The weaver manual

The weaver contributors
main

Contents

What weaver is 3
- What it gives you 3
- How it treats arguments 3
- Validating a configuration 3

Get started 5
- Add the package 5
- Run it 6
- Where to go next 6
- API reference for every version 6

Verify the source bundle 7
- What the evidence covers 7
- Verify a release before using it 8
- Chain into your own inventory 9

Learn from the examples 10
- Which copy to use 10
- What the output looks like 10

Contribute 12
- Set up 12
- Two compilers 12
- Run the checks 13
- Where tests live 13
- Change an example 14
- Search for inputs nobody thought of 14
- Build this manual 14

Maintain and release 16
- What each workflow shows 16
- Update the tooling 17
- Make a release 18
- How the site is put together 19

The weaver manual

weaver parses the command line of a Roc program. One record builder declares the options, parameters and subcommands, and gives back typed data, help text, usage and actionable diagnostics.

How to use this manual

- **Deciding whether to use weaver**: Start with What weaver is.
- **Using it for the first time**: Follow Get started.
- **Looking for code to adapt**: Browse Learn from the examples.
- **Changing weaver itself**: Read Contribute.
- **Releasing or looking after the repository**: Read Maintain and release.
- **Upgrading**: Read the published release notes.

What weaver is

weaver is an ergonomic command-line argument parser for Roc, in the style of Rust's clap. Roc has no code generation at compile time, so weaver uses the record builder syntax instead: the same expression builds the configuration and the parser, so they cannot disagree, and the result is checked by the type checker.

weaver was created by Sam Mohr as smores56/weaver. This repository is a maintained fork that keeps it working with the current Roc compiler and publishes tested releases.

What it gives you

From one declaration of a command line, weaver produces:

- **Typed data.** Options and parameters parse to `U64`, `Dec`, `Str`, flags, counts, lists and your own types, and the record you get back has exactly the fields you declared.
- **Help and version pages.** `-h`/`--help` and `-V`/`--version` are added for you, and every subcommand has its own help page.
- **Diagnostics.** A missing option, a malformed value, an unknown argument or a misplaced parameter is reported with what went wrong, the usage line, and a tip for fixing it.
- **Subcommands**, nested to any depth, each with its own options and parameters.

Set `text_style` to `Color` for ANSI-styled help and diagnostics, or choose `Plain` when output is redirected or your application honours `NO_COLOR`.

How it treats arguments

weaver parses exactly the arguments passed to `Cli.parse_or_display_message`; it never drops the first one. `basic-cli` supplies `List(OsStr)` without the executable path, so pass them straight through with `OsStr.to_raw`. That keeps UTF-8, native Unix bytes and Windows UTF-16 code units intact as a `path.Path`. Only drop the first element when your platform documents that it is the executable path.

Use `Opt.arg` or `Param.arg` to keep a raw value as a `Path`. String parsers such as `Opt.str` and `Param.str` decode at the parser boundary and report a value that is not valid UTF-8 as a usage error.

Everything after a subcommand's name belongs to that subcommand. Give a parent's options and parameters before the subcommand's name; after it, they are reported as not recognized or not expected, against the subcommand's usage.

A separate token beginning with `-` is another option, not an option's value. Pass an option-like value as `--name=value`, or put `--` before it: both `--beta=-2.5` and `--beta -- -2.5` are unambiguous.

Help prose and diagnostics wrap at 80 UTF-8 bytes. weaver keeps explicit line breaks and never splits a token, so an unusually long token can run past that boundary rather than being corrupted.

Validating a configuration

`Cli.assert_valid` is convenient for a configuration written in the source, which is either valid or a bug. An application that builds its configuration at run time should handle the

result of `Cli.finish` and show the error with `ErrorFormatter.render_cli_validation_err(err, Color)`, or `Plain`.

Get started

This chapter takes you from nothing to a program that parses its command line. You need a current Roc nightly installed as `roc`.

Add the package

A Roc application names its packages in its header, each by the URL of a release. Save this as `hello.roc`:

The latest release is 0.10.1

```
app [main!] {
    pf: platform "https://github.com/roc-lang/basic-cli/releases/download/0.26.0/
GddgvsGjiNXrHFW6au5i5dUDbTs8FinQdrgE2Gqmw53t.tar.zst",
    weaver: "https://github.com/lukewilliamboswell/weaver/releases/download/
0.10.1/7CdYzjBAJmprmnK45oTpmH6BXt76ZASN2Ck5uZQEoWsK.tar.zst",
}

import pf.OsStr
import pf.Stdout
import weaver.Cli
import weaver.Opt

main! : List(OsStr) => Try({}, _)
main! = |args| {
    match Cli.parse_or_display_message(cli_parser, args, OsStr.to_raw) {
        Ok(alpha) => Stdout.line!("alpha is ${alpha.to_str()}")
        Err(InvalidUsage(message)) => {
            Stdout.line!(message)?
            Err(Exit(1))
        }
        Err(Help(message)) | Err(Version(message)) => Stdout.line!(message)
    }
}

cli_parser : Cli.CliParser(U64)
cli_parser = Cli.assert_valid(
    Cli.finish(
        Opt.u64({ short: "a", long: "alpha", help: "Set the alpha level.",
default: NoDefault }),
        { name: "hello", version: "v0.0.1", authors: [], description: "",
text_style: Color },
    ),
)
```

For release 0.10.1:

- API reference

- Examples, each depending on this release
- Release notes

The URL ends in a hash of the package's contents. Roc checks what it downloads against that hash, so the URL can only ever mean one thing.

Run it

```
roc hello.roc -- -a 3
roc hello.roc -- --help
roc hello.roc -- -a three
```

The first prints `alpha is 3`. The second prints the help page weaver wrote from the parser. The third reports that the value given to `-a/--alpha` is not a valid number, shows the usage line and exits with status 1.

Where to go next

- Learn from the examples has a complete program for each part of the package, from a single option to nested subcommands.
- The API reference linked below lists every function of every module.

API reference for every version

Each release keeps its own API reference, so the pages for the version you depend on never change under you.

- main, which is not released yet
- 0.10.1
- 0.10.0
- 0.9.1

Verify the source bundle

The release workflow generates supply-chain evidence for the exact Roc source bundle it publishes. Releases before 0.9.1 do not carry this evidence. Keep the bundle and its evidence together when admitting weaver into your own build.

What the evidence covers

Each source bundle has these sidecars:

- `<bundle>.spdx.json`: SPDX 3.
- `<bundle>.spdx-2.3.json`: SPDX 2.3 for compatibility with existing consumers.
- `<bundle>.cdx.json`: CycloneDX 1.7, from the same inventory.
- `<bundle>.sbom-inventory.json`: resolved components, relationships and coverage.
- `<bundle>.sbom-members.json`: every archive member's name, size and SHA-256.
- `<bundle>.sbom-manifest.json`: Blueprint's manifest binding the SPDX 3 and CycloneDX sidecars to subject bytes and inventory.
- `<bundle>.sbom-evidence.json`: the artifact/sidecar hashes, source commit and producing workflow run and attempt, in the shared `roc-automation-sbom-evidence-v1` format. It binds all three SBOM formats.

`weaver-provenance.sigstore.json` authenticates the tested release files, including the sidecars and evidence. `weaver-sbom.sigstore.json` binds the SPDX composition statement to the source bundle. Both are retained as release assets. The expected signer is this repository's `.github/workflows/release.yml` on `main`, at the release's source commit, using a GitHub-hosted runner.

The source inventory is independent of a CPU target. It covers the shipped first-party modules and their declared license, `UPL-1.0 AND MIT`: weaver is UPL-1.0, and keeps the MIT notice of the upstream project it was forked from. The release supplies both as `LICENSE`. Archive membership and bytes are checked against the committed package source.

The bundle depends on the packages named in the header of `package/main.roc`, which it does not contain. The compiler fetches each by its immutable URL, and checks it against the content hash in that URL, when an application is built. `Blueprint.roc` also pins each of those files as a `SourceFile`, by the SHA-256 its release publishes, and every pull request checks that `roc deps` lists exactly the pinned files.

Each dependency is a `dependsOn` component in the inventory, with the SHA-256 of the pinned file. Where the supplier publishes an SBOM beside its bundle, that document is imported, bound to the pinned file, so its graph and coverage statements become part of this one: the chain continues into whatever you build. The imported documents are recorded in `sbom/imports.json`, and the release refuses a supplier document whose bytes changed since it was reviewed. A supplier that publishes no SBOM has a reviewed `unavailable_sboms` entry in `sbom/package.json` saying so; its component is recorded from the reviewed bundle.

The compiler is a build dependency, identified by its version and actual executable SHA-256. Its native dependency closure is explicitly incomplete; this source inventory does not establish a complete compiler/toolchain inventory or the composition of an executable compiled downstream. The examples, API archive and PDF are separate

deliverables; their provenance is retained, but this source SBOM does not describe their contents.

Verify a release before using it

Select the bundle filename from the GitHub release. Its Roc content-hash name and its SHA-256 are different identities: use the SHA-256 in the SBOM to check the delivered archive bytes.

```
repo=lukewilliamboswell/weaver
version='<release version>'
bundle='<bundle filename from the release>'
commit=$(gh api "repos/$repo/commits/$version" --jq .sha)

gh release download "$version" --repo "$repo" \
  --pattern "$bundle" --pattern "$bundle.spdx-2.3.json" \
  --pattern "$bundle.sbom-evidence.json" \
  --pattern "$bundle.sbom-inventory.json" \
  --pattern "$bundle.sbom-members.json" \
  --pattern "$bundle.sbom-manifest.json" \
  --pattern "$bundle.spdx.json" --pattern "$bundle.cdx.json" \
  --pattern 'weaver-*.sigstore.json'

gh attestation verify "$bundle" --repo "$repo" \
  --signer-workflow "$repo/.github/workflows/release.yml" \
  --source-ref refs/heads/main --source-digest "$commit" \
  --signer-digest "$commit" --deny-self-hosted-runners \
  --bundle weaver-provenance.sigstore.json

gh attestation verify "$bundle.spdx-2.3.json" --repo "$repo" \
  --signer-workflow "$repo/.github/workflows/release.yml" \
  --source-ref refs/heads/main --source-digest "$commit" \
  --signer-digest "$commit" --deny-self-hosted-runners \
  --bundle weaver-provenance.sigstore.json

gh attestation verify "$bundle" --repo "$repo" \
  --signer-workflow "$repo/.github/workflows/release.yml" \
  --source-ref refs/heads/main --source-digest "$commit" \
  --signer-digest "$commit" --deny-self-hosted-runners \
  --predicate-type https://spdx.dev/Document/v2.3 \
  --bundle weaver-sbom.sigstore.json

expected=$(jq -r '.packages[] | select(.SPDXID == "SPDXRef-Subject")
| .checksums[] | select(.algorithm == "SHA256") | .checksumValue'
"$bundle.spdx-2.3.json")
```

Verify the source bundle

```
actual=$(shasum -a 256 "$bundle" | cut -d ' ' -f 1)
test "$actual" = "$expected"
```

With Blueprint's verified native SBOM utility on PATH, validate the downloaded manifest against the subject and resolved inventory, and check all three sidecars together:

```
blueprint-sbom check --manifest "$bundle.sbom-manifest.json" \
  --artifacts . --inventory "$bundle.sbom-inventory.json" "$bundle.spdx-2.3.json"
```

Also validate the standard schema and read the coverage statements. CI checks the SPDX 2.3 rendering against an unchanged, digest-pinned official schema using an independent pinned validator. Blueprint separately checks subject bytes, resolved inventory and graph agreement across all three formats. Keep your own admitted copy, expected signer policy and verification results; a short-lived Actions artifact is not the release evidence channel.

Chain into your own inventory

Preserve weaver's immutable source location, release version, bundle SHA-256, license and upstream document identity. weaver's own SBOM already imports the supplier documents of its dependencies, so importing it brings theirs too. Import the verified supplier graph, including its coverage gaps. Pin the consumed file by SHA-256 and bind the supplier SBOM to that exact artifact; checking only the SBOM's own checksum does not bind it to a package. Keep build-only dependencies distinct from included code and from dependencies fetched later.

Blueprint accepts a source supplier into source and native inventories, while preserving its `source` identity and checking exact supplier hashes and versions. Native architecture compatibility checks remain strict. The regression fixture in `tests/supply-chain/blueprint-source-import/` covers source-to-source, source-to-Linux-x86_64 and source-to-Linux-arm64 admission, plus integrity refusals. Keep source archives as build inputs and describe the resulting executable's included code/assets and native closure separately.

Learn from the examples

Each example is a complete program that shows one way to use weaver. Read them top to bottom, or copy one as a starting point.

Example	What it shows
`single-arg.roc`	The smallest parser: one required option, mapped into a tag
`basic.roc`	A record of an option, a flag, an optional parameter and a list of parameters
`default-values.roc`	Options and parameters with fixed and generated defaults
`subcommands.roc`	Nested subcommands, each with its own options, parameters and help
`deploy.roc`	A realistic deployment command line, and the diagnostics a user sees

Which copy to use

The examples exist in two places, and they are not the same.

- **The examples for a release**: Each release attaches an archive, `weaver-examples-<version>.zip`, linked from Add the package. Its examples depend on that release's package URL and never change. Use these to start a project.
- **The examples in the repository**: These depend on the package beside them, `weaver: "../package/main.roc"`, so they always match the unreleased code on `main`. They may use something no release has yet. Use these when you are changing weaver.

What the output looks like

```
$ roc examples/basic.roc -- file1.txt file2.txt -f -a 123
Successfully parsed! Here's what I got:

{ alpha: 123, file: Ok("file1.txt"), files: ["file2.txt"], force: True }

$ roc examples/basic.roc -- --help
basic v0.1.0
Some One <some.one@mail.com>

This is a basic example of what you can build with Weaver. You get safe
parsing, useful error messages, and help pages all for free!

Usage:
  basic -a NUM [-f] [-h/--help] [-V/--version] [<file>] [<files>...]

Arguments:
  <file>      The file to process.
  <files...>  The rest of the files.
```

```
Options:
  -a NUM         Set the alpha level.
  -f             Force the task to complete.
  -h, --help     Show this help page.
  -V, --version  Show the version.
```

The archive also holds `spec.json`, which records exactly what each example printed when the release was made, with cases covering successful parses, help and version pages, malformed values, missing arguments, unknown options, nested subcommands and delimiters.

Contribute

This chapter is for someone changing weaver: what to install, how to run the checks a pull request must pass, and where each kind of test lives.

Use issues for bugs and design discussion, and pull requests for changes. Say what behaviour changes and how you checked it. A bug fix or a new behaviour comes with a test.

Set up

You need three things:

- The `blueprint` command from roc-blueprint, at the release named by the platform URL at the top of `Blueprint.roc`. Download the binary for your system from that release and check it against the release's `blueprint.sha256`, make it executable and put it on `PATH`.
- A current Roc nightly on your `PATH` as `roc`.
- The host tools listed in `Expect` in `Blueprint.roc`: `chmod`, `gh`, `git`, `gzip`, `realpath`, `sh`, `shasum`, `tar`, `test`, `touch`, `unzip` and `zip`.

Nix is not needed. The file and archive tools supplied by Linux and macOS work; GNU coreutils is not required. Install GitHub CLI (`gh`) separately. Blueprint downloads the pinned tooling compiler and its Roc packages on first use. Run `blueprint update` after changing its declarations, and commit the lock file. A checkout already has the lock it needs.

Two compilers

Command	Chosen by	Used for
`roc`	You: the latest nightly	Checking, testing, bundling and running the package and its examples
`roc-stable`	The release named in `Blueprint.roc`	Running the scripts under `scripts/`

The package is always tested with the newest compiler, because that is what its users install. `roc-stable` exists so the repository's own tooling keeps working on the day a nightly breaks something. It never checks or builds the package, and it only exists inside `blueprint run` and `blueprint shell`.

Nothing pins a compiler for the package or the examples. Each script names `roc-stable`'s release in its `roc` header field, so that running one with a different compiler says why it misbehaves.

The released SBOM utility is a checksum-locked native executable from the Blueprint release named in `Blueprint.roc`. `blueprint run prepare-sbom` checks the source and prepares the reviewed composition. The release workflow uses roc-automation's shared `sbom-sidecars` action to generate and check the sidecars; `blueprint run sbom-tool` exposes the native tool for local inspection. Blueprint embeds its configuration compiler.

Run the checks

```
blueprint run check      # source, scripts, fuzz corpus and site checks
blueprint run release-package -- test-examples  # recorded example output
blueprint shell          # or enter the environment and run scripts directly
blueprint tasks          # list every task
```

`blueprint run check` runs `scripts/check_all.roc`, which:

1. Fails if a compiler pin appears outside `scripts/`, or a script's pin differs from `Blueprint.roc`.
2. Checks that the package depends on exactly the bundles `Blueprint.lock` pins.
3. Tests, checks and formats the scripts with `roc-stable`.
4. Formats, checks and tests the package with `roc`.
5. Builds each fuzz target and replays its committed corpus.
6. Builds the API documentation site.

The examples, the bundle and the archives are checked by the shared release workflow that `release.yml` calls, on every pull request: it runs every example against the working tree and compares its output with `tests/spec.json`, then bundles the package, builds the examples archive and runs the archived examples, unchanged, against that bundle.

A pull request can fail because a new nightly arrived, with no fault in the change. Compare with the latest run of the nightly workflow and with `main` before treating a failure as yours.

Where tests live

Kind	Where	How to add one
Unit tests	`expect` beside the code in `package/`	Write an `expect` in the module. `roc test package/main.roc` runs them all.
Example output	`tests/spec.json`	Add a case to the example's entry, then record its output; see Change an example.
Properties	`fuzz/<name>/main.roc`	A roc-fuzz target with a small `corpus/`; see Search for inputs nobody thought of.
Script logic	`expect` in the modules under `scripts/src/`	Keep effects out of the module, and add it to the list in `check_scripts!`.

Change an example

An example's exact output is recorded, so a change to what it prints must be recorded too:

```
blueprint run release-package -- test-examples --update
```

Blueprint provisions the locked, released `release-package` utility used by the shared release workflow. It tests with the `roc` on `PATH` and rewrites `tests/spec.json` from what the examples print now. Review the diff like any other change. An example that cannot run for a reason outside this repository is skipped in the specification with a `reason` and an `issue` link; a skip without both is rejected.

Examples are for users of the package. Keep each one short, about one thing, and commented for someone reading it with no other context. A test belongs in `package/` or `tests/`, not in `examples/`.

Search for inputs nobody thought of

Each target under `fuzz/` states a rule that must hold for every input, and roc-fuzz searches for an input that breaks it. The targets turn the fuzzer's bytes into a list of arguments, some of them not valid UTF-8, and parse them as a command line would. Every pull request only builds each target and replays its committed corpus. To search, for a target `<name>`:

```
roc build --fuzz fuzz/<name>/main.roc --output=<name>
./<name> run fuzz/<name>/corpus --time=60
```

When it finds a failure, fix the bug and keep the minimised input as an ordinary `expect` in the package.

Build this manual

Keep documentation focused on current reader tasks. Remove superseded guidance and status narratives; keep release changes in release records and investigations or unresolved work in issues. Follow the shared documentation guide.

The manual is the AsciiDoc under `docs/`, one chapter per file, rendered by the shared build-docs action. Every pull request builds it, and the site can be downloaded from the `Documentation` workflow run. Blueprint provisions the locked native renderer from the roc-automation release named in `Blueprint.roc`; no container or sibling checkout is needed:

```
blueprint run install-snippet
blueprint run docs -- --docs-version main
blueprint run serve
```

```
```

The README's screenshot, `assets/pretty-errors.svg`, is the deploy example's source and real output. Re-record it with `blueprint run capture` whenever that example or its diagnostics change.

The first command writes the parts of the manual that depend on what has been released into `docs/generated/`, which needs `gh` to be authenticated. The second renders HTML and chapter Markdown into `.docs-out/site`, and the PDF into `.docs-out/weaver.pdf`. The last adds the API reference for `main` and serves the site on a free port, or the one named by `SITE_PORT`; with `FETCH_RELEASE_DOCS=1` it also restores the API reference of every release.

Maintain and release

This chapter is for a maintainer: what each workflow shows, how the tooling is updated, and how a release is made. The repository follows the roc-automation manual, which gives the reasons behind each practice.

What each workflow shows

Workflow	Runs on	What it shows
`release.yml`	Manual dispatch; pull requests as a dry run	The working tree passes `blueprint run check` and its examples on the latest nightly, and the exact bundle, examples archive, documentation and SBOM sidecars proposed for publication work
`nightly.yml`	Daily	The latest release's examples archive still works on the latest nightly
`docs.yml`	Pull requests, pushes to `main`, published releases	The manual and site build; `main` and a final release deploy them

`release.yml` is a thin caller of roc-automation's shared release workflow at a full commit SHA. The shared workflow owns the jobs; this repository owns its package, examples, `tests/spec.json`, notes and the Blueprint tasks the workflow calls: `check`, `prepare-sbom`, `install-snippet` and `docs-path`. The required status checks are `Release / Build and test the source`, `Release / Test the release candidate`, `Validate source SBOMs` and `Build the manual and site`. The pull-request SBOM job independently validates the SPDX 2.3 schema and checks the supplier-import regression fixtures. The shared workflow checks sidecar bindings before publication.

The package is pure Roc, so CI uses Linux without Nix. The local setup action pins a reviewed roc-automation revision and selects the Blueprint release from the platform URL in `Blueprint.roc`. The shared installer verifies and caches the CLI by release, system and executable digest. Blueprint embeds its configuration compiler; the separate `roc-stable` script compiler remains pinned, and the package nightly floats.

SBOM generation uses roc-automation's shared `sbom-sidecars` action. The `prepare-sbom` task checks the bundle against the committed source and writes the reviewed composition, and the committed `sbom/imports.json` records the suppliers' SBOMs; the shared action owns generation and binding.

Downloaded utilities and compiler archives use Blueprint's verified artifact cache, keyed by workflow, job, Blueprint release and lock. Extracted trees are recreated and verified, while Roc package sources have a separate cache. A cold job downloads the locked utilities; a warm job restores and verifies them. Changing the lock invalidates the corresponding cache key. The floating nightly may still need a fresh download when its version changes.

The documentation jobs use `blueprint-tool` to resolve `docs-path` to the verified renderer in the store. CI reports executable digest and bytes, provisioning duration, native rendering time and peak RSS, output file sizes, and separate preparation and API generation timings. Rendering uses a precompiled Roc binary. The site's HTML, PDF, Markdown and `llms.txt` come from the same AsciiDoc sources.

`nightly.yml` calls roc-automation's pinned published-release workflow. It needs the canonical suite in a release's examples archive; the workflow has nothing to test until a release made by `release.yml` is published. Successful compiler diagnostics are retained; incomplete or failed setup cannot be reported as a compatibility pass. Exact release and nightly overrides provide read-only diagnostic reruns.

Update the tooling

Nothing updates the tooling automatically. Each of these is a pull request.

- **Refresh the lock**: Run `blueprint update` and commit `Blueprint.lock`.
- **Change the tooling compiler**: Change the tag in `Fetch("roc-stable", Recipes.roc_nightly(...))` in `Blueprint.roc` and the `roc` field of every script to the same nightly. The check fails until they agree. Run `blueprint update roc-stable` and fix any script the new compiler breaks.
- **Upgrade Blueprint**: Change the platform URL at the top of `Blueprint.roc`, and the pinned shared setup action in `.github/actions/setup-blueprint/action.yml`. Use a shared revision that records that release's checksums, or supply reviewed `sha256-*` overrides in the local action when adopting a release ahead of its registry. Remove overrides when the shared registry covers the release.
- **Upgrade roc-automation**: Move every `lukewilliamboswell/roc-automation` reference in `.github/` to the same reviewed full SHA, and the `roc-docs` and `release-package` releases in `Blueprint.roc` to the matching `tools-*` tag. Run `blueprint update roc-docs release-package` and compare each declared system's SHA-256 with the release's `sha256sums.txt`. The downstream dashboard reports a repository as current only when every reference matches roc-automation's `main`. Run `blueprint update` and review `blueprint lock show --json` (format 2) as well as the lock diff.
- **Update a package dependency**: Change its URL in the header of `package/main.roc` and in its `SourceFile` in `Blueprint.roc`, with the SHA-256 the supplier's release publishes for the file, and its `version` in `sbom/package.json`. Run `blueprint update`. Then delete `sbom/imports.json` and regenerate it with roc-automation's `fetch-dependency-sboms` action at the revision `release.yml` pins, as that workflow's composition step describes, and review the supplier documents it records. If the supplier publishes no SBOM, give the component an `unavailable_sboms` entry with the

reason instead. `blueprint run check` fails until the header and the lock agree, and the release dry run fails until the imports match what the suppliers publish.
- **Update a script dependency**: Change its URL in every script header and under `RocPackages` in `Blueprint.roc`. Task-only Roc bundles are content-addressed and are not recorded in the lock.

Make a release

1. Choose the version. `roc bump` compares the package with the latest release and says how far the version must move:

```
roc bump --old <latest release bundle URL> package/main.roc
```

2. Write the notes in `docs/releases/<version>.adoc`, register them in the release-input manifest `docs/releases.adoc`, and merge that pull request. Write them for someone deciding whether and how to upgrade. Use `{package-url}` wherever the release's bundle URL belongs.
3. Dispatch the **Release** workflow from `main` with that version.

The workflow then:

1. Runs `blueprint run check`, then rejects a version that already exists, one with no notes, and one that `roc bump --expect` says understates the change.
2. Runs the examples against the working tree and compares them with `tests/spec.json`.
3. Builds the bundle, the examples archive, the documentation archive, the SBOM sidecars and this manual as a PDF, and renders the notes with the real bundle URL and the `roc` it used. That record replaces a header pin as the statement of what the release was built and tested with.
4. Tests the examples archive, unchanged, against the bundle, in a read-only job that checks out this repository to provision the locked utility and runs the examples from the archive.
5. Publishes. A separate job, which checks out nothing from this repository and runs none of its code, attests the tested files, creates a draft release at the tested commit, uploads every asset, downloads each back and compares it with the tested file, and only then publishes.
6. `docs.yml` then rebuilds and deploys the site, unless the version is a release candidate.

A release holds:

- `<hash>.tar.zst`: The package bundle. Its name is its content hash.
- `weaver-examples-<version>.zip`: The examples, depending on this bundle, with their recorded output.
- `weaver-docs-<version>.zip`: The API reference for this version, which the site restores.
- `weaver-<version>.pdf`: This manual.
- **Source-bundle SBOMs**: The bundle's `.spdx.json`, `.spdx-2.3.json` and `.cdx.json` sidecars describe it in SPDX 3, SPDX 2.3 and CycloneDX. Its `.sbom-manifest.json` binds those documents to the bundle.
- `LICENSE`: The UPL-1.0 terms, with the MIT notice of the upstream project weaver was forked from.

Build provenance for every asset is an attestation GitHub stores for the repository, not a release asset: `gh attestation verify <file> --repo lukewilliamboswell/weaver` checks a downloaded file against it.

No follow-up pull request is needed, because nothing on `main` names its own release. A published release is never changed: if one is wrong, publish a new version.

On a pull request the workflow builds and tests the same things under the version `0.0.1-dryrun`, renders the newest notes, and publishes nothing.

A release candidate, such as `<version>-rc1`, goes through the same workflow. It is marked as a prerelease, is never the latest release, and needs its own notes file.

How the site is put together

The site is this manual, rendered by the shared build-docs action, with the API reference added beside it by `blueprint run site`:

- `main/` holds fresh API documentation for the working tree.
- Each release's API documentation sits under its version, such as `<version>/`, restored from the archive that release attached. Its address keeps working without its HTML being stored in Git, and generated documentation is never committed.

The manual leads with the latest release: its first program, its API reference and its examples archive. Those parts are written by `blueprint run install-snippet` before the manual is rendered, because a bundle URL holds a hash nobody can write by hand.

`docs.yml` builds the site on every pull request and deploys it from `main` and after a final release is published, since a release changes which one is the latest.