


 GoogleContainerTools / **kaniko** Public

Build Container Images In Kubernetes

 Apache-2.0 license

☆ 11.7k stars 🍴 1.2k forks

☆ Star

 Notifications<> **Code** ⌚ Issues 553 🔗 Pull requests 27 ▶ Actions 📁 Projects 2 ⚠ Security 📊 Insights main ▾

Go to file




dependabot[bot] ...

✖ last month 🕒

[View code](#)

☰ README.md

 **NOTE: kaniko is not an officially supported Google product**  Unit tests passing  Integration tests passing  Build images failing go report A+

kaniko is a tool to build container images from a Dockerfile, inside a container or Kubernetes cluster.

kaniko doesn't depend on a Docker daemon and executes each command within a Dockerfile completely in userspace. This enables building container images in environments that can't easily or securely run a Docker daemon, such as a standard Kubernetes cluster.

kaniko is meant to be run as an image: `gcr.io/kaniko-project/executor`. We do **not** recommend running the kaniko executor binary in another image, as it might not work.

We'd love to hear from you! Join us on [#kaniko Kubernetes Slack](#)

🚧 Please fill out our [quick 5-question survey](#) so that we can learn how satisfied you are with kaniko, and what improvements we should make. Thank you! 🙏

If you are interested in contributing to kaniko, see [DEVELOPMENT.md](#) and [CONTRIBUTING.md](#).

Table of Contents *generated with DocToc*

- [kaniko - Build Images In Kubernetes](#)
 - 🚧 NOTE: kaniko is not an officially supported Google product 🚧
 - [Community](#)
 - [How does kaniko work?](#)
 - [Known Issues](#)
 - [Demo](#)
 - [Tutorial](#)
 - [Using kaniko](#)
 - [kaniko Build Contexts](#)
 - [Using Azure Blob Storage](#)
 - [Using Private Git Repository](#)
 - [Using Standard Input](#)
 - [Running kaniko](#)
 - [Running kaniko in a Kubernetes cluster](#)
 - [Kubernetes secret](#)
 - [Running kaniko in gVisor](#)
 - [Running kaniko in Google Cloud Build](#)
 - [Running kaniko in Docker](#)
 - [Caching](#)
 - [Caching Layers](#)
 - [Caching Base Images](#)
 - [Pushing to Different Registries](#)
 - [Pushing to Docker Hub](#)
 - [Pushing to Google GCR](#)
 - [Pushing to GCR using Workload Identity](#)
 - [Pushing to Amazon ECR](#)
 - [Pushing to Azure Container Registry](#)
 - [Pushing to JFrog Container Registry or to JFrog Artifactory](#)
 - [Additional Flags](#)
 - [Flag --build-arg](#)
 - [Flag --cache](#)
 - [Flag --cache-dir](#)
 - [Flag --cache-repo](#)
 - [Flag --cache-copy-layers](#)
 - [Flag --cache-run-layers](#)
 - [Flag --cache-ttl duration](#)
 - [Flag --cleanup](#)
 - [Flag --compressed-caching](#)
 - [Flag --context-sub-path](#)

- Flag `--customPlatform`
- Flag `--digest-file`
- Flag `--dockerfile`
- Flag `--force`
- Flag `--git`
- Flag `--image-name-with-digest-file`
- Flag `--image-name-tag-with-digest-file`
- Flag `--insecure`
- Flag `--insecure-pull`
- Flag `--insecure-registry`
- Flag `--label`
- Flag `--log-format`
- Flag `--log-timestamp`
- Flag `--no-push`
- Flag `--oci-layout-path`
- Flag `--push-retry`
- Flag `--registry-certificate`
- Flag `--registry-mirror`
- Flag `--reproducible`
- Flag `--single-snapshot`
- Flag `--skip-tls-verify`
- Flag `--skip-tls-verify-pull`
- Flag `--skip-tls-verify-registry`
- Flag `--skip-unused-stages`
- Flag `--snapshotMode`
- Flag `--tar-path`
- Flag `--target`
- Flag `--use-new-run`
- Flag `--verbosity`
- Flag `--ignore-var-run`
- Flag `--ignore-path`
- Flag `--image-fs-extract-retry`
- Debug Image
- Security
 - Verifying Signed Kaniko Images
- Kaniko Builds - Profiling
- Comparison with Other Tools
- Community
- Limitations
 - mtime and snapshotting
- References

Community

We'd love to hear from you! Join [#kaniko on Kubernetes Slack](#)

How does kaniko work?

The kaniko executor image is responsible for building an image from a Dockerfile and pushing it to a registry. Within the executor image, we extract the filesystem of the base image (the FROM image in the Dockerfile). We then execute the commands in the Dockerfile, snapshotting the filesystem in userspace after each one. After each command, we append a layer of changed files to the base image (if there are any) and update image metadata.

Known Issues

- kaniko does not support building Windows containers.
- Running kaniko in any Docker image other than the official kaniko image is not supported (ie YMMV).
 - This includes copying the kaniko executables from the official image into another image.
- kaniko does not support the v1 Registry API ([Registry v1 API Deprecation](#))

Demo



Tutorial

For a detailed example of kaniko with local storage, please refer to a [getting started tutorial](#).

Please see [References](#) for more docs & video tutorials

Using kaniko

To use kaniko to build and push an image for you, you will need:

1. A [build context](#), aka something to build
2. A [running instance of kaniko](#)

kaniko Build Contexts

kaniko's build context is very similar to the build context you would send your Docker daemon for an image build; it represents a directory containing a Dockerfile which kaniko will use to build your image. For example, a `COPY` command in your Dockerfile should refer to a file in the build context.

You will need to store your build context in a place that kaniko can access. Right now, kaniko supports these storage solutions:

- GCS Bucket
- S3 Bucket
- Azure Blob Storage
- Local Directory
- Local Tar
- Standard Input
- Git Repository

Note about Local Directory: this option refers to a directory within the kaniko container. If you wish to use this option, you will need to mount in your build context into the container as a directory.

Note about Local Tar: this option refers to a tar gz file within the kaniko container. If you wish to use this option, you will need to mount in your build context into the container as a file.

Note about Standard Input: the only Standard Input allowed by kaniko is in `.tar.gz` format.

If using a GCS or S3 bucket, you will first need to create a compressed tar of your build context and upload it to your bucket. Once running, kaniko will then download and unpack the compressed tar of the build context before starting the image build.

To create a compressed tar, you can run:

```
tar -C <path to build context> -zcvf context.tar.gz .
```

Then, copy over the compressed tar into your bucket. For example, we can copy over the compressed tar to a GCS bucket with `gsutil`:

```
gsutil cp context.tar.gz gs://<bucket name>
```

When running kaniko, use the `--context` flag with the appropriate prefix to specify the location of your build context:

Source	Prefix	Example
Local Directory	dir://[path to a directory in the kaniko container]	dir:///workspace
Local Tar Gz	tar://[path to a .tar.gz in the kaniko container]	tar://path/to/context.tar.gz

Source	Prefix	Example
Standard Input	tar://[stdin]	tar://stdin
GCS Bucket	gs://[bucket name]/[path to .tar.gz]	gs://kaniko-bucket/path/to/context.tar.gz
S3 Bucket	s3://[bucket name]/[path to .tar.gz]	s3://kaniko-bucket/path/to/context.tar.gz
Azure Blob Storage	https://[account].[azureblobhostsuffix]/[container]/[path to .tar.gz]	https://myaccount.blob.core.windows.net/con
Git Repository	git://[repository url][#reference][#commit-id]	git://github.com/acme/myproject.git#refs/he id>

If you don't specify a prefix, kaniko will assume a local directory. For example, to use a GCS bucket called kaniko-bucket , you would pass in `--context=gs://kaniko-bucket/path/to/context.tar.gz` .

Using Azure Blob Storage

If you are using Azure Blob Storage for context file, you will need to pass [Azure Storage Account Access Key](#) as an environment variable named `AZURE_STORAGE_ACCESS_KEY` through Kubernetes Secrets

Using Private Git Repository

You can use `Personal Access Tokens` for Build Contexts from Private Repositories from [GitHub](#).

You can either pass this in as part of the git URL (e.g., `git://TOKEN@github.com/acme/myproject.git#refs/heads/mybranch`) or using the environment variable `GIT_TOKEN` .

You can also pass `GIT_USERNAME` and `GIT_PASSWORD` (password being the token) if you want to be explicit about the username.

Using Standard Input

If running kaniko and using Standard Input build context, you will need to add the docker or kubernetes `-i, --interactive` flag. Once running, kaniko will then get the data from `STDIN` and create the build context as a compressed tar. It will then unpack the compressed tar of the build context before starting the image build. If no data is piped during the interactive run, you will need to send the EOF signal by yourself by pressing `ctrl+D` .

Complete example of how to interactively run kaniko with `.tar.gz` Standard Input data, using docker:

```
echo -e 'FROM alpine \nRUN echo "created from standard input"' > Dockerfile | tar -cf - Dock
--interactive -v $(pwd):/workspace gcr.io/kaniko-project/executor:latest \
--context tar://stdin \
--destination=<gcr.io/$project/$image:$tag>
```

Complete example of how to interactively run kaniko with `.tar.gz` Standard Input data, using Kubernetes command line with a temporary container and completely dockerless:

```

echo -e 'FROM alpine \nRUN echo "created from standard input"' > Dockerfile | tar -cf - Dock
--rm --stdin=true \
--image=gcr.io/kaniko-project/executor:latest --restart=Never \
--overrides='{
  "apiVersion": "v1",
  "spec": {
    "containers": [
      {
        "name": "kaniko",
        "image": "gcr.io/kaniko-project/executor:latest",
        "stdin": true,
        "stdinOnce": true,
        "args": [
          "--dockerfile=Dockerfile",
          "--context=tar://stdin",
          "--destination=gcr.io/my-repo/my-image"
        ],
        "volumeMounts": [
          {
            "name": "cabundle",
            "mountPath": "/kaniko/ssl/certs/"
          },
          {
            "name": "docker-config",
            "mountPath": "/kaniko/.docker/"
          }
        ]
      }
    ],
    "volumes": [
      {
        "name": "cabundle",
        "configMap": {
          "name": "cabundle"
        }
      },
      {
        "name": "docker-config",
        "configMap": {
          "name": "docker-config"
        }
      }
    ]
  }
}'

```

Running kaniko

There are several different ways to deploy and run kaniko:

- [In a Kubernetes cluster](#)
- [In gVisor](#)
- [In Google Cloud Build](#)
- [In Docker](#)

Running kaniko in a Kubernetes cluster

Requirements:

- Standard Kubernetes cluster (e.g. using [GKE](#))
- [Kubernetes Secret](#)
- A [build context](#)

Kubernetes secret

To run kaniko in a Kubernetes cluster, you will need a standard running Kubernetes cluster and a Kubernetes secret, which contains the auth required to push the final image.

To create a secret to authenticate to Google Cloud Registry, follow these steps:

1. Create a service account in the Google Cloud Console project you want to push the final image to with Storage Admin permissions.
2. Download a JSON key for this service account
3. Rename the key to `kaniko-secret.json`
4. To create the secret, run:

```
kubectl create secret generic kaniko-secret --from-file=<path to kaniko-secret.json>
```

Note: If using a GCS bucket in the same GCP project as a build context, this service account should now also have permissions to read from that bucket.

The Kubernetes Pod spec should look similar to this, with the args parameters filled in:

```
apiVersion: v1
kind: Pod
metadata:
  name: kaniko
spec:
  containers:
    - name: kaniko
      image: gcr.io/kaniko-project/executor:latest
      args:
        - "--dockerfile=<path to Dockerfile within the build context>"
        - "--context=gs://<GCS bucket>/<path to .tar.gz>"
        - "--destination=<gcr.io/$PROJECT/$IMAGE:$TAG>"
      volumeMounts:
        - name: kaniko-secret
          mountPath: /secret
      env:
        - name: GOOGLE_APPLICATION_CREDENTIALS
          value: /secret/kaniko-secret.json
      restartPolicy: Never
  volumes:
    - name: kaniko-secret
      secret:
        secretName: kaniko-secret
```

This example pulls the build context from a GCS bucket. To use a local directory build context, you could consider using configMaps to mount in small build contexts.

Running kaniko in gVisor

Running kaniko in [gVisor](#) provides an additional security boundary. You will need to add the `--force` flag to run kaniko in gVisor, since currently there isn't a way to determine whether or not a container is running in gVisor.

```
docker run --runtime=runc -v $(pwd):/workspace -v ~/.config:/root/.config \
gcr.io/kaniko-project/executor:latest \
--dockerfile=<path to Dockerfile> --context=/workspace \
--destination=gcr.io/my-repo/my-image --force
```

We pass in `--runtime=runc` to use gVisor. This example mounts the current directory to `/workspace` for the build context and the `~/.config` directory for GCR credentials.

Running kaniko in Google Cloud Build

Requirements:

- A [build context](#)

To run kaniko in GCB, add it to your build config as a build step:

```
steps:
- name: gcr.io/kaniko-project/executor:latest
  args:
  [
    "--dockerfile=<path to Dockerfile within the build context>",
    "--context=dir://<path to build context>",
    "--destination=gcr.io/$PROJECT/$IMAGE:$TAG",
  ]
```

kaniko will build and push the final image in this build step.

Running kaniko in Docker

Requirements:

- [Docker](#)

We can run the kaniko executor image locally in a Docker daemon to build and push an image from a Dockerfile.

For example, when using gcloud and GCR you could run kaniko as follows:

```
docker run \
-v "$HOME"/.config/gcloud:/root/.config/gcloud \
-v /path/to/context:/workspace \
gcr.io/kaniko-project/executor:latest \
--dockerfile /workspace/Dockerfile \
--destination "gcr.io/$PROJECT_ID/$IMAGE_NAME:$TAG" \
--context dir:///workspace/
```

There is also a utility script [run_in_docker.sh](#) that can be used as follows:

```
./run_in_docker.sh <path to Dockerfile> <path to build context> <destination of final image>
```

NOTE: `run_in_docker.sh` expects a path to a Dockerfile relative to the absolute path of the build context.

An example run, specifying the Dockerfile in the container directory `/workspace`, the build context in the local directory `/home/user/kaniko-project`, and a Google Container Registry as a remote image destination:

```
./run_in_docker.sh /workspace/Dockerfile /home/user/kaniko-project gcr.io/$PROJECT_ID/$TAG
```

Caching

Caching Layers

kaniko can cache layers created by `RUN` (configured by flag `--cache-run-layers`) and `COPY` (configured by flag `--cache-copy-layers`) commands in a remote repository. Before executing a command, kaniko checks the cache for the layer. If it exists, kaniko will pull and extract the cached layer instead of executing the command. If not, kaniko will execute the command and then push the newly created layer to the cache.

Note that kaniko cannot read layers from the cache after a cache miss: once a layer has not been found in the cache, all subsequent layers are built locally without consulting the cache.

Users can opt into caching by setting the `--cache=true` flag. A remote repository for storing cached layers can be provided via the `--cache-repo` flag. If this flag isn't provided, a cached repo will be inferred from the `--destination` provided.

Caching Base Images

kaniko can cache images in a local directory that can be volume mounted into the kaniko pod. To do so, the cache must first be populated, as it is read-only. We provide a kaniko cache warming image at `gcr.io/kaniko-project/warmer`:

```
docker run -v $(pwd):/workspace gcr.io/kaniko-project/warmer:latest --cache-dir=/workspace/c
```

`--image` can be specified for any number of desired images. This command will cache those images by digest in a local directory named `cache`. Once the cache is populated, caching is opted into with the same `--cache=true` flag as above. The location of the local cache is provided via the `--cache-dir` flag, defaulting to `/cache` as with the cache warmer. See the `examples` directory for how to use with kubernetes clusters and persistent cache volumes.

Pushing to Different Registries

kaniko uses Docker credential helpers to push images to a registry.

kaniko comes with support for GCR, Docker `config.json` and Amazon ECR, but configuring another credential helper should allow pushing to a different registry.

Pushing to Docker Hub

Get your docker registry user and password encoded in base64

```
echo -n USER:PASSWORD | base64
```

Create a `config.json` file with your Docker registry url and the previous generated base64 string

Note: Please use v1 endpoint. See #1209 for more details

```
{
  "auths": {
    "https://index.docker.io/v1/": {
      "auth": "xxxxxxxxxxxxxxxxxx"
    }
  }
}
```

Run kaniko with the `config.json` inside `/kaniko/.docker/config.json`

```
docker run -ti --rm -v `pwd`: /workspace -v `pwd`/config.json:/kaniko/.docker/config.json:ro
```

Pushing to Google GCR

To create a credentials to authenticate to Google Cloud Registry, follow these steps:

1. Create a [service account](#) or in the Google Cloud Console project you want to push the final image to with `Storage Admin` permissions.
2. Download a JSON key for this service account
3. (optional) Rename the key to `kaniko-secret.json`, if you don't rename, you have to change the name used the command(in the volume part)
4. Run the container adding the path in `GOOGLE_APPLICATION_CREDENTIALS` env var

```
docker run -ti --rm -e GOOGLE_APPLICATION_CREDENTIALS=/kaniko/config.json \
-v `pwd`: /workspace -v `pwd`/kaniko-secret.json:/kaniko/config.json:ro gcr.io/kaniko-project
--dockerfile=Dockerfile --destination=yourimagename
```

Pushing to GCR using Workload Identity

If you have enabled Workload Identity on your GKE cluster then you can use the workload identity to push built images to GCR without adding a `GOOGLE_APPLICATION_CREDENTIALS` in your kaniko pod specification.

Learn more on how to [enable](#) and [migrate existing apps](#) to workload identity.

To authenticate using workload identity you need to run the kaniko pod using the Kubernetes Service Account (KSA) bound to Google Service Account (GSA) which has `Storage.Admin` permissions to push images to Google Container registry.

Please follow the detailed steps [here](#) to create a Kubernetes Service Account, Google Service Account and create an IAM policy binding between the two to allow the Kubernetes Service account to act as the Google service account.

To grant the Google Service account the right permission to push to GCR, run the following GCR command

```
gcloud projects add-iam-policy-binding $PROJECT \
--member=serviceAccount:[gsa-name]@$PROJECT.iam.gserviceaccount.com \
```

```
--role=roles/storage.objectAdmin
```

Please ensure, kaniko pod is running in the namespace and with a Kubernetes Service Account.

Pushing to Amazon ECR

The Amazon ECR [credential helper](#) is built into the kaniko executor image.

1. Configure credentials

- i. You can use instance roles when pushing to ECR from a EC2 instance or from EKS, by [configuring the instance role permissions](#) (the AWS managed policy `EC2InstanceProfileForImageBuilderECRContainerBuilds` provides broad permissions to upload ECR images and may be used as configuration baseline). Additionally, set `AWS_SDK_LOAD_CONFIG=true` as environment variable within the kaniko pod. If running on an EC2 instance with an instance profile, you may also need to set `AWS_EC2_METADATA_DISABLED=true` for kaniko to pick up the correct credentials.
- ii. Or you can create a Kubernetes secret for your `~/.aws/credentials` file so that credentials can be accessed within the cluster. To create the secret, run: `shell kubectl create secret generic aws-secret --from-file=<path to .aws/credentials>`

The Kubernetes Pod spec should look similar to this, with the args parameters filled in. Note that `aws-secret` volume mount and volume are only needed when using AWS credentials from a secret, not when using instance roles.

```
apiVersion: v1
kind: Pod
metadata:
  name: kaniko
spec:
  containers:
    - name: kaniko
      image: gcr.io/kaniko-project/executor:latest
      args:
        - "--dockerfile=<path to Dockerfile within the build context>"
        - "--context=s3://<bucket name>/<path to .tar.gz>"
        - "--destination=<aws_account_id.dkr.ecr.region.amazonaws.com/my-repository:my-tag>"
      volumeMounts:
        # when not using instance role
        - name: aws-secret
          mountPath: /root/.aws/
  restartPolicy: Never
  volumes:
    # when not using instance role
    - name: aws-secret
      secret:
        secretName: aws-secret
```

Pushing to Azure Container Registry

An ACR [credential helper](#) is built into the kaniko executor image, which can be used to authenticate with well-known Azure environmental information.

To configure credentials, you will need to do the following:

1. Update the `credStore` section of `config.json` :

```
{ "credsStore": "acr" }
```

A downside of this approach is that ACR authentication will be used for all registries, which will fail if you also pull from DockerHub, GCR, etc. Thus, it is better to configure the credential tool only for your ACR registries by using `credHelpers` instead of `credsStore` :

```
{ "credHelpers": { "mycr.azurecr.io": "acr-env" } }
```

You can mount in the new config as a configMap:

```
kubectl create configmap docker-config --from-file=<path to config.json>
```

2. Configure credentials

You can create a Kubernetes secret with environment variables required for Service Principal authentication and expose them to the builder container.

```
AZURE_CLIENT_ID=<clientId>
AZURE_CLIENT_SECRET=<clientSecret>
AZURE_TENANT_ID=<tenantId>
```

If the above are not set then authentication falls back to managed service identities and the MSI endpoint is attempted to be contacted which will work in various Azure contexts such as App Service and Azure Kubernetes Service where the MSI endpoint will authenticate the MSI context the service is running under.

The Kubernetes Pod spec should look similar to this, with the args parameters filled in. Note that `azure-secret` secret is only needed when using Azure Service Principal credentials, not when using a managed service identity.

```
apiVersion: v1
kind: Pod
metadata:
  name: kaniko
spec:
  containers:
    - name: kaniko
      image: gcr.io/kaniko-project/executor:latest
      args:
        - "--dockerfile=<path to Dockerfile within the build context>"
        - "--context=s3://<bucket name>/<path to .tar.gz>"
        - "--destination=mycr.azurecr.io/my-repository:my-tag"
      envFrom:
        # when authenticating with service principal
        - secretRef:
            name: azure-secret
      volumeMounts:
        - name: docker-config
          mountPath: /kaniko/.docker/
  volumes:
    - name: docker-config
      configMap:
```

```
name: docker-config
restartPolicy: Never
```

Pushing to JFrog Container Registry or to JFrog Artifactory

Kaniko can be used with both [JFrog Container Registry](#) and JFrog Artifactory.

Get your JFrog Artifactory registry user and password encoded in base64

```
echo -n USER:PASSWORD | base64
```

Create a `config.json` file with your Artifactory Docker local registry URL and the previous generated base64 string

```
{
  "auths": {
    "artprod.company.com": {
      "auth": "xxxxxxxxxxxxxxxx"
    }
  }
}
```

For example, for Artifactory cloud users, the docker registry should be: `<company>.<local-repository-name>.io`.

Run kaniko with the `config.json` inside `/kaniko/.docker/config.json`

```
docker run -ti --rm -v `pwd`: /workspace -v
`pwd`/config.json:/kaniko/.docker/config.json:ro gcr.io/kaniko-project/executor:latest --
dockerfile=Dockerfile --destination=yourimagename
```

After the image is uploaded, using the JFrog CLI, you can [collect](#) and [publish](#) the build information to Artifactory and trigger [build vulnerabilities scanning](#) using JFrog Xray.

To collect and publish the image's build information using the Jenkins Artifactory plugin, see instructions for [scripted pipeline](#) and [declarative pipeline](#).

Additional Flags

Flag `--build-arg`

This flag allows you to pass in ARG values at build time, similarly to Docker. You can set it multiple times for multiple arguments.

Note that passing values that contain spaces is not natively supported - you need to ensure that the IFS is set to null before your executor command. You can set this by adding `export IFS=''` before your executor call. See the following example

```
export IFS=''
/kaniko/executor --build-arg "MY_VAR='value with spaces'" ...
```

Flag --cache

Set this flag as `--cache=true` to opt into caching with kaniko.

Flag --cache-dir

Set this flag to specify a local directory cache for base images. Defaults to `/cache`.

This flag must be used in conjunction with the `--cache=true` flag.

Flag --cache-repo

Set this flag to specify a remote repository that will be used to store cached layers.

If this flag is not provided, a cache repo will be inferred from the `--destination` flag. If `--destination=gcr.io/kaniko-project/test`, then cached layers will be stored in `gcr.io/kaniko-project/test/cache`.

This flag must be used in conjunction with the `--cache=true` flag.

Flag --cache-copy-layers

Set this flag to cache copy layers.

Flag --cache-run-layers

Set this flag to cache run layers (default=true).

Flag --cache-ttl duration

Cache timeout in hours. Defaults to two weeks.

Flag --cleanup

Set this flag to clean the filesystem at the end of the build.

Flag --compressed-caching

Set this to false in order to prevent tar compression for cached layers. This will increase the runtime of the build, but decrease the memory usage especially for large builds. Try to use `--compressed-caching=false` if your build fails with an out of memory error. Defaults to true.

Flag --context-sub-path

Set a sub path within the given `--context`.

Its particularly useful when your context is, for example, a git repository, and you want to build one of its subfolders instead of the root folder.

Flag --customPlatform

Allows to build with another default platform than the host, similarly to docker build `--platform xxx` the value has to be on the form `--customPlatform=linux/arm`, with acceptable values listed here: [GOOS/GOARCH](https://github.com/google/goos/goarch).

It's also possible specifying CPU variants adding it as a third parameter (like `--customPlatform=linux/arm/v5`). Currently CPU variants are only known to be used for the ARM architecture as listed here: [GOARM](#)

The resulting images cannot provide any metadata about CPU variant due to a limitation of the OCI-image specification.

This is not virtualization and cannot help to build an architecture not natively supported by the build host. This is used to build i386 on an amd64 Host for example, or arm32 on an arm64 host.

Flag `--digest-file`

Set this flag to specify a file in the container. This file will receive the digest of a built image. This can be used to automatically track the exact image built by kaniko.

For example, setting the flag to `--digest-file=/dev/termination-log` will write the digest to that file, which is picked up by Kubernetes automatically as the `{{.state.terminated.message}}` of the container.

Flag `--dockerfile`

Path to the dockerfile to be built. (default "Dockerfile")

Flag `--force`

Force building outside of a container

Flag `--git`

Branch to clone if build context is a git repository (default branch=,single-branch=false,recurse-submodules=false)

Flag `--image-name-with-digest-file`

Specify a file to save the image name w/ digest of the built image to.

Flag `--image-name-tag-with-digest-file`

Specify a file to save the image name w/ image tag and digest of the built image to.

Flag `--insecure`

Set this flag if you want to push images to a plain HTTP registry. It is supposed to be used for testing purposes only and should not be used in production!

Flag `--insecure-pull`

Set this flag if you want to pull images from a plain HTTP registry. It is supposed to be used for testing purposes only and should not be used in production!

Flag `--insecure-registry`

You can set `--insecure-registry <registry-name>` to use plain HTTP requests when accessing the specified registry. It is supposed to be used for testing purposes only and should not be used in production! You can set it multiple times for multiple registries.

Flag --label

Set this flag as `--label key=value` to set some metadata to the final image. This is equivalent as using the `LABEL` within the Dockerfile.

Flag --log-format

Set this flag as `--log-format=<text|color|json>` to set the log format. Defaults to `color`.

Flag --log-timestamp

Set this flag as `--log-timestamp=<true|false>` to add timestamps to `<text|color>` log format. Defaults to `false`.

Flag --no-push

Set this flag if you only want to build the image, without pushing to a registry.

Flag --oci-layout-path

Set this flag to specify a directory in the container where the OCI image layout of a built image will be placed. This can be used to automatically track the exact image built by kaniko.

For example, to surface the image digest built in a [Tekton task](#), this flag should be set to match the image resource `outputImageDir`.

Note: Depending on the built image, the media type of the image manifest might be either `application/vnd.oci.image.manifest.v1+json` or `application/vnd.docker.distribution.manifest.v2+json`.

Flag --push-retry

Set this flag to the number of retries that should happen for the push of an image to a remote destination. Defaults to `0`.

Flag --registry-certificate

Set this flag to provide a certificate for TLS communication with a given registry.

Expected format is `my.registry.url=/path/to/the/certificate.cert`

Flag --registry-mirror

Set this flag if you want to use a registry mirror instead of the default `index.docker.io`. You can use this flag more than once, if you want to set multiple mirrors. If an image is not found on the first mirror, Kaniko will try the next mirror(s), and at the end fallback on the default registry.

Expected format is `mirror.gcr.io` for example.

Note that you can't specify a URL with scheme for this flag. Some valid options are:

- `mirror.gcr.io`
- `127.0.0.1`
- `192.168.0.1:5000`
- `mycompany-docker-virtual.jfrog.io`

Flag --reproducible

Set this flag to strip timestamps out of the built image and make it reproducible.

Flag --single-snapshot

This flag takes a single snapshot of the filesystem at the end of the build, so only one layer will be appended to the base image.

Flag --skip-tls-verify

Set this flag to skip TLS certificate validation when pushing to a registry. It is supposed to be used for testing purposes only and should not be used in production!

Flag --skip-tls-verify-pull

Set this flag to skip TLS certificate validation when pulling from a registry. It is supposed to be used for testing purposes only and should not be used in production!

Flag --skip-tls-verify-registry

You can set `--skip-tls-verify-registry <registry-name>` to skip TLS certificate validation when accessing the specified registry. It is supposed to be used for testing purposes only and should not be used in production! You can set it multiple times for multiple registries.

Flag --skip-unused-stages

This flag builds only used stages if defined to `true`. Otherwise it builds by default all stages, even the unnecessary ones until it reaches the target stage / end of Dockerfile

Flag --snapshotMode

You can set the `--snapshotMode=<full (default), redo, time>` flag to set how kaniko will snapshot the filesystem.

- If `--snapshotMode=full` is set, the full file contents and metadata are considered when snapshotting. This is the least performant option, but also the most robust.
- If `--snapshotMode=redo` is set, the file mtime, size, mode, owner uid and gid will be considered when snapshotting. This may be up to 50% faster than "full", particularly if your project has a large number files.
- If `--snapshotMode=time` is set, only file mtime will be considered when snapshotting (see [limitations related to mtime](#)).

Flag --tar-path

Set this flag as `--tar-path=<path>` to save the image as a tarball at path. You need to set `--destination` as well (for example `--destination=image`). If you want to save the image as tarball only you also need to set `--no-push`.

Flag --target

Set this flag to indicate which build stage is the target build stage.

Flag --use-new-run

Use the experimental run implementation for detecting changes without requiring file system snapshots. In some cases, this may improve build performance by 75%.

Flag --verbosity

Set this flag as `--verbosity=<panic|fatal|error|warn|info|debug|trace>` to set the logging level. Defaults to `info`.

Flag --ignore-var-run

Ignore `/var/run` when taking image snapshot. Set it to `false` to preserve `/var/run/*` in destination image. (Default `true`).

Flag --ignore-path

Set this flag as `--ignore-path=<path>` to ignore path when taking an image snapshot. Set it multiple times for multiple ignore paths.

Flag --image-fs-extract-retry

Set this flag to the number of retries that should happen for the extracting an image filesystem. Defaults to `0`.

Debug Image

The kaniko executor image is based on scratch and doesn't contain a shell. We provide `gcr.io/kaniko-project/executor:debug`, a debug image which consists of the kaniko executor image along with a busybox shell to enter.

You can launch the debug image with a shell entrypoint:

```
docker run -it --entrypoint=/busybox/sh gcr.io/kaniko-project/executor:debug
```

Security

kaniko by itself **does not** make it safe to run untrusted builds inside your cluster, or anywhere else.

kaniko relies on the security features of your container runtime to provide build security.

The minimum permissions kaniko needs inside your container are governed by a few things:

- The permissions required to unpack your base image into its container
- The permissions required to execute the RUN commands inside the container

If you have a minimal base image (SCRATCH or similar) that doesn't require permissions to unpack, and your Dockerfile doesn't execute any commands as the root user, you can run kaniko without root permissions. It should be noted that Docker runs as root by default, so you still require (in a sense) privileges to use kaniko.

You may be able to achieve the same default seccomp profile that Docker uses in your Pod by setting [seccomp](#) profiles with annotations on a [PodSecurityPolicy](#) to create or update security policies on your cluster.

Verifying Signed Kaniko Images

kaniko images are signed for versions $\geq 1.5.2$ using [cosign](#)!

To verify a public image, install [cosign](#) and use the provided [public key](#):

```
$ cat cosign.pub
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE9aAfAcgAxIFMTstJUv8l/AMqnSKw
P+vLu3NnnBDHCfREQpV/AJuiZ1UtgGpFpHlJLCNPmFkzQTnfyN5idzNl6Q==
-----END PUBLIC KEY-----

$ cosign verify -key ./cosign.pub gcr.io/kaniko-project/executor:latest
```

Kaniko Builds - Profiling

If your builds are taking long, we recently added support to analyze kaniko function calls using [Slow Jam](#) To start profiling,

1. Add an environment variable `STACKLOG_PATH` to your [pod definition](#).
2. If you are using the kaniko `debug` image, you can copy the file in the `pre-stop` container lifecycle hook.

Comparison with Other Tools

Similar tools include:

- [BuildKit](#)
- [img](#)
- [orca-build](#)
- [umoci](#)
- [buildah](#)
- [FTL](#)
- [Bazel rules_docker](#)

All of these tools build container images with different approaches.

BuildKit (and `img`) can perform as a non-root user from within a container but requires seccomp and AppArmor to be disabled to create nested containers. `kaniko` does not actually create nested containers, so it does not require seccomp and AppArmor to be disabled.

`orca-build` depends on `runc` to build images from Dockerfiles, which can not run inside a container (for similar reasons to `img` above). `kaniko` doesn't use `runc` so it doesn't require the use of kernel namespacing techniques. However, `orca-build` does not require Docker or any privileged daemon (so builds can be done entirely without privilege).

`umoci` works without any privileges, and also has no restrictions on the root filesystem being extracted (though it requires additional handling if your filesystem is sufficiently complicated). However, it has no `Dockerfile`-like build tooling (it's a slightly lower-level tool that can be used to build such builders -- such as `orca-build`).

`Buildah` specializes in building OCI images. Buildah's commands replicate all of the commands that are found in a Dockerfile. This allows building images with and without Dockerfiles while not requiring any root privileges. Buildah's ultimate goal is to provide a lower-level `coreutils` interface to build images. The flexibility of building images without Dockerfiles allows for the integration of other scripting languages into the build process. Buildah follows a simple fork-exec model and does not run as a daemon but it is based on a comprehensive API in `golang`, which can be vendored into other tools.

`FTL` and `Bazel` aim to achieve the fastest possible creation of Docker images for a subset of images. These can be thought of as a special-case "fast path" that can be used in conjunction with the support for general Dockerfiles `kaniko` provides.

Community

[kaniko-users](#) Google group

To Contribute to `kaniko`, see [DEVELOPMENT.md](#) and [CONTRIBUTING.md](#).

Limitations

mtime and snapshotting

When taking a snapshot, `kaniko`'s hashing algorithms include (or in the case of `--snapshotMode=time`, only use) a file's `mtime` to determine if the file has changed. Unfortunately, there is a delay between when changes to a file are made and when the `mtime` is updated. This means:

- With the time-only snapshot mode (`--snapshotMode=time`), `kaniko` may miss changes introduced by `RUN` commands entirely.
- With the default snapshot mode (`--snapshotMode=full`), whether or not `kaniko` will add a layer in the case where a `RUN` command modifies a file **but the contents do not** change is theoretically non-deterministic. This *does not affect the contents* which will still be correct, but it does affect the number of layers.

Note that these issues are currently theoretical only. If you see this issue occur, please [open an issue](#).

References

- [Kaniko - Building Container Images In Kubernetes Without Docker](#).

Releases 38

 **v1.9.1** Latest
on Sep 26, 2022

[+ 37 releases](#)

Packages

No packages published

Contributors 245



+ 234 contributors

Languages

