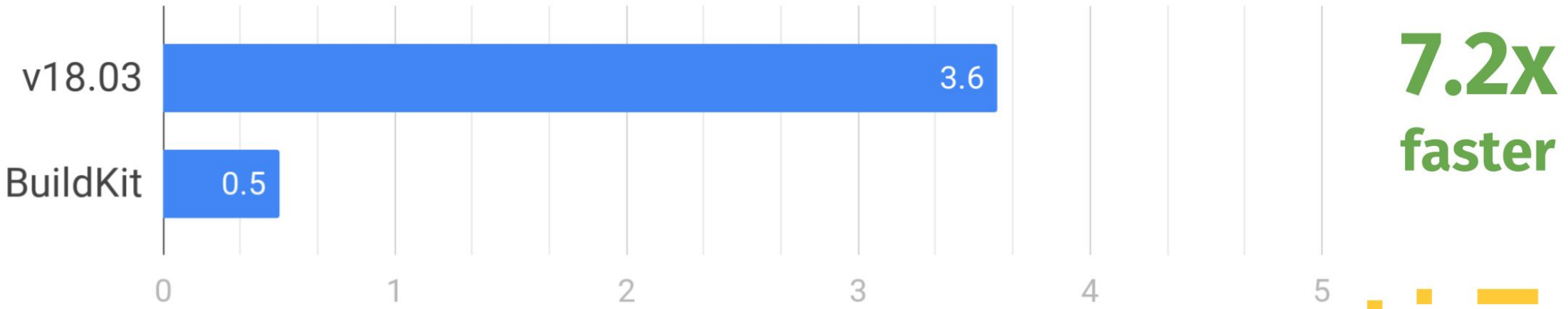
Time for full build from empty state

2.0x
faster

Measured on DO 4vcpu droplet

BuildKit
Performance example

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

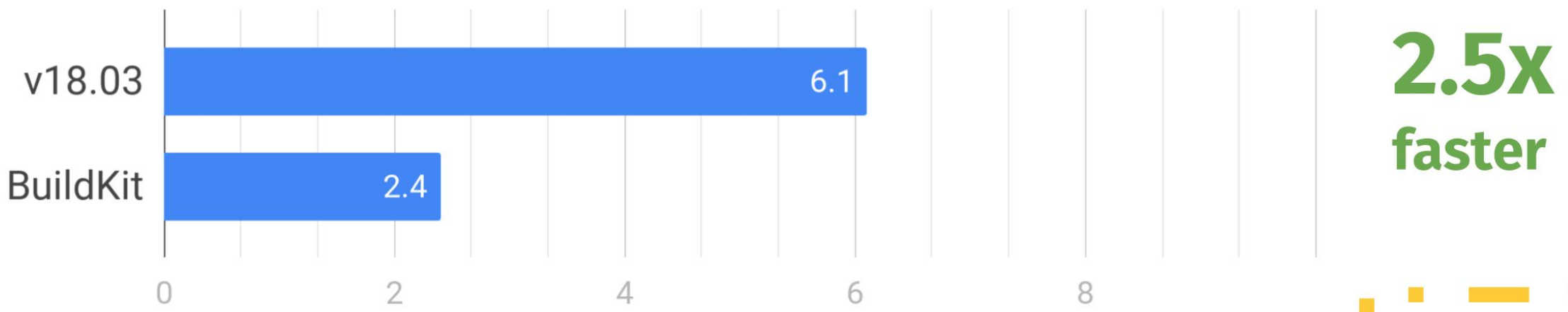## Repeated build with matching cache

v18.03 — 3.6
BuildKit — 0.5

0 1 2 3 4 5

7.2x
faster

Measured on DO 4vcpu droplet

docker con SF 18

BuildKit
Performance example

Based on **github.com/moby/moby** Dockerfile, master branch. **Smaller** is better.
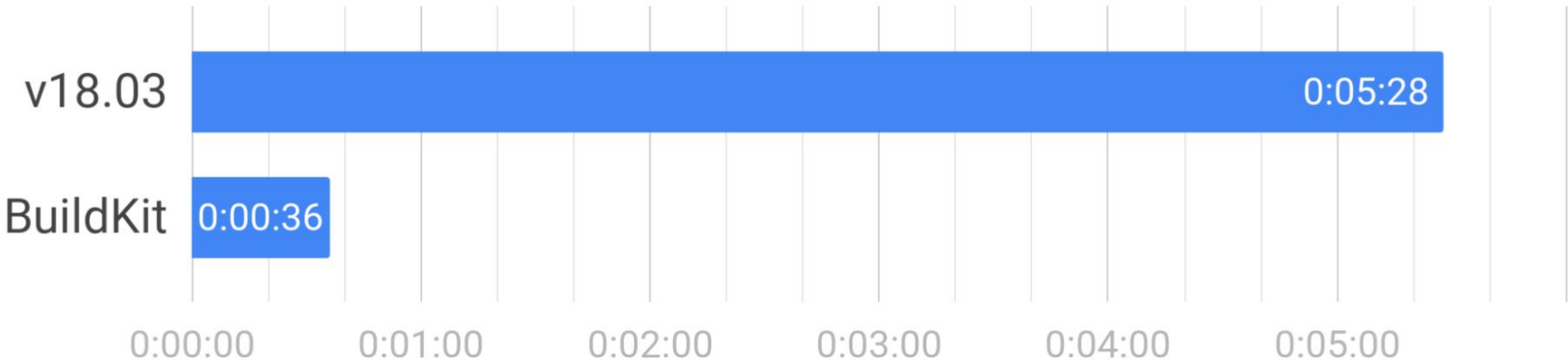
Repeated build with new source code

v18.03 — 6.1

BuildKit — 2.4

2.5x
faster

Measured on DO 4vcpu droplet

BuildKit
Performance example

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

Fresh build with --cache-from from remote source

v18.03   0:05:28

BuildKit   0:00:36

0:00:00   0:01:00   0:02:00   0:03:00   0:04:00   0:05:00

9.1x
faster

Measured on DO 4vcpu droplet

RUN --mount=type=cache

- Allows preserving caches of compilers and package managers

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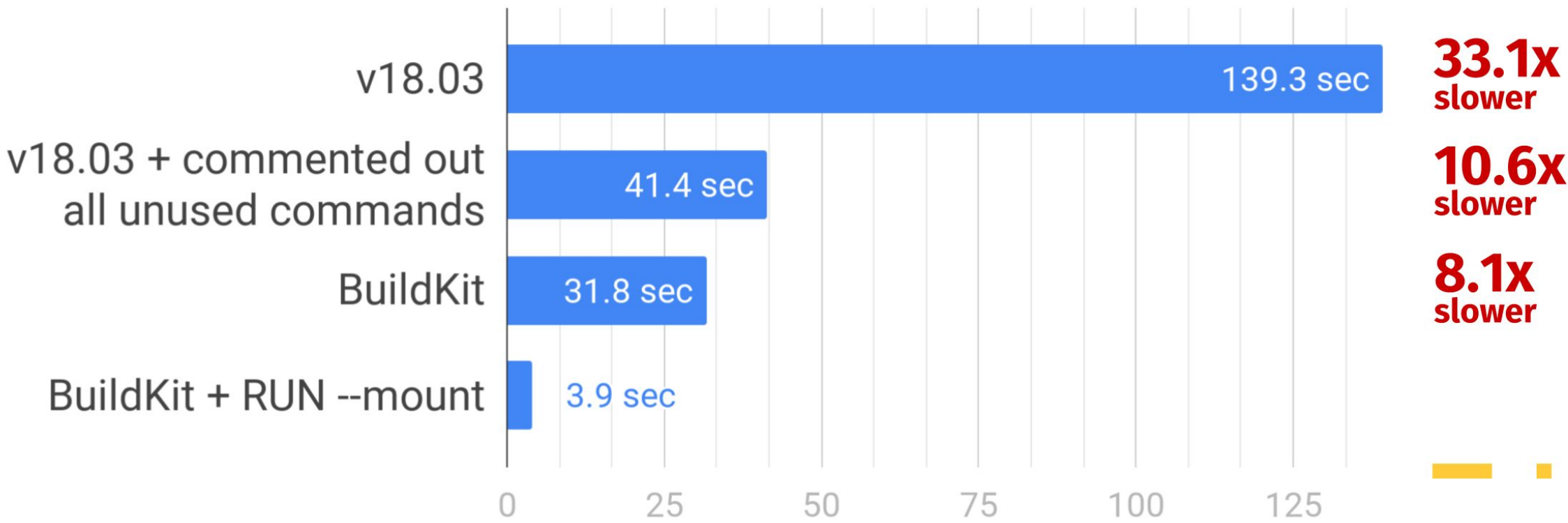
RUN --mount=type=cache

- Allows preserving caches of compilers and package managers

```
# syntax = docker/dockerfile:1.1-experimental
FROM ubuntu
RUN rm -f /etc/apt/apt.conf.d/docker-clean; \
  echo 'Binary::apt::APT::Keep-Downloaded-Packages "true";' > \
  /etc/apt/apt.conf.d/keep-cache
RUN \
  --mount=type=cache,target=/var/cache/apt \
  --mount=type=cache,target=/var/lib/apt \
  apt-get update && apt-get install -y gcc
```

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

Measured on DO 4vcpu droplet

dockercon18 SF

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 ...
```

RUN --mount=type=secret

- 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 \
  --build-arg \
  MY_AWS_CREDENTIALS=$(cat ~/.aws/credentials)
```

RUN --mount=type=ssh

- Akin to `--mount=type=secret` but specific to SSH
- Supports passphrase

```
# syntax = docker/dockerfile:1.1-experimental
...
RUN --mount=type=ssh git clone ssh://github.com/...
```

```
$ eval $(ssh-agent)
$ ssh-add ~/.ssh/id_rsa
(Enter your passphrase)
$ buildctl build --ssh default=$SSH_AUTH_SOCK ...
```

Non-Dockerfiles

- LLB can be also compiled from non-Dockerfiles

- Several languages are being proposed
  - Buildpacks
  - Mockerfile
  - Gockerfile

- You can also create your own language

Buildpacks

- Ported from Heroku/CloudFoundry Buildpacks
- No support for Cloud Native Buildpacks yet

```
# syntax = tonistiigi/pack
---
applications:
- name: myapp
  memory: 128MB
  disk_quota: 256MB
  random-route: true
  buildpack: python_buildpack
  command: python hello.py
```

Mockerfile

- `apt-get` in highly declarative YAML

```
# syntax = r2d4/mocker
apiVersion: v1alpha1
images:
- name: demo
  from: ubuntu:16.04
  package:
    repo:
    - deb [arch=amd64] http://storage.googleapis.com/bazel-apt stable jdk1.8
    gpg:
    - https://bazel.build/bazel-release.pub.gpg
    install:
    - bazel
```

Gockerfile

- Really simple
- Specific to Golang

```
# syntax = po3rin/gocker
repo: github.com/foo/bar
path: ./cmd/baz
version: v0.0.1
```

Part 2
Deploying BuildKit on Kubernetes

Why build images on Kube?

Two kinds of motivation:

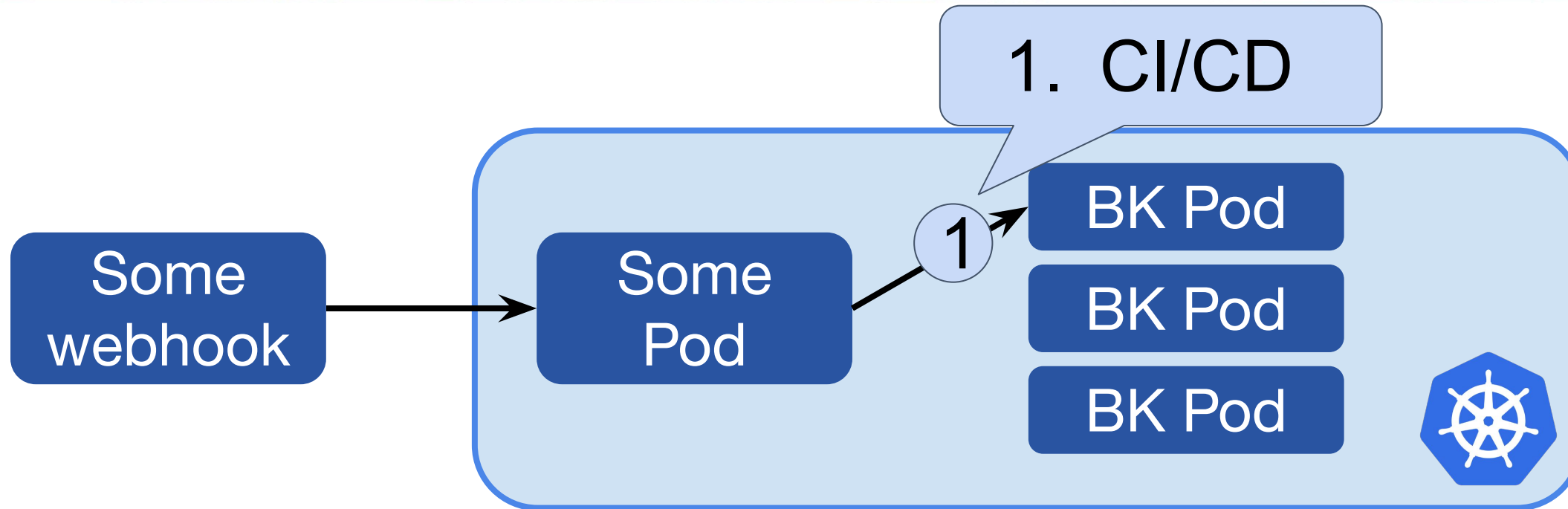1. CI/CD

2. Developer Experience

Why build images on Kube?

1. CI/CD

Some webhook → Some Pod →(1)→ BK Pod / BK Pod / BK Pod

Why build images on Kube?

1. CI/CD

Some webhook → Some Pod

1 → BK Pod
2 → BK Pod
BK Pod

2. Developer Experience

poor CPU, RAM, Wi-FI, battery
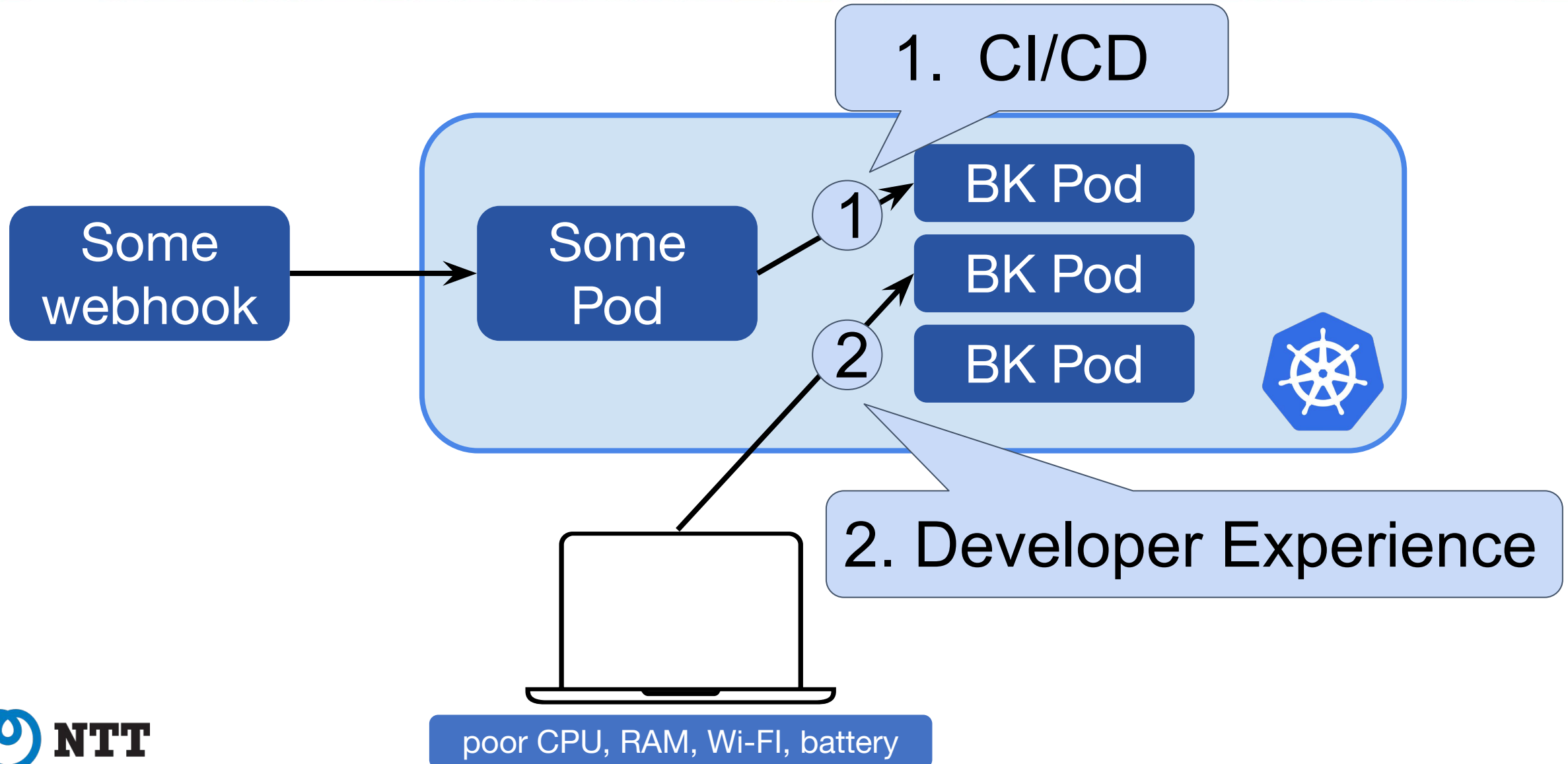
Issue with docker build on Kube

- 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

Part 2.1
Rootless mode

Rootless mode

- BuildKit can be executed as a non-root user

- No extra `securityContext` configuration needed

- Protect the host from potential BuildKit vulns

Demo

myth 1: requires securityContext.privileged

- Not true since BuildKit v0.4.0

- But you need to disable "Process Sandbox":

  launch `buildkitd` with

  `--oci-worker-no-process-sandbox`

  - Disable unsharing PIDNS and mounting `/proc`

    for `RUN` instructions

Process sandbox
(needs to be disabled)

Process sandbox

worker container (e.g. `RUN gcc ...`)

BuildKit daemon
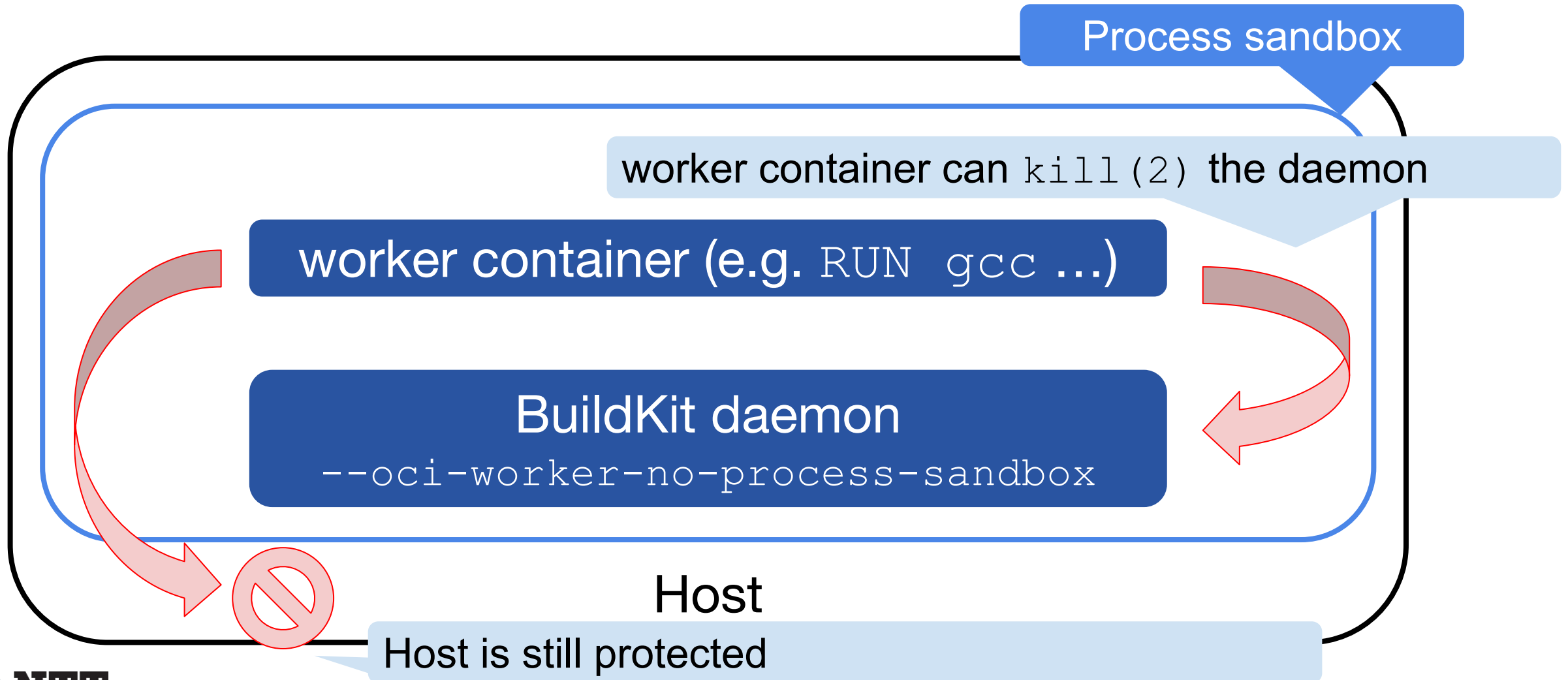
Host

myth 1: requires securityContext.privileged

Process sandbox

worker container can `kill(2)` the daemon

worker container (e.g. `RUN gcc ...`)

BuildKit daemon
`--oci-worker-no-process-sandbox`

Host

Host is still protected

- To enable Process Sandbox,

  `securityContext.procMount` **needs** to be set to

  `Unmasked`

  - Requires Kubernetes v1.12+ with Docker v18.06+ /

    containerd v1.2+ / CRI-O v1.12

myth 2: seccomp and AppArmor need to be disabled

~~myth 2~~: seccomp and AppArmor need to be disabled

- Not a myth :P

- seccomp (and AppArmor) is typically disabled by default on Kubernetes anyway
  - In Kubernetes world, seccomp is still in alpha status and AppArmor is in beta
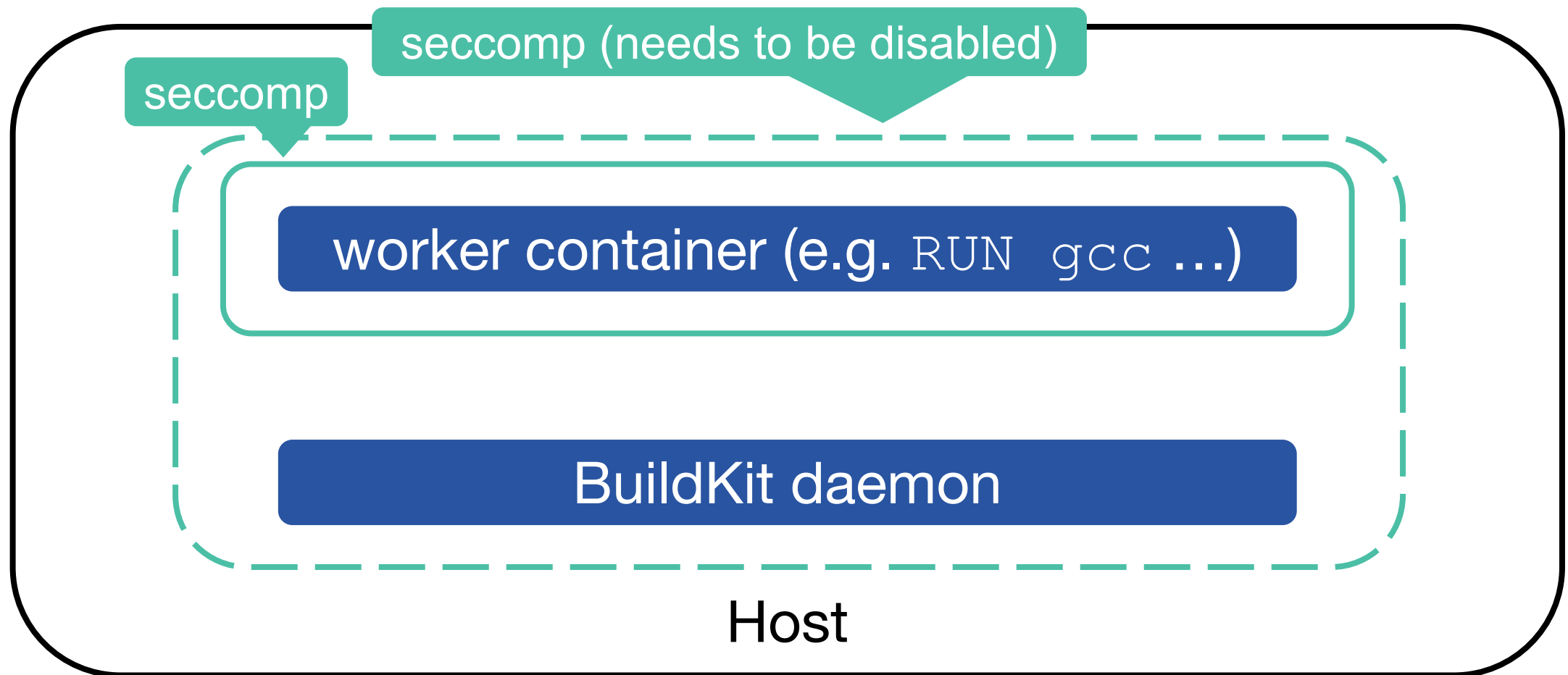
~~myth 2~~: seccomp and AppArmor need to be disabled

seccomp (needs to be disabled)

seccomp

worker container (e.g. `RUN gcc …`)

BuildKit daemon

Host

~~myth 2~~: seccomp and AppArmor need to be disabled

seccomp

worker containers are still protected with seccomp

worker container (e.g. `RUN gcc ...`)

BuildKit daemon

Host

NTT
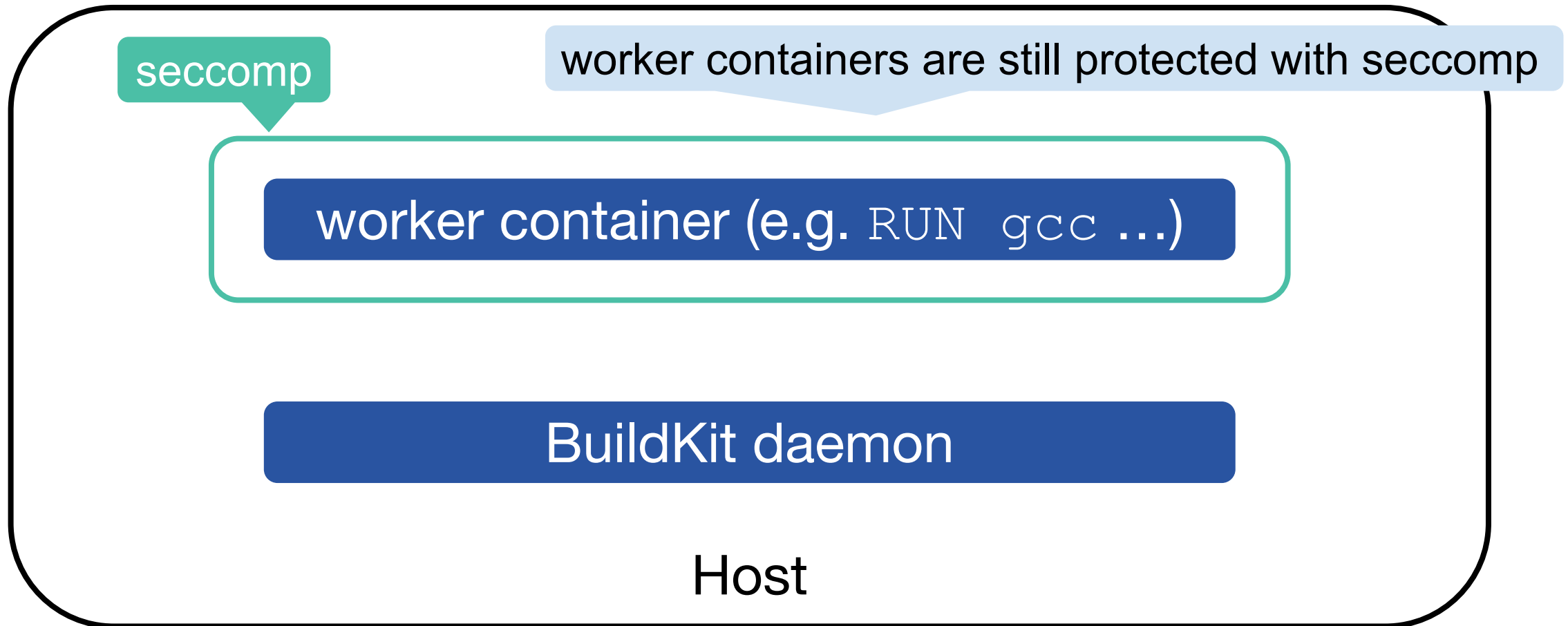
Future work: gVisor integration?
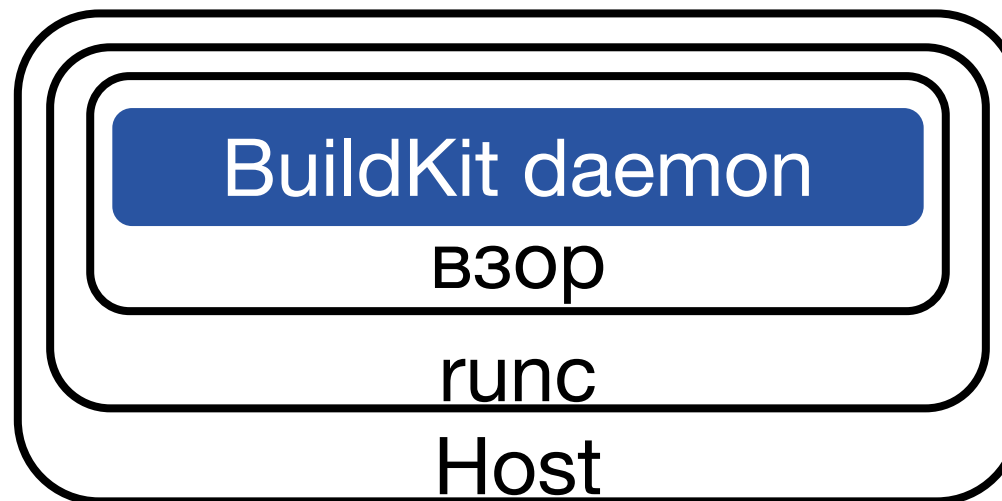
- gVisor: Yet another Linux kernel implementation in userspace

- взор (*vzor*): gVisor-based sandbox for runc containers

  https://github.com/tonistiigi/vzor
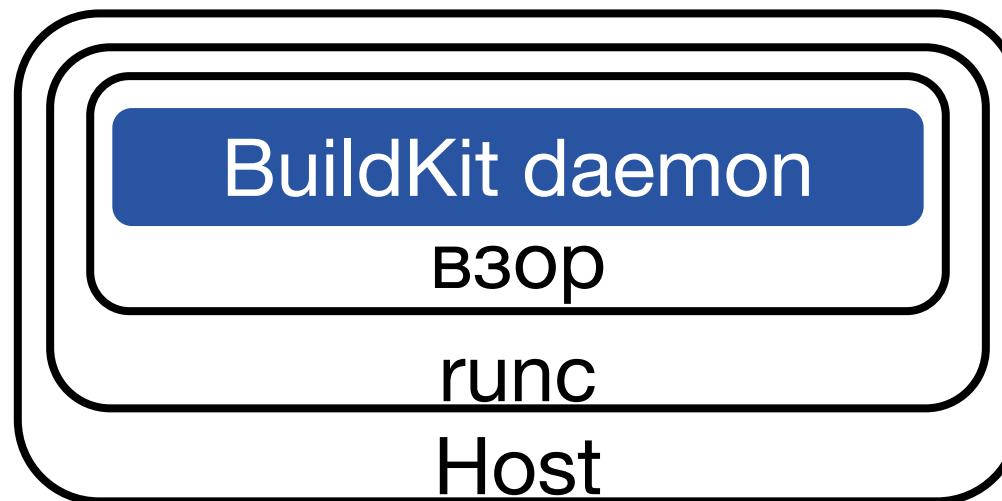
Future work: gVisor integration?

- No need to disable seccomp/AppArmor for runc

- Can also mitigate kernel vulns

```
┌─────────────────────────────────┐
│  ┌───────────────────────────┐  │
│  │  ┌─────────────────────┐  │  │
│  │  │   BuildKit daemon   │  │  │
│  │  └─────────────────────┘  │  │
│  │          взор             │  │
│  └───────────────────────────┘  │
│             runc                │
│             Host                │
└─────────────────────────────────┘
```

Future work: gVisor integration?

- Currently BuildKit fails with `EINVAL` due to syscall incompatibility

- Or User-Mode Linux?
  - Full Linux compatibility
  - 20 yo, still alive :)

Rootless BuildKit vs Kaniko

- Kaniko runs as the root 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

Part 2.2
Deployment strategy

Deployment strategy

Deployment?

DaemonSet?

StatefulSet?

Job?

Deployment strategy

- `Deployment`
  - Most typical deployment

- `DaemonSet`
  - Better `Pod` placement
  - But unlikely to hit daemon-local cache if you have a bunch of replicas
  - So might not be always optimal for large clusters w/ complex Dockerfiles

Deployment strategy

- `StatefulSet`
  - Consistent `Pod` names
  - Good for Consistent Hashing (discussed later)

Deployment strategy

- `Job` ("Daemonless")
  - `buildctl` and ephemeral `buildkitd` in a single container in an ephemeral `Pod`
  - No need to manage the life cycles of the daemons
  - Needs PR: moby/buildkit#1005
    - or github.com/genuinetools/img (lacks some upstream features)

How to connect to BuildKit?

- BuildKit daemon can listen on TCP (with TLS)

- The entire operation (build & push) just needs a single gRPC connection

- So you can create Kubernetes `Service` for connecting to BuildKit `Deployment / DaemonSet / StatefulSet`
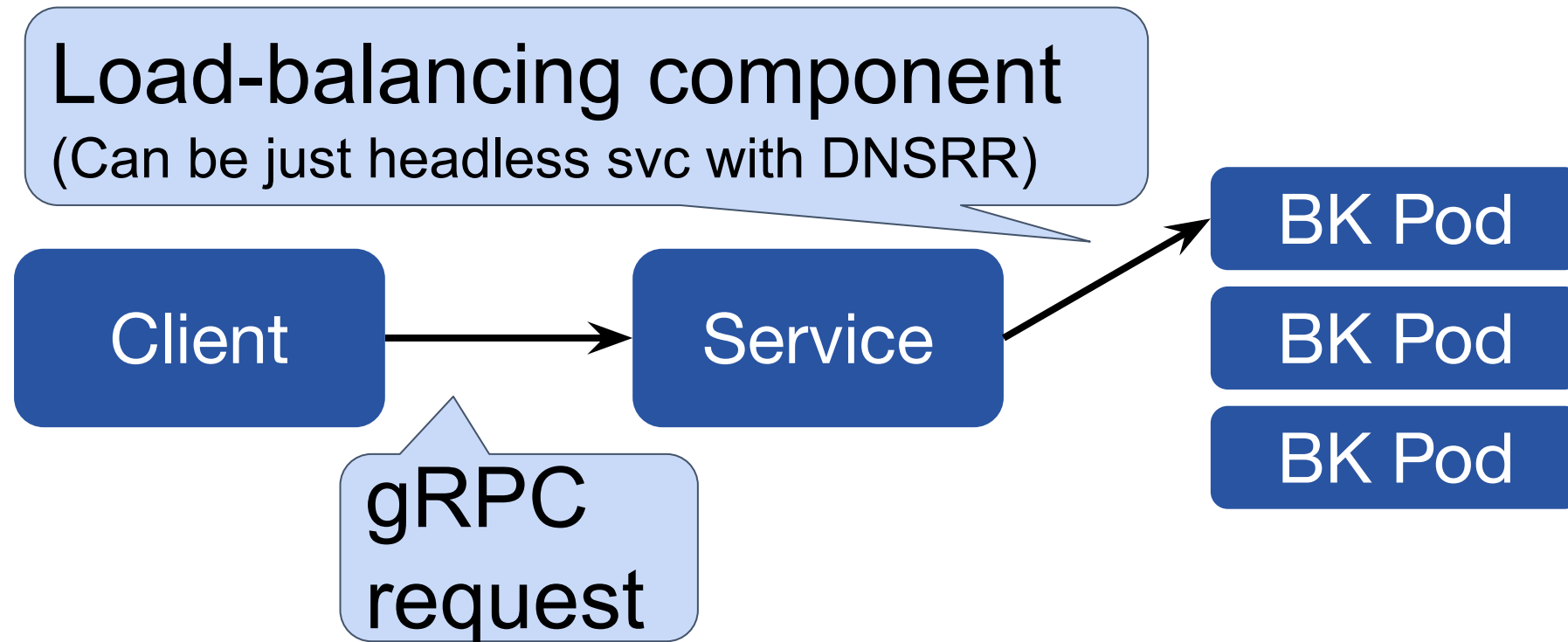
How to connect to BuildKit?

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

Client → Service

gRPC
request

BK Pod

BK Pod

BK Pod

NTT

51

How to connect to BuildKit?

- But you don't need to necessarily create `Service`

- `buildctl` CLI can directly connect to a daemon in a `Pod` without `Service`
  - Internally invokes `kubectl exec`

How to connect to BuildKit?

```
$ kubectl run \
    --generator=run-pod/v1 \
    --image=moby/buildkit:master-rootless \
    bk -- --oci-worker-no-process-sandbox
```

```
$ export BUILDKIT_HOST=kube-pod://bk
$ buildctl build ...
```

- `docker buildx` is the next generation CLI for integrating BuildKit to Docker
  - Supports building multi-arch image with remote ARM machines
  - "Bake": compose-like build

- `docker buildx` will support connecting to BuildKit on Kubernetes in the same UX

Part 2.3
Caching

Remote cache

- Cache can be shared via either registry or shared FS

- Similar to classic `docker build --cache-from` but more chance of hitting cache

- For building non-container artifacts (it's a valid use-case), FS cache might be useful
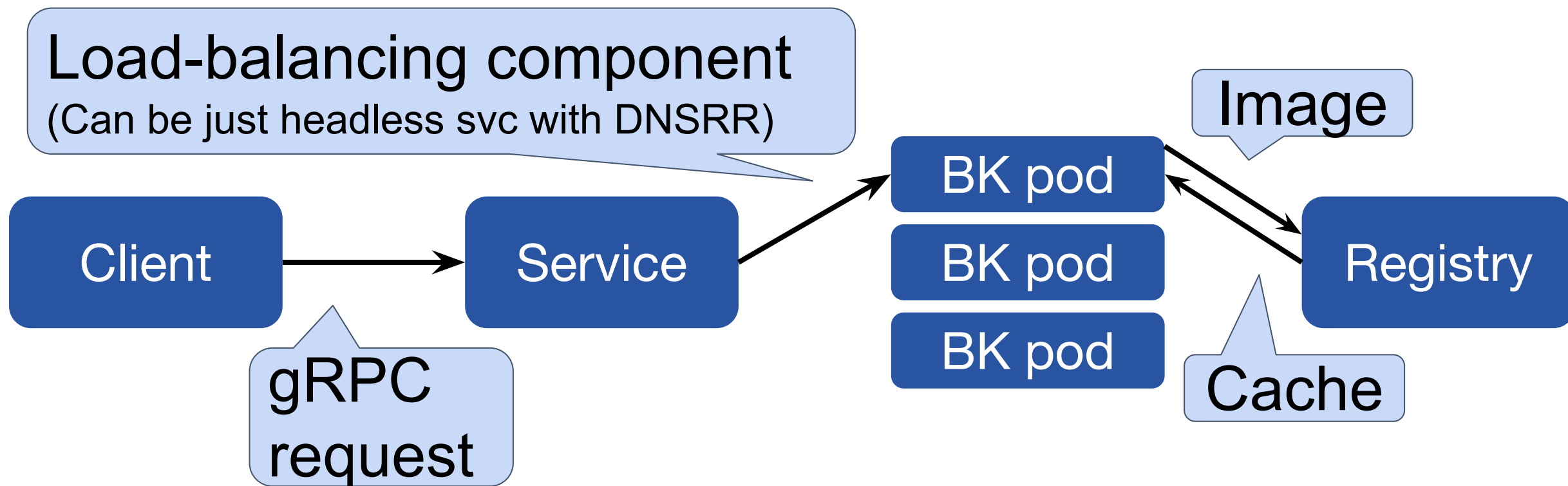
Remote cache

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

Image

gRPC
request

Cache

Client → Service → BK pod / BK pod / BK pod → Registry

Remote cache

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

- Example from Part 1 slides:
  - No cache: 2m50s
  - Remote cache: :   36s
  - Daemon-local cache: 0.5s

Consistent hashing

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

- So the build request can always hit the daemon-local cache in the `Pod`
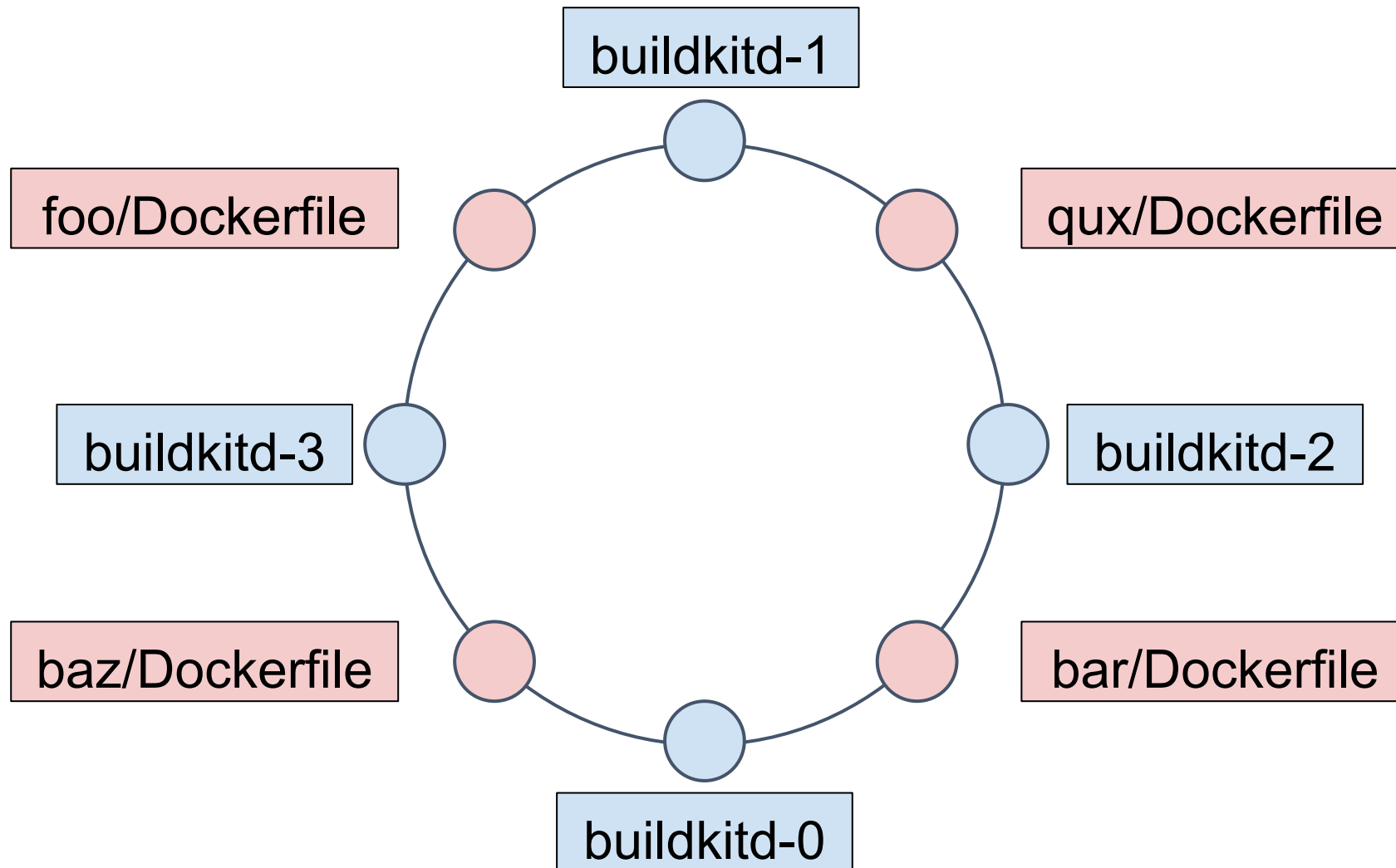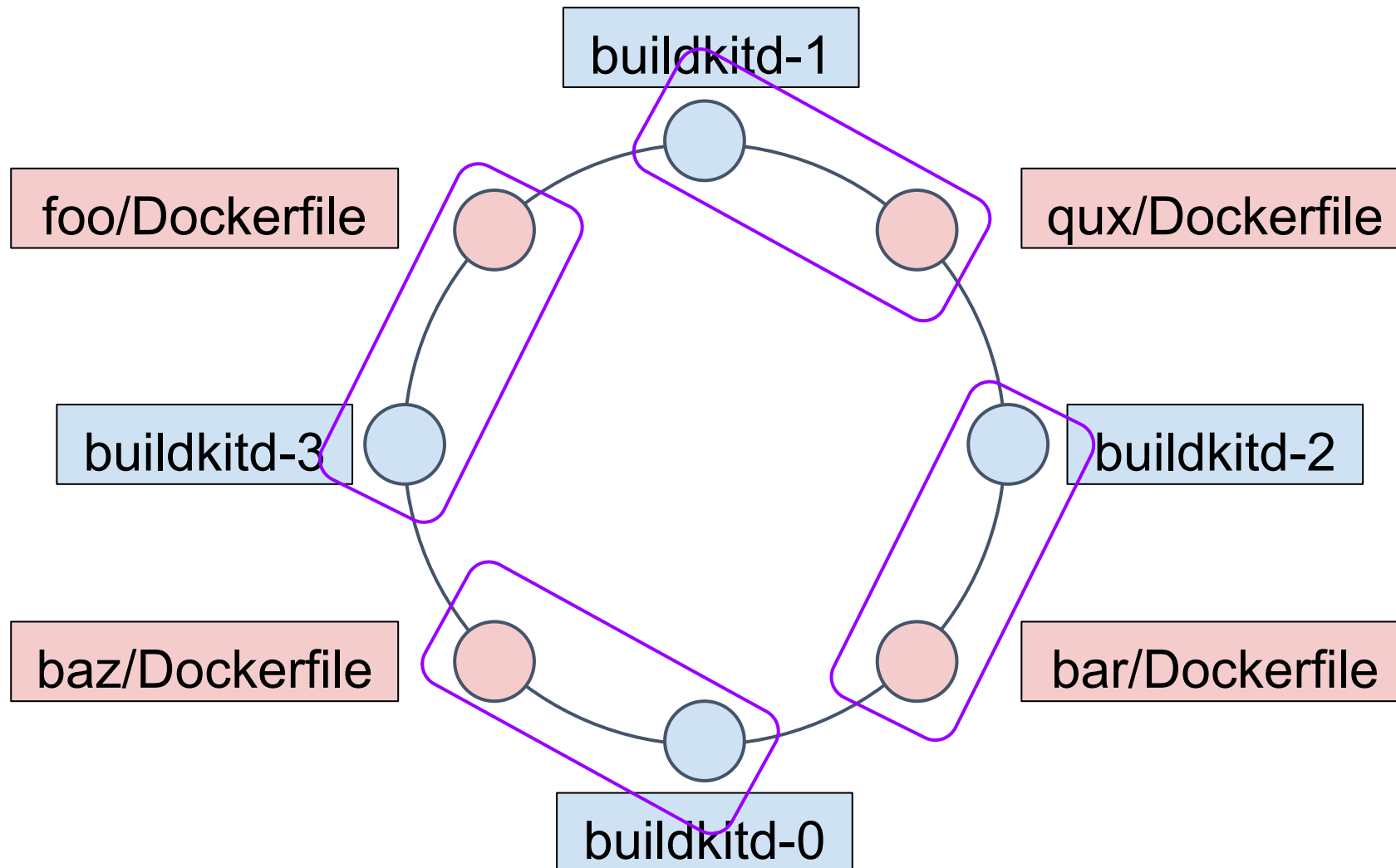
Consistent hashing

Consistent hashing

buildkitd-1

foo/Dockerfile

qux/Dockerfile

buildkitd-3

buildkitd-2

baz/Dockerfile

bar/Dockerfile

buildkitd-0

Consistent hashing

Consistent hashing

buildkitd-1

foo/Dockerfile

qux/Dockerfile

buildkitd-2

baz/Dockerfile

bar/Dockerfile

quux/Dockerfile

buildkitd-0
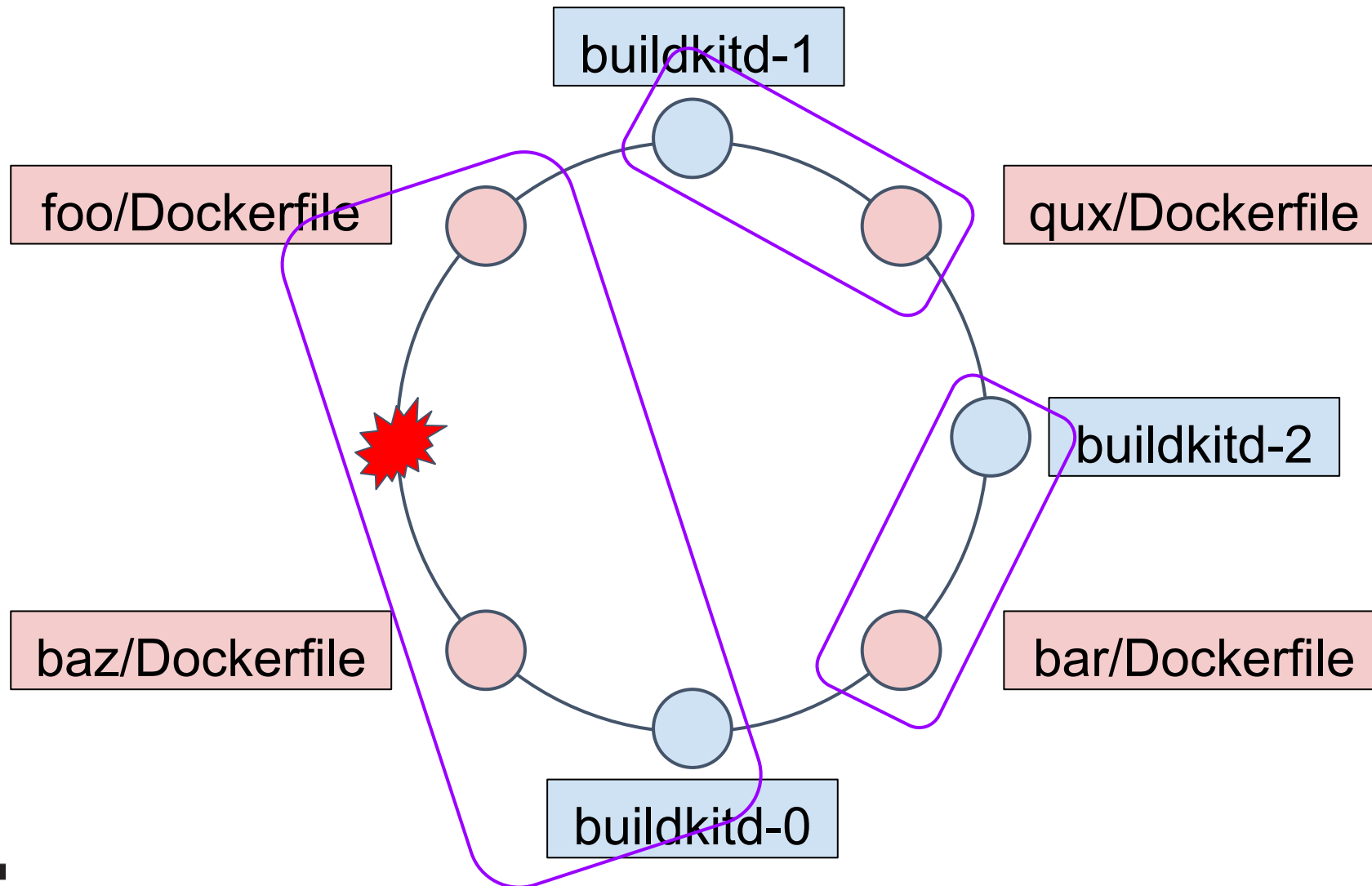
Consistent hashing

- Caveats:
  - High I/O overhead on specific set of nodes
  - Some nodes might not be used at all

- **See** `examples/kube-consistent-hashing` **in the** `moby/buildkit` **repo**

Remote cache vs Consistent hashing?

- If your cache registry is fast enough for your Dockerfiles, remote cache w/ load-balancing might be better

- If you don't like transferring cache, consistent hashing might be better

Part 2.4
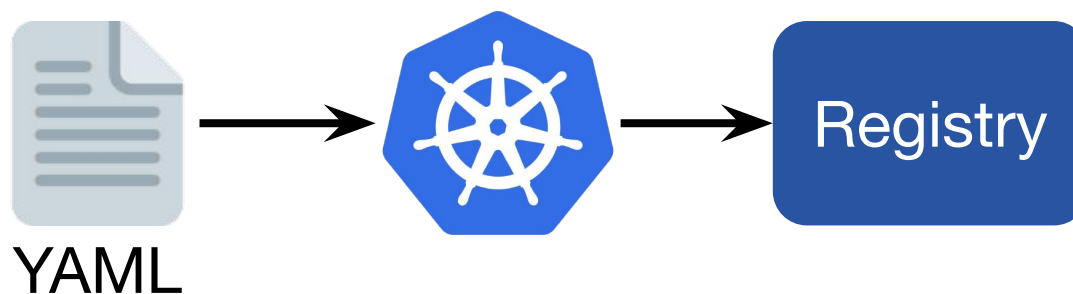# CRD

YAMLIFY ALL THE THINGS

YAML → Registry

CRD

Container Builder Interface (CBI)

- The first common build CRD

- Supports Docker, BuildKit, Buildah, kaniko, img, Google Cloud Container Builder, Azure Container Registry Build, and OpenShift S2I

- Complex design with a bunch of microservices

- Now being deprecated
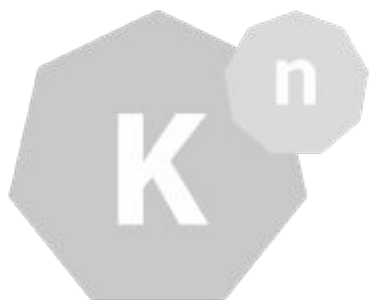
CRD

Container Builder Interface (CBI)

- Simpler than CBI and easily extensible
- The build component (not entire Knative) might be going to be replaced by Tekton?

- Spun out from Knative
- Much more simple and extensible

CRD

Container Builder Interface (CBI)

- Simpler than CBI and easily extensible
- The build component (not entire Knative) might be going to be replaced by Tekton?

- Spun out from Knative
- Much more simple and extensible

Tekton

```
apiVersion: tekton.dev/v1alpha1

kind: TaskRun

metadata:

  name: foobar

spec:

  taskRef:

    name: buildkit

...
```

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

Tekton

```
inputs:

  resources:

  - name: source

    resourceSpec:

      type: git

      params:

      - name: url

        value: git@github.com:foo/bar.git
```

SSH credential is loaded from the `Secret` associated with the `ServiceAccount`

Tekton

```
outputs:

  resources:

  - name: image

    resourceSpec:

      type: image

      params:

      - name: url

        value: registry.example.com/foo/bar
```

Registry credential is loaded from the `Secret` associated with the `ServiceAccount`
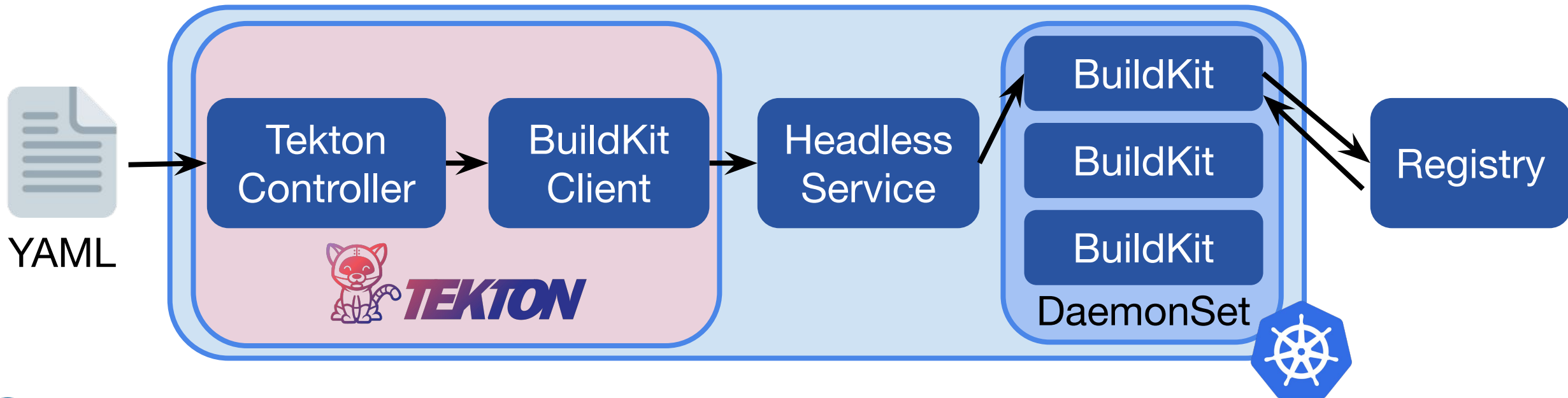
Wrap-up

- BuildKit is fast and secure
- Several deployment plans, w/ and w/o daemon
- Example:

Join us: https://github.com/moby/buildkit