Calango

Packaging Guide

Building the macOS `.dmg`
and Linux `.deb` installers

Leandro Seixas

Calango 26.8.46

Abstract

Calango ships as two native installers built with CMake/CPack: a drag-and-drop disk image (`.dmg`) for macOS and a Debian package (`.deb`) for Ubuntu/Debian. This guide covers the prerequisites, the exact commands, what the packaging pipeline does under the hood, how to verify the artifacts, and the failure modes we have already hit — with their fixes. Everything described here lives in `CMakeLists.txt` (install and CPack sections) and in the `packaging/` directory of the repository.

Contents

1 Overview

Both installers come out of the standard CMake flow: *configure* → *build* → *cpack*. The platform decides the generator:

Platform	CPack generator	Artifact
macOS (`-DCALANGO_MACOS_BUNDLE=ON`)	`DragNDrop`	`calango-26.8.46-macos-arm64.dmg`
Linux	`DEB`	`calango_26.8.46_amd64.deb`

Each artifact is accompanied by a `.sha256` checksum file. The package version comes from `project(calango VERSION ...)` in `CMakeLists.txt` — the single source of truth for the version, baked into the binary as `CALANGO_VERSION` (About dialog, project files) and reused by CPack for the installer file names.

The supporting files are:

- `packaging/macos/make_icns.sh` — renders `calango.icns` from the master PNG icon in `assets/calango/` (used when no committed `.icns` exists).
- `packaging/macos/embed_python_framework.sh` — makes the `.app` independent of the build machine's Python (Section 5).
- `packaging/linux/calango.desktop` — application launcher entry.
- `packaging/linux/calango-mime.xml` — registers `application/x-calango-project` for `.calproj` project files, plus MIME types for the 12 other structure / trajectory / volumetric formats Calango reads (Section 3.6).
- `packaging/linux/postinst`, `postrm` — refresh the desktop/MIME/icon caches on install and removal.
- `packaging/linux/audit_elf_deps.py`, `packaging/macos/audit_macho_deps.py` — fail the packaging step if a shipped binary needs a library the installer does not declare (Section 2).

2 Dependency audit

The packaged `calango` binary links no BLAS/LAPACK implementation at all: the native DFT and DFTB engines that were LAPACK's only consumers in this codebase have been removed, and nothing calls `find_package(LAPACK)` any more. Nothing here has a numerical-library vendor to choose, or to get wrong.

The audit below is not about that, and does not go away with it. It guards a general class of defect — *a library from the build machine leaking into a shipped artifact without being declared* — which applies to every library the package touches: Qt, HDF5, the Python framework's extension modules, the compiler runtime. That is why it runs on every build rather than on the ones somebody thought were risky.

2.1 What the scripts check

`packaging/linux/audit_elf_deps.py` and `packaging/macos/audit_macho_deps.py` run automatically at the end of `build_deb.sh`/`build_dmg.sh`, after `cpack` produces the artifact and before it is published, and fail the packaging step — not just warn — if any shipped ELF/Mach-O needs a library that is not declared in the installer's own dependency metadata, bundled in the package itself, or part of the base OS. Intel MKL and the Intel OpenMP runtime (`libmkl_*`, `libiomp5`, `libimf`, `libirc`, `libsvml`, `libintlc`) are hard-banned in the Linux audit regardless of `Depends:` — a packaged build must never need them.

```
python3 packaging/linux/audit_elf_deps.py installers/calango_<ver>_<arch>.deb
python3 packaging/macos/audit_macho_deps.py installers/calango_<ver>_macOS.dmg
```

The Linux script needs `dpkg` for an authoritative package-ownership check (the real gate, run on the Linux packaging machine); without it, it falls back to a soname-family heuristic — useful for exercising the script's own logic elsewhere, not a substitute for the real gate. The macOS script needs no such fallback: `otool`, always present, is enough to tell a relative/system load path from a build-machine one.

2.2 Load paths and rpaths on macOS

The macOS audit checks *two* things, and the second is the one that matters. Every `LC_RPATH` entry on a bundled Mach-O must be `@loader_path`- or `@executable_path`-relative; and every relative dependency must actually *resolve* to a file present in the bundle. A bundle-relative load path is proof of *form*, not of *resolution*: a library can carry a correct-looking `@rpath/...` dependency and still be found only through a leftover absolute rpath pointing at the build machine, which no clean Mac has.

`packaging/macos/strip_leftover_rpaths.sh` enforces the first half before the audit checks it. It is wired into `CMakeLists.txt`'s install step — after `macdeployqt`'s two passes, before the final `codesign` — and deletes any non-relocatable `LC_RPATH` entry from every bundled Mach-O.

Known gap, not silently swept under the audit. `build_dmg.sh` passes the audit script two `-exclude` flags that skip the embedded Python framework and its payload directory entirely, so **that tree is not audited**. The exclusion is a concession, not a statement that the tree is clean: an interpreter built against Homebrew's OpenSSL, sqlite, zstd, mpdecimal or xz carries exactly the same class of undeclared dependency in its stdlib extension modules, and the `-probe-python` launch check catches "ASE won't import", not "every dylib is relocatable".

Build the framework via the documented default flow — the relocatable `python-build-standalone` tree (Section 5) — and re-run the audit with no `-exclude` flags to confirm a given build is actually clean there. Removing the two flags is what closes this gap for good.

3 macOS: building the `.dmg`

3.1 Prerequisites

- Xcode command-line tools (provides `sips`, `iconutil`, `codesign`, `hdiutil`).
- Qt 6 with the Svg module; Homebrew's `brew install qt` puts it at `/opt/homebrew/opt/qt`.
- CMake $\geq$ 3.21.
- A Python virtualenv with ASE installed, e.g. the project `.venv` (`python3 -m venv .venv && .venv/bin/pip install ase`).

3.2 Commands

The supported one-shot route is the driver script, which provisions the relocatable Python payload, builds, packages, and mounts the finished image to prove the shipped app can import ASE:

```
packaging/macos/build_dmg.sh
```

It also picks the build directory for you, which matters — see the warning below. To drive the steps by hand instead, from the repository root:

```
BUILD=~/Library/Caches/calango/build-dmg

cmake -S . -B $BUILD \
  -DCMAKE_BUILD_TYPE=Release \
  -DPython3_EXECUTABLE=$PWD/.venv/bin/python \
  -DCMAKE_PREFIX_PATH=/opt/homebrew/opt/qt \
  -DCALANGO_MACOS_BUNDLE=ON
```

```
cmake --build $BUILD -j 8

cd $BUILD && cpack
```

The staging tree must not be cloud-synced. A file provider (Dropbox, iCloud Drive, OneDrive, Google Drive) stamps its own extended attributes on every file it manages, and re-applies them faster than they can be cleared. `codesign` rejects those as *"resource fork, Finder information, or similar detritus"*. Since the checkout itself lives in Dropbox on this machine, a build directory named relative to it (`build-dmg`) puts `_CPack_Packages/` inside the synced tree, and the run fails at the very last step — after the full build, `macdeployqt` and the Python embed — with

```
calango.app: resource fork, Finder information, or similar detritus not allowed
error: calango.app fails strict signature validation
```

The fix is to stage outside the synced folder, as above. `build_dmg.sh` does this automatically: a `BUILD_DIR` you set yourself is refused with an explanation, and the default is relocated to `~/Library/Caches/calango/` with a notice. An already built tree does not need reconfiguring — `cpack -B` moves only the staging directory and the output:

```
cd build-dmg && cpack -G DragNDrop -B ~/Library/Caches/calango/cpack-stage
```

A finished `.dmg` is a single signed file that syncing cannot spoil, so only the staging tree has to move.

3.3 The resulting image

The result is `calango-<version>-macos-arm64.dmg`. Mounting it shows `calango.app` next to an `/Applications` symlink — the standard drag-to-install layout. There is deliberately no click-through license sheet (`CPACK_DMG_SLA_USE_RESOURCE_FILE_LICENSE OFF`): a license SLA blocks scripted `hdiutil attach` and adds friction for an MIT-licensed application.

3.4 Cleaning up afterwards

The scratch tree is several GB — a Release build of the application, the CPack staging copy of the bundle, and the provisioned Python payload — and every byte of it is reproducible. `build_dmg.sh` therefore deletes it as its last step, once the image has been built, mounted and probed:

```
==> SUCCESS
    Artifact : /path/to/repo/installers/calango_<version>_macOS.dmg
    Checksum : /path/to/repo/installers/calango_<version>_macOS.dmg.sha256
    Size     : 65M
    Verified : valid macOS disk image (hdiutil imageinfo)
    Probe    : python: 3.14.6 ase: 3.29.0
    Cleaned  : ~/Library/Caches/calango/macos-bundle-a1b2c3d4 (freed 3.4G)
               next run re-provisions the embedded Python; --keep-cache avoids that
```

Three properties of that step are deliberate:

- **It runs last, and only on success.** A run that failed anywhere keeps its tree, because that tree is exactly what is needed to diagnose the failure.
- **It deletes the per-checkout subdirectory, not the root.** The scratch directory is keyed by the repository's path (`macos-bundle-<hash>`) so that several checkouts can coexist; removing `~/Library/Caches/calango` wholesale would take another checkout's

cache with it. The root itself is then removed only if it is left empty.

- **It only ever deletes inside that root.** A `BUILD_DIR` pointing anywhere else is reported and kept — it may be a working build directory that predates the packaging run.

Pass `-keep-cache` while iterating on packaging itself. The tree caches the ∼500 MB embedded-Python payload, so a kept tree makes the following run skip the download and reuse the compiled objects as well. Cleaning up by hand after a manual `cpack` is the same idea:

```
rm -rf ~/Library/Caches/calango/cpack-stage
```

3.5 What the install step does

`cpack` internally runs the CMake install rules into a staging directory (`_CPack_Packages/`) and then wraps the result in a disk image. The install rules perform, in order:

1. **Stage the bundle.** `calango.app` is installed with the Finder icon (`calango.icns`, committed or rendered at build time by `make_icns.sh`) and the `Info.plist` metadata (bundle id `org.seixasresearch.calango`).

2. **Optionally embed a Python environment** into `Contents/Resources/python`, when the cache variable `CALANGO_EMBEDDED_PYTHON_DIR` is set (Section 5).

3. **Deploy Qt with `macdeployqt` — twice.** The tool copies the Qt frameworks and plugins into the bundle and rewrites their load paths to `@executable_path`. Two passes with `-libpath=/opt/homebrew/opt/qt/lib` are required: frameworks copied into the bundle lose the `@loader_path` context their `@rpath` dependencies resolve against, so transitive dependencies (e.g. QtGui → QtDBus, plugin dependencies) are only found when the second pass re-scans the already-copied frameworks. The first pass prints a wall of `ERROR: Cannot resolve rpath` lines — this is expected noise; judge success by the verification steps below.

4. **Embed the linked `Python.framework`.** `embed_python_framework.sh` copies the framework version the binary links against (typically Homebrew's) into `Contents/Frameworks`, replaces Homebrew's dangling `site-packages` symlink with a real directory, recreates the canonical framework symlinks, retargets the load command with `install_name_tool`, and finally re-signs the whole bundle ad hoc (`codesign --force --deep --sign -`). Without the re-sign, Apple-silicon macOS refuses to load the modified binaries.

3.6 File associations

`packaging/macos/Info.plist.in` declares document types for every format Calango reads that has a real, associable filename extension: a `CFBundleDocumentTypes` entry and a matching `UTExportedTypeDeclarations` entry each — 13 pairs, split across two tiers:

Tier	`LSHandlerRank`	What
Owner	`Owner`	`.calproj` — Calango's own project format
Alternate	`Default` / `Alternate`	`.extxyz` (`Default`), `.xyz`, `.cif`, `.vasp`, `.traj`, `.pwi`, `.lammpstrj`, `.gjf`, `.res`, `.cube`, `.xsf`, `.h5`/`.hdf5` (all `Alternate`)

`Alternate` adds Calango to Finder's "Open With" menu for a format without taking over the user's existing default handler — every one of these is a community/scientific format other applications (VESTA, OVITO, Avogadro, Mercury, . . .) also read. `.calproj` is the only format Calango defines, so it alone claims `Owner`.

Two things are deliberately *not* declared. **Extension-less filenames** — Launch Services

associates by extension or UTI, never by exact basename, so `POSCAR`/`CONTCAR` and the VASP volumetric family (`CHGCAR`/`LOCPOT`/`PARCHG`/`ELFCAR`) cannot be double-click-associated at all; `.vasp` and `.cube` cover the extension case of the same two formats, and these still open fine through the Open dialog, a CLI argument, or drag-and-drop. **Extensions too generic to claim system-wide** — `.in`, `.out`, `.data`, `.dump`, `.cell`, `.com`, `.json`, `.dat`, `.top` — even though `src/gui/FileOpenRouting.hpp` still maps several of them to an explicit ASE format hint when Calango opens them itself; declaring them in `LSItemContentTypes` would put Calango in "Open With" for large numbers of files that are not its concern.

No established system or community UTI was found to *import* for any of these formats (checked specifically for HDF5, since `.h5`/`.hdf5` are generic containers Calango does not own), so every UTI is Calango's own `org.seixasresearch.calango.*`, declared via `UTExportedTypeDeclarations`. Four reuse a real chemical-MIME-type convention for `public.mime-type` — `chemical/x-xyz`, `chemical/x-cif`, `chemical/x-gaussian-input` and `chemical/x-gaussian-cube`; the rest get an `application/x-*` string of Calango's own.

Installing by dragging the `.app` out of the `.dmg` registers the bundle with Launch Services automatically, no separate step needed. Verified directly against exactly this ad-hoc-signed, unnotarized build:

```
LSREGISTER=/System/Library/Frameworks/CoreServices.framework/Versions/A/\
  Frameworks/LaunchServices.framework/Versions/A/Support/lsregister

codesign --force --deep --sign - calango.app
"$LSREGISTER" -f calango.app
"$LSREGISTER" -dump | grep -A2 'claimed UTIs'
```

All 13 UTIs came back in `claimed UTIs:`, each with the right extension/MIME tag. **Caveat:** a freshly registered, unnotarized bundle carries a `launch-disabled` flag until Gatekeeper approves it — the document *associations* appear in "Open With" immediately, but Finder may still need the one-time right-click → Open approval (or an ordinary first launch of the app) before it will actually run Calango for a double-clicked document.

On the application side, `calango::CalangoApplication` (`src/CalangoApplication.hpp`) intercepts `QFileOpenEvent` — the Apple Event Qt surfaces for a double-click, "Open With", a Dock drop, or (one event per file) several files opened together — and routes each path to `MainWindow::loadFile`, exactly like a CLI argument, so each file opens in its own tab. An event arriving before `MainWindow` exists (the normal case on a cold Finder launch) is queued (`src/FileOpenQueue.hpp`) and replayed once the window is attached. `loadFile` itself decides project vs. structure and picks the ASE format hint (`src/gui/FileOpenRouting.hpp`) — the same routing used for the Open dialog, so format detection, error dialogs and recent-files behave identically regardless of how the file arrived.

3.7 Verification

```
APP=_CPack_Packages/Darwin/DragNDrop/calango-<version>-macos-arm64/calango.app

# 1. Signature must verify strictly, including nested code
codesign --verify --deep --strict "$APP"

# 2. The packaged binary must start and find a Python with ASE
"$APP/Contents/MacOS/calango" --probe-python

# 3. The image must mount without an SLA prompt
hdiutil attach -readonly -nobrowse calango-<version>-macos-arm64.dmg
```

To rebuild the package from a clean state, clear the staging area first with `cmake -E rm -rf _CPack_Packages` — stale staging can mask deployment bugs. `build_dmg.sh` additionally runs `audit_macho_deps.py` over the finished image and fails the build on a build-machine load path (Section 2).

4 Linux: building the `.deb`

4.1 Prerequisites (Ubuntu/Debian)

```
sudo apt install build-essential cmake ninja-build \
  qt6-base-dev libqt6svg6-dev libqt6opengl6-dev \
  libgl1-mesa-dev python3-dev python3-venv \
  dpkg-dev file
```

`dpkg-dev` provides `dpkg-shlibdeps`, which CPack uses to compute the runtime `Depends:` line automatically (`CPACK_DEBIAN_PACKAGE_SHLIBDEPS ON`).

Build against the system Python, not Conda/Homebrew. The `.deb` must link the distribution's `libpython` (`/usr/lib/.../libpython3.x.so.1.0`, owned by the `libpython3.x` package) so that `dpkg-shlibdeps` can resolve it into a proper `Depends:` line. If CMake is pointed at a Conda or Homebrew interpreter, the binary links a `libpython` under `$HOME` that `dpkg-shlibdeps` cannot find — packaging fails with `dpkg-shlibdeps: error: cannot find library` — and the resulting package would not be installable on other machines anyway. `python3-dev` installs the system interpreter and its `libpython`; configure against `/usr/bin/python3` (Section 4.2). ASE is only a runtime `Recommends` (`apt install python3-ase`); it is not needed to build the package.

4.2 Commands

Configure a *fresh*, Linux-native build directory — never reuse a tree configured on another machine or OS (see the note below) — and point CMake at the system interpreter:

```
cmake -S . -B build-deb \
  -DCMAKE_BUILD_TYPE=Release \
  -DPython3_EXECUTABLE=/usr/bin/python3

cmake --build build-deb -j$(nproc)

cd build-deb && cpack
```

Do not run `cpack` in a build directory copied from another host. Cloud-synced checkouts (Dropbox) carry the macOS `build/` tree along with the source. Its `CMakeCache.txt` and `CPackConfig.cmake` bake in absolute macOS tool paths, so a Linux `cpack` there fails with `CPack Error: Cannot find description file name: [/opt/homebrew/share/cmake/Templates/...]`. A separate `build-deb` directory, configured by the Linux `cmake`, avoids this entirely.

The result is `calango_26.8.46_<arch>.deb`. Install it via `apt` rather than `dpkg -i`, so the dependencies are resolved:

```
sudo apt install ./calango_<version>_amd64.deb
```

4.3 Package contents and desktop integration

Path in package	Purpose
`/usr/bin/calango`	The application binary
`/usr/share/applications/calango.desktop`	Launcher entry (`Exec=calango %F`)
`/usr/share/mime/packages/calango-mime.xml`	MIME types for `.calproj` and 12 other readable formats
`/usr/share/pixmaps/calango.png`	Application icon (resolves the `.desktop` `Icon=calango` lookup)
`/usr/lib/calango/python/`	Embedded Python (optional, see Section 5)

The MIME registration means double-clicking a `.calproj` workspace, or one of the 12 other associated formats (the same Alternate-tier list as the macOS `Info.plist`, Section 3.6), in the file manager opens it in Calango; `MainWindow::loadFile` routes `.calproj` arguments to the project loader and everything else through the ASE structure reader, picking a format hint (`src/gui/FileOpenRouting.hpp`) when the extension alone is ambiguous. The declarations in `calango-mime.xml` are self-contained rather than assuming the optional `chemical-mime-data` package is installed, so the association resolves even on a system that never installed it. The `postinst`/`postrm` maintainer scripts refresh `update-mime-database`, `update-desktop-database` and `gtk-update-icon-cache` so the association is live immediately after install (modern dpkg triggers usually do this anyway; the scripts make it robust on minimal systems).

Unless an embedded Python is bundled, the package declares `Depends: python3 (>= 3.9)` and `Recommends: python3-ase`. Without ASE the GUI starts but structure I/O and job submission are disabled; users can also point Calango at any interpreter via `CALANGO_PYTHON`.

4.4 Verification

```
dpkg-deb --info  calango_<version>_amd64.deb   # control fields, Depends
dpkg-deb --contents calango_<version>_amd64.deb

# after installing:
calango --probe-python
xdg-mime query default application/x-calango-project
```

Status note: the DEB pipeline has been built and verified end-to-end on Ubuntu (`dpkg-deb -info` shows the auto-computed `Depends:`, including `libpython3.12t64` and the Qt 6 runtime libraries). Running it in CI on a clean Ubuntu 22.04/24.04 image remains worthwhile to catch host-specific assumptions. `build_deb.sh` additionally runs `audit_elf_deps.py` over the finished `.deb` and fails the build if any shipped ELF needs a library the control file does not declare (Section 2).

5 The embedded Python environment

Calango embeds CPython at runtime. The interpreter is resolved in this order (documented in `src/python_bridge/PythonEngine.hpp`):

1. `$CALANGO_PYTHON` — explicit interpreter path;
2. `$VIRTUAL_ENV/bin/python` — active virtualenv;

3. a Python bundled by the installers: `Contents/Resources/python/bin/python3` (macOS) or `<prefix>/lib/calango/python/bin/python3` (Linux);
4. the interpreter CMake was configured against (`CALANGO_DEFAULT_PYTHON`, meaningful only on the build machine).

To ship a fully self-contained installer, pass a *relocatable* Python distribution that has ASE installed:

```
cmake -S . -B $BUILD ... \
  -DCALANGO_EMBEDDED_PYTHON_DIR=/path/to/standalone-python
```

A distribution from the `python-build-standalone` project is the recommended source. A plain project `.venv` is *not* relocatable — its `bin/python` and `pyvenv.cfg` refer back to the build machine's base interpreter.

`build_dmg.sh` does all of this for you when the variable is unset. It downloads a matching `python-build-standalone` tree, pip-installs `CALANGO_EMBEDDED_PACKAGES` into it, and caches the result in the build directory. Skipping this is what separates a `.dmg` that only runs on the build machine from a shippable one — the app still launches either way, it just cannot import ASE — and the mount-and-probe check at the end of the script is what tells the two apart.

Independent of this option, the macOS pipeline always embeds the `Python.framework` that the binary itself links against (`libpython` via `Development.Embed`), so the packaged app launches on machines without Homebrew. For the interpreter to also find ASE there, either bundle an environment as above or let users install ASE and set `CALANGO_PYTHON`.

6 Known limitations and next steps

- **Code signing:** the `.app` is signed ad hoc. Distribution outside a lab requires a Developer ID certificate and notarization (`codesign -sign "Developer ID..."`, `notarytool submit`, `stapler`); the ad-hoc step in `embed_python_framework.sh` is the place to swap the identity in.
- **DEB validation:** run the Linux pipeline once in CI (Ubuntu 22.04/24.04) and consider `lintian` for policy checks.
- **Full ASE bundling:** wiring a `python-build-standalone` + ASE tree into both installers (and building against its `libpython`) would remove the last runtime dependency on a user-provided Python.