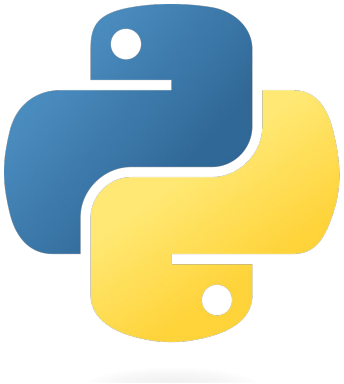
UltiMaker

Cura Build System

Conan & GitHub Actions

Cura Ecosystem

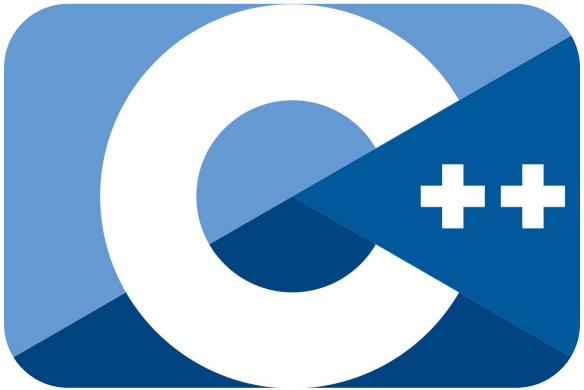


Own Python Modules
- Cura
- Uranium
- libCharon

Third-party
- Numpy
- Shapely
- …

Own C++ Python-bindings
- PyArcus
- PySavitar
- pynest2d

Third-party
- Python
- PyQt6
- …



Own C++ libraries and executables
- CuraEngine
- libArcus
- libSavitar
- libnest2d

Third-party libraries
- Python
- Qt6
- …

Managing Python dependencies

Package managers:

- **Pip** (we use this)
- Pipenv
- Poetry

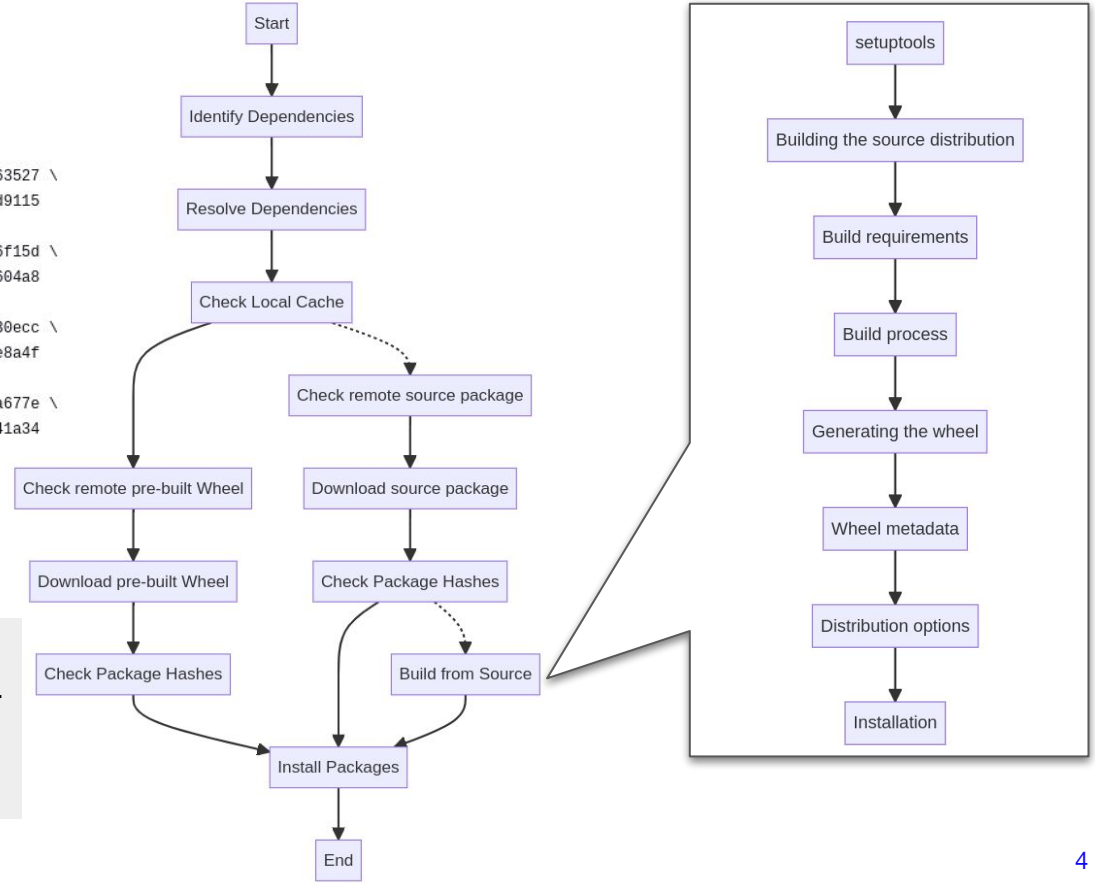
Artifactory:

- PyPi.org (we use this for most)
- Various generic Artifactories (Google, AWS, …)
- Simple bucket Google (we use for precompiled wheels for Numpy+MKL+Intel)
- Git repository (we use this for libCharon, GCodeAnalyzer, CuraEngine_grpc_definitions)
- Jfrog Artifactory (we have it we don’t use it)

Installing Python dependencies

What happens if you do a:
pip install -r requirements.txt

```
urllib3==1.25.9 \
    --hash=sha256:3018294ebefce6572a474f0604c2021e33b3fd8006ecd11d62107a5d2a963527 \
    --hash=sha256:88206b0eb87e6d677d424843ac5209e3fb9d0190d0ee169599165ec25e9d9115
mypy-extensions==0.4.3 \
    --hash=sha256:090fedd75945a69ae91ce1303b5824f428daf5a028d2f6ab8a299250a846f15d \
    --hash=sha256:2d82818f5bb3e369420cb3c4060a7970edba416647068eb4c5343488a6c604a8
tomli==2.0.1 \
    --hash=sha256:939de3e7a6161af0c887ef91b7d41a53e7c5a1ca976325f429cb46ea9bc30ecc \
    --hash=sha256:de526c12914f0c550d15924c62d72abc48d6fe7364aa87328337a31007fe8a4f
typing-extensions==3.10.0.2 \
    --hash=sha256:49f75d16ff11f1cd258e1b988ccff82a3ca5570217d7ad8c5f48205dd99a677e \
    --hash=sha256:f1d25edafde516b146ecd0613dabcc61409817af4766fbbcfb8d1ad4ec441a34
```



Info

All (in-)direct packages are specified in our *requirements.txt* because we check the hashes to guard against dependency injections

Managing C++ dependencies

Package managers:

- **Conan** (we use this)
  - C++ dependency manager
  - Open Source with active community
    - Slack #conan on cpplang
  - Written in Python
    - Easily extendable
  - 1000+ recipes of open-source projects
    - CCI Conan-Center-Index
- vcpkg
- Buckaroo
- Hunter
- Build2

Artifactory:

- Jfrog Artifactory

C++ dependency manager challenges

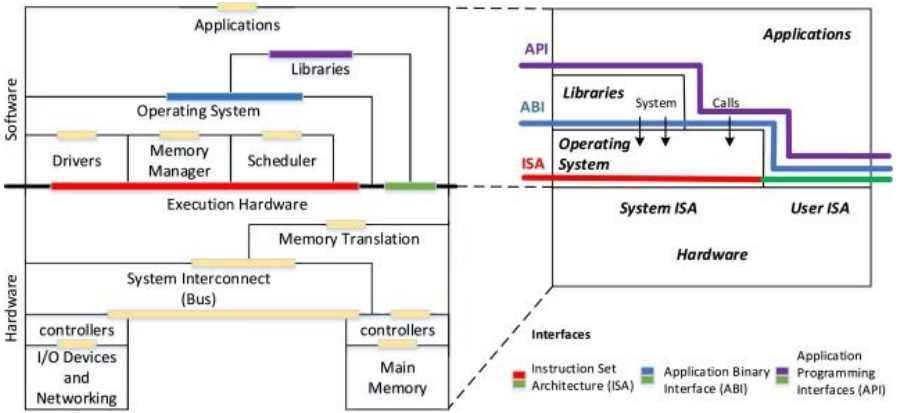
API (Application Programming Interface)

A “contract” or a set of rules and protocols that allow different software applications to communicate and interact with each other

```
#ifndef CALCULATOR_H
#define CALCULATOR_H

class Calculator {
public:
    // Constructor
    Calculator();

    // Basic arithmetic operations
    int add(int a, int b);
    int subtract(int a, int b);
    int multiply(int a, int b);
    int divide(int a, int b);
};

#endif
```

ABI (Application Binary Interface)

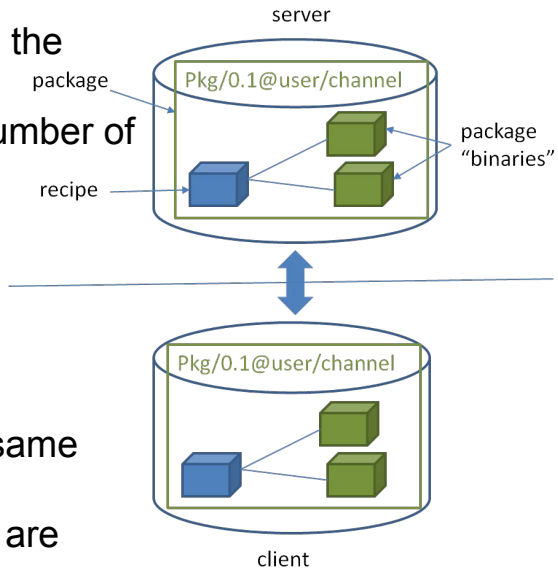
A low-level interface between different binary software components, such as libraries, operating systems and hardware.



Source: https://www.sciencedirect.com/topics/computer-science/application-binary-interface

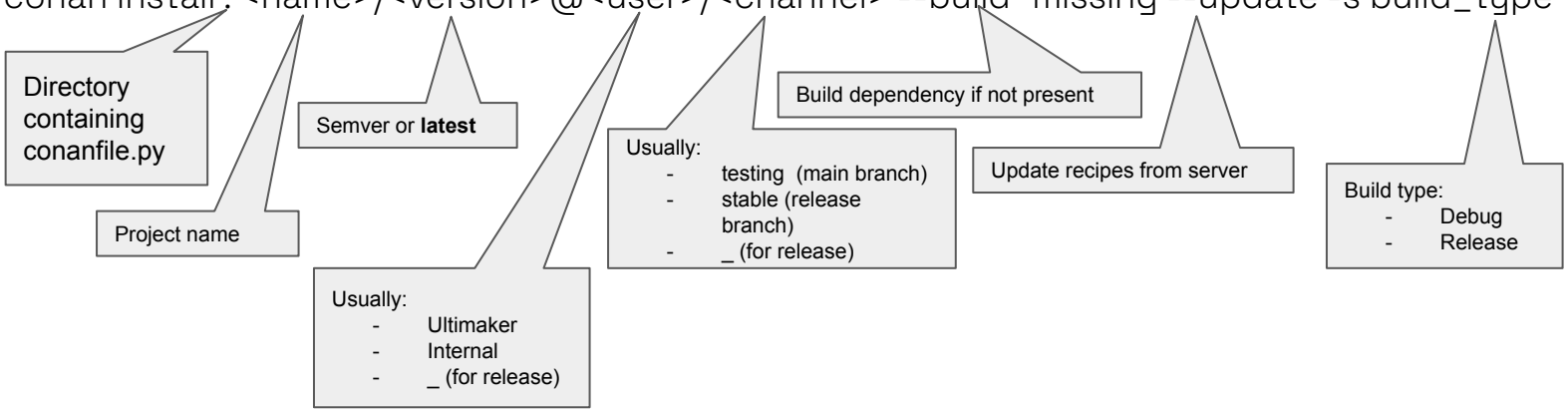
Conan

- A package is defined by a `conanfile.py`
- Which defines the package's dependencies, sources, how to build the binaries from sources, etc.
- One package "`conanfile.py`" recipe can generate any arbitrary number of binaries, one for each different platform and configuration:
  - Operating system
  - Architecture
  - Compiler
  - Build type
  - …
- These binaries can be created and uploaded to a server with the same commands in all platforms
- Only necessary binaries for the current platform and configuration are downloaded
- If the compatible binary is not available, the package can be built from sources in the client too



Conan usage

- All Cura related projects/repository have a conanfile.py
- These projects share the same build/system configuration
  Which can be installed with:
  conan config install https://github.com/Ultimaker/conan-config.git
- Such a project can usually be initialized by running:
  conan install . <name>/<version>@<user>/<channel> --build=missing --update -s build_type=Debug



- After this run the regular CMake, Ninja and/or build commands

Creating a package

A conan package can be created from any project with a conanfile.py

conan create . <name>/<version>@<user>/channel --build=missing --update

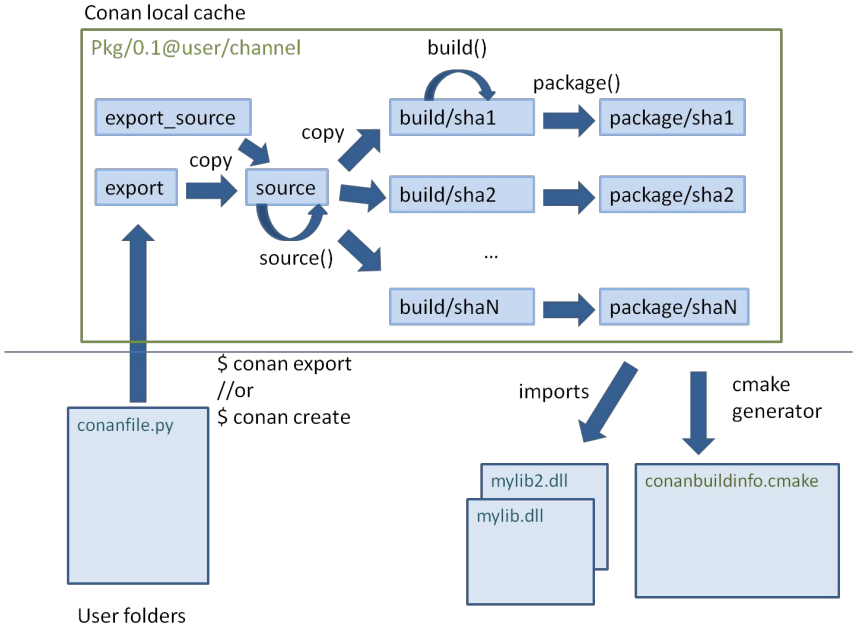
Info

Such a package will be created and placed in the local conan cache, such that it can be used by other projects latter on.

Info

These package can be shared with everyone by uploading them to the Artifactory

conan upload <name>/<version>@<user>/channel -r cura



CONAN 1.X CHEATSHEET

Conan 2.0 is now out! please head to conan.io to learn more

JFrog | CONAN C/C++ Package Manager

Show Local Client Configuration

Conan application configuration

```
$ conan config get
```

Contents of a profile (eg. default)

```
$ conan profile show default
```

Remote Repositories

```
$ conan remote list
```

Add and modify configurations

Install collection of configs

```
$ conan config install <url>
```

Change a single config value

```
$ conan config set general.revisions_enabled=1
```

Add a remote

```
$ conan remote add my_remote <url>
```

Provide credentials for remote

```
$ conan user -p <password> -r my_remote <username>
```

Display information from recipes or references

Displays attributes of conanfile.py

```
$ conan inspect <path> -a <attribute>
```

Displays content of conanfile.py for a reference

```
$ conan get <reference>
```

Display dependency graph info for a recipe

```
$ conan info <path_or_reference>
```

Search Packages

Search for packages in a remote

```
$ conan search zlib -r conancenter
```

Consume Packages

Install package using just a reference

```
$ conan install <package_reference>
```

Install list of packages from conanfile

```
$ cat conanfile.txt
[requires]
zlib/1.2.11
$ conan install <path_to_conanfile>
```

Consume packages in build system via generators

```
$ cat conanfile.txt
[requires]
zlib/1.2.11
[generators]
cmake_find_package
msbuild
make
```

Install requirements and generate files

```
$ mkdir build && cd build
$ conan install ..
```

Run your build system (one of the following)

```
$ cmake .. && cmake --build .
$ msbuild myproject.sln
$ make
```

Create a package

Create a recipe (conanfile.py) from templates

```
$ conan new <reference> -m <template>
```

Just export the recipe to local cache

```
$ conan export <path_to_conanfile>
```

Create package from recipe for one configuration
Also implicitly does install and export steps

```
$ conan create . -pr <profile>
```

Upload a Package

One or more with wildcard support, with binaries

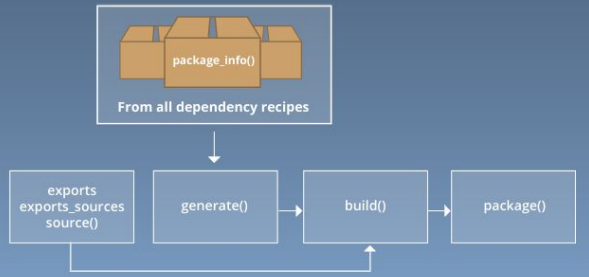
```
$ conan upload zlib* -r remote --all
```

Copy packaged files out of Conan cache

Using the deploy generator

```
$ conan install zlib/1.2.11@ -g deploy
```

Conan Recipe Methods in Package Creation



Extending Conan with Python Virtual Environment generator

Problem:

- **Cura** is Python based
- It has **Pip** managed dependencies
- But, we also have our own dependencies:
  - Compiled Python modules such as: **PyArcus, PySavitar, pynest2d** which depends on **libArcus, libSavitar, libnest2d**
  - **Python** interpreter (managed with Conan)
  - **CuraEngine** a terminal application, with its own C++ dependencies

Solution:

- Custom generator which will generate an `activate` or `Activate.ps1` script with all the correct paths
- Setup a Virtual Python Environment with the Python interpreter managed by Conan
- Install Pip managed dependencies for Cura and Uranium
- Run it with `conan install ... -g VirtualPythonEnv`

https://github.com/Ultimaker/conan-config/blob/master/generators/VirtualPythonEnv.py

Extending Conan with PyCharm run target generator

Problem:

- **Cura** is Python based
- IDE of choice in UltiMaker is **PyCharm**
- It has **Pip** managed dependencies
- But, we also have our own dependencies:
  - Compiled Python modules such as: **PyArcus, PySavitar, pynest2d** which depends on **libArcus, libSavitar, libnest2d**
  - **Python** interpreter (managed with Conan)
  - **CuraEngine** a terminal application, with its own C++ dependencies

Solution:

- Custom generator which will generate a PyCharm run targets for Cura and Unit Tests in the `.run` folder with all the correct paths
- Run it with `conan install ... -g PyCharmRunEnv`

https://github.com/Ultimaker/conan-config/blob/master/generators/PyCharmRunEnv.py

Extending Conan with GitHub Action Virtual environment

Problem:

- Activating the conan build or run environment in a GitHub action after running `conan install ...` or `conan create ...`
- Each step in a GitHub workflow is a separate process, running: `source activate` or `Activate.ps1` won't transfer to the next step

Solution:

- Custom generator which will generate a file that can be used to write the paths to the `GITHUB_ENV` file.
- The `GITHUB_ENV` file is read at the start of each workflow step
- Run it with `conan install ... -g GitHubActionsRunEnv -g GitHubActionsBuildEnv`

https://github.com/Ultimaker/conan-config/blob/master/generators/GitHubActionsBuildEnv.py
https://github.com/Ultimaker/conan-config/blob/master/generators/GitHubActionsRunEnv.py

Extending Conan with translation tools

- **Cura** & **Uranium** use i18n translation strings
- These projects have a build dependency `translationextractor`
- Which is a custom build tool that runs with a `conan install ...` command on a Unix shell
  - Extract the strings from the source files using `xgettext`, this will update the `*.pot` files
  - Update the translation from the `*.pot` files using `msmerge`, this will update the `*.po` files

https://github.com/Ultimaker/conan-ultimaker-index/blob/main/recipes/translationextractor/conanfile.py

Info

Compiling the `*.po` files to `*.mo` files such that they are actually used by Cura is done with `msgfmt` during the build step

https://github.com/Ultimaker/Cura/blob/63e78d313c66588b8425d08a948fe16e3c2f2e00/conanfile.py#L328

Extending Conan with SIP Build tools and pyproject.toml toolchain

- We have a couple of C++ projects (**libArcus, libnest2d, libSavitar, …**) which we use as Python modules
- The mapping of the C++ with Python is done using *.sip definition files.
- These definition files are then used to **generate C++ code** linking against Python, ctypes and our own library.
- Generation of this C++ from the *.sip files is done in the projects (**PyArcus, pynest2d, PySavitar**)
- It uses the custom conan build tool: pyprojecttoolchain, which will generate the pyproject.toml and the C++ code.

https://github.com/Ultimaker/conan-ultimaker-index/blob/main/recipes/pyprojecttoolchain/conanfile.py

- This generated C++ code needs to be compiled, the regular **Sip** build process uses setuptools, which uses distutils, setting rpath for compiled libraries on **Mac** does not work as intended.
- We created a custom CMake script which let CMake take care of the linking and building

https://github.com/Ultimaker/conan-ultimaker-index/tree/main/recipes/sipbuildtool

Creating an standalone Cura Application

- Pyinstaller bundles a Python application and all dependencies
- Works for Linux, Windows and Mac
- Configuration is managed with an UltiMaker-Cura.spec file
  - This is a Python file used by pyinstaller, to collect libraries, resources, sign and notarize them
  - We use Jinja2 to let conan generate the content of that spec file, such that it links against the version specific dependencies
- Running pyinstaller will create an dist folder with a standalone application

https://github.com/Ultimaker/Cura/blob/main/UltiMaker-Cura.spec.jinja
https://github.com/Ultimaker/Cura/blob/63e78d313c66588b8425d08a948fe16e3c2f2e00/conanfile.py#L167

□ Info

Mac code needs to be signed and notarized, this requires files to be in specific directories. Pyinstaller has a bug which we doesn't do this correctly. We use duck typing to fix this very specifically for Cura:
https://github.com/Ultimaker/Cura/blob/63e78d313c66588b8425d08a948fe16e3c2f2e00/UltiMaker-Cura.spec.jinja#L72

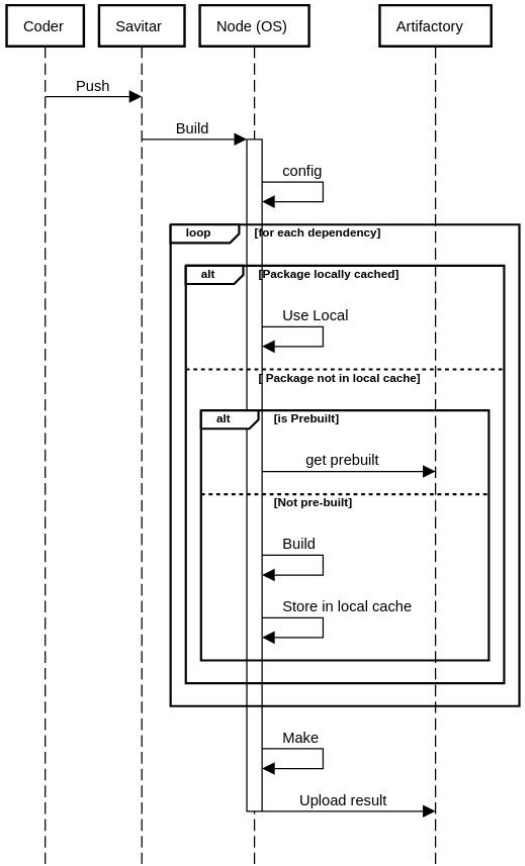
□ Info

Pyinstaller bootstraps the cura_app.py and creates an executable. Unfortunately a lot of viruses also use use pyinstaller. This means that virus scanners can get false positives for UltiMaker-Cura.exe

That is why we always test the binaries on virustotal.com
https://www.virustotal.com/gui/home/upload

GitHub Actions

- We are open-source GitHub Actions are **free**
- Each push to one of our repositories will create a new conan package



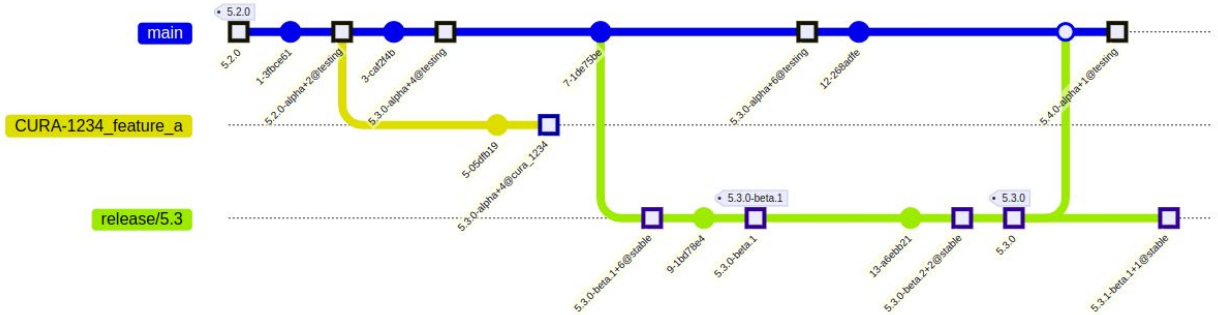
https://github.com/Ultimaker/Cura/blob/main/.github/workflows/conan-package-create.yml

https://github.com/Ultimaker/Cura/blob/main/.github/workflows/conan-package.yml

Determining a version for a Conan package

Conan `user` and `channel` differ per branch, this ensures that the `(latest)` alias is branch specific.

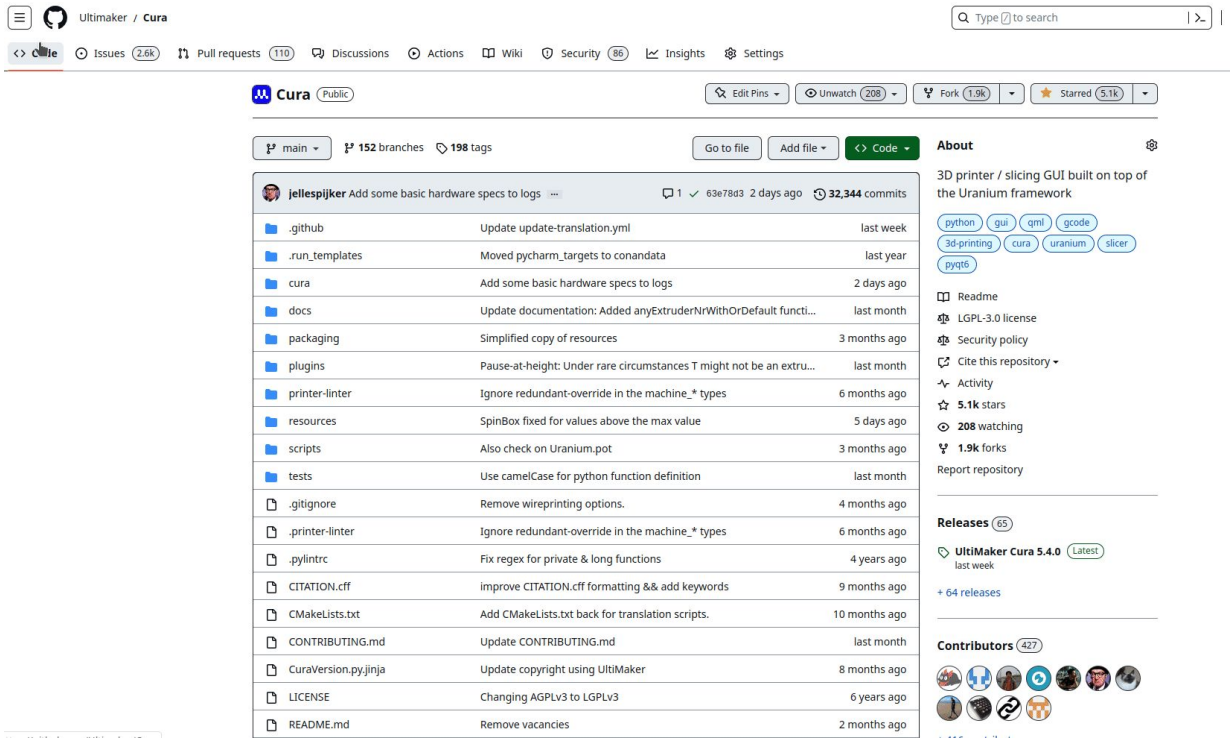
- Pushes on branch `main` are considered `testing` e.q.: `<name>/<major>.<minor>.<patch>-alpha+<commit_hash>@utlimaker/testing` or `<name>/latest@utlimaker/testing` for the latest `main` package
- Pushes on release branch `<major>.<minor>` are considered `stable` e.q.: `<name>/<major>.<minor>.<patch>-beta.<prerelease_no>+<commit_hash>@utlimaker/stable` or `<name>/latest@utlimaker/stable` for the latest release package
- Pushes on a FR/bug-fix branch `CURA-1234` will use the `cura_1234` channel e.q.: `<name>/<major>.<minor>.<patch>-alpha+<commit_hash>_cura_1234@utlimaker/cura_1234` or `<name>/latest@utlimaker/cura_1234` for the latest package
- Pull requests will use the `pr_<github_pr_number>_<commit_hash>`



https://github.com/Ultimaker/Cura/blob/main/.github/workflows/conan-recipe-version.yml

Creating an Cura installer with GitHub Actions

- Use the GH Action UI
- Run for each OS on a specific runner
- The steps are the same as you would perform locally
- List of used deps are in the GH action as a summary
- Artifacts can be downloaded in the summary



https://github.com/Ultimaker/Cura/blob/main/.github/workflows/cura-all-installers.yml
https://github.com/Ultimaker/Cura/blob/main/.github/workflows/cura-installer.yml