Rockcraft

Release 0.0.1.dev1

Canonical Ltd.

Mar 22, 2023

CONTENTS

Rockcraft is a tool to create ROCKs - a new generation of secure, stable and OCI-compliant container images, powered by Ubuntu.

Using the same language as Snapcraft and Charmcraft, Rockcraft (currently under heavy development) offers a true declarative way for building efficient container images. By making use of existing Ubuntu tools like LXD and Multipass, Rockcraft is able to compartmentalize your typical container image build into multiple parts, each one being comprised of several independent lifecycle steps (pull, build, stage and prime), allowing you to declaratively perform complex operations, at build time, without the need for massaging or stripping down the build environment from where your final container image originates.

Off with the cumbersome and explicit scripting, on with the new declarative builds. While preserving the familiar Ubuntu experience, Rockcraft eliminates the need for imperative builds, allowing everyone to declaratively define a container image that is built from Ubuntu, for Ubuntu users. Rockcraft implements source-to-image best-practice designs, handling all the repetitive and boilerplate steps of a build and directing your focus to what really matters - the image's content.

Rockcraft is for everyone wanting to build production-grade container images, regardless of their experience as a software developer. From independent software vendors to cloud-native developers and occasional container users.

CHAPTER

ONE

TUTORIALS

If you want to learn the basics from experience, then our tutorials will help you acquire the necessary competencies from real-life examples with fully reproducible steps.

1.1 Create a "Hello World" ROCK

1.1.1 Prerequisites

- snap enabled system (https://snapcraft.io)
- LXD installed (https://linuxcontainers.org/lxd/getting-started-cli/)
- skopeo installed (https://github.com/containers/skopeo)
- Docker installed (https://snapcraft.io/docker)
- a text editor

1.1.2 Install Rockcraft

Install Rockcraft on your host:

```
sudo snap install rockcraft --classic --edge
```

1.1.3 Project Setup

Create a new directory and write the following into a text editor and save it as `rockcraft.yaml`:

```
name: hello
summary: Hello World
description: The most basic example of a ROCK.
version: "1.0"
base: ubuntu:20.04
license: Apache-2.0
cmd: [/usr/bin/hello, -t]
platforms:
  amd64:  # Make sure this value matches your computer's architecture

parts:
```

(continues on next page)

(continued from previous page)

```
  hello:
    plugin: nil
    overlay-packages:
      - hello
```

1.1.4 Pack the ROCK with Rockcraft

To build the ROCK, run:

```
rockcraft pack
```

The output should look as follows:

```
Launching instance...
Retrieved base ubuntu:20.04
Extracted ubuntu:20.04
Executed: pull hello
Executed: overlay hello
Executed: build hello
Executed: stage hello
Executed: prime hello
Executed parts lifecycle
Created new layer
Cmd set to ['/usr/bin/hello', '-t']
Labels and annotations set to ['org.opencontainers.image.version=1.0', 'org.
↪opencontainers.image.title=hello', 'org.opencontainers.image.ref.name=hello', 'org.
↪opencontainers.image.licenses=Apache-2.0', 'org.opencontainers.image.created=2022-06-
↪30T09:07:38.124741+00:00']
Exported to OCI archive 'hello_1.0_amd64.rock'
```

At the end of the process, a file named `hello_1.0_amd64.rock` should be present in the current directory. That's your ROCK, in oci-archive format (a tarball).

1.1.5 Run the ROCK in Docker

First, import the recently created ROCK into Docker:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:hello_1.0_
↪amd64.rock docker-daemon:hello:1.0
```

Now run the `hello` command from the ROCK:

```
docker run --rm hello:1.0
```

Which should print:

```
hello, world
```

1.2 Install packages slices into a ROCK

In this tutorial, you will create a lean ROCK that contains a fully functional OpenSSL installation, and you will verify that it is functional by loading the ROCK into Docker and using it to validate the certificates of the Ubuntu website.

1.2.1 Prerequisites

- snap enabled system (https://snapcraft.io)
- LXD installed (https://linuxcontainers.org/lxd/getting-started-cli/)
- skopeo installed (https://github.com/containers/skopeo)
- Docker installed (https://docs.docker.com/get-docker/)
- a text editor

1.2.2 Install Rockcraft

Install Rockcraft on your host:

```
snap install rockcraft --classic --edge
```

1.2.3 Project Setup

Create a new directory, write the following into a text editor and save it as `rockcraft.yaml`:

```
name: chisel-openssl
summary: OpenSSL from Chisel slices
description: A "bare" ROCK containing an OpenSSL installation created from Chisel slices.
license: Apache-2.0

version: "0.0.1"
base: bare
build_base: "ubuntu:22.04"
entrypoint: [/usr/bin/openssl]
platforms:
  amd64:

env:
  - SSL_CERT_FILE: /etc/ssl/certs/ca-certificates.crt

parts:
  openssl:
    plugin: nil
    stage-packages:
      - openssl_bins
      - ca-certificates_data
```

Note that this Rockcraft file uses the `openssl_bins` and `ca-certificates_data` Chisel slices to generate an image containing only files that are strictly necessary for a functional OpenSSL installation. See *What is Chisel?* for details on the Chisel tool.

1.2.4 Pack the ROCK with Rockcraft

To build the ROCK, run:

```
rockcraft
```

The output will look similar to:

```
Launching instance...
Retrieved base bare for amd64
Extracted bare:latest
Executed: pull openssl
Executed: overlay openssl
Executed: build openssl
Executed: stage openssl
Executed: prime openssl
Executed parts lifecycle
Created new layer
Entrypoint set to ['/usr/bin/openssl']
Cmd set to []
Environment set to ['SSL_CERT_FILE=/etc/ssl/certs/ca-certificates.crt']
Labels and annotations set to ['org.opencontainers.image.version=0.0.1', 'org.
↪opencontainers.image.title=chisel-openssl', 'org.opencontainers.image.ref.name=chisel-
↪openssl', 'org.opencontainers.image.licenses=Apache-2.0', 'org.opencontainers.image.
↪created=2022-09-30T17:57:57.070040+00:00', 'org.opencontainers.image.base.
↪digest=719e29cbdf81d2c046598c274ae82bdcdfe7bf819058a0f304c57858b633d801']
Exported to OCI archive 'chisel-openssl_0.0.1_amd64.rock'
```

The process might take a little while, but at the end, a new file named `chisel-openssl_0.0.1_amd64.rock` will be present in the current directory. That's your OpenSSL ROCK, in oci-archive format.

1.2.5 Run the ROCK in Docker

First, import the recently created ROCK into Docker:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:chisel-
↪openssl_0.0.1_amd64.rock docker-daemon:chisel-openssl:latest
```

Now you can run a container from the ROCK:

```
docker run --rm  chisel-openssl
```

The output will be OpenSSL's default help message, which starts like this:

```
help:

Standard commands
asn1parse         ca                ciphers           cmp
cms               crl               crl2pkcs7         dgst
dhparam           dsa               dsaparam          ec
ecparam           enc               engine            errstr
fipsinstall       gendsa            genpkey           genrsa
help              info              kdf               list
```

(continues on next page)

(continued from previous page)

```
mac                nseq               ocsp               passwd
pkcs12             pkcs7              pkcs8              pkey
pkeyparam          pkeyutl            prime              rand
rehash             req                rsa                rsautl
s_client           s_server           s_time             sess_id
<... many more lines of output>
```

As you can see, OpenSSL has many features. Use one of them to check that Ubuntu's website has valid SSL certificates:

```
docker run --rm chisel-openssl s_client -connect ubuntu.com:443 -brief
```

The output will look similar to the following:

```
CONNECTION ESTABLISHED
Protocol version: TLSv1.3
Ciphersuite: TLS_AES_256_GCM_SHA384
Peer certificate: CN = ubuntu.com
Hash used: SHA256
Signature type: RSA-PSS
Verification: OK
Server Temp Key: X25519, 253 bits
```

The `Verification: OK` line indicates that the OpenSSL installation inside your ROCK was able to validate Ubuntu Website's certificates successfully.

1.3 Publish a ROCK to a registry

1.3.1 Prerequisites

- skopeo installed (https://github.com/containers/skopeo)
- Docker installed (https://docs.docker.com/get-docker/)

1.3.2 Push a ROCK to Docker Hub

The output of `rockcraft pack` is a ROCK in its oci-archive archive format.

```
skopeo --insecure-policy copy oci-archive:<your_rock_file.rock> docker://<container_
↪registry>/<repo>:<tag>
```

Output:

```
Getting image source signatures
Copying blob e65b2e587073 skipped: already exists
Copying blob 01f981dde5a5 skipped: already exists
Copying config 5da22a9016 done
Writing manifest to image destination
Storing signatures
```

CHAPTER

TWO

HOW-TO GUIDES

If you have a specific goal but are already familiar with Rockcraft, our How-to guides have more in-depth detail than our tutorials and can be applied to a broader set of applications.

They'll help you achieve an end result but may require you to understand and adapt the steps to fit your specific requirements.

2.1 How to get started - quick guide

See the *Tutorials* for a full getting started guide.

2.1.1 Getting started

Rockcraft is **the tool** for building Ubuntu-based and production-grade OCI images, aka ROCKs!

Rockcraft is distributed as a snap. For packing new ROCKs, it makes use of "providers" to execute all the steps involved in the ROCK's build process. At the moment, the supported providers are LXD and Multipass.

Requirements

Before installing the Rockcraft snap, make sure you have the necessary tools and environment to install and run Rockcraft.

First things first, if you are running Ubuntu, Snap is already installed and ready to go:

```
snap --version
```

You'll get something like:

```
snap    2.57.1
snapd   2.57.1
series  16
ubuntu  22.04
kernel  5.17.0-1016-oem
```

If this is not the case, then please check https://snapcraft.io/docs/installing-snap-on-ubuntu.

For what concerns providers, LXD is the default one for Rockcraft, so start by checking if it is available:

```
lxd --version
```

The output will be something like:

```
5.5
```

And that it is enabled:

```
systemctl status snap.lxd.daemon.service
```

The output should look like:

```
 snap.lxd.daemon.service - Service for snap application lxd.daemon
    Loaded: loaded (/etc/systemd/system/snap.lxd.daemon.service; static)
    Active: active (running) since Wed 2022-09-07 16:02:29 CEST; 6 days ago
    ...
```

If LXD is not installed, then run:

```
snap install lxd
```

And if LXD is not running, try starting it via:

```
lxd init --minimal  # drop the --minimal for an interactive configuration
```

May you find any problems with LXD, please check https://ubuntu.com/lxd.

Choose a Rockcraft release

Pick a Rockcraft release, either from the snap store or via `snap search rockcraft`.

Keep in mind the chosen channel, as riskier releases are more prone to breaking changes.

Also, note that the Rockcraft's snap confinement is set to "classic" (this is important for the installation step).

Installation steps

Having chosen a Rockcraft release, you must now install it via the snap CLI (or directly via the Ubuntu Desktop store):

```
sudo snap install rockcraft --channel=<chosen channel> --classic
```

For example:

```
sudo snap install rockcraft --channel=latest/edge --classic
```

Testing Rockcraft

Once installed, you can make sure that Rockcraft is actually present in the system and ready to be used:

```
rockcraft --version
```

The output will be similar to:

```
rockcraft 0.0.1.dev1
```

2.2 How to build a ROCK with Rockcraft's GitHub Action

Within your GitHub repository, make sure you have GitHub Actions enabled.

Navigate to `.github/workflows`, open the YAML file where you want the ROCK build to take place, and add the following steps:

```
steps:
  - uses: actions/checkout@v3
  - uses: canonical/craft-actions/rockcraft-pack@main
```

Commit and push the changes. This will trigger a new workflow run using Rockcraft to pack your ROCK based on the `rockcraft.yaml` file at your project's root.

To learn how to publish this ROCK outside the GitHub build environment and how to pass additional input parameters to this action, please refer to the action's documentation.

2.3 How to build the documentation

Create a virtual environment and activate it:

```
python3 -m venv docs/env
source docs/env/bin/activate
```

Install the documentation requirements:

```
pip install -r requirements-focal.txt -r requirements-doc.txt
```

Once the requirements are installed, you can use the provided `Makefile` to build the documentation:

```
make docs  # the home page can be found at docs/_build/html/index.html
```

Even better, serve it locally on port 8080. The documentation will be rebuilt on each file change, and will reload the browser view.

```
make rundocs
```

Note that `make rundocs` automatically activates the virtual environment, as long as it already exists.

2.4 How to create a package slice for Chisel

If your package doesn't yet have the slice definitions you actually need to **create your own slice definition** (which you can, later on, propose to be merged upstream for everyone else to use *How to make custom slice definitions available for everyone*).

Let's assume you are trying to create a slice definition for installing the OpenSSL binary into your ROCK!

2.4.1 Make sure the slice definition doesn't exist yet

To avoid re-creating a slice, check the following to see if something that fits your needs already exists:

1. Look into the upstream chisel-releases repository
2. Switch to the branch corresponding to the desired Ubuntu release for your ROCK
3. Search your package name within the list of slice definitions files
    - if you find it, open it and try to find a slice name containing the bits and pieces you need from that package

2.4.2 Structure of a slice definitions file

There can be only **one slice definitions file** for each Ubuntu package. All of the slice definitions files follow the same structure:

```yaml
# (req) Name of the package.
# The slice definition file should be named accordingly (eg. "openssl.yaml")

package: <package-name>

# (req) List of slices
slices:

    # (req) Name of the slice
    <slice-name>:

        # (opt) Optional list of slices that this slice depends on
        essential:
          - <pkgA_slice-name>
          - ...

        # (req) The list of files, from the package, that this slice will install
        contents:
            </path/to/content>:
            </path/to/another/multiple*/content/**>:
            </path/to/moved/content>: {copy: /bin/original}
            </path/to/link>: {symlink: /bin/mybin}
            </path/to/new/dir>: {make: true}
            </path/to/file/with/text>: {text: "Some text"}
            </path/to/mutable/file/with/default/text>: {text: FIXME, mutable: true}
            </path/to/temporary/content>: {until: mutate}

        # (opt) Mutation scripts, to allow for the reproduction of maintainer scripts,
        # based on Starlark (https://github.com/canonical/starlark)
        mutate: |
            ...
```

2.4.3 Find the dependencies of your package

Find the dependencies of the package for which you want to create a new slice definition (`openssl` in this guide) with this command:

```
apt show openssl
```

The output will be similar to:

```
package: openssl
Version: 3.0.2-0ubuntu1.7
Origin: Ubuntu
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer: Debian OpenSSL Team <pkg-openssl-devel@alioth-lists.debian.net>
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Depends: libc6 (>= 2.34), libssl3 (>= 3.0.2-0ubuntu1.2)
```

From the above output, you can confirm that `openssl` **depends on** `libc6` and `libssl3`. So when creating your slice definitions file for OpenSSL, you will need to remember to include those packages' slices as a dependency as well, whenever needed. Let's do that in the following section.

2.4.4 Create your slice definition

You now have everything needed to actually define the OpenSSL slice that will install the content you are looking to have in the final ROCK. Since you are looking to install just the OpenSSL binaries from the `openssl` package, what about naming this new slice **bins**? Let's go for it:

1. **What is the name of your slice definitions file?** It is a YAML file called `openssl.yaml`
2. **What package name should be defined inside this file?** The package name is `openssl`
3. **What is your slice name?** It should be called `bins`
4. **What contents do you need from the OpenSSL package?** Just the binaries - `/usr/bin/c_rehash` and `/usr/bin/openssl`
5. **Does your slice depend on any other package slice?** Yes, OpenSSL depends on `libc6` and `libssl3`
    - **Do these two packages have slice definitions files upstream?** Yes, there is already a slice definitions file for libc6 and another one for libssl3. If these dependencies were not present in the upstream Chisel release, you would also need to create their corresponding slice definitions
    - **Which slices do you depend on then?** Since you only want the OpenSSL binaries, you might only need the libraries from `libc6` and `libssl3`, as well as the configuration files from `libc6` and `openssl` themselves.

Create a new YAML file named `openssl.yaml`, with the following content:

```yaml
package: openssl
slices:
    bins:
        essential:
          - libc6_libs
          - libc6_config
          - libssl3_libs
          - openssl_config
        contents:
            /usr/bin/c_rehash:
```

(continues on next page)

(continued from previous page)

```
            /usr/bin/openssl:

    config:
        contents:
            /etc/ssl/private/:
            /etc/ssl/openssl.cnf:
            /usr/lib/ssl/certs:
            /usr/lib/ssl/openssl.cnf:
            /usr/lib/ssl/private:
```

Notice the unforeseen new slice `config`. Because your OpenSSL binaries depend on the OpenSSL configuration files, and those were not yet present anywhere in the Chisel releases upstream, you also need to create that slice! You may also ask **"why not put those configuration files inside the "bins" slice"**? You could! But we recommend, as a best practice, to separate and group contents according to their nature, as you may tomorrow need to create a new slice definition that only needs the OpenSSL configurations and not the binaries.

And that's it. This is your brand new slice definitions file, which will allow Chisel to install **just** the OpenSSL binaries (and their dependencies) into your ROCK! To learn about how to actually use this new slice definition file and publish it upstream for others to use, please check the following guides.

2.5 How to install your own package slice

When a specific package slice is not available on the upstream Chisel releases, you will more likely end up creating your own slice definition.

Once you have it though, the most obvious question is: **how can I install this custom slice with Chisel?**

Let's assume you want to install the OpenSSL binaries slice created in here…

First, clone the Chisel releases repository:

```
# Let's assume we are working with Ubuntu 22.04
git clone -b ubuntu-22.04 https://github.com/canonical/chisel-releases/
```

This repository acts as the database of slice definitions files for each Chisel release (Chisel releases are named analogously to Ubuntu releases, and mapped into Git branches within the repository).

Chisel will only recognize slices belonging to a Chisel release, so you need to copy your slice definitions file - `openssl.yaml` in this example - into the `chisel-releases/slices` folder. Note that if a slice definitions file with the same name already exists, it most likely means that the package you're slicing has already been sliced before, and in this case, you only need to merge your changes into that existing file.

At this point, you should be able to find your custom OpenSSL slice `bins` in the local Chisel release:

```
grep -q "bins" chisel-releases/slices/openssl.yaml && echo "My slice exists"
```

If you wanted to test it with Chisel alone, you could now simply run

```
# Testing with Chisel directly:
mkdir -p my-custom-openssl-fs
chisel cut --release ./chisel-releases --root my-custom-openssl-fs openssl_bins
```

You should end up with a folder named "my-custom-openssl-fs" containing a few folders, amongst which there would be `./usr/bin/openssl`.

To install the custom package slice into a ROCK though, you need to use Rockcraft!

Start by initializing a new Rockcraft project:

```
rockcraft init
```

After this command, you should find a new `rockcraft.yaml` file in your current path.

Adjust the `rockcraft.yaml` file according to the following content (feel free to adjust the metadata, but pay special attention to the `parts` section):

```
name: custom-openssl-rock
base: bare
build_base: "ubuntu:22.04"
version: '0.0.1'
summary: A chiselled ROCK with a custom OpenSSL slice
description: |
    A ROCK containing only the binaries (and corresponding dependencies) from the␣
↪OpenSSL package.
    Built from a custom Chisel release.
license: GPL-3.0
platforms:
    amd64:
parts:
    build-context:
        plugin: nil
        source: chisel-releases/
        source-type: local
        override-build:
            chisel cut --release ./ --root ${CRAFT_PART_INSTALL} openssl_bins
```

The "build-context" part allows you to send the local `chisel-releases` folder into the builder. The "override-build" enables you to install your custom slice. Please not that this level of customization is only needed when you want to install from a custom Chisel release. If the desired slice definitions are already upstream, then you can simply use `stage-packages`, as demonstrated in *here*.

Build your ROCK with:

```
rockcraft
```

The output will be:

```
Launching instance...
Retrieved base bare for amd64
Extracted bare:latest
Executed: pull build-context
Executed: pull pebble
Executed: overlay build-context
Executed: overlay pebble
Executed: build build-context
Executed: build pebble
Executed: stage build-context
Executed: stage pebble
Executed: prime build-context
Executed: prime pebble
```

(continues on next page)

(continued from previous page)

```
Executed parts lifecycle
Created new layer
Labels and annotations set to ['org.opencontainers.image.version=0.0.1', 'org.
↪opencontainers.image.title=custom-openssl-rock', 'org.opencontainers.image.ref.
↪name=custom-openssl-rock', 'org.opencontainers.image.licenses=GPL-3.0', 'org.
↪opencontainers.image.created=2022-11-25T14:00:47.470814+00:00', 'org.opencontainers.
↪image.base.digest=c4d1cae85485fb5bf8483a440f7e47b0fd2592ff114117cd4763604fbf6ae7a4']
Exported to OCI archive 'custom-openssl-rock_0.0.1_amd64.rock'
```

Test that the OpenSSL binaries have been correctly installed with the following:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:custom-
↪openssl-rock_0.0.1_amd64.rock docker-daemon:chisel-openssl:latest
```

The output will be:

```
Getting image source signatures
Copying blob 253d707d7e97 done
Copying blob 7044a53e1935 done
Copying config c114b59704 done
Writing manifest to image destination
Storing signatures
```

And after:

```
docker run --rm chisel-openssl openssl
```

The output of the Docker command will be OpenSSL's default help message:

```
help:

Standard commands
asn1parse         ca                ciphers           cmp
cms               crl               crl2pkcs7         dgst
dhparam           dsa               dsaparam          ec
ecparam           enc               engine            errstr
fipsinstall       gendsa            genpkey           genrsa
help              info              kdf               list
mac               nseq              ocsp              passwd
pkcs12            pkcs7             pkcs8             pkey
pkeyparam         pkeyutl           prime             rand
rehash            req               rsa               rsautl
s_client          s_server          s_time            sess_id
<... many more lines of output>
```

And that's it! You've now built your own ROCK from a custom Chisel release. Next step: share your slice definitions file with others!

2.6 How to make custom slice definitions available for everyone

At this stage, you have created some package slice definitions and you have a custom Chisel release in your local development environment. You have also tested this custom Chisel release, and it works! You believe there are others who could really use it as well, so **how can you make it accessible to everyone**?

It is as simple as proposing your changes into the upstream Chisel releases repository:

1. Fork this repository https://github.com/canonical/chisel-releases and clone your fork:

```
# Let's assume we are working with Ubuntu 22.04
git clone -b ubuntu-22.04 https://github.com/<your_github_username>/chisel-releases.git
```

2. Create a branch:

```
cd chisel-releases
git checkout -b create-openssl-bins-slice
```

3. Add and commit your modifications:

```
cp <path/to/your/slice/definitions/file> slices
git add slices/
git commit -m "feat: add new slice definitions for 'name_of_the_package'"
git push origin create-openssl-bins-slice
```

Create a pull request and wait for it to be merged.

And that's it! Your custom Chisel release and new slice definitions are now available in Chisel, and anyone can use them. **Congrats**! And thank you for your contribution.

CHAPTER

THREE

REFERENCE

A Rockcraft project is defined in a YAML file named `rockcraft.yaml` at the root of the project tree in the filesystem.

This *Reference* section is for when you need to know which options can be used, and how, in this `rockcraft.yaml` file.

3.1 Format specification

```
# The name of the ROCK.
name: <name>

# (Optional) The human-readable title of the ROCK. Defaults to name.
title: <title>

# Short summary describing the ROCK.
summary: <summary>

# Long multi-line description of the ROCK.
description: |
  <description>

# The ROCK version, used within the ROCK OCI tag.
version: <version>

# The system image and version the application will be layered on.
base: ubuntu:18.04 | ubuntu:20.04 | ubuntu:22:04 | bare

# (Optional) The system and version on top of which the application
# will be built. Defaults to base.
build-base: ubuntu:18.04 | ubuntu:20.04 | ubuntu:22:04

# The license, in SPDX format, of the software packaged inside the ROCK.
# This field is case insensitive.
license: <license>

# (Optional) The container entry point.
entrypoint: [<path>, ...]

# (Optional) The container command line, used as arguments for
# entrypoint. If the entrypoint is not defined, the first item in
```

(continues on next page)

(continued from previous page)

```yaml
# the cmd list the command to execute.
cmd: [<arg>, ...]

# (Optional) A list of keys and values defining the container's
# runtime environment variables.
env:
  - <var name>: <value>

# List of architecture-specific ROCKs to be built.
# Supported architectures are: amd64, arm64, arm, i386, ppc64le, riscv64 and s390x.
platforms:
  # If matching a valid architecture name, it must be the same as "build-for".
  <entry>:
    # Host architectures where the ROCK can be built.
    # Required when "build-for" is specified, otherwise it defaults to <entry>
    build-on: [<arch>, ...]
    # (Optional) Target architecture the ROCK will be built for.
    # Defaults to <entry>.
    build-for: <arch>

# The parts used to build the application.
parts:
  <part name>:
    ...
```

3.1.1 Example

```yaml
name: hello
title: Hello World
summary: An Hello World ROCK
description: |
  This is just an example of a Rockcraft project
  for an Hello World ROCK.
version: latest
base: bare
build-base: ubuntu:22.04
license: Apache-2.0
entrypoint: [/usr/bin/hello, -t]
env:
  - VAR1: value
  - VAR2: "other value"
platforms:
  amd64:
  arm:
    build-on: ["arm", "arm64"]
  ibm:
    build-on: ["s390x"]
    build-for: s390x

parts:
```

(continues on next page)

(continued from previous page)

```
  hello:
    plugin: nil
    stage-packages:
      - hello
```

3.2 Rockcraft parts

The main building blocks of a ROCK are *parts*.

If this sentence sounds familiar, it's because **it is familiar**! Rockcraft parts are inherited from other existing Craft tools like Snapcraft and Charmcraft.

Rockcraft *parts* go through the same lifecycle steps as Charmcraft and Snapcraft parts.

The way the *parts*' keys and values are used in the *rockcraft.yaml* is exactly the same as in *`snapcraft.yaml`*_ (here is how you define a *part*).

Albeit being fundamentally identical to Snapcraft parts, Rockcraft parts actually offer some extended functionality and keywords:

- **stage-packages**: apart from offering the well-known package installation behavior, in Rockcraft the `stage-packages` keyword actually supports chiseled packages as well (*learn more about Chisel*). To install a package slice instead of the whole package, simply follow the Chisel convention *<packageName>_<sliceName>*.

3.2.1 Example

```
parts:
  chisel-openssl-binaries-only:
    plugin: nil
    stage-packages:
      - openssl_bins
      - ca-certificates_data

  package-hello:
    plugin: nil
    stage-packages:
      - hello
```

NOTE: at the moment, it is not possible to mix packages and slices in the same stage-packages field.

3.3 Rockcraft commands

3.3.1 Lifecycle commands

Lifecycle commands can take an optional parameter `<part-name>`. When a part name is provided, the command applies to the specific part. When no part name is provided, the command applies to all parts.

clean

Removes a part’s assets. When no part is provided, the entire build environment (e.g. the LXD instance) is removed.

pull

Downloads or retrieves artifacts defined for each part.

overlay

Execute operations defined for each part on a layer over the base filesystem, potentially modifying its contents.

build

Builds artifacts defined for each part.

stage

Stages built artifacts into a common staging area, for each part.

prime

Prepare, for each part, the final payload to be packed as a ROCK, performing additional processing and adding metadata files.

pack

This is the default lifecycle command for `rockcraft`.

Process parts and create the ROCK as an OCI archive file containing the project payload with the provided metadata.

3.3.2 Other commands

init

Initializes a rockcraft project with a boilerplate `rockcraft.yaml` file.

help

Shows information about a command.

CHAPTER

FOUR

EXPLANATION

Getting past the technical matters surrounding Rockcraft, from a higher perspective, you might be asking *"but what is this after all?"* and *"why do I need it?"*.

Let's then use this page to go a bit deeper into the concepts and definitions behind Rockcraft.

4.1 What is a ROCK?

Rockcraft builds ROCKs, but **what is a ROCK**?

In short, a ROCK is just an OCI-compliant container image! Period.

A ROCK can live on any existing container registry, very much like any other Docker image out there. You can run a ROCK very much like any other container image…for example: `docker run <rock> ...` will work just fine.

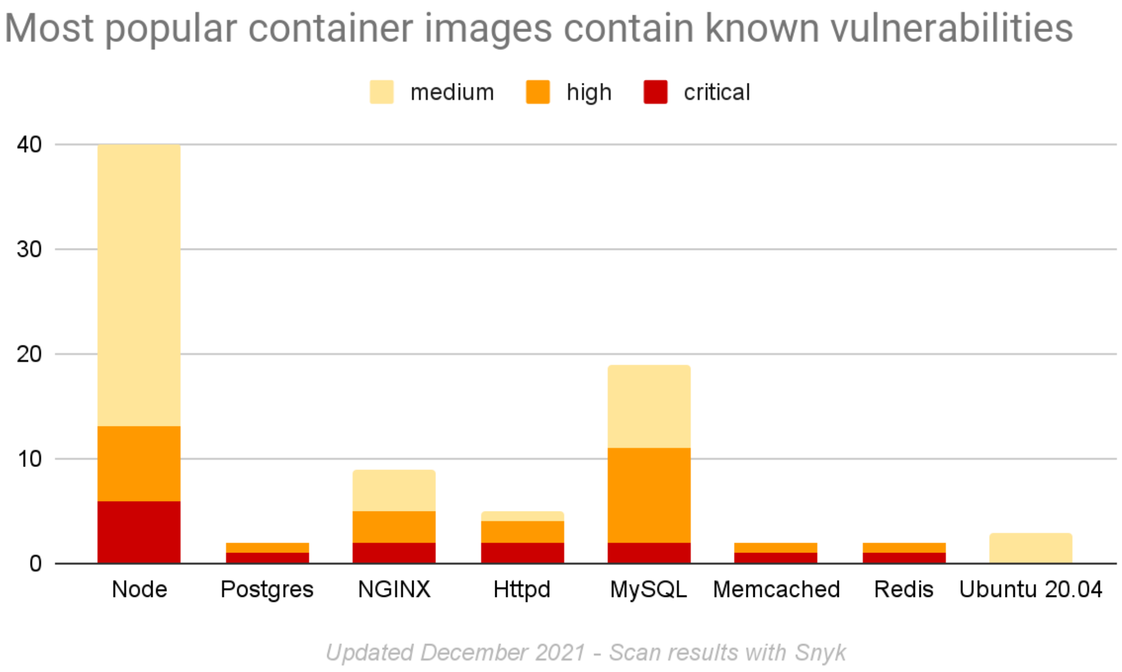
And the same applies to container image builds, in the sense that if you already have image build recipes (like Dockerfiles) and you want to start basing your own images on an existing ROCK, that will work just fine too!

4.1.1 So why do I need ROCKs?

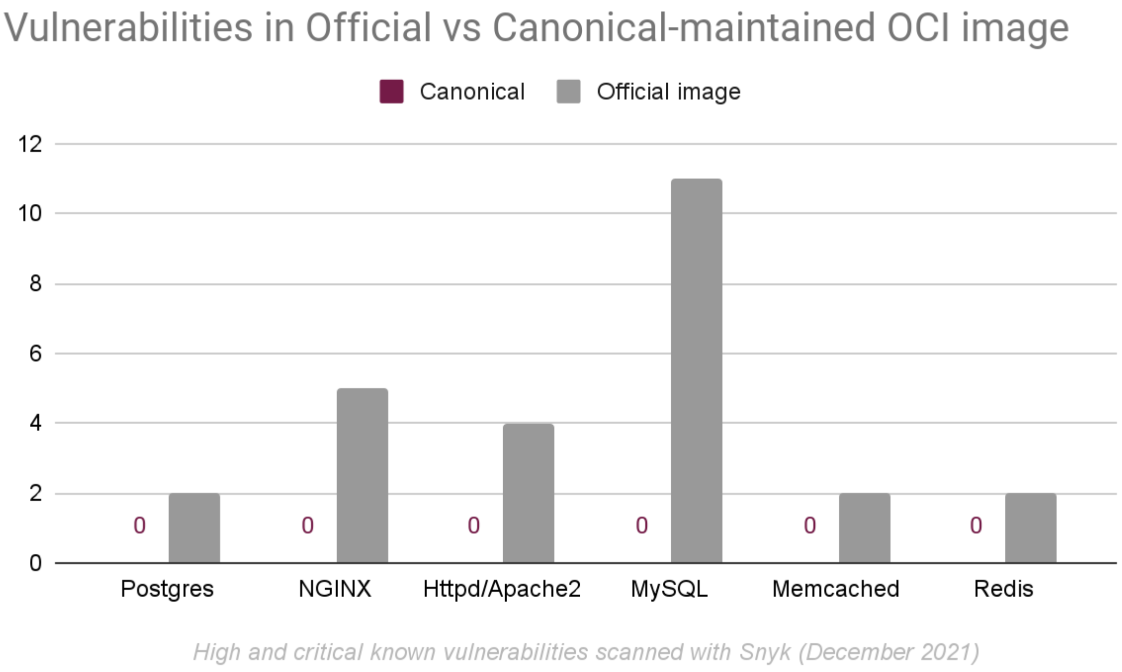
Now, this is where things get interesting. To answer this question, we first need to look at the current state of the art with respect to the existing container image offerings out there.

It is easy to find public studies (like Unit 42 / Znet and Snyk's state of open source security report 2020) where the findings state a concerning number of containers at risk deployed in cloud infrastructures.

In fact, both these studies and our own assessments (dated from December 2021) show that the most popular images in Docker Hub contain known vulnerabilities, with Ubuntu being the only one without any critical or high ones.



Sure, consumers could venture to fix these vulnerabilities themselves, but not only would this increase the cost and proliferation of images, but it wouldn't be easy to accomplish due to the lack of expertise in the subject matter. The right approach is to actually fix the vulnerabilities at their source! And Canonical has already started doing this. If we compare some of the Docker Official container images vs some of the ones maintained by Canonical, we can verify that the latter have no high/critical vulnerabilities in them!



So this is where the motivation for a new generation of OCI images (aka ROCKs) starts - the need for more secure container images! And while this need might carry the biggest weight in the container users' demands, other values come into play when selecting the best container image, such as:

- stability
- size
- compliance
- provenance

You can find these values and their relevance in this report.

This brings us to the problem statement behind ROCKs:

> *How might we redesign secure container images for Kubernetes developers and application maintainers, considering the Top 10 Docker images are full of vulnerabilities, except Ubuntu?*

A ROCK is:

- **secure** and **stable**: based on the latest and greatest Ubuntu releases;
- **OCI-compliant**: compatible with all the popular container management tools (Docker, Kubernetes, etc.);
- **dependable**: built on top of Ubuntu, with a predictable release cadence and timely security updates;
- **production-grade**: tested and secured by default.

4.2 Do I need to use Rockcraft?

If you want to build a proper ROCK, yes, we'd recommend you do. This is not to say you wouldn't be able to build ROCK-like container images with your own tools, but Rockcraft has been developed precisely to offer an easy way to build production-grade container images.

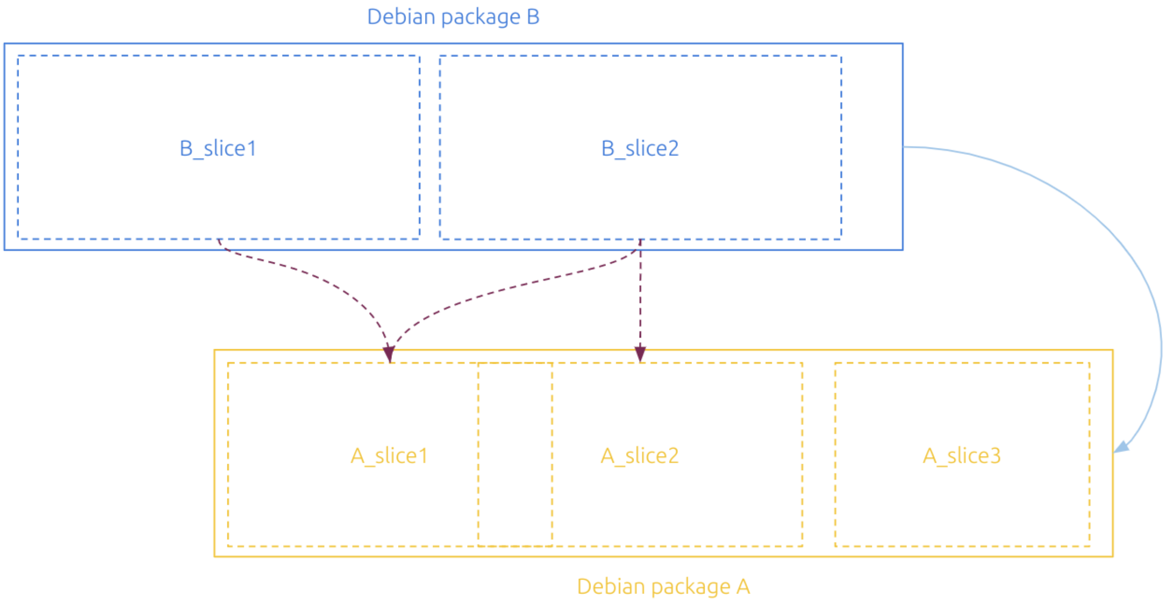
Furthermore, Rockcraft is built on top of existing concepts and within the same family as Snapcraft and Charmcraft, such that its adoption becomes seamless for those already used to building Snaps and Charms.

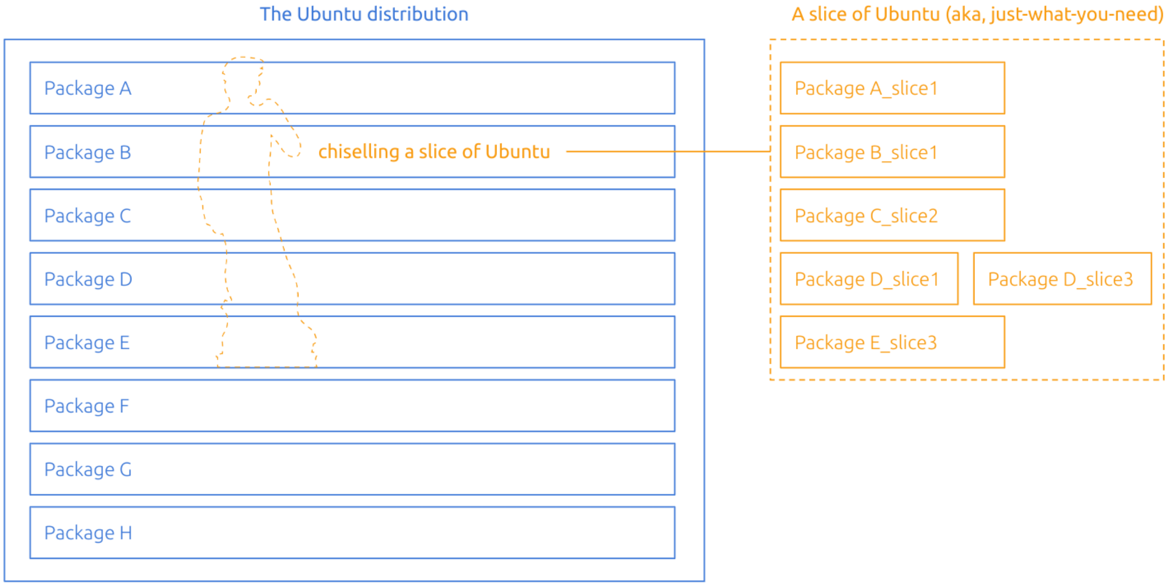
4.3 What is Chisel?

As the name says, Chisel is a tool for carving and cutting. But carving and cutting what? Even though we are talking about ROCKs, it's not like these are actual solid masses one can physically interact with…

Chisel is a software tool for carving and cutting **Debian packages**!

One of the key value propositions of Rockcraft is the ability to build truly minimal container images while honoring the Ubuntu experience. Well, when having a closer look at a Debian package, it is easy to understand that this artifact is purely an archive that can be inspected, navigated and deconstructed. Having noted this, we've come up with the idea of **Package Slices** - minimal, complimentary and loosely coupled sets of files, based on the package's metadata and content. Slices are basically subsets of the Debian packages, with their own content and set of dependencies to other internal and external slices.





This image depicts a simple case, where both packages *A* and *B* are deconstructed into multiple slices. At a package level, *B* depends on *A*, but in reality, there might be files in *A* that *B* doesn't actually need (eg. *A_slice3* isn't needed for *B* to function properly). With this slice definition in place, Chisel is able to extract a highly-customized and specialized Slice of the Ubuntu distribution, which one could see as a block of stone from which we can carve and extract small and relevant parts we need to run our applications. It is ideal to support the creation of smaller but equally functional container images.

> *"The sculpture is already complete within the marble block, before I start my work. It is already there, I just have to chisel away the superfluous material."*
>
> - Michelangelo

In the end, it's like having a slice of Ubuntu - get *just what you need*. You can *have your cake and eat it too*!

4.3.1 How to use Chisel?

Chisel has been integrated with Rockcraft in a way that it becomes seamless to users. Packages and slices can be both installed via the `stage-packages` field without any ambiguities because slices follow an underscore-driven naming convention. For instance, `openssl` means the whole OpenSSL package, while `openssl_bins` means just the binaries slice of the OpenSSL package. And that's it. Rockcraft will then take care of the installation and priming of your content into the ROCK. There's an example *here*.

Chisel isn't, however, specific to Rockcraft. It can be used on its own! It relies on a database of slices that are indexed per Ubuntu release. So for example, the following command:

```
chisel cut --release ubuntu-22.04 --root myrootfs libgcc-s1_libs libssl3_libs
```

would look into the Ubuntu Jammy archives, fetch the provided packages and install only the desired slices into the `myrootfs` folder.

To learn more about Chisel and how it works, have a look at https://github.com/canonical/chisel.

Do you need a package slice that doesn't exist yet? Please feel free to propose your slice definition in https://github.com/canonical/chisel-releases.

Tutorial **Get started** with a hands-on introduction to Rockcraft

How-to guides **Step-by-step guides** covering key operations and common tasks

Reference **Technical information** about the `rockcraft.yaml` format

Explanation **Discussion and clarification** of key topics

CHAPTER FIVE

PROJECT AND COMMUNITY

Rockcraft is a member of the Canonical family. It's an open source project that warmly welcomes community projects, contributions, suggestions, fixes and constructive feedback.

- Ubuntu Code of Conduct.
- Canonical contributor licenses agreement.