

The roc-automation manual

THE ROC-AUTOMATION CONTRIBUTORS

Version 3: An opinionated approach for managing Roc packages and platforms

Table of Contents

1. How a Roc repository is maintained	2
1.1. The model in one page	2
1.2. Three compilers, three jobs	3
1.3. What each workflow answers	4
1.4. A pure Roc package	5
1.5. What is shared and what a repository owns	5
1.6. What the model does not do	6
2. Pin the tooling environment	7
2.1. Declare the environment	7
2.2. Get <code>blueprint</code>	8
2.3. Work locally	9
2.4. Use the environment in CI	10
3. Lay out the repository	11
3.1. The layout	11
3.2. Where a compiler pin belongs	11
3.3. Roc applications in the repository	12
3.4. Write automation as Roc scripts	12
3.5. What is committed	14
3.6. What goes in a bundle	14
4. Examples and the examples archive	15
4.1. What an example is for	15
4.2. Why examples live in two places	16
4.3. Repository examples test the working tree	16
4.4. Record expected output	16
4.5. The examples archive	16
4.6. Build and test the exact archive	17
4.7. Dry runs	18
4.8. An archive is never reformatted	18
5. Test in pull-request CI	19
5.1. What runs on every pull request	19
5.2. Heavy runs stay out of pull-request CI	20
5.3. Build with <code>--opt=dev</code>	20
5.4. Unit tests	20
5.5. Integration tests	20
5.6. Skip a broken case, visibly	21

5.7. Report progress and results	22
5.8. Measure allocations	23
5.9. Property-based tests with roc-fuzz	23
5.10. Formatting	25
6. The three workflows	27
6.1. Continuous integration	28
6.2. Release	28
6.3. Nightly compatibility	35
7. Optional workflows	37
7.1. Documentation site	37
7.2. Fuzzing	38
7.3. Benchmarks	39
7.4. Linker inputs	39
8. Maintain a platform	40
8.1. Host bindings	40
8.2. Cross-compile once, run on every target	41
8.3. Several platforms in one repository	42
8.4. Template repositories	43
8.5. Native build inputs	43
8.6. Separate the artifacts by what can invalidate them	46
8.7. Make content identity the ordinary consumer contract	48
8.8. Publish provenance without confusing it with integrity	49
8.9. Choose the correct maintainer path	50
8.10. Keep routine pull requests ignorant of production	51
8.11. Preserve reviewed history and authority	52
8.12. Keep rebuild selection narrow and complete	53
8.13. Keep final release preparation small	53
9. Publish linker inputs from a pull request	55
9.1. Producer contract	55
9.2. Trusted publisher contract	56
9.3. Recover interrupted publication	58
9.4. Review and merge the adoption	58
9.5. Routine consumption	59
10. Manual maintenance tasks	60
10.1. Update the lock	60
10.2. Change the tooling compiler	61
10.3. Upgrade Blueprint	61
10.4. Update script dependencies	62
10.5. Regenerate host bindings	62
11. Repository settings	63
11.1. Actions	63
11.2. Branch rules	63

11.3. Releases	64
11.4. Pages	64
11.5. Dependencies	64
11.6. Reporting	64
11.7. Confirm the settings	64
12. Adopt the model in a repository	66
12.1. Steps	66
12.2. Acceptance	67
12.3. Evidence for each claim	67
13. Shared actions	69
13.1. actions/setup-blueprint	69
13.2. Install the nightly compiler	70
13.3. Actions that run a Roc script	71
13.4. actions/nightly-report	71
13.5. actions/build-docs	73
13.6. actions/publish-release	74
13.7. actions/publish-build-inputs	76
14. Verify released artifacts	77
14.1. Separate admission from routine use	77
14.2. Routine CI policy	78
14.3. Appropriate attestation use	78
14.4. Review checklist	78
15. Automation trust model	80
15.1. The three workflows	80
15.2. Compilers	82
15.3. The blueprint binary	82
15.4. Documentation deployment	83
15.5. Linker-input release publisher	83
15.6. Released artifact verification	83
15.7. The pin-bump controller and the release guard	84
16. Write project documentation	88
16.1. Give each page one purpose	88
16.2. Define the page before drafting	89
16.3. Start from the reader's knowledge	89
16.4. Show structure and behavior	90
16.5. Write for the shared renderer	90
16.6. One manual, several audiences	91
16.7. Keep the README short	92
16.8. Cover the public project experience	92
16.9. Keep claims testable	93
16.10. Review each change	94
17. Project-practice evidence	95

17.1. What the model supplies and what a project still owes	95
17.2. Keep an evidence inventory	96
17.3. Review and bots	97
18. Contribute to roc-automation	98
18.1. Propose a change	98
18.2. Change the manual	98
18.3. Change a shared action	98
18.4. Change a script	99
18.5. Release the tools	100
18.6. Release a shared action	101
19. Release notes	102
19.1. roc-automation 3	102
19.2. roc-automation 2	104
19.3. roc-automation 1	104

This manual describes an opinionated approach for managing a Roc package or platform repository: how its tooling is pinned, what its pull-request, release and nightly workflows prove, and what every release contains. roc-automation holds the shared actions that implement the parts every repository needs identically.

How to use this manual

New to the model

Start with [How a Roc repository is maintained](#) for the compilers involved, the three workflows, and what the model leaves out.

Pure Roc package

A [pure Roc package](#) says which parts of the model a package with no host code uses, and which it skips.

Package maintainer

Read [Pin the tooling environment](#), [Lay out the repository](#), [Examples and the examples archive](#), [Test in pull-request CI](#) and [The three workflows](#) in order.

Platform maintainer

Read the same chapters, then [Maintain a platform](#) for host bindings, target matrices and native build inputs, and [Publish linker inputs from a pull request](#).

Routine upkeep

[Manual maintenance tasks](#) lists the changes that only a maintainer makes, and how to make each one.

Adopting the model

Follow [Adopt the model in a repository](#) and its acceptance list.

Reviewing trust boundaries

Read [Automation trust model](#) and [Verify released artifacts](#).

Writing a project manual

[Write project documentation](#) covers the conventions this manual itself follows.

Changing roc-automation

[Contribute to roc-automation](#) covers the manual and the shared actions.

Chapter 1. How a Roc repository is maintained

This manual is end-to-end guidance for managing Roc packages, platforms and the infrastructure that supports them: the tooling environment, the repository layout, continuous integration, releases, documentation, and the routine upkeep in between.

Keeping a Roc project in good shape means juggling several things that move at different speeds. The compiler is still changing, and breaking changes are routine. A bug can sit in the compiler, in a platform, in a dependency, or in the project itself. Releases rarely line up: a package may need a platform release that has not happened, or a fix may be merged long before users can download it. Each repository that solves these problems alone solves them slightly differently, and each difference is something a maintainer has to remember.

A common baseline is the answer this manual offers. It records the practices that have worked across these repositories and the reasons behind them, so that a lesson learnt in one place is applied everywhere. The practices are opinionated, and they are not finished: when experience shows a better way, the baseline changes here first and the repositories follow.

One decision shapes most of the rest. The package or platform always meets the newest compiler, because that is what its users will install. The repository's own tooling stays on a compiler the maintainer chose and moves only when the maintainer moves it, so that scripts, checks and release steps keep working on the day a new compiler breaks something.

1.1. The model in one page

Concern	Practice
Package compiler	The latest published Roc nightly. The package, platform and examples carry no pin.
Tooling compiler	<code>roc-stable</code> : one release named in <code>Blueprint.roc</code> and repeated in each script's <code>roc</code> header field, changed by a reviewed pull request.
Tools and toolchains	Declared in <code>Blueprint.roc</code> , pinned by <code>Blueprint.lock</code> , identical for contributors and CI.

Concern	Practice
Automation	Roc scripts under <code>scripts/</code> , run by <code>roc-stable</code> .
Repository examples	Depend on the working tree by relative path.
Pull requests	Scripts pass on <code>roc-stable</code> ; the working tree passes on the latest nightly; a platform's committed host bindings match what that nightly generates.
A release	One immutable release holding the bundle, an examples archive pinned to that bundle, a documentation archive, and the manual where there is one. Every asset is tested as the exact bytes that are published.
Nightly	Runs the latest release's own examples archive on the latest nightly. A failure means the package needs a patch release.
Upkeep	Blueprint, the lock, the tooling compiler, the scripts and a platform's host bindings are each changed by hand, in a pull request.

1.2. Three compilers, three jobs

A repository involves three Roc compilers. Each has one job, and they are never interchangeable.

Compiler	Selected by	Used for
Blueprint's own; it has no command name	The <code>blueprint</code> binary	Evaluating <code>Blueprint.roc</code>
<code>roc-stable</code>	The release tag named in <code>Blueprint.roc</code>	Running the repository's automation scripts
<code>roc</code> , the latest nightly	Nothing: it is whatever <code>roc-lang/nightlies</code> published most recently	Checking, testing, bundling and running the package, the platform and their examples

A maintainer chooses the second and observes the third. The first is hidden: `blueprint` fetches it when needed, as described in [Get `blueprint`](#).

Blueprint has a compiler of its own because Blueprint is itself written in Roc. It has to evaluate `Blueprint.roc` before any environment exists, so it cannot take its compiler from the environment that file describes: that would be a bootstrap problem. Each `blueprint` release therefore fixes the one compiler it was built with.

Three compilers is more than anyone wants, and the arrangement is accepted as temporary. It

follows from Roc having only nightly builds. Once Roc 0.1 is released, Blueprint and most repository tooling are expected to settle on that release, which leaves two compilers to think about: the released one that tooling runs on, and the nightly that packages are tested against.

`roc-stable` exists so that repository tooling keeps working when a nightly breaks something. The package deliberately floats, because the question a maintainer needs answered is whether the package works with the compiler users will install today. "Stable" means only that the repository does not move the tag without review. There is no shared registry of stable versions, and repositories need not agree.

"Latest nightly" means the most recent release published by `roc-lang/nightlies`. This model treats that as the last good build and does not check upstream CI separately.

CI installs that compiler as `roc` with the official `roc-lang/setup-roc` action, which is also how it is named on a contributor's machine. The Blueprint environment provides `roc-stable` and no `roc`, so the two names never collide.

1.3. What each workflow answers

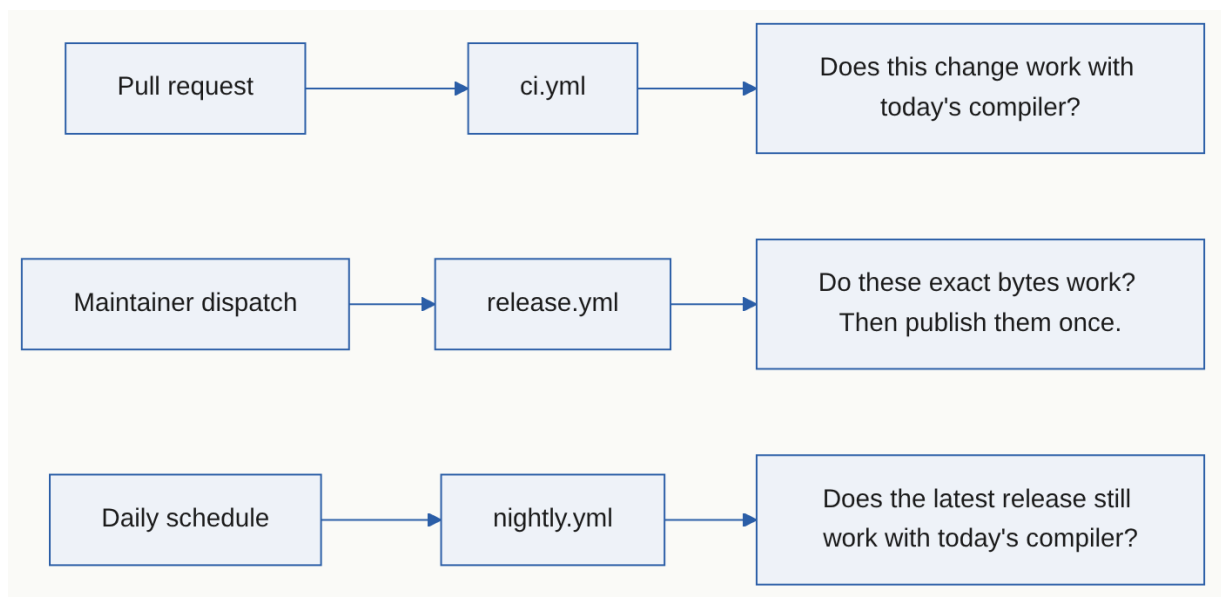


Figure 1. The three workflows and the question each one answers

`ci.yml` tests the working tree. `release.yml` tests and publishes artifacts. `nightly.yml` tests what users already have. Each can fail without the others being wrong: a new nightly can break `ci.yml` on an untouched pull request, and `nightly.yml` can fail while the default branch is green because the fix has not been released. [The three workflows](#) describes each in full.

1.4. A pure Roc package

Most repositories are the simplest kind, and the model is at its smallest for them. A repository is a pure Roc package when all of these hold:

- it publishes a package and has no host or platform code;
- it fetches no native build inputs;
- none of its code behaves differently from one target to another.

Such a package is expected to work wherever Roc does. A difference between targets is a compiler defect to report upstream, not something the package’s own CI is there to find. It therefore tests on one runner.

Concern	A pure Roc package
Runners	One. <code>ci.yml</code> , the test job of <code>release.yml</code> and <code>nightly.yml</code> each run on <code>ubuntu-latest</code> only.
Required status checks	The job names themselves. With no matrix there is no aggregate job.
Examples	Run on a published platform, such as <code>basic-cli</code> , and are tested on the one runner.
Chapters that do not apply	Maintain a platform and Publish linker inputs from a pull request : host bindings, native build inputs, rebuild selection and <code>actions/publish-build-inputs</code> .

Everything else applies unchanged: the two compilers, the Blueprint environment, the Roc scripts, the three workflows, and what a release contains.

The cost is stated here so that it is a decision and not an accident: a compiler defect that appears on one target only is not detected by this package’s CI.

A package leaves this profile as soon as it gains behaviour that can differ by target, for example when it starts a process, interprets file paths, or depends on the width of a pointer. It then adds a runner for each target it claims, and takes on the matrix and the aggregate job described in [Continuous integration](#).

1.5. What is shared and what a repository owns

Anything two repositories need identically belongs in `roc-automation`, referenced by full commit SHA. A repository keeps only what knows its own package.

Lives in roc-automation	Lives in the repository
<code>actions/setup-blueprint</code> : install Nix and the verified <code>blueprint</code> binary	<code>Blueprint.roc</code> and <code>Blueprint.lock</code>
<code>actions/nightly-report</code> : keep the nightly tracking issue	The test job of <code>nightly.yml</code> , its matrix and its schedule
<code>actions/build-docs</code> : render an AsciiDoc manual as a site and PDF	<code>docs/*.adoc</code>
<code>actions/publish-release</code> : stage a draft release, verify every asset, then publish	What a release contains, and the jobs that build and test it
<code>actions/publish-build-inputs</code> : publish linker inputs from a pull request	The producer workflow and its build recipe
The <code>check-settings</code> task: compare a repository's GitHub settings with this manual	The settings themselves, applied by a maintainer
This manual	Thin caller workflows, <code>scripts/</code> , tests and expected output

[Shared actions](#) is the reference for each action.

1.6. What the model does not do

- It does not make package builds reproducible. The package compiler floats by design; only the tooling environment is pinned.
- It does not verify upstream compiler quality beyond using a published nightly.
- It does not support holding a package on an older compiler line. There are no compiler-compatibility branches: a release works with the compilers named in its notes, and a patch release follows when a newer compiler breaks it.
- It does not describe application repositories that publish no package. They can use the environment and script conventions, and have no release or nightly workflow.

Chapter 2. Pin the tooling environment

`Blueprint.roc` declares every tool a contributor and CI need. `Blueprint.lock` pins them. Both are committed. The `blueprint` command from `roc-blueprint` realises that environment, the same way on a contributor's machine and on every CI runner.

The environment holds the tooling compiler, the tools the scripts call, and the native toolchains a platform host or a test host needs. It does not hold the package compiler, which is never pinned.

2.1. Declare the environment

```
app [config] { pf: platform "<roc-blueprint release URL>" }

config = [
  Name("roc-example"),
  Overlay("roc", "github:roc-lang/roc-overlay"),
  Environment(
    "dev",
    [
      Overlays(["roc"]),
      Command("roc-stable", "rocpkgs.nightly-2026-10-09-258ab27"),
      RocPackages([
        "https://github.com/roc-lang/basic-cli/releases/download/<version>/<hash>.tar.zst",
        "https://github.com/roc-lang/http/releases/download/<version>/<hash>.tar.zst",
      ]),
      Tools(["coreutils", "gh", "git", "unzip", "zip"]),
    ],
  ),
  Shell("default", [Use("dev")]),
  Task("check", [Use("dev"), Run(["scripts/check_all.roc"])]),
  Task("bundle", [Use("dev"), Run(["scripts/build_bundle.roc"])]),
  Task("test-bundle", [Use("dev"), Run(["scripts/test_bundle.roc"])]),
  Task("package-examples", [Use("dev"), Run(["scripts/package_examples.roc"])]),
  Task("package-docs", [Use("dev"), Run(["scripts/package_docs.roc"])]),
  Task("test-published", [Use("dev"), Run(["scripts/test_published.roc"])]),
]
```

Each setting has one job:

`Command("roc-stable", ...)`

Names the exact nightly release used for tooling. The compiler comes from `roc-overlay`, which packages recorded nightlies under their release tags. `Command` exposes that one release as `roc-`

stable without adding the tool's own executables, so the environment provides no bare roc.

RocPackages

Locks the Roc bundles the scripts depend on, so a script resolves them without downloading. List every bundle, including dependencies of dependencies; `roc-stable deps scripts/check_all.roc` prints them. Blueprint does not read package headers and cannot add a missing one.

Tools

Every program a script starts, and every toolchain a build needs: for example `pandoc`, `zig`, a Rust toolchain or `go`. A tool that a script calls and the environment does not declare is a defect, because CI and contributors then run different versions.

Task

A named entry point. CI calls tasks, never script paths, so the workflow and a contributor run the same command.

The two files divide the work. `Blueprint.roc` states which release is the tooling compiler and which tools are needed, where a reviewer can read it. `Blueprint.lock` pins the overlay revision that supplies that release's download URL and hash, the package source, and each locked bundle.

Nothing that `blueprint` generates is committed. Its generated Nix files stay in `.blueprint/`, which the repository's `.gitignore` lists.

2.2. Get blueprint

Blueprint needs Nix on the machine, so install [Nix](#) first. There are then two ways to get the `blueprint` command. Whichever you use, take the same roc-blueprint release as the platform URL in the repository's `Blueprint.roc` header.

2.2.1. With Nix

Nix can run or install `blueprint` straight from the release tag, with nothing to download or verify by hand:

```
# run it once, without installing
nix run github:lukewilliamboswell/roc-blueprint/<version> -- run check

# or install it into your profile
nix profile install github:lukewilliamboswell/roc-blueprint/<version>
```

This is the simplest route for a contributor. Nix builds the command from the tagged source the first time, which takes longer than a download, and the flake provides it for x86_64 Linux, arm64

Linux and Apple Silicon macOS.

Blueprint does not run on Intel macOS. nixpkgs no longer supports that system, so there is no environment for Blueprint to realise there and no `blueprint-x86_64-darwin` binary is released.

2.2.2. As a release binary

Each roc-blueprint release also attaches prebuilt `blueprint-<system>` binaries and a `blueprint.sha256` file. Download the one for your system, check it against the published SHA-256, make it executable and put it on `PATH`.

CI always uses the release binary, through `actions/setup-blueprint`, which checks it against a reviewed SHA-256 on every run. Nobody builds the CLI from source in CI.

2.2.3. What the command needs

The binary needs Nix on the machine. It needs neither Python nor Roc installed. To evaluate `Blueprint.roc` it uses a `roc` on `PATH` when that is exactly the nightly the binary was built with; otherwise it fetches that nightly itself from a roc-overlay revision pinned inside the binary. The `ROC` environment variable overrides this choice and must name that same nightly.

This compiler is separate from `roc-stable` for a reason: Blueprint is written in Roc and must run before the environment it describes exists. See [Three compilers, three jobs](#).

A release's binary must be used with that release's platform URL in the `Blueprint.roc` header. The two move together; see [Upgrade Blueprint](#).

2.3. Work locally

A contributor needs Nix, the `blueprint` command, and a current nightly installed as `roc`, the way Roc is normally installed. Nothing else has to be set up.

```
blueprint run check    # run a declared task in the pinned environment
blueprint shell        # or enter the environment and run scripts directly
blueprint tasks        # list the declared tasks
```

`blueprint` is the outer command. Run it from an ordinary shell: there is no `nix develop` step first and no nested shell.

Inside the environment `roc-stable` is the pinned compiler, and the contributor's own `roc` remains visible for package work.

2.4. Use the environment in CI

Every job that runs repository scripts sets up Blueprint first, then installs the latest nightly, then calls a task:

1. Set up Nix and the verified `blueprint` binary with `actions/setup-blueprint`.
2. Install the latest nightly as `roc` with the official `roc-lang/setup-roc` action; see [Install the nightly compiler](#).
3. Run `blueprint run <task>`.

A matrix names only runners Blueprint supports: x86_64 and arm64 Linux, and Apple Silicon macOS. It cannot include an Intel macOS runner. A platform can still cross-compile a host for that target in [the builder job](#), but no job runs the repository's scripts there.

The job needs no cache account and no token for this. A Nix cache is an optimisation a repository may add; a cache that requires `id-token: write` widens a test job's token and must be an explicit choice.

A change to `Blueprint.lock` changes the tools every build uses. Where a repository fingerprints native build inputs, `Blueprint.lock` is one of the fingerprinted files; see [Keep rebuild selection narrow and complete](#).

Chapter 3. Lay out the repository

A consistent layout lets the same checks, scripts and workflows apply to every repository, and lets a maintainer who knows one repository find their way in another.

3.1. The layout

Blueprint.roc	development environment and named tasks
Blueprint.lock	pinned inputs
package/main.roc	a package; or platform/main.roc for a platform
examples/	applications that depend on the working tree
scripts/*.roc	executable automation entry points
scripts/src/*.roc	modules shared by those scripts
tests/	package-specific cases and expected output
docs/*.adoc	the manual, where the repository has one

A repository that publishes several packages or platforms gives each its own root directory, such as `package/` and `tzdb/package/`, or `platform-web/` and `platform-gui/`. Each root is bundled, versioned and released together with the others in one release; see [Several bundles in one release](#).

3.2. Where a compiler pin belongs

The `roc: "..."` header field appears in exactly one place: the scripts under `scripts/`, where it names the same release as `roc-stable`. Nothing else is pinned. The package, a platform, the examples and every other Roc application carry no `roc` field, and the repository has no `.roc-version` file and no `.github/roc-nightly.json`.

The rule for the package applies to platforms as much as packages. `roc check` exits with an error on a warning, and a dependency whose header pins a different nightly produces a version-mismatch warning. A pinned platform therefore fails every application that floats, and a package cannot float until every platform its examples use has dropped the field.

The check script enforces both halves. It fails when a `roc` field or `.roc-version` appears outside `scripts/`, and it fails when a script's pin differs from the tooling compiler; see [Keep the script pins in step](#).

3.3. Roc applications in the repository

Every Roc application in the repository depends on the working tree by relative path. Pull-request CI checks them with the latest nightly compiler. That covers `examples/` and also anything outside it: fuzz targets, benchmarks, documentation examples, test cases, and a site generator.

In the repository: the package is a relative path

```
app [main!] {
  cli: platform "https://github.com/roc-lang/basic-cli/releases/download/<version>/<hash>.tar.zst",
  ansi: "../package/main.roc",
}
```

The package is a relative path. Every other dependency, including the platform, is an immutable release URL with its content hash.

These are not what the nightly workflow tests. They change with every pull request and may use APIs that no release has. The nightly workflow runs the examples archive attached to the latest release. In the archive the same example names that release's exact bundle URL, and nothing else differs:

In a release's examples archive: the package is that release's bundle

```
app [main!] {
  cli: platform "https://github.com/roc-lang/basic-cli/releases/download/<version>/<hash>.tar.zst",
  ansi: "https://github.com/<owner>/roc-ansi/releases/download/<tag>/<hash>.tar.zst",
}
```

The release workflow writes the second form when it builds the archive; the repository only ever contains the first. See [The examples archive](#).

The single exception is `scripts/`. A script never depends on the repository's own package by relative path, because a script must keep working while the package is broken.

An example that needs more than one file is a directory: `examples/report/` holds `main.roc`, its sibling modules and any data files it reads. A local helper package that several examples share sits beside them and is referenced by relative path.

3.4. Write automation as Roc scripts

Each entry point under `scripts/` is an executable Roc application:

```
#!/usr/bin/env roc-stable
app [main!] {
  cli: platform "https://github.com/roc-lang/basic-cli/releases/download/<version>/<hash>.tar.zst",
  roc: "nightly-2026-10-09-258ab27",
}
```

- Start with the `roc-stable` shebang and commit the file as executable.
- Set the `roc` header field to the release that `Blueprint.roc` names as `roc-stable`.
- Reference every dependency by immutable release URL with its content hash, and list the same URLs under `RocPackages` in `Blueprint.roc`.
- Run package work with `roc`, which is the latest nightly in CI and on a contributor's machine. Never use `roc-stable` to check, test or bundle the package.
- Put pure logic in `scripts/src/` modules with `expect` tests, and run those tests from the check script.
- Pass arguments after `--`, for example `scripts/test_goldens.roc -- --update`.
- Declare every entry point as a `Task` in `Blueprint.roc`.

Automation is not written in Python, bash or PowerShell. A script that exists only to wrap one command is deleted in favour of a `Task` that runs the command.

3.4.1. Keep the script pins in step

A script is written for one compiler and one set of dependencies. Someone who runs it with a newer `roc`, instead of through `blueprint`, may find it broken for reasons that have nothing to do with their change. The `roc` field makes the compiler say so: a compiler other than the one the field names refuses the file and names the release it targets, which points at the cause before anyone debugs the script.

The field is useful only while it is true, so one cheap check keeps it true. The check script reads the tag from the `Command("roc-stable", ...)` setting in `Blueprint.roc` and requires every file under `scripts/` that has an `app` header to carry exactly that tag in its `roc` field. It reads text and runs no compiler, so it costs nothing in pull-request CI.

Two things follow from the compiler enforcing the field:

- Format the scripts with `roc-stable fmt`. `roc fmt` on another compiler rewrites the field to that compiler's own release, and the check then fails.
- The field belongs only on a file that `roc-stable` runs. A program under `scripts/` that the latest nightly builds, or that a compiler the repository itself produces builds, cannot carry it: that compiler would refuse the file. Keep such a program outside `scripts/` where that is practical. Where it stays, the check script lists it by path as an exception and requires that it has no `roc` field at all.

`Blueprint.roc` remains the authority. The header fields are copies that the check holds to it, and [Change the tooling compiler](#) changes them together.

3.4.2. The check script

One script, conventionally `scripts/check_all.roc` behind the `check` task, runs everything a pull request must pass: the static checks in this chapter, `roc fmt --check` and `roc-stable fmt --check`, the script tests, `roc test` on the package root, every example and integration case, a smoke run of each benchmark and fuzz target, and a dry-run build of what a release would publish. [What runs on every pull request](#) describes each.

3.5. What is committed

Committed	Not committed
Source, <code>Blueprint.roc</code> , <code>Blueprint.lock</code>	<code>.blueprint/</code> and anything else <code>blueprint</code> generates
Expected output and golden files under <code>tests/</code>	Build output, bundles, archives
Generated data the package ships, with the recipe and inputs that produced it	Rendered API documentation and the built site
A platform's generated host bindings	Native binaries, which are released independently and consumed through a content lock; see No binary blobs in the repository

Generated data that is committed has a script that regenerates it and a check that fails when the committed copy is stale. Rendered documentation is published as a release asset and a Pages deployment, never committed.

3.6. What goes in a bundle

A bundle holds what a user of the package needs and nothing else: the package's Roc modules and any data files they import. Fuzz targets, benchmarks and test modules stay outside the package root, so that `roc bundle` on the root produces the release artifact with no exclusion list.

Chapter 4. Examples and the examples archive

Examples are realistic demonstrations of the package for the people who will use it. They are where most new visitors land, and for many they are the first Roc code written against the package that they read. This chapter covers what an example is for, and how the same example is kept correct in two places: the repository, and each release.

4.1. What an example is for

The audience of an example is a user of the package who is probably not familiar with it. It is not a contributor.

An example is not a test, a benchmark or a fixture. Those have their own directories and their own standards. Code that exists to cover an edge case, to measure something, or to feed another tool does not belong under `examples/`, however convenient it is to put it there.

Manage examples with the care their position deserves:

- **Realistic.** Each one solves a problem a user would recognise, with input and output worth looking at.
- **Focused.** Each one demonstrates one thing. Several short examples serve a reader better than one that shows everything.
- **Documented.** Each one carries ample comments that walk the reader through the code: what it is about to do, why it is done that way, and what to change to adapt it. Write them for someone reading top to bottom with no other context.
- **High quality.** An example is copied into real projects. It should be code the maintainers would be content to see copied.
- **Few.** Every example is a promise to keep it working. Add one when it teaches something the others do not.

4.2. Why examples live in two places

The repository and its releases are rarely in step. Pull requests land continuously; releases happen now and then. At most commits the package in the repository is ahead of the last release.

An example in the repository must work with the package at that same commit, so it may use an API that no release has yet. A user reading it beside the latest release could copy code that does not compile for them. An example published with a release has the opposite property: it works with that release and will never change.

One set of files cannot be both. The repository therefore keeps examples that follow the working tree, and each release publishes a frozen copy of them that names that release.

4.3. Repository examples test the working tree

Every example names the package by relative path, as described in [Roc applications in the repository](#). A pull request that changes the package and breaks an example fails on that pull request.

Committed examples are never rewritten after a release. No release follow-up pull request exists, and the default branch never names its own release URL.

4.4. Record expected output

An example that only compiles proves little. Record what each example prints, and compare it on every run. Examples are run by the same driver as the integration tests; see [Integration tests](#).

- Keep expected output under `tests/`, in one specification the scripts read: the arguments each example runs with, its standard input where it has one, and the exact output expected.
- The check script runs every example and fails on a difference.
- The same script accepts `--update` to rewrite the expected output, so that an intended change is a reviewed diff.
- An example that cannot run in CI, such as one needing a display or a console runtime, is still checked and built, and the specification says so explicitly. It is never silently skipped.

4.5. The examples archive

Each release attaches an examples archive named `<name>-examples-<version>.zip`, where `<name>` is the repository, or the package when the repository publishes several. It contains:

- complete copies of the examples, including sibling modules and data files, in which the package dependency is that release's bundle URL;
- the expected-output specification those examples were tested with; and
- a short README naming the release, the bundle URL and how to run an example.

Nothing else in an example changes. The archived examples differ from the repository's at the tested commit only in the package dependency, and the check script verifies that.

The one line that differs between the repository and the archive

```
app [main!] {
  cli: platform "https://github.com/roc-lang/basic-cli/releases/download/<version>/<hash>.tar.zst",
-  ansi: "../package/main.roc",
+  ansi: "https://github.com/<owner>/roc-ansi/releases/download/<tag>/<hash>.tar.zst",
}
```

A user who wants a starter downloads the archive for the release they are installing. The archive is also the fixed input to [nightly compatibility](#), which is why it carries its own expected output: the default branch's specification describes the default branch, not the release.

4.6. Build and test the exact archive

The archive that is tested is the archive that is published. Nothing is rebuilt between the test and the upload.

1. Build the bundle from the tested commit with `roc bundle`. Its filename is its content hash and the release tag is the version being released, so the final URL, `.../releases/download/<tag>/<hash>.tar.zst`, is known before anything is published. A release with several bundles puts a package name before the hash; see [Several bundles in one release](#).
2. Build the archive with that real release URL in every example.
3. Serve the bundle from a loopback address.
4. Run the archived examples with `--replace-dep <release URL> <loopback URL>`, which `roc check`, `roc test` and `roc build` accept. The files are unchanged; only the download location differs for the duration of the test.
5. Compare the output with the archived specification.
6. Publish the same bundle and the same archive.

Run the test with an empty Roc package cache, so that the bundle is fetched and verified and not found from an earlier run.

4.7. Dry runs

A pull request runs the release steps with a placeholder version and publishes nothing. Roc rejects `0.0.0` in a package URL, so the placeholder is `0.0.1-dry-run`.

4.8. An archive is never reformatted

An archive is immutable, so it is checked, tested, built and run, and never passed to `roc fmt`. A later formatter change would fail the nightly check for a reason that no patch release can fix.

Chapter 5. Test in pull-request CI

Code that only compiles proves little. That is true of examples, and equally of tests, benchmarks, fuzz targets and fixtures. Anything in the repository that is not run goes stale without anyone noticing, and is found broken on the day it is needed.

The rule is that every Roc program in the repository runs in pull-request CI, even if only briefly. This chapter says what runs, how much of it, and how the tests are organised.

5.1. What runs on every pull request

Check	What it runs
Formatting	<code>roc fmt --check</code> over the repository's Roc sources
Unit tests	<code>roc test</code> on the package or platform root
Examples	Every example, built once and run against its recorded cases
Integration tests	The test driver, over its full list of cases
Allocation ratchets	The cases that record an allocation ceiling, on a test platform that counts allocations
Benchmarks	A smoke run: each benchmark builds and completes one small iteration
Fuzz targets	A smoke run: each target builds and replays its committed corpus
Scripts	The <code>expect</code> tests of the modules under <code>scripts/src/</code>

The `check` task runs all of these, so a contributor runs the same set with `blueprint run check` before opening a pull request.

5.2. Heavy runs stay out of pull-request CI

A long fuzzing campaign or a full benchmark run never runs on a pull request. They take too long, their results vary from run to run, and a pull request should not wait on them or fail because of noise. They run on a schedule or on request, as described in [Fuzzing](#) and [Benchmarks](#).

What pull-request CI owes them is a smoke run. A benchmark that has not been built since the API it measures changed will not run on the night it is scheduled. A few seconds per target on every pull request keeps each of them building and running, so the scheduled job measures and searches instead of discovering that it is broken.

5.3. Build with `--opt=dev`

Build and run with `--opt=dev` unless there is a reason not to. It compiles quickly, which is what a test run spends most of its time on, and the checks in this chapter ask whether the program is correct, not how fast it is.

Use `--opt=speed` only where the optimised build is the thing under test: a benchmark's measured run, a long fuzzing campaign, or a release artifact that ships optimised. A benchmark's smoke run in pull-request CI uses `--opt=dev` like everything else.

5.4. Unit tests

Run `roc test` once, on the root: `package/main.roc` for a package, `platform/main.roc` for a platform. Roc runs the `expect` tests of every module the root imports, transitively, so there is no need to list files or to run the command per module. A module that the root does not reach is either dead code or missing from the package, and either is worth knowing.

A repository with several roots runs `roc test` once on each.

5.5. Integration tests

An integration test runs a built program and looks at what it does: its output, its exit code, the files it writes. Organise these around an external driver and a list of cases, not as one program per test.

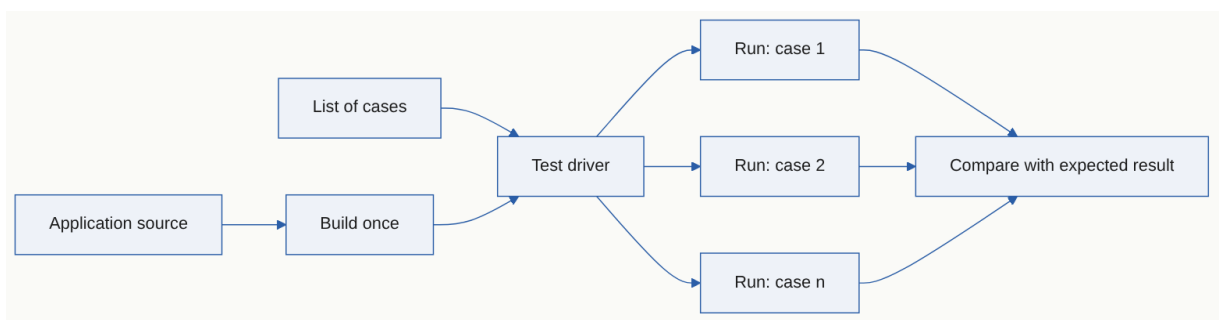


Figure 2. One build, many cases

- **Build once.** The driver builds the application a single time and runs the resulting executable for every case. Compiling dominates the cost of a test run, so this is the difference between a suite that takes seconds and one that takes minutes.
- **Cases are data.** Each case names its arguments, its standard input, any files it needs, and the expected output and exit code. Adding a case is adding an entry, not writing a program.
- **Cover failure as well as success.** The same executable is run on the error paths: bad input, missing files, invalid arguments. A case that expects a particular error message and a non-zero exit code is as ordinary as one that expects success.
- **The driver owns the environment.** Because it sits outside the program under test, the driver decides how cases run. It can give each case its own process and working directory, run cases in parallel, set up and tear down what a case needs, and apply a timeout. None of that leaks into the application.
- **An update mode.** The driver accepts `--update` to rewrite expected results, so an intended change in behavior is a reviewed diff.

The driver is a Roc script under `scripts/`, and the cases live under `tests/`. The expected-output specification for examples, described in [Record expected output](#), is one such list of cases run by the same driver.

5.6. Skip a broken case, visibly

Sometimes a case is broken for a reason that cannot be fixed today: a compiler bug, a dependency that has not been released, a platform limitation. Deleting the case loses it, and leaving it failing blocks every pull request. Skip it, and make the skip impossible to forget.

- Mark the skip where the case is defined, with a `TODO` note and a link to a GitHub issue that tracks the cause.

```

# TODO: skipped until the compiler handles nested tag unions here.
# https://github.com/roc-lang/roc/issues/<number>
{ name: "nested-tags", skip: Skip("compiler bug, roc-lang/roc#<number>"), ... }

```

- A skip always carries a reason. The harness rejects a skip with no note.
- The harness prints every skipped case with its note, and counts skips in the summary. Nobody should have to search the source to learn what is not being tested or why.
- A skip is temporary. When the linked issue closes, the skip is removed in the pull request that confirms the case passes again.

The same applies outside the harness: an example, benchmark or fuzz target that is switched off says why, in a `TODO` with an issue link, at the place it is switched off.

5.7. Report progress and results

A harness is read far more often than it is written, usually by someone waiting for it or trying to find out what failed. Its output is part of its job.

What a run looks like

```
Running 20 cases
[ 1/20] parse-empty-input ..... PASS      12ms
[ 2/20] parse-nested-lists ..... PASS      48ms
[ 3/20] render-wide-table ..... FAIL    1_204ms
[ 4/20] nested-tags ..... SKIP          compiler bug, roc-lang/roc#1234
...
[20/20] rejects-missing-file ..... PASS      9ms

Failed
  render-wide-table: expected exit code 0, got 1
    tests/cases/render-wide-table

Skipped
  nested-tags: compiler bug, roc-lang/roc#1234

Summary: 18 passed, 1 failed, 1 skipped in 10_312ms
Slowest: render-wide-table 1_204ms, build 6_870ms
```

- **Show progress as it happens.** Print each step with its position, such as `3/20`, and its result as soon as it finishes. A run that prints nothing for a minute looks the same as a run that has hung.
- **End with a summary.** Report how many cases passed, failed and were skipped, list each failure with enough detail to act on, and list each skip with its note. The summary is what a reader sees first in a CI log, so it stands alone.
- **Report timing.** Give the time of each case, the total, and the slowest steps, including the build.
- **Use one unit for time.** Print every duration in milliseconds, with digit separators and aligned to the right: `12ms`, `1_204ms`, `10_000ms`. With a single unit the eye compares lengths, and a slow case stands out. A column that mixes `850ms`, `1.2s` and `2m` hides exactly the outlier worth seeing.
- **Use color.** `FAIL` is red, `PASS` green, `SKIP` yellow. Color is an aid and never the only signal: the word

is always printed, and color is turned off when output is not a terminal or when `NO_COLOR` is set.

- **Exit accurately.** Any failure gives a non-zero exit code. Skips alone do not.

5.8. Measure allocations

Memory behavior is a quality of Roc code in its own right, and it is invisible unless it is measured. A change that keeps a reference alive a little longer can turn an in-place update into a copy of a whole list, on every call. The result is still correct, every test still passes, and the package is now several times slower for its users.

Most packages and platforms should therefore test on a platform that counts allocations, and hold the counts with ratchets.

- Use a test platform whose host records allocations, reallocations and frees while a case runs, and reports the counts to the harness.
- Record a ceiling for the cases that matter: the hot paths, and the operations users will call in a loop.
- The harness fails a case whose count exceeds its ceiling, and prints the ceiling beside the measured value.
- A ceiling is a ratchet. It is compared as "no more than", never for equality, because the compiler floats and a newer one may do better. When a count improves, lower the ceiling in the same pull request so the gain cannot be lost quietly.
- Raising a ceiling is a reviewed decision with a stated reason, not a way to make a red build green.

Allocation counts are deterministic for a given compiler, which makes them a far steadier signal than timing. They run in pull-request CI with everything else; wall-clock benchmarks do not.

5.9. Property-based tests with roc-fuzz

Unit tests and integration cases check the inputs someone thought of. Most bugs live in the inputs nobody thought of: the empty list, the string that ends in a delimiter, the two values that are equal, the length one past a boundary. A property-based test closes that gap. You state a rule that must hold for every input, and a tool searches for an input that breaks it.

`roc-fuzz` is that tool for Roc. It is a platform: a fuzz target is a small Roc application that names what values to try, what code to run, and what must be true afterwards.

Part	Role
Generator	Produces typed Roc values to try: numbers, strings, lists, records, or shapes built from them
Operation	Calls the package code under test
Property	Decides whether the result is acceptable for that input

5.9.1. Why it suits pure Roc code

Most of a Roc package is pure functions, and pure functions are the best case for this kind of testing.

- **Nothing to set up.** A pure function needs no files, network, clock or state. The target calls it with a generated value and inspects the result.
- **Failures reproduce exactly.** The same input always gives the same output, so a saved failing input is a complete bug report. There is no flakiness to rule out.
- **Properties are easy to state.** Pure code invites rules that hold for every input: decoding an encoded value returns the original; formatting twice changes nothing the second time; a parser returns an error and never crashes; an optimised implementation agrees with a simple one.
- **The search is guided.** roc-fuzz watches which parts of the compiled program each input reaches and keeps the inputs that reach something new, so it works its way into branches that random values would rarely hit.
- **Failures arrive small.** roc-fuzz saves a failing input, shows it as a Roc value, replays it, and minimises it to the smallest case that still fails.

The benefit is largest exactly where a package carries the most risk: parsers, encoders and decoders, data structures, arithmetic on sizes and indexes, and anything that handles text.

5.9.2. What a target looks like

```
app [target] {
  fuzz: platform "https://github.com/lukewilliamboswell/roc-
fuzz/releases/download/<version>/<hash>.tar.zst",
  pkg: "../..../package/main.roc",
}

import fuzz.Fuzz
import pkg.Stack

# After pushing an item, popping must return that item and the original stack.
```

```

test : List(U8) -> Fuzz.Outcome
test = |initial| {
  item = 0
  pushed = Stack.push(initial, item)

  match Stack.pop(pushed) {
    Ok({ value, rest }) if value == item and rest == initial => Fuzz.keep
    Ok(_) => crash "pop did not undo push"
    Err(Empty) => crash "the stack was empty after push"
  }
}

target = Fuzz.target_with({
  name: "stack-push-pop",
  generator: Fuzz.list(Fuzz.u8, 8),
  test,
  show: |initial| Str.inspect(initial),
})

```

The target depends on the working tree by relative path, like every other Roc application in the repository, and on a roc-fuzz release by URL. It lives under `fuzz/`, outside the package root, so it is never bundled.

5.9.3. How targets are managed

- Write one operation and one property per target, so that a failure says what went wrong.
- Add a target when a module has a rule that should hold for all inputs. Every parser and every encode and decode pair should have one.
- When roc-fuzz finds a bug, fix it and keep the minimised input as an ordinary `expect` test. The unit test stops that bug returning; the target keeps searching for the next one.
- Commit a small corpus for each target. Pull-request CI builds every target and replays that corpus as its smoke run.
- Run the real search on a schedule, as described in [Fuzzing](#).

A weak property finds nothing, and a passing fuzz run is not a proof. The [roc-fuzz manual](#) covers writing generators and properties, running campaigns, and reading a saved failure.

5.10. Formatting

Run `roc fmt --check` in pull-request CI, over every Roc source the package compiler builds, and `roc-stable fmt --check` over the scripts, whose `roc` field another compiler's formatter would rewrite. It fails the pull request that introduces unformatted code, which is the only moment at which fixing it is cheap.

Do not run it in `nightly.yml`. A new nightly can change the formatter, and a formatting difference

says nothing about whether a release still works. The check would raise an alarm that no patch release answers. When a new formatter does change the output, the next pull request sees it, and reformatting is a small pull request of its own.

Chapter 6. The three workflows

Every package and platform repository has the same three workflows. Each answers one question, and none of them changes the repository.

Workflow	Trigger	What it proves	Token access
<code>ci.yml</code>	Pull request; push to the default branch	Scripts pass on <code>roc-stable</code> . The working tree and its examples pass on the latest nightly. A platform's committed host bindings match that nightly.	<code>contents: read</code>
<code>release.yml</code>	Manual dispatch; pull request as a dry run	The exact bundle, examples archive and documentation proposed for publication work.	<code>contents: read</code> . The publish job alone has <code>contents: write</code> , behind a protected environment.
<code>nightly.yml</code>	Daily schedule; manual dispatch	The latest published release and its examples archive work with the latest nightly.	Test jobs <code>contents: read</code> ; the report job <code>issues: write</code>

Rules that hold in all three:

- Every `uses:` reference is a full commit SHA.
- Top-level `permissions` is `{}`, and each job states what it needs.
- Checkout sets `persist-credentials: false`.
- A job that runs repository code has no write access. A job with write access runs no repository code.
- The nightly tag is resolved once, in the first job, and passed to every other job as the `nightly-tag` input of `roc-lang/setup-roc`. Matrix jobs never each resolve "latest" for themselves, or one run could test several compilers.
- Each job calls a `blueprint run <task>` and holds no logic of its own.

Workflows beyond these three, such as a documentation site or scheduled fuzzing, are described

in [Optional workflows](#).

6.1. Continuous integration

`ci.yml` tests the working tree.

1. Set up Blueprint and install the latest nightly, as in [Use the environment in CI](#).
2. Run `blueprint run check`.

[What runs on every pull request](#) lists what the `check` task covers.

A [pure Roc package](#) runs this as one job on `ubuntu-latest`, and that job's name is the required status check.

A repository whose code can behave differently by target runs it as a matrix, one job per runner, and ends with a single aggregate job that requires every matrix job to have succeeded. The aggregate job's name is the required status check, so the ruleset does not need editing when the matrix changes.

A platform adds the host-bindings check described in [Host bindings](#). It runs here and nowhere else.

A pull request can fail because a new nightly arrived, with no fault in the change itself. Compare with the latest `nightly.yml` run and with the default branch before treating a failure as a regression in the pull request.

6.2. Release

`release.yml` builds what a release would contain, tests those exact bytes, and publishes them once. A maintainer dispatches it from the default branch with a version. A pull request runs the same build and tests as a dry run and publishes nothing.

6.2.1. What a release contains

One immutable GitHub release per version holds:

Asset	Purpose
The bundle, under the filename <code>roc bundle</code> emits	What users depend on. The filename ends in its content hash, which is never changed. Several bundles in one release says when the filename also carries a name.

Asset	Purpose
<code><name>-examples-<version>.zip</code>	Examples pinned to this release's bundle, with their expected output.
<code><name>-docs-<version>.zip</code>	The generated API reference and site for this version, so the documentation of any release can be restored.
<code><name>-<version>.pdf</code>	The manual, where the repository has one.
Release notes	An authored account of what changed, with the real bundle URLs and the compiler versions used.
SBOM and build-provenance attestations	Where the repository produces them.

Managing releases carefully is a large part of what makes a package dependable. A user meets a release through three things: its version number, its notes, and the URL they paste into a header. Each is handled deliberately.

6.2.2. Choose the version with `roc bump`

A version number is a promise about compatibility, and `roc bump` checks the promise against the code. It compares the public API of the package with a previous release and reports the semantic-version bump the difference requires: patch, minor or major.

```
# what bump does this change need, and what is the next version?
roc bump --old <previous release bundle URL> package/main.roc

# in release.yml: fail unless the chosen version bumps far enough
roc bump --old <previous release bundle URL> --expect 0.13.0 package/main.roc
```

- Run it before choosing a version, so the number follows from the change and not from habit.
- `release.yml` runs it with `--expect` in its validate step, against the latest release, and stops when the requested version understates the change. A breaking change can then never go out as a patch.
- A repository with several packages runs it once per package root.
- It compares only the modules the package header exposes. A change in behavior behind an unchanged signature is still the maintainer's call, and belongs in the notes.

A package version is independent of any compiler version. A changed package is always a new version. Nothing is ever republished under an existing version.

6.2.3. Write the release notes

Each release has its notes in `docs/releases/<version>.adoc`, for example `docs/releases/0.12.0.adoc`. They are committed in a pull request before the release is dispatched, and the workflow refuses to release a version whose notes file is missing. A top-level `docs/releases.adoc` chapter includes each file, so the notes of every release are part of the manual; see [Write for the shared renderer](#).

The notes are an authored narrative, written for a user deciding whether and how to upgrade. They are not a list of commits or merged pull requests.

- Open with what this release is for, in a sentence or two.
- Describe what changed in terms of what a user can now do, or must now do differently.
- For a breaking change, show the code before and after.
- Say what was fixed, naming the visible symptom.
- Thank contributors, and link issues where a reader may want the background.

`docs/releases/0.12.0.adoc`

```
= roc-example 0.12.0

This release adds streaming decoders and removes the deprecated `Table.render`.

== Use this release

[source,roc]
----
app [main!] {
  cli: platform "https://github.com/roc-lang/basic-cli/releases/download/<version>/<hash>.tar.zst",
  example: "{package-url}",
}
----

== Streaming decoders

...
```

The bundle URL cannot be written by hand, because it contains the content hash of a bundle that does not exist until the release builds it. The notes therefore use a placeholder, and the workflow fills it in:

- `{package-url}` stands for the released bundle's URL. A release with several bundles has one placeholder for each, named after the package, such as `{tzdb-url}`.
- After building the bundles, `release.yml` renders the notes with each placeholder set to the real URL, tag and hash. A placeholder that is left unresolved, or one the notes never use, fails the release.
- The rendered notes become the body of the GitHub release, followed by a section headed

"Compilers" that holds the output of `roc version` and `roc-stable version`. That record is the statement of what the release was built and tested with; no header pin makes it.

- A dry run renders the notes too, so a mistake in them fails on the pull request. It renders one file: the notes for the highest version under `docs/releases/`, which are the notes of the next release whenever a pull request adds or edits them. It asks nothing of GitHub to choose that file.

The committed file keeps its placeholders. It is never rewritten after the release, and no follow-up pull request is needed.

6.2.4. The order of a release

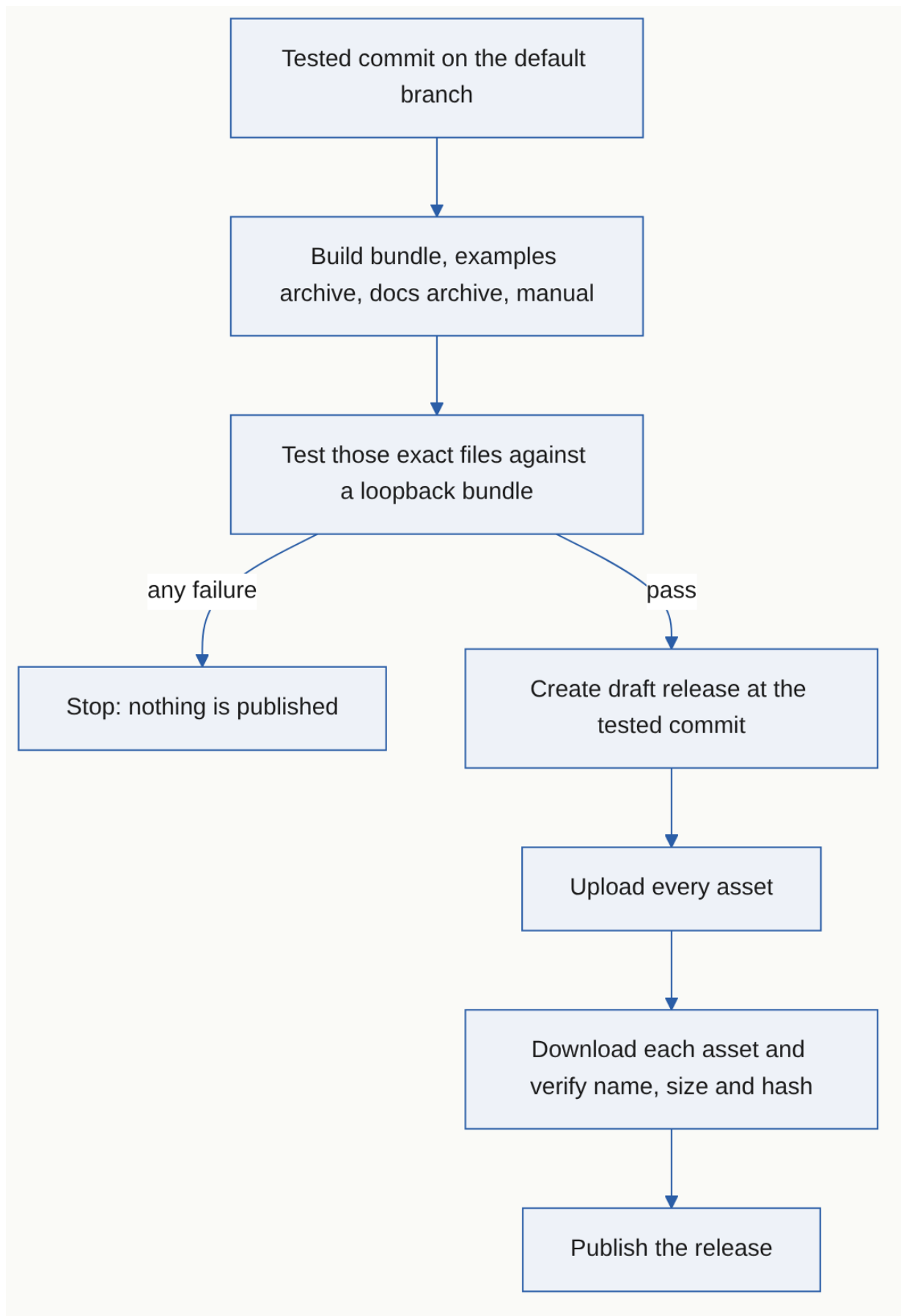


Figure 3. A release tests exact bytes, then publishes once

1. **Validate.** Require the default branch. Reject a version whose tag or release already exists.

Require `docs/releases/<version>.adoc`. Check the version with `roc bump --expect`. Record the event commit and use that SHA in every later job.

2. **Check.** Run `blueprint run check` on that commit.
3. **Build.** Build the bundle with the latest nightly, then the examples archive with the real release URL, the documentation archive, and the manual. Render the release notes with the real bundle URLs. Upload them as workflow artifacts.
4. **Test.** On every supported target, download those artifacts and test them as described in [Build and test the exact archive](#).
5. **Publish.** A separate job, with `contents: write` and a protected environment, downloads the tested artifacts. It checks out nothing and runs no repository script. It hands the version, the rendered notes and the asset files to `actions/publish-release`, which stages a draft at the tested commit, verifies every uploaded asset against the tested file, and then publishes.
6. **Update the site.** For a final release, rebuild the documentation site so that it points at the new release, and deploy it; see [What the site points at](#). A release candidate does not change the site.

The publish job consumes the artifacts of its own run. It never rebuilds, and never resolves a moving branch.

Enable immutable releases in the repository, so that a published release cannot gain, lose or replace an asset. Hash checks remain necessary even with immutable hosting.

6.2.5. Several bundles in one release

A repository that publishes several packages or platforms builds one bundle and one examples archive for each, in the same run, under the same version. Each archive names only its own bundle, plus any sibling bundle its examples need. The release notes list every bundle URL by name. All of them are tested before any of them is published.

Two bundles under one tag need different names as well as different hashes. Roc identifies a package by its URL with the version and the content hash removed. Two files named only `<hash>.tar.zst` under one tag are therefore one package with two contents, and Roc refuses the pair as soon as one depends on the other:

The same package version is being served with two different content hashes

Give each bundle a prefix that names its package and ends in a hyphen, such as `example-tzdb-<hash>.tar.zst`. Roc reads the hash as the run of base58 characters at the end of the filename, so the prefix must end in a character that is not base58, and a hyphen is one. The hash itself is never changed.

- At most one bundle in a release may go without a prefix. In a repository that starts with several packages, name every one.
- The prefix is part of the package's identity. Adding, removing or changing it makes a different package as far as Roc is concerned, so choose it before the first release, and treat a change to it as a breaking release.
- `roc bundle` writes `<hash>.tar.zst`. The script that builds the release copies that file to its prefixed name, and the prefixed file is what is tested and published.
- `roc bundle` resolves the dependencies of the package it bundles. A package that names a sibling by its final release URL is bundled after that sibling, with the sibling's bundle already in the package cache, because the URL does not exist until the release is published.
- `roc unbundle` accepts only the bare `<hash>.tar.zst` name. A script that unpacks a prefixed bundle copies it to that name first.

6.2.6. Release candidates

A release candidate, such as `0.12.0-rc1`, is a prerelease published for users to test before the maintainers commit to the release. It answers the question the workflows cannot: does this version work in real projects, and is the API right?

- Publish a candidate through `release.yml` with a prerelease version. It follows every step above and holds the same assets, so what users test is built exactly as the final release will be.
- Mark it as a prerelease on GitHub. It is never the repository's latest release.
- Announce it where users will see it, and say what feedback is wanted and for how long.
- Fix what the feedback finds on the default branch, and publish `-rc2`. A candidate is immutable like any other release and is never amended.
- When a candidate has stood without change, release the final version from the same commit. Its bundle is a new artifact under a new tag, and it goes through the release workflow in full.

Everything that speaks of "the latest release" means the latest release that is not a prerelease.

`nightly.yml` tests that one, and the documentation site links to that one. A candidate is found from the releases page and from its announcement.

6.2.7. Recover an interrupted publication

If publication stops, inspect exactly what exists before retrying. The publish job's error says which of these states it left.

- **A draft and no tag.** Nothing is public. The error names the draft, which holds whatever was uploaded before the failure. Look at it to see how far the job got, delete it, and re-run the failed

publish job of the same workflow run. That job publishes the files the run already tested. The publish action never adopts a draft it finds, so the draft is deleted first.

- **A published release.** It is immutable and is never repaired. If it is complete, the version is released and nothing is re-run. If it is not, publish a new version.
- **Unknown.** When the answer to the publish request was lost, look at the releases page and treat what is there as one of the two states above.

Changed bytes are always a new version. Do not publish a draft by hand, replace an asset, or rebuild from a moving branch. If the run's artifacts have expired, delete the draft and dispatch the release again from the default branch. The version was never published, so it is still free, and the new run builds and tests its own files.

6.3. Nightly compatibility

`nightly.yml` answers one question each day: does the release users download today still work with today's compiler? A failure means the package or platform needs a patch release.

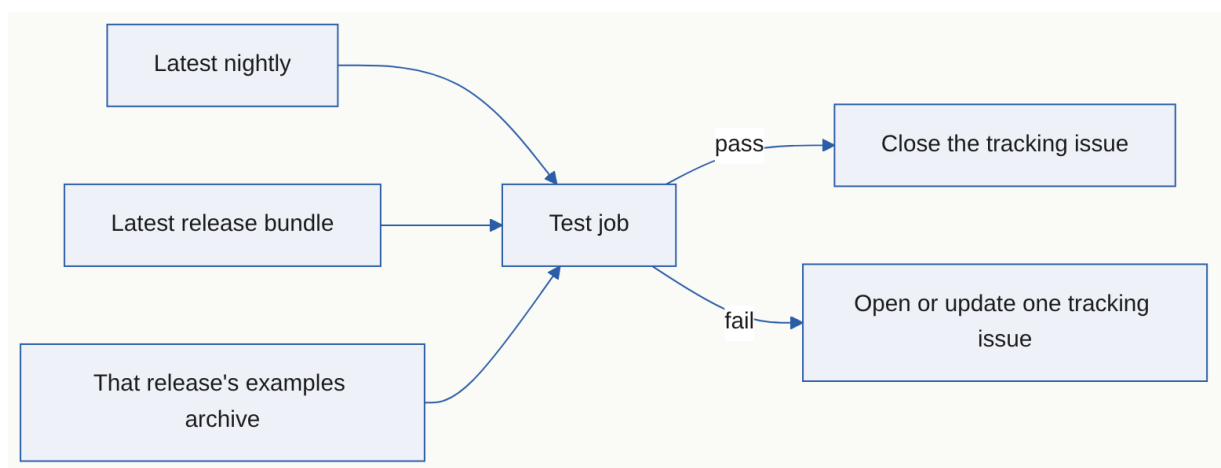


Figure 4. The inputs to one nightly run and its two outcomes

- Resolve the latest nightly tag and the latest release once. Accept both as optional dispatch inputs, so a maintainer can re-run a past combination.
- Skip prereleases when choosing the latest release.
- Download that release's examples archive and run it unchanged, comparing with the expected output inside the archive. Do not rebind it to the working tree, and do not test the default branch's examples here: new APIs on the default branch are not evidence about a release.
- Use the default branch's scripts, on `roc-stable`, to drive the test.
- Run on every target the release is tested on: the one runner of a [pure Roc package](#), or the same matrix as `ci.yml`.
- A platform's host bindings are not checked here. A released host is fixed; what matters is

whether its applications still build and run.

The archive is immutable, so a failure is never a reason to change it. The fix is a patch release that works with the new compiler.

6.3.1. Reporting

A separate job reports the result by calling `actions/nightly-report`, which keeps one tracking issue: opened on the first failure, commented on by each later failure, and closed by the next pass. The action defines which issue that is and what each outcome does. The workflow supplies the rest:

- The report job has `if: ${{ always() }}`, so that it runs when the test job fails, and holds `issues: write` and nothing else.
- It checks out nothing and runs no repository or archive code.
- The test job gives the step that runs the examples an `id` and exposes that step's `outcome` as a job output, with the nightly tag and the release tag. The report job passes those three values to the action and nothing else: never text produced by the examples under test.
- The outcome is the step's, not the test job's result. A job that failed while installing the compiler or downloading the archive has a failed result and an examples step that never ran. The action reports that as a setup failure and leaves the compatibility issue alone.

A repository with no release yet passes with a notice and does not call the action.

This workflow never opens a pull request, commits, merges, or publishes.

Chapter 7. Optional workflows

Some repositories need more than the three core workflows. Each addition below has a fixed name and purpose, so that the same file means the same thing everywhere. All of them follow the rules in [The three workflows](#): full-SHA references, `permissions: {}` at the top, and no repository code in a job that can write.

Workflow	Trigger	Purpose	Token access
<code>docs.yml</code>	Push to the default branch; pull request	Build the site and manual; deploy to GitHub Pages from the default branch.	Build contents: <code>read</code> ; deploy pages: <code>write</code> , <code>id-token: write</code>
<code>fuzz.yml</code>	Schedule; manual dispatch	Run fuzz targets for a bounded time and keep reproducers.	<code>contents: read</code>
<code>bench.yml</code>	Schedule; manual dispatch	Measure performance against a recorded baseline.	<code>contents: read</code>
<code>codeql.yml</code> , <code>scorecard.yml</code>	Push, pull request, schedule	Static analysis and repository-practice scoring.	As each tool documents
<code>link-inputs.yml</code>	Pull request touching producer inputs; manual dispatch	Build, test and attest native linker inputs.	<code>contents: read</code> , <code>attestations: write</code> , <code>id-token: write</code>
<code>publish-link-inputs.yml</code>	Manual dispatch from the default branch	Publish an attested candidate and commit its lock.	See Trusted publisher contract

7.1. Documentation site

`docs.yml` has two jobs.

The build job checks out the repository, builds the site, and uploads it as a Pages artifact. A repository with an AsciiDoc manual uses `actions/build-docs`, which also produces the PDF. A pull

request runs this job, so a broken manual fails review, and the artifact can be downloaded to preview the change.

The deploy job runs only for the default branch. It has `pages: write` and `id-token: write`, uses the `github-pages` environment, checks out nothing, and deploys the artifact the build job produced.

7.1.1. What the site points at

A visitor to the site is almost always a user, and a user has the latest release. The site therefore leads with the latest release, not with the default branch:

- its primary API reference is the `roc docs` output generated for the latest release, restored from that release's documentation archive;
- its examples link is the examples archive attached to the latest release; and
- its install snippet names the latest release's bundle URL.

"Latest" excludes prereleases; see [Release candidates](#).

The site may also carry the default branch's documentation, under a path and a label that mark it as unreleased, and the documentation of earlier releases, each restored from its own archive. Nothing rendered is committed: the site is assembled at build time from the working tree and from release assets.

A new release changes what "latest" means, so `release.yml` rebuilds and deploys the site after it publishes, in a job that holds only the Pages permissions and deploys an artifact built by a read-only job.

7.2. Fuzzing

Fuzz targets are written with `roc-fuzz`; see [Property-based tests with roc-fuzz](#) for why and how.

`fuzz.yml` runs each target for a fixed duration on a schedule. It restores and saves the corpus through the Actions cache, uploads any reproducer as a workflow artifact, and fails when a target fails. Fuzz targets are Roc applications that depend on the working tree, so the latest nightly builds them.

`ci.yml` gives every target a smoke run: it builds the target and replays the committed corpus, which is fast and deterministic. That keeps the targets building between scheduled runs. The scheduled run is the only place new inputs are searched for, and the only place a target is built with `--opt=speed`.

7.3. Benchmarks

`bench.yml` measures on a schedule and on request, never on a pull request. `ci.yml` gives every benchmark a smoke run, one small iteration built with `--opt=dev`, so that a benchmark cannot fall behind the API it measures.

A measurement that depends on the compiler, such as an allocation count, is compared as a ratchet: the build fails when the value is worse than the recorded ceiling, and a maintainer lowers the ceiling when it improves. It is never compared for equality, because the compiler floats and an exact value would fail on a compiler that did better.

7.4. Linker inputs

Platforms that consume prebuilt native inputs have a producer workflow and a publisher workflow. These are the only route by which native build inputs are released. [Maintain a platform](#) explains when a platform needs them, and [Publish linker inputs from a pull request](#) specifies both.

Chapter 8. Maintain a platform

A platform follows the same model as a package: the same environment, layout, examples archive and three workflows, and no compiler pin in the platform or its examples. This chapter covers what a platform adds, which is a native host. A host brings generated bindings, a target matrix, and build inputs that are too expensive or too sensitive to rebuild on every pull request.

8.1. Host bindings

A host is compiled against bindings that `roc glue` generates from `platform/main.roc`: a Zig, Rust or C description of the types and entry points the Roc side expects. The compiler generates them, so a new nightly can change them when nothing in the repository changed.

The bindings are committed. The host build reads the committed file and never runs `roc glue` itself, so a host builds the same way on every runner and a reviewer sees every change to the interface as a diff.

8.1.1. Check the bindings in pull-request CI

`ci.yml` has one job that regenerates the bindings with the latest nightly and requires the committed file to match:

1. Obtain the glue specification that belongs to the resolved nightly, from that nightly's source archive, and verify the archive's digest.
2. Run `roc glue <spec> <output directory> platform/main.roc` with the latest nightly.
3. Run `git diff --exit-code` on the committed bindings.

This check runs on pull requests and on pushes to the default branch. It does not run in `nightly.yml`: a released host is fixed, and what [nightly compatibility](#) asks is whether applications still build and run against it.

The job is part of the required aggregate check. When a nightly changes the generated bindings, every open pull request fails this job until the bindings are upgraded. That is intended: the default branch should not accept changes on top of an interface that no longer matches the compiler.

8.1.2. Upgrade the bindings

Regenerating the bindings is a manual task, because a changed interface usually needs host code to change with it.

1. Run the bindings script with its update flag, using the latest nightly.
2. Review the diff. A changed layout or signature needs matching host changes.
3. Build the host for every target and run the examples.
4. Open a pull request containing the bindings and any host changes. Merge after `ci.yaml` passes.
5. If the released platform no longer works with the new nightly, `nightly.yaml` has already reported it. Release a new platform version.

8.2. Cross-compile once, run on every target

A platform advertises a set of targets, and each workflow validates all of them. The usual shape is one builder job that cross-compile for every target, followed by a matrix of runner jobs that execute the result on each one.

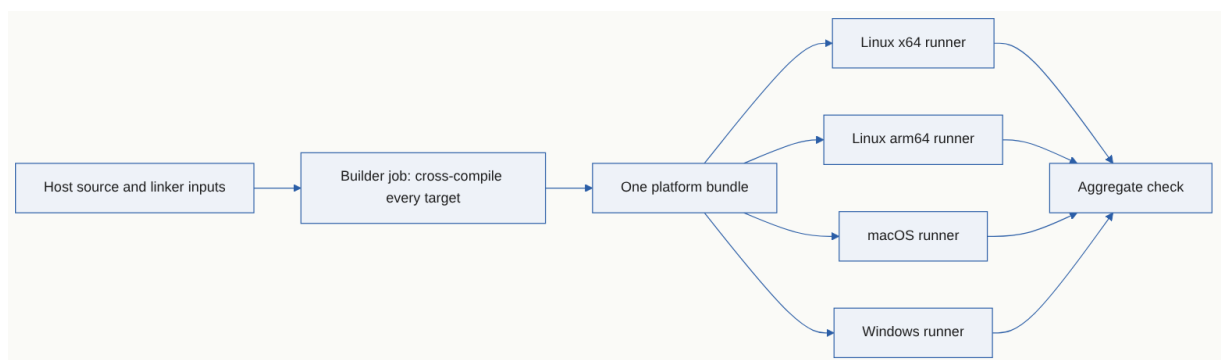


Figure 5. One builder, a matrix of runners

8.2.1. The builder job

- One job, on one runner, cross-compile the host for every supported target. A single toolchain version then produces every host, and the expensive work happens in one place where its cache is easy to manage.
- It assembles the hosts with the locked linker inputs and bundles once. The bundle is uploaded as a workflow artifact.
- Where a target cannot be cross-compiled, because it needs an SDK or a toolchain that exists on one operating system only, that host is built by a job on that system and handed to the builder as an artifact. This is the exception to plan around, not the default.

8.2.2. The runner jobs

- One job per supported target downloads the bundle and runs applications against it. A runner job never rebuilds the host or the bundle, so every target tests the same bytes, and those are the bytes a release publishes.
- One aggregate job requires every runner job to have passed. Its name is the required status check.

The purpose of the matrix is narrow: to show that the host and the linker inputs for each target link and run correctly there. Cross-compilation proves a host compiles for a target; only running on that target proves that it links, starts, and talks to the operating system as expected.

8.2.3. What the runners run

Run the examples. They exercise the platform through its public API the way an application does, and between them they usually reach enough of the host to show that a target works. They are already written, already have recorded expected output, and are what a user will try first.

Add dedicated tests where the examples leave a gap: a host service no example uses, a code path that differs per operating system, a failure that should be reported in a particular way. Keep them few and aimed at the host, because everything in the matrix runs once per target.

What does not belong in the matrix is anything that does not depend on the target. Unit tests of pure Roc code, formatting and the static checks run once, on one runner.

8.2.4. Runner environments

- State each runner’s runtime needs in the workflow: a display server, a software GPU, an audio sink, a browser. A simulated environment is reported as simulated; it is not evidence about the real one.
- An example that can be built but not run on a hosted runner is built and checked, and its expected-output specification records that it is not run.

The native toolchains come from `Blueprint.roc`, as described in [Declare the environment](#), so a contributor and every CI job compile the host with the same versions.

8.3. Several platforms in one repository

A repository that publishes more than one platform, such as a web platform and a native platform that share an engine, gives each its own root, its own bundle and its own examples archive, as in [Several bundles in one release](#). Sharing an engine does not make two platform APIs or their application roots interchangeable. Give examples separate roots and package URLs even when

they share application modules.

8.4. Template repositories

A template's examples are the product: they are what a user copies. A template follows the model without exception. Its examples depend on the working tree by relative path, and each release's examples archive is the starter a user downloads.

8.5. Native build inputs

Platforms need native builds before Roc can link an application, and native builds are where CI time goes. A maintainer has a limited budget of runner minutes and of patience, and spends it well only by planning what gets built when, and by watching how that changes as the project grows. This section is about building each thing once, reusing it until it changes, and testing and publishing the same bytes.

8.5.1. No binary blobs in the repository

As a general rule, a platform repository does not commit binaries. A committed library or object file has no visible origin: a reviewer cannot tell what source it came from, what built it, or whether it changed for a reason.

Binaries such as linker inputs are built by their own workflow, independent of ordinary CI, and published as an immutable release that carries:

- the archives for every supported target;
- a build-provenance attestation for each of them;
- the licenses of everything inside; and
- an SBOM.

The software supply chain can then be followed and verified end to end: from a source commit, through the workflow run that built it, to the exact bytes a platform links.

Routine use goes through content hashes. The repository commits a lock that names each archive, its release and its SHA-256, and every build verifies the bytes against that lock. A hash check is local and fast, it makes the archives safe to cache, and it needs no network service. The attestation API is reserved for the moments that need it: admitting a new release into the lock, auditing one, or an end user who wants to verify what they downloaded. It is not called on every build. [Separate admission from routine use](#) sets out that division.

There are exceptions. A platform may depend on libraries that another project already publishes,

for example the prebuilt libraries of a graphics library it wraps. Those may be vendored in the repository, so that builds do not make repeated network calls on someone else's project. A vendored binary records where it came from: the upstream project, the release, the download URL, its hash and its license. It is the upstream project's published artifact, unmodified, and never something the repository built and then committed.

8.5.2. Two releases, three build steps

A platform's host and its Roc source usually change together and are released together: one platform release holds both. Linker inputs are different. They change rarely, cost the most to build, and are published as their own release.

So there are two kinds of release, and three things that are built:

What is built	How it is released	In an ordinary pull request
Linker inputs	Their own immutable release, selected by a content lock	Download the locked archives and verify their hashes. Never rebuild them.
The host, such as <code>libhost.a</code>	As part of the platform release	Build it in its own job, and skip that job when the host's fingerprint has not changed.
The platform bundle	The platform release: the Roc source with the host and linker inputs	Assemble from the outputs above and test. Bundling compiles nothing native.

The host is not given a release lifecycle of its own. It is given a job of its own. The point of the separation is that each step can be skipped when its inputs have not changed.

8.5.3. Skip what has not changed

Compute a content fingerprint of everything that can affect the host: its source, generated bindings, dependency versions, toolchain, target and build options. Use the fingerprint as a cache key.

- When the fingerprint matches a cached build, the host job restores it and finishes in seconds. A pull request that touches only Roc source, examples or documentation compiles no native code.
- When it does not match, the job builds the host and saves it under the new fingerprint.
- Later jobs take the host as an artifact from that job. They do not know or care whether it was

built or restored.

Keep rebuild selection narrow and complete says what a fingerprint must include. A cache is storage, not a source of trust: restored bytes come from the same repository's own earlier job, and anything that crosses a release boundary is verified against a locked hash.

Separate steps and caches pay off beyond runner minutes. A failure points at the step that failed. A contributor without the native toolchain can still work on the Roc side. Review is easier when a pull request shows which step it invalidated.

8.5.4. Watch what CI costs

Plan the pipeline, then keep watching it, because it drifts as changes are made.

- Know roughly how long a routine pull request takes, and what takes the time. The harness and each workflow report durations for this reason.
- A routine change, one that touches no native code, should be fast. If pull-request CI takes a long time for routine changes, something is wrong and should be investigated: a fingerprint that includes too much, a cache key that changes on every commit, a job that rebuilds what an earlier job already built, a matrix that grew.
- Treat a slow pipeline as a defect with a cause, not as the price of the project. Slow CI is paid for on every pull request by every contributor.
- Spend resources where they prove something. Run the expensive matrix when native inputs change and before a release; run the cheap path the rest of the time.

`libhost.a` and `host.lib` are host outputs, not linker-input release assets. Operating-system import libraries, runtime libraries, interface stubs and resource objects required by `platform/main.roc` are linker inputs, even when the platform repository holds their generation recipes. Classify by ownership and invalidation, not by the fact that the final link receives the file.

Choose the smallest lifecycle that fits the project:

Project shape	Lifecycle
Pure Roc package	Bundle source. No native producer.
Package with a test-only host	Build the host in a read-only job of <code>ci.yml</code> . It is never published.
Platform with a small host and no external inputs	Build the host in its own fingerprinted job, bundle once, test the bundle, then publish those bytes.
Platform with expensive external inputs	The same, with the linker inputs released independently and consumed through a reviewed lock.

An existing binary is not a verified artifact until its build recipe, digest and target coverage are established. An explicit source build remains valid.

8.6. Separate the artifacts by what can invalidate them

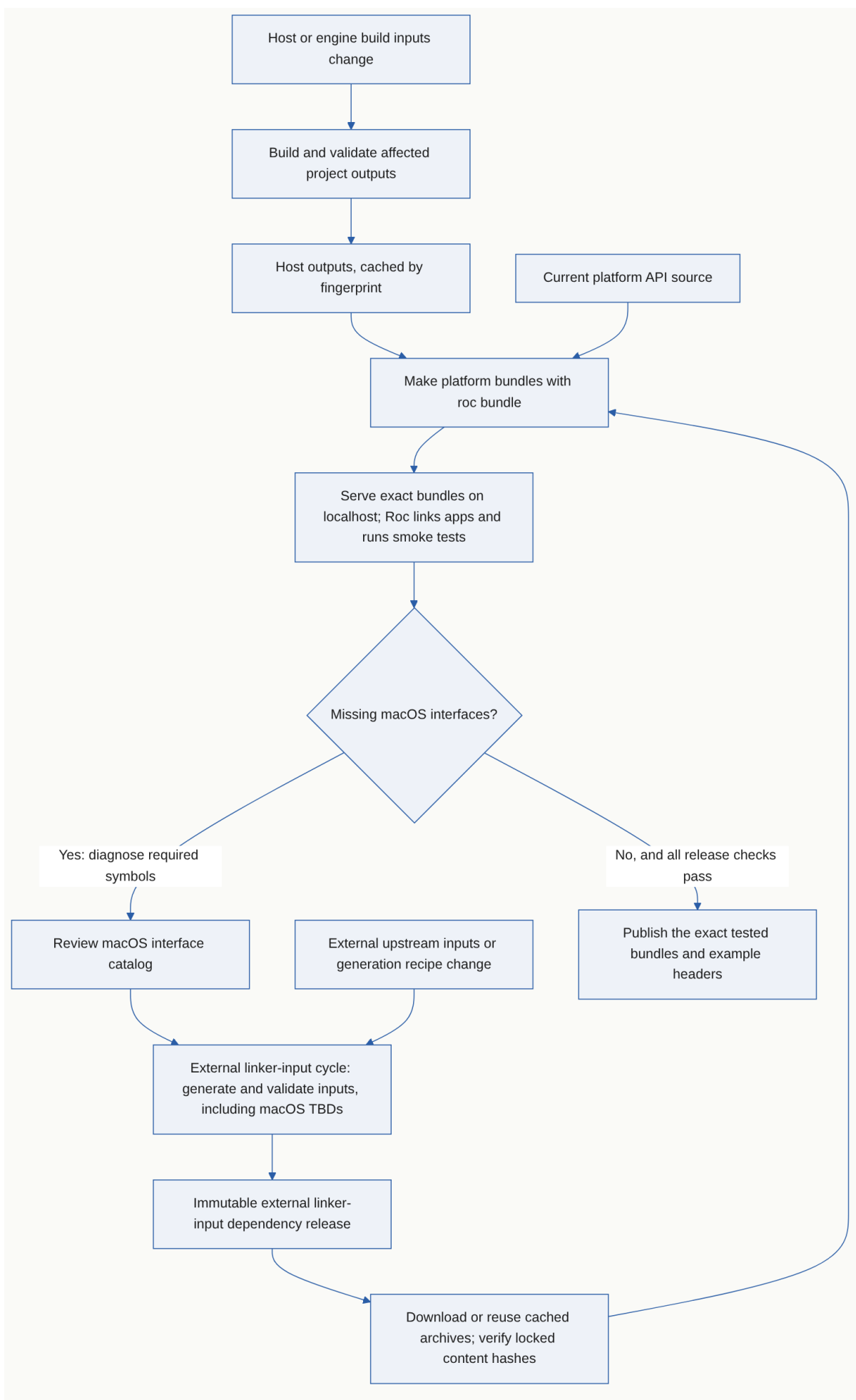
All of these files can eventually reach a linker, but they have different owners and rebuild triggers. Name the categories explicitly in scripts, locks and docs.

Artifact	What it contains	What requires a new build
Project host and engine outputs, such as <code>libhost.a</code> , <code>libengine.a</code> or a Wasm host library	Compiled project implementation for a target	Relevant source, generated inputs, dependency versions, toolchain, target ABI or build options change
External linker inputs, such as import libraries, runtime link files and macOS <code>.tbd</code> files	The external libraries or interfaces used by the final application link	Their upstream inputs, target contract, generation recipe or required interface catalog changes
Roc platform bundle	Platform API source plus the selected target outputs and linker inputs	Package source or selected input bytes change
Application root and companion modules	An application targeting a particular platform API and URL	Application source or selected package URL changes

A host edit can require new host archives while reusing the existing external linker inputs. Rebuild an engine archive only when its own inputs change; if a producer groups multiple outputs, document that granularity. A new nightly invalidates host outputs only when the compiler contributes to their build identity; a change to the generated bindings is handled as described in [Host bindings](#).

For platforms that package macOS interfaces, review the required catalog before generating and validating `.tbd` files. That generation belongs inside the external linker-input cycle. When released separately, publish the generated files as immutable inputs. For a static host archive, these files normally serve Roc's final link rather than host compilation. A producer that itself links executables or shared libraries may also consume linker interfaces. Record its actual SDK, headers, interfaces and native build tools in its identity; do not assume the host build and the linker inputs have no dependencies.

For a platform with a separate host step and an independent linker-input release, the flow is:



An unresolved symbol during the application link is evidence to investigate. Confirm the library, symbol and supported target before changing the catalog; missing project symbols, mismatched archives and ABI errors can produce similar failures. Do not invent an export or expand the catalog automatically to make a test pass. Producer validation can catch mistakes earlier, but a successful host build alone does not establish that Roc can link the final application.

8.7. Make content identity the ordinary consumer contract

Commit reviewed lock data with each dependency selection. Include the artifact name, target, exact release location, cryptographic digest such as SHA-256, byte size, and expected archive contents or extraction boundaries. Record available source revision and producer build-input fingerprint separately from the archive digest. The fingerprint answers whether a build can be reused; the digest identifies the exact bytes a consumer accepts. Neither a version label nor a fingerprint alone verifies a downloaded archive.

Recompute freshness fingerprints where the repository owns or vendors the producer inputs, such as its own host source. For external binaries, consumers verify the reviewed selection, digest, target and inventory; they need not fetch upstream source or install its toolchain to recompute the producer's fingerprint. Recorded producer metadata remains a claim subject to the chosen provenance policy.

Final Roc bundles have a compiler-defined content identity: preserve the hash that `roc bundle` puts in the filename, and use one filename in both localhost and published URLs. Roc uses its base58-encoded BLAKE3 hash to verify the downloaded bundle. An additional SHA-256 in release metadata is useful but cannot replace that URL hash. See the [Roc bundle implementation](#) and [download contract](#). Application users consuming that bundle need no separate native-input fetcher.

Consumers should resolve the locked artifact from a local cache, a mirror or the recorded release URL, check size and digest before extraction, and reject a mismatch, including on cache hits. Extract into fresh staging with bounded member counts and expanded sizes. Accept only declared regular files and directories with unique, portable paths; reject escapes, path collisions, links and special files. Validate the complete inventory before making the staged result available, and do not trust an old extracted tree solely because its cached archive verified. Keep licenses and required runtime files in the declared inventory. Fail clearly when a required artifact is unavailable or stale; an ordinary consumer must not silently rebuild it or pick a newer release.

The reviewed lock is the trust anchor. A checksum downloaded beside an archive from the same mutable location does not independently authorize its contents. A matching digest establishes equality with reviewed bytes, not their safety, source origin or correctness. Review the initial artifact selection and subsequent lock changes as dependency changes.

Hash verification is local and should work without a GitHub API call, token, attestation service or network connection when the selected bytes are already available. Downloading a missing artifact still requires a source of those bytes. This reduces service dependencies; it does not make uncached downloads offline. Apply the same principle to other downloaded inputs, including build tools where practical, while keeping the required trust policy explicit.

8.8. Publish provenance without confusing it with integrity

An attestation can bind an artifact digest to claims about its producer, workflow, source revision and build. Publish it for users who want to verify that provenance. Such verification must check the expected signer/workflow identity, source and subject digest under a stated policy, not merely that some signature exists. Keep ordinary locked consumption independent of that service check unless the consumer explicitly adopts a stricter provenance policy. Verification required by an adopted policy must fail visibly rather than silently downgrade to hashes. Provenance does not inherently require an online check on every run: GitHub supports [offline verification](#) using previously obtained attestation bundles and trusted roots. Define how that verification material is obtained and updated when adopting such a policy.

Describe only the evidence that was actually produced. A packaging workflow that downloads and bundles an archive can attest to that assembly; its attestation does not establish how the downloaded archive was compiled. Preserve the original producer evidence and identify the selected dependencies in the package manifest. A source revision in a lock is a recorded claim until supported by the producer's evidence. Attestations do not by themselves prove reproducible builds, complete test coverage, or absence of defects.

Publish each dependency release under a unique, immutable identity with the exact validated archives, lock data, licenses and available provenance. Do not replace assets or move tags. Where the hosting service supports enforced release immutability, enable it and verify the resulting state; a naming convention alone does not enforce immutability. Hash checks remain necessary even with immutable hosting. On GitHub, [stage an immutable release as a draft](#), attach all assets, verify the complete inventory and tag commit, then publish. An interrupted draft may resume after existing assets match the recorded candidate. A complete published release needs only verification and any remaining follow-up; an incomplete published immutable release cannot accept missing assets and needs a new release identity. Changed candidate bytes also need a new identity. Preserve the original evidence and revalidate starters whenever their final URLs change.

8.8.1. Publish a pull-request candidate without granting it release authority

A repository may need a linker-input release before the source change that requires it can merge. An unprivileged workflow on the same-repository pull-request branch builds, tests, and attests the exact candidate. A trusted workflow dispatched from the default branch then verifies the producer run, source SHA and ref, workflow identity, attestations, manifest, and content hashes before publishing those bytes. It never executes scripts supplied by the candidate artifact.

The trusted workflow may add the generated content lock to the producer branch as one lease-guarded, GitHub-signed commit. That commit changes only the configured lock path and remains subject to ordinary review and checks. After merge, the lock permanently selects the branch-built release; do not rebuild or relabel it on the default branch. A content-derived tag permits recovery only when the complete candidate is byte-identical.

`actions/publish-build-inputs` implements this boundary for producers that emit `build-input-release.json` and its declared target archives. The consumer wrapper exposes only a pull-request number and pins the controller action to a reviewed full SHA. Fork changes must first move to a same-repository branch because the controller never writes to forks.

The complete producer, publisher, generated-lock, recovery, review, and merge contract is in [Publish linker inputs from a pull request](#).

8.9. Choose the correct maintainer path

Use the smallest lifecycle that corresponds to the changed ownership boundary:

Change	Required maintainer action
Roc source, examples, tests, docs, or an unrelated workflow	Ordinary pull-request validation using the committed host and linker-input locks. No native dependency release.
A new nightly breaks the released platform	<code>nightly.yml</code> reports it. Fix the source and release a new platform version. Native inputs are rebuilt only if the compiler contributes to their fingerprint.
A new nightly changes the generated host bindings	Upgrade the bindings , with any host change, in one pull request.
Host implementation, host dependency, host toolchain, host ABI, or exported project symbol	The host job rebuilds, because its fingerprint changed. The new host ships in the next platform release. Reuse the existing linker inputs unless their own inputs changed.

Change	Required maintainer action
Linker-input source, upstream binary, interface catalog, resource source, generation recipe, target matrix, or toolchain	Build the complete linker-input candidate on a same-repository pull-request branch, then run the trusted publisher and review its signed lock commit.
<code>Blueprint.lock</code> , where it is part of a build-input fingerprint	The same producer and publisher cycle, in the pull request that changes the lock.
<code>platform/main.roc</code> linker entries	Check classification first. A new project implementation symbol normally changes the host; a new external library or interface normally changes linker inputs. Validate that every referenced released path exists for every advertised target.
Platform release	Require a current linker-input lock, build or restore the host, assemble once, exercise applications on every target, and publish the exact tested bundles.

For a stale linker-input lock, stage the material change in a same-repository pull request first. An unprivileged producer run at that head can build, test and attest a candidate without publishing. Then dispatch the trusted publisher **from the default branch** with the pull-request number. It reruns the producer at the exact head, verifies the candidate, publishes the immutable linker-input release, and adds a signed lock-only commit to the pull request. Revalidate the updated pull request against the released lock before merging. A platform bundle is a separate, later release that uses the selected lock. [Publish linker inputs from a pull request](#) defines the admission and review checks.

An input fingerprint mismatch must be explicit. For a host, the read-only host job simply builds it, because host source commonly changes with the pull request. For a locked linker-input release, routine validation fails with instructions to run the producer and publisher cycle. It must not bootstrap a replacement as a side effect of a cache miss. An implicit rebuild would substitute unreviewed bytes for the dependency the lock selects, make results depend on whichever runner missed its cache, and hide that the declared dependency is stale.

8.10. Keep routine pull requests ignorant of production

Routine consumers need only the lock schema, a downloader/cache, digest and archive validation, and the selected extracted files. They do not need a native SDK, linker-input generator, publication token, attestation lookup, or knowledge of how the archives were produced. Keep producer jobs in their own workflow and path scope; do not call them from general CI merely to

populate a cache.

Every cache read is untrusted until the archived bytes match the locked size and SHA-256. A cache miss downloads the exact locked release asset. A download failure fails the job. A digest mismatch removes or ignores the bad entry and fails closed; it never selects another release. Extraction and manifest checks happen after the archive hash succeeds and before files become visible to the build.

Derive cache keys from the locked content identity rather than a branch, workflow run, or mutable release label. This prevents unrelated revisions from sharing a slot merely because their human-readable names match and makes identical reviewed bytes reusable across PRs. Still rehash a cache hit: cache keys select storage but do not authenticate its contents, and a poisoned, truncated, or incorrectly restored entry can otherwise bypass the download-time check.

8.11. Preserve reviewed history and authority

Use merge commits for pull requests whose evidence names exact commits. A merge commit keeps the attested producer commit and the publisher's signed lock-only commit reachable under their original identities. Squashing manufactures a different commit and discards that correspondence. A maintainer may still tidy an ordinary development branch before review; the automation must not depend on history being rewritten after it has bound evidence to a SHA.

Keep the authorities distinct:

- producer and ordinary validation jobs may execute candidate code and are read-only;
- the trusted publisher job invokes only its full-SHA-pinned shared action, with release and lock-write permission;
- the platform release publisher consumes already selected dependencies and publishes only explicitly dispatched, tested platform bundles.

This split is what makes running candidate build code acceptable: a compromise of that job cannot write a release or a branch. The publisher has powerful credentials, so it accepts only inert declared files and never executes the pull request. Least-privilege permissions reduce what is possible if the pinned action or the hosting account is compromised; they do not make unreviewed code safe to run in that job.

A shared action is tried from a reviewed roc-automation pull request by pinning the caller to that pull request's full commit SHA. A branch name or abbreviated SHA is never pinned. If review changes the shared implementation, pin the new commit and repeat the trial.

8.12. Keep rebuild selection narrow and complete

Producers computing reuse identities must include every input that can affect an output: relevant source paths and contents, generated inputs, dependency locks, build recipes, tool versions, target configuration and build flags. Record file paths and modes as needed to distinguish renames, deletions and symlinks. Validate the permitted source-file kinds. Include relevant environment and SDK identities; an unpinned runner image or ambient library can change output without a source edit. Where the toolchain comes from Blueprint, `Blueprint.lock` is one of those inputs, so a lock update that changes the toolchain makes the selected outputs stale. Use a versioned fingerprint recipe and compare the same input definitions. A whole-commit identity, together with the non-source inputs above, is a conservative starting point, but causes needless rebuilds for unrelated edits. Narrow the source selection only after testing the input inventory. Include the source revision separately for traceability.

Treat workflow triggers and fingerprints as different controls. Path filters decide when to run a producer audit; fingerprints decide whether the selected published outputs still correspond to current inputs. A test-harness edit may justify rerunning an audit without requiring a new dependency release. Trace shared scripts and generated inputs transitively so a narrow filter cannot hide a real build change. For repository-owned inputs, record the published fingerprint in lock data or a digest-bound manifest so a shallow checkout can compare current inputs without fetching an old producer commit. A stale host blocks reuse; PR validation may explicitly build a new candidate in a read-only job without publishing it.

For a narrowed fingerprint, require meaningful checks for both directions: a relevant source, tool pin or build-option change invalidates reuse, while an unrelated example or documentation change preserves it. Check archive digest rejection and target/path confinement. Exercise final linking against reused artifacts; a selector test alone does not

8.13. Keep final release preparation small

Once the linker inputs are locked and the host job has built or restored the host, the rest of the release needs only the latest nightly to bundle and link applications, the artifact verification step, and the runtime dependencies of its tests. It downloads and hash-verifies the locked linker inputs. SDK generators belong in the linker-input producer; a stale lock sends the maintainer back to that cycle.

Bundle once, then pass those exact archives to the target runners. Each runner serves the applicable bundles on a loopback address and runs the archived examples against them, as in [Build and test the exact archive](#). Validate every advertised target before publication. Check package size limits and target contents early, so that publication does not discover that a bundle cannot carry its advertised hosts.

Keep preparation, validation and publication connected by recorded artifact digests and the exact source SHA. Record archive digests, selected dependency identities, the source revision and the compiler versions in the release notes. The remaining steps are those of every release; see [The order of a release](#).

Chapter 9. Publish linker inputs from a pull request

This chapter specifies the operational contract for `actions/publish-build-inputs`. Read [Native build inputs](#) first for the ownership, rebuild, and consumer model. The [trust model](#) is normative for the publisher's authority boundaries.

A platform repository can need new linker-input bytes before the change that uses them is mergeable. The producer therefore runs on a same-repository pull-request branch, while publication remains controlled by a workflow dispatched from the trusted default branch. The release belongs to that exact PR commit. It is not a temporary preview and is not rebuilt or promoted after merge.

This resolves a deliberate ordering problem: the material change and its build recipe must be reviewed together, but routine validation cannot consume the new bytes until they have an immutable release and lock. Building on the PR preserves that exact source identity. Dispatching publication from the default branch keeps release credentials and controller selection under already-reviewed policy. A single PR job doing both would allow changed producer code to publish or write its own asserted digest.

9.1. Producer contract

The unprivileged producer workflow builds and tests every supported target from the PR branch. It uploads one flat Actions artifact containing top-level regular files only:

- `build-input-release.json`; and
- one deterministic archive for every target declared by the manifest.

The manifest is canonical JSON with this envelope:

```
{
  "schema_version": 1,
  "kind": "link-inputs",
  "source": {
    "repository": "OWNER/PLATFORM",
    "sha": "0123456789abcdef0123456789abcdef01234567",
    "ref": "refs/heads/change-link-inputs",
```

```

    "workflow": "OWNER/PLATFORM/.github/workflows/link-inputs.yml",
    "input_fingerprint": "<versioned producer-input identity>"
  },
  "assets": {
    "x64glibc": {
      "asset": "link-inputs-x64glibc.tar",
      "sha256": "<lowercase SHA-256>",
      "size": 1234
    }
  }
}

```

Use the platform's complete supported target set rather than publishing a partial replacement under the same contract. The input fingerprint covers every source, recipe, tool pin, generated input, target option, and other value that can affect the archives. It determines whether the selected release is current for the working tree; each asset SHA-256 identifies the exact bytes.

The producer must have no release, repository-write, or pull-request-write permission. Generate GitHub build-provenance attestations for the manifest and each target archive. Producer jobs may compile and execute PR code because they do not possess publication authority.

9.2. Trusted publisher contract

The platform owns a small `workflow_dispatch` publisher workflow on its default branch. That workflow uses the `actions/publish-build-inputs` composite action from `roc-automation` at one reviewed full commit SHA and supplies only:

- the same-repository pull-request number;
- the producer workflow filename;
- the candidate Actions artifact name;
- the repository-relative lock path; and
- the release tag prefix.

Give that job `actions: write`, `attestations: read`, `contents: write`, and `pull-requests: read`; set all other permissions to none. A typical caller is:

```

name: Publish linker inputs

on:
  workflow_dispatch:
    inputs:
      pull-request:
        description: Same-repository material-change PR number
        required: true
        type: string

```

```
permissions: {}

jobs:
  publish:
    runs-on: ubuntu-24.04
    permissions:
      actions: write
      attestations: read
      contents: write
      pull-requests: read
    steps:
      - uses: OWNER/roc-automation/actions/publish-build-inputs@REVIEWED_FULL_SHA
        with:
          pull-request: ${ inputs.pull-request }
          producer-workflow: link-inputs.yml
          candidate-artifact: link-input-release
          lock-path: link-inputs.lock.json
          release-prefix: linker-inputs
```

The composite action obtains its controller implementation from that exact `roc-automation` revision. Its privileged job does not check out the platform PR, execute platform scripts, import candidate modules, or source files from the candidate artifact.

The controller requires a manual dispatch on the live default-branch commit. It resolves the open PR, dispatches the named producer workflow at the exact current PR head, captures the resulting run, and rejects a run whose repository, event, workflow, ref, SHA, or conclusion differs. It validates the artifact's complete file set, bounded extraction, manifest schema, target set, byte sizes, and hashes. It then verifies every attestation against the expected repository, producer workflow, source ref, and source SHA.

The release tag is the configured prefix followed by `-sha256-` and the SHA-256 of the exact manifest bytes. The controller creates a draft, uploads the manifest, generated lock, and declared target archives, verifies their complete names and sizes, re-downloads every release asset into a clean directory, verifies its SHA-256 against the candidate, then publishes. Existing mismatched tags or releases are rejected; an interrupted draft can be resumed only when every published byte matches the candidate. Enable immutable releases in the platform repository.

`actions/publish-build-inputs` is the single supported linker-input publication contract. Keeping one path matters because archive admission, draft recovery, attestation verification, immutable publication, and lock adoption are one trust boundary; parallel publishers would duplicate security-sensitive policy and could silently diverge.

Finally, the controller creates one lease-guarded, GitHub-signed commit on the PR branch. It modifies only the configured lock path and verifies the resulting signature, file set, parent, and live PR head. Any competing branch update stops publication adoption rather than being overwritten.

The signature establishes the commit creator and integrity; the lock-only shape limits what that privileged identity can change. Neither protection substitutes for the other. The lease prevents

the publisher from attaching a valid lock to a newer, unverified PR head or overwriting maintainer work that arrived during the release operation.

9.3. Recover interrupted publication

The optional `producer-run` action input selects an existing successful producer run instead of dispatching another build. The run must be a completed successful manual dispatch of the configured workflow, from the same repository, branch, and exact current PR head. Recovery repeats candidate inventory, content, and attestation verification; it does not reuse a prior admission decision.

Expose `producer-run` as an optional string in the trusted default-branch caller and pass it through to the action. Leave it empty for a new build. A moved PR head requires a new matching producer run.

Draft recovery searches all release-list pages. New drafts use the release ID returned by creation, without requiring immediate visibility in tag or release list projections. Asset downloads use their numeric IDs, so a draft needs no published Git tag. Every recovered asset is downloaded and hashed before the same release ID is published and checked for immutability. Unexpected or partial asset sets fail without replacement or deletion.

9.4. Review and merge the adoption

The generated lock records the manifest identity, source identity, input fingerprint, release, repository, and every target archive's filename, SHA-256, and size. Publication and lock creation are one admission transaction, but the lock is not adopted until the updated PR passes its required checks and is merged.

Review the PR as a dependency change:

1. Confirm that the source changes actually require or intentionally regenerate linker inputs and that the producer-input inventory is complete.
2. Inspect the producer and publisher run links, exact source SHA, attestations, target coverage, and consumer-facing link tests.
3. Confirm that the publisher added only the lock and that its commit is verified.
4. Rerun routine validation on the new PR head. It must consume the release through the lock and must not rebuild linker inputs.
5. Merge with a merge commit so the exact attested producer commit and signed lock commit remain in history.

Do not rebuild, copy, retag, or promote the release after merge. The merged lock permanently selects the PR-built bytes. If source or build inputs change again, produce a new candidate and content-derived release. Byte-identical recovery may reuse the same content identity; changed bytes require a new identity.

Keeping the PR-built release is what preserves the chain from source SHA through producer tests and attestations to the reviewed lock. Rebuilding on the default branch would create different bytes with different evidence even if the source appeared equivalent, while promotion would add ceremony without strengthening a content-derived identity. Permanent releases consume storage, but avoid ambiguity about which candidate was actually admitted and support later audit or rollback.

Fork pull requests cannot receive the signed lock commit. Move an accepted change to a same-repository branch before publication. Do not grant a fork or PR workflow write credentials as a workaround.

9.5. Routine consumption

Ordinary pull requests, nightly compatibility runs, local builds, and platform release assembly read the committed lock. They restore or download each selected archive, recompute size and SHA-256 even on a cache hit, validate the archive and manifest, and install it through fresh staging. They do not call the attestation service and do not fall back to a source build when the cache is empty.

A cache key is an optimization derived from the reviewed content identity. The lock, not the cache key or release name, authorizes bytes. A missing archive is a download/cache failure; a stale input fingerprint is a request for an intentional new linker-input release. See [Separate admission from routine use](#) for the distinction between admission evidence and routine integrity checks.

Chapter 10. Manual maintenance tasks

The package compiler moves by itself, every day. Nothing else does. The tasks in this chapter are the only way the repository's tooling changes, and each is a pull request a maintainer opens and reviews.

No workflow performs them on a schedule. A pull request opened by a workflow with the repository's `GITHUB_TOKEN` does not start CI, so an automated update would arrive untested and wait for a maintainer anyway. More to the point, each of these changes decides which code runs in CI, which is a decision to review and not a chore to automate.

Task	Files changed	When
Update the lock	<code>Blueprint.lock</code>	To take newer tool versions, or to make a newer nightly selectable as <code>roc-stable</code>
Change the tooling compiler	<code>Blueprint.roc</code> , the <code>roc</code> field of every script, sometimes <code>Blueprint.lock</code>	When scripts need a newer compiler, or periodically to stay close to current
Upgrade Blueprint	<code>Blueprint.roc</code> header, the <code>blueprint</code> version and checksums	For a new <code>roc-blueprint</code> release
Update script dependencies	Script headers, <code>RocPackages</code> in <code>Blueprint.roc</code> , <code>Blueprint.lock</code>	For a new release of a platform or package the scripts use
Regenerate host bindings	A platform's committed bindings	When <code>ci.yml</code> reports that the latest nightly generates different bindings

10.1. Update the lock

1. Run `blueprint update`, or `blueprint update <input>` to move one input and keep every other pin.
2. Run `blueprint run check`.

3. Open a pull request whose only changed file is `Blueprint.lock`.
4. Merge after `ci.yml` passes.

A lock update moves the overlay revision, the package source and tool versions. It never changes which release is `roc-stable`, because that tag is written in `Blueprint.roc`. It does not change the `blueprint` binary or the platform URL.

Where the repository fingerprints native build inputs, a lock change can make those inputs stale. `ci.yml` reports that, and the fix is a new linker-input release in the same pull request; see [Choose the correct maintainer path](#).

10.2. Change the tooling compiler

1. Choose a published nightly release.
2. Change the tag in the `Command("roc-stable", ...)` setting of `Blueprint.roc`.
3. Change the `roc` field in every script header to the same tag. The check script fails until they all match.
4. If the locked overlay revision predates that release, run `blueprint update roc` and commit `Blueprint.lock` in the same pull request.
5. Fix any script the new compiler breaks in that pull request, so the tag and the scripts move together.
6. Merge after `ci.yml` passes.

This is a maintainer's decision for each repository. Repositories need not agree.

10.3. Upgrade Blueprint

A release's `blueprint` binary must be used with that release's platform URL. `actions/setup-blueprint` reads the release from that URL, so the URL is the only place to change:

1. Choose a `roc-blueprint` release.
2. Move the workflows' reference to `actions/setup-blueprint` to a revision that records that release, if the current one does not. Dependabot proposes this change; `actions/setup-blueprint` explains where the checksums are reviewed.
3. Change the platform URL in the `Blueprint.roc` header.
4. Run `blueprint update` if the new release refuses the existing lock.
5. Merge after `ci.yml` passes.

No workflow file names the Blueprint version or a checksum.

Contributors then download the binary from the same release.

10.4. Update script dependencies

1. Change the release URL in each script header that uses the dependency.
2. Change the same URL under `RocPackages` in `Blueprint.roc`, and add any new dependencies of that dependency. `roc-stable deps scripts/check_all.roc` lists them.
3. Run `blueprint update` and commit `Blueprint.lock`.
4. Merge after `ci.yml` passes.

10.5. Regenerate host bindings

See [Upgrade the bindings](#).

Chapter 11. Repository settings

Workflow files do not enforce a process by themselves. The settings in this chapter are part of the model, and a repository has not adopted it until they are in place and have been exercised with a real pull request.

11.1. Actions

- Set the default workflow token to read-only. Each job that needs to write states its own permission.
- If the repository restricts which actions may run, allow each shared action at its reviewed full SHA. An allowlist entry without `@...` does not match an action reference.
- Do not enable "Allow GitHub Actions to create and approve pull requests". No workflow in this model opens a pull request.

11.2. Branch rules

Protect the default branch with a ruleset:

- Require a pull request.
- Require status checks to pass on the current commit, with the branch up to date. Name the aggregate job of `ci.yml`, or its one job in a [pure Roc package](#), and the build job of the `release.yml` dry run.
- Require signed commits.
- Grant no bypass to any bot: neither an app nor a deploy key is on the ruleset's bypass list.

Verify the rules with a real pull request: a failing check must block the merge, and a check name that no job produces must be noticed before it is relied on. If a required name identifies both a check run and a commit status, GitHub requires both.

Platform repositories that publish linker inputs allow merge commits. The publisher binds its evidence to exact commit SHAs, and a squash would replace them; see [Preserve reviewed history and authority](#).

11.3. Releases

- Enable immutable releases.
- Create a `release` environment restricted to the default branch, with a required reviewer where the project has more than one maintainer. The publish job of `release.yml` names it. Restrict it by name: choose "Selected branches and tags" and add the default branch alone. "Protected branches only" is wider, because it admits every branch a ruleset later covers.
- Protect tags with a ruleset that targets all tags and restricts updates and deletions, so that an existing version tag cannot be moved or deleted.

11.4. Pages

- Set the Pages source to GitHub Actions.
- Restrict the `github-pages` environment to the default branch, by name as for `release`.

11.5. Dependencies

Keep a weekly `github-actions` entry in Dependabot. Shared actions are pinned by full SHA, and Dependabot proposes each new one as a pull request. Review it like any other dependency update, and never replace a full SHA with a branch or tag.

11.6. Reporting

Enable private vulnerability reporting and keep `SECURITY.md` accurate. [Project-practice evidence](#) lists what else a public project should be able to show.

11.7. Confirm the settings

The `check-settings` task of `roc-automation` reads a repository's settings from GitHub and reports each line of this chapter as passing or failing. Run it from a checkout of `roc-automation` and name the repository:

```
blueprint run check-settings -- <owner>/<name>
```

Three answers only a maintainer can give are flags:

`--platform`

The repository publishes linker inputs. The report notes whether merge commits are allowed.

--reviewer-required

The project has more than one maintainer. The `release` environment must have a required reviewer.

--no-site

The repository publishes no site. The Pages lines are left out.

Each failing line is followed by its fix: the settings page to change, or the `gh api` command that changes it. A setting that GitHub would not show is reported as unknown, never as passing. The task exits with an error while any line fails or is unknown.

- It only reads. Every request it makes is a `GET`. Applying a setting stays a maintainer's decision, made on the settings page or by running the printed command.
- It uses the maintainer's own `gh` login, which must be able to administer the repository. GitHub shows a ruleset's bypass list and the Actions permissions to nobody else. A workflow's token cannot be granted that access, so the task runs on a maintainer's machine and not in CI.
- It shows that the rules exist, not that they are enforced. A real pull request with a failing check is still what shows that the merge is blocked.
- It compares the required check names with the jobs that ran on the pull request behind the latest commit on the default branch. It cannot tell whether those are the right jobs to require.
- It confirms that `SECURITY.md` exists, not that it is accurate.

Chapter 12. Adopt the model in a repository

This chapter is for a maintainer bringing a repository in line with the model. It assumes the earlier chapters. Work from a fresh branch of the remote default branch, and open one pull request per numbered step so that a failure points at one change.

A [pure Roc package](#) follows every step, with one runner in each workflow. The sentences below that begin "A platform" and the acceptance item that begins "For a platform" do not apply to it.

12.1. Steps

1. **Environment.** Add `Blueprint.roc` and `Blueprint.lock` with `roc-stable`, the script dependencies under `RocPackages`, and every tool and toolchain the repository uses. Using the released `blueprint` binary, confirm that `blueprint run` executes a shebang script. Remove any `flake.nix`, `flake.lock`, `.envrc` or other environment definition this replaces, and add `.blueprint/` to `.gitignore`.
2. **Scripts.** Before porting a script, decide whether it earns its place: consolidate scripts that overlap, and delete one that only repeats another test. Port what remains to Roc following [Write automation as Roc scripts](#). Carry each script's tests across first, as `expect` tests or cases the check script runs. Delete the original only when its replacement runs in CI.
3. **Pins.** Remove `.roc-version`, `.github/roc-nightly.json` and every `roc` header field outside `scripts/`. Set the `roc` field of every script to the `roc-stable` tag. Make the check script enforce both. A platform also commits its generated host bindings and adds the check that regenerates and compares them.
4. **Examples.** Point every Roc application in the repository at the working tree by relative path. Record expected output. Remove any step that rewrites committed examples after a release.
5. **Release artifacts.** Add tasks that build the bundle, the examples archive, the documentation archive and the manual, and a task that tests the archive against a loopback bundle.
6. **Workflows.** Replace the repository's workflows with `ci.yml`, `release.yml` and `nightly.yml`, plus any [optional workflows](#) it needs. Call `actions/setup-blueprint` with no inputs, after the checkout; it installs the release that `Blueprint.roc` names.
7. **Repository settings.** Apply [Repository settings](#), update the required check names, and delete branches that earlier automation owned. Confirm the result with the `check-settings` task, as

Confirm the settings describes.

12.2. Acceptance

Record a link for each item in the final pull request.

- `blueprint run check` passes locally, using the released `blueprint` binary and a current nightly `roc` on `PATH`.
- A search finds no `.roc-version`, no `roc: "` header field outside `scripts/`, no committed flake files, no rendered documentation, and no Python, bash or PowerShell automation.
- `ci.yml` passes on the pull request, and its log shows the `blueprint` checksum verified before `blueprint run check`.
- A `release.yml` dry run produces every release asset, and the archived examples differ from the repository's only in the package dependency.
- A manual `nightly.yml` run passes against the latest release.
- A deliberately failing nightly run opens exactly one tracking issue, a second failure comments on it, and a passing run closes it.
- `blueprint run check-settings --owner <owner> /<name>`, run from `roc-automation`, passes every line for the repository.
- A real pull request with a failing check is blocked from merging.
- For a platform: a deliberate edit to the committed bindings fails `ci.yml`.

12.3. Evidence for each claim

A claim about the repository is only as good as the check behind it.

Claim	Evidence
CI runs the reviewed <code>blueprint</code> binary	The setup step verifies the download's SHA-256 against the reviewed value before running it
Tooling uses the pinned compiler	Scripts start with the <code>roc-stable</code> shebang; CI prints <code>roc-stable version</code>
Nothing but the scripts is pinned	The check script fails when a <code>roc</code> header field or <code>.roc-version</code> exists outside <code>scripts/</code>
Every script names the tooling compiler	The check script fails unless each script's <code>roc</code> field equals the <code>roc-stable</code> tag in <code>Blueprint.roc</code>

Claim	Evidence
Repository examples test the working tree	The check script fails unless each application names the relative package path
Examples behave as documented	The check script compares each example's output with the recorded specification
A platform's bindings match the compiler	<code>ci.yml</code> regenerates them with the latest nightly and fails on a difference
Published examples work	<code>release.yml</code> tests the archive before upload; <code>nightly.yml</code> tests the published archive
Published bytes are the tested bytes	The publish job downloads each uploaded asset and compares it with the tested artifact
The release names its compilers	Release notes contain both version strings
The repository settings are in place	The <code>check-settings</code> task reads them from GitHub and passes every line
The latest release works on the latest nightly	The most recent <code>nightly.yml</code> run, and the absence of an open tracking issue

Chapter 13. Shared actions

roc-automation provides the pieces every repository needs identically. Reference each one by full commit SHA, and upgrade that reference through a pull request. A shared change never reaches a repository silently.

```
- uses: lukewilliamsboswell/roc-automation/actions/<name>@REVIEWED_FULL_SHA
```

13.1. actions/setup-blueprint

Installs Nix and the `blueprint` binary that `Blueprint.roc` names, so that `blueprint run <task>` works in the next step. It builds nothing.

```
- uses: actions/checkout@REVIEWED_FULL_SHA
  with:
    persist-credentials: false
- uses: lukewilliamsboswell/roc-automation/actions/setup-blueprint@REVIEWED_FULL_SHA
- run: blueprint run check
```

The call takes no inputs. The action reads the roc-blueprint release from the platform URL in the checked-out `Blueprint.roc`, and it records the reviewed SHA-256 of every binary of each release it installs. `Blueprint.roc` is therefore the one place a repository names its Blueprint release, and a workflow cannot disagree with it.

Input	Meaning
<code>version</code>	The roc-blueprint release to install. Defaults to the release named by the platform URL in <code>Blueprint.roc</code> at the root of the workspace. Set it only in a job that has no checkout.
<code>sha256-x86_64-linux</code>	Reviewed SHA-256 of the binary for x86_64 Linux runners. Overrides the recorded value.
<code>sha256-aarch64-linux</code>	Reviewed SHA-256 of the binary for arm64 Linux runners. Overrides the recorded value.

Input	Meaning
<code>sha256-aarch64-darwin</code>	Reviewed SHA-256 of the binary for Apple Silicon macOS runners. Overrides the recorded value.

Outputs are `blueprint-path`, the absolute path of the installed binary, `system`, the release system name chosen for the runner, and `version`, the release that was installed.

The recorded checksums live in `actions/setup-blueprint/releases.json`, one entry for each roc-blueprint release, copied from that release's `blueprint.sha256` file. A new roc-blueprint release is reviewed once, in the pull request that adds its entry there, and reaches a repository when that repository moves its reference to the action. The reference is a full commit SHA, so the recorded checksums are pinned by the same review as the action's code.

A release the referenced revision does not record fails the step, unless the caller supplies a `sha256-<system>` input for the runner. Use those inputs to install a release ahead of the action, such as a release candidate. A system for which no binary is released also fails.

The action downloads the binary for the runner's system into a private temporary file, compares its SHA-256 with the reviewed value, and only then makes it executable and puts it on `PATH`. A mismatch fails the step and installs nothing. It takes no token and needs no cache account. [The blueprint `binary`](#) states what the checksum does and does not establish.

13.2. Install the nightly compiler

The package compiler is installed by the official `roc-lang/setup-roc` action, not by an action in this repository. Reference it by full commit SHA like any other.

```
- uses: roc-lang/setup-roc@REVIEWED_FULL_SHA
  with:
    version: nightly-new-compiler
    nightly-tag: ${needs.resolve.outputs.nightly-tag }
```

It installs the compiler on `PATH` as `roc`. With `nightly-tag` empty it installs the latest release of `roc-lang/nightlies`; with a tag it installs that release.

A workflow with more than one job resolves the latest tag once, in its first job, and passes it to every job through `nightly-tag`, so that one run uses one compiler:

```
- id: nightly
  env:
    GH_TOKEN: ${github.token }
  run: echo "tag=$(gh api repos/roc-lang/nightlies/releases/latest --jq .tag_name)" >>
    "$GITHUB_OUTPUT"
```

13.3. Actions that run a Roc script

`actions/nightly-report` and `actions/publish-release` are Roc scripts in `roc-automation`, `scripts/nightly_report.roc` and `scripts/publish_release.roc`, written as [Write automation as Roc scripts](#) describes. Each runs from source in the calling job, from the action's own copy of `roc-automation` at the referenced commit. The action:

1. installs Nix, with the install action `actions/setup-blueprint` uses, at the same full commit SHA;
2. downloads the `blueprint` binary that `roc-automation`'s `Blueprint.roc` names and checks it against its recorded SHA-256, as [The blueprint binary](#) describes; and
3. runs its task, `blueprint run nightly-report` or `blueprint run publish-release`, in that copy of `roc-automation`.

`roc-automation`'s `Blueprint.lock` at the referenced commit pins what the task starts: `roc-stable`, which runs the script, the platform the script is built on, the `gh` CLI, which makes every request to GitHub, and `coreutils`. The full SHA in the `uses:` line therefore fixes the script and everything under it. Nothing comes from the calling repository: its `Blueprint.roc` is not read, and the binary is installed in a directory of the action's own and is not put on `PATH`, so a `blueprint` the job set up earlier is left as it was.

The script is given the action's inputs and the token in its environment, never as arguments. `gh` reads the token from `GH_TOKEN` and sends requests to the GitHub host it is configured for, which is `GitHub.com` unless the job sets `GH_HOST`.

These actions need a runner Blueprint supports, which is x86-64 or arm64 Linux or Apple Silicon macOS, and `python3` for the installer, which GitHub's hosted runners provide. On a runner with an empty Nix store, installing Nix and downloading the environment is most of the job's time: the environment is about a gigabyte once unpacked, so allow a minute or two. The script itself finishes in seconds.

13.4. `actions/nightly-report`

Keeps the one tracking issue of a [nightly compatibility](#) workflow. It checks out nothing from the calling repository and runs none of its code.

```
report:
  needs: test
  if: ${{ always() }}
  runs-on: ubuntu-latest
  permissions:
    issues: write
  steps:
    - uses: lukewilliamboswell/roc-automation/actions/nightly-report@REVIEWED_FULL_SHA
      with:
        outcome: ${{ needs.test.outputs.outcome }}
```

```
nightly-tag: ${{ needs.test.outputs.nightly }}
release-tag: ${{ needs.test.outputs.release }}
```

Input	Default	Meaning
outcome	Required	Outcome of the step that ran the examples
nightly-tag	Required	The Roc nightly the examples ran on, named in the issue text
release-tag	Required	The release whose examples ran, named in the issue text
title	The latest release fails on the latest Roc nightly	Exact title of the tracking issue
token	\${{ github.token }}	Token used to read and write issues

Outputs are `action`, which is `opened`, `commented`, `closed` or `none`, and `issue-number`, the issue that action concerned. `issue-number` is empty when `action` is `none`.

The tracking issue is the open issue whose title equals `title` exactly and whose author is a bot account. The action never comments on, closes or reopens any other issue. An issue with the same title that a person opened is ignored, as are a pull request and a closed issue.

outcome	What the action does
success	Comments on the tracking issue and closes it. With no tracking issue open it does nothing, and <code>action</code> is <code>none</code> .
failure	Comments on the tracking issue. With no tracking issue open it opens one.
Anything else	Fails the step with an error and reads or writes no issue. An empty value, <code>skipped</code> and <code>cancelled</code> all mean that the examples did not run to a verdict, which is a setup failure and not evidence about the release.

The issue body and every comment name the release and the nightly and link to the workflow run. Tags are limited to 100 letters, digits and `.`, `_`, `+`, `/`, `-` characters, and a title to 256 characters with no control, format or private-use characters. A malformed input, or an empty token, fails the step before any request is made.

The action reads the open issues a page at a time. A listing it cannot read in full, or a record in it that is not an issue record, fails the step: it never opens an issue while it cannot tell whether one exists.

The calling job needs `issues: write` and no other permission. It has no checkout step, and it passes the action only the step outcome and the two tags, never text produced by the examples.

[Reporting](#) states what the calling workflow must provide, and [Actions that run a Roc script](#) what the action needs of the runner.

13.5. actions/build-docs

Renders a repository's `docs/*.adoc` sources as an HTML site and a PDF manual with a shared theme, Roc syntax highlighting and Mermaid diagrams. It can also place compiler-generated API documentation under `site/api`.

```
- id: docs
  uses: lukewilliamboswell/roc-automation/actions/build-docs@REVIEWED_FULL_SHA
  with:
    pdf-filename: my-project.pdf
    docs-version: main
    api-entrpoint: platform/main.roc
- uses: actions/upload-pages-artifact@REVIEWED_FULL_SHA
  with:
    path: ${ steps.docs.outputs.site-directory }
```

Input	Default	Meaning
<code>docs-directory</code>	<code>docs</code>	Source directory, relative to the repository root
<code>entrpoint</code>	<code>index.adoc</code>	The manual's entrpoint, relative to <code>docs-directory</code>
<code>output-directory</code>	<code>.docs-out</code>	Generated output, relative to the repository root
<code>pdf-filename</code>	<code>manual.pdf</code>	Filename of the generated PDF
<code>docs-version</code>	<code>unreleased</code>	Label passed to the sources as the <code>docs-version</code> attribute
<code>api-entrpoint</code>	<code>empty</code>	Roc package or platform root for the generated API reference
<code>roc-command</code>	<code>roc</code>	Roc executable used when <code>api-entrpoint</code> is set
<code>theme-directory</code>	<code>empty</code>	A replacement theme directory, relative to the repository root

Outputs are `site-directory` and `pdf-path`, both absolute paths.

The action needs Docker and Python on the runner, which GitHub’s Ubuntu runners provide. A replacement theme directory must contain `docs.css`, `docs-theme.yml`, `mermaid-config.json` and `fonts/`. [Write for the shared renderer](#) lists what the renderer expects of the sources.

13.6. actions/publish-release

Publishes one release from files an earlier job built and tested. It stages them in a draft, reads every asset back and compares it with the tested file, and only then publishes. It checks out nothing from the calling repository and runs none of its code.

```
publish:
  needs: [build, test]
  if: github.event_name == 'workflow_dispatch'
  runs-on: ubuntu-24.04
  environment: release
  permissions:
    contents: write
  steps:
    - uses: actions/download-artifact@REVIEWED_FULL_SHA
      with:
        name: release-candidate
        path: candidate
    - uses: lukewilliamboswell/roc-automation/actions/publish-release@REVIEWED_FULL_SHA
      with:
        version: ${ needs.build.outputs.release-version }
        prerelease: ${ needs.build.outputs.is-prerelease }
        notes-file: candidate/release-notes.md
        assets: |
          candidate/dist/*
          candidate/docs/*.pdf
```

The job names the `release` environment, holds `contents: write` and nothing else, and has no checkout step. Its only inputs are the artifact the build job uploaded and the test jobs verified. What the assets are, how they are built and tested, and any attestation of them stay in the repository’s earlier jobs.

Input	Default	Meaning
<code>version</code>	required	The release version, such as <code>0.12.0</code> or <code>0.12.0-rc1</code>
<code>prerelease</code>	required	Exactly <code>true</code> or <code>false</code> . It must agree with whether version has a prerelease suffix.
<code>notes-file</code>	required	Path of the rendered release notes, which become the body of the release

Input	Default	Meaning
<code>assets</code>	<code>required</code>	Globs of the tested files, one per line, relative to the workspace
<code>tag</code>	<code>version</code>	The tag to create: the version, or <code><name>-<version></code> in a repository that releases more than one thing
<code>target</code>	<code>\${{ github.sha }}</code>	Full SHA of the tested commit the tag points at
<code>token</code>	<code>\${{ github.token }}</code>	A token with contents: <code>write</code> on the repository

Outputs are `release-url`, the URL of the published release, and `digests`, the path of a listing of the published assets in the format `sha256sum` writes and `sha256sum --check` reads.

The tag is the bare version, without a `v`. A repository that publishes a second, independently versioned package gives that package's releases a name with the `tag` input, such as `kit-0.1.0`. The tag must end with the version, so the two cannot disagree.

The action does the following, in this order, and stops at the first failure.

1. **Check the inputs.** The version, tag, target and prerelease value must be well formed and agree with each other. The notes file must be a non-empty regular file of UTF-8 text.
2. **Collect the assets.** Every glob must match at least one file. Every match must be a regular, non-empty file and not a symbolic link, and no two may share a filename. Filenames are limited to letters, digits, `.`, `_` and `-`, because GitHub rewrites other characters and the published name would then differ from the tested one. The SHA-256 of each file is recorded now. The token and the repository name are checked last, so every check that needs no request has passed before the first request is made.
3. **Require a new version.** The target must be a commit of the repository. The tag must not exist, and no release for it may exist, published or draft.
4. **Create a draft** at the target commit, with the notes file as its body.
5. **Upload** every asset to the draft.
6. **Verify the draft.** It must hold exactly the collected filenames. Each asset is downloaded into a new, empty directory and must equal the tested file in size and SHA-256. A download is cut off one byte past the tested size.
7. **Publish.** A prerelease version is published as a prerelease and is not made the latest release. Any other version is published as the latest release.
8. **Confirm.** The published release must carry the requested tag and be immutable. The digest listing is written to the job summary.

Nothing is public before the publish step. With immutable releases enabled, publishing a draft is the moment its tag and assets become permanent, which is why every check comes before it. A failure after the draft exists leaves the draft unpublished and the tag uncreated; the error names the draft. [Recover an interrupted publication](#) says what to do with it.

The action does not adopt an existing draft, replace an asset, or add to a published release. Once it has created the draft, every request names that draft by its numeric identifier, never by tag. It publishes the notes file as written and does not ask GitHub to generate notes: release notes are authored, as [Write the release notes](#) describes.

A glob in `assets` is a path relative to the workspace. Within one path segment `*` matches any run of characters and `?` any one character, and neither matches the dot at the start of a name. `.` segments are ignored. Nothing else is a pattern: `*`, `[`, a backslash, a `..` segment and an absolute path are each an error, so a glob names only paths under the workspace.

Every request is a `gh api` call. `gh` adds the token to requests for the GitHub host and does not repeat it when GitHub redirects an asset download to the storage host that serves the bytes. A repository without immutable releases fails the last step, after the release is published; [Releases](#) lists the setting. [Actions that run a Roc script](#) states what the action needs of the runner.

13.7. actions/publish-build-inputs

Publishes native linker inputs built on a same-repository pull request as an immutable, content-addressed release, and commits only the resulting lock to that pull request. The job that holds release credentials never checks out or runs the pull request's code.

Input	Meaning
<code>pull-request</code>	Number of the same-repository pull request that holds the producer source
<code>producer-workflow</code>	Filename of the producer workflow to dispatch on that branch
<code>producer-run</code>	Optional: an existing successful producer run for the current head, used for recovery
<code>candidate-artifact</code>	Name of the artifact the producer uploads
<code>lock-path</code>	The only repository path the action may commit
<code>release-prefix</code>	Prefix of the content-derived release tag

[Publish linker inputs from a pull request](#) specifies the producer, the caller workflow and its permissions, recovery, and review.

Chapter 14. Verify released artifacts

Use an immutable content hash as the identity of every released compiler, package, platform, native library, tool, or other binary input consumed by builds and tests. Record that hash in reviewed source, a lock file, or another version-controlled manifest together with the artifact URL and release identity.

14.1. Separate admission from routine use

An attestation answers who or which workflow produced an artifact and from which source context. A content hash answers whether the bytes being consumed are the exact bytes that were reviewed. These checks belong at different stages:

1. **Publish:** produce the artifact, checksum, provenance attestation, SBOM, and immutable release metadata.
2. **Adopt:** review the release, verify its attestation and source identity, test the artifact, and commit its content hash and URL in a reviewed change.
3. **Consume:** download or load the artifact, calculate its hash locally, and compare it with the reviewed pin before extraction or execution.
4. **Reuse:** recheck cached bytes against the same pin. A cache is storage, not a source of trust.
5. **Update:** repeat admission only when intentionally adopting different bytes.

The reviewed hash is the trust anchor for ordinary builds after adoption. Attestation does not replace content verification, and repeating the same remote attestation lookup does not strengthen the identity of already pinned bytes. Admission asks whether the producer identity and source are acceptable; routine use asks whether storage returned those admitted bytes. Keeping those questions separate avoids turning every build into an online provenance-policy decision while still detecting corrupted or substituted downloads and cache entries.

14.2. Routine CI policy

Routine CI and local builds must not call a remote signing, transparency, or attestation service when a reviewed content hash is available. This avoids making builds depend on service availability, rate limits, authentication, or a network response for evidence already evaluated during adoption.

A routine consumer should:

- reject a missing or malformed content hash;
- verify a newly downloaded file before moving it into a shared cache;
- verify a cached file again before using it;
- validate the archive inventory, internal manifests, target metadata, and release identity where those exist;
- extract into a temporary directory before replacing working inputs; and
- record the URL, release, and content hash in generated provenance without implying that a runtime attestation check occurred.

If the digest does not match, fail without extracting, executing, or installing the artifact. Do not fall back to a floating URL, a newly discovered release, or a remote attestation lookup.

14.3. Appropriate attestation use

Use an attestation service during release publication, explicit lock adoption, security investigation, periodic provenance audit, or manual verification by a downstream user. Export attestation bundles with releases when practical so that provenance can be checked offline and retained independently of the service.

If policy requires provenance verification on every run despite a reviewed content hash, make that exceptional network dependency explicit and separate it from the ordinary validation path. Document its availability and credential requirements and do not describe it as content-integrity verification.

14.4. Review checklist

Before adopting a released artifact, record evidence that:

- the release and tag are immutable and identify the intended source revision;
- the producing workflow and runner policy are acceptable;

- the attestation subject digest equals the downloaded artifact digest;
- the artifact passed its consumer-facing tests;
- the reviewed lock contains the exact URL, filename, release, and SHA-256; and
- routine CI verifies that SHA-256 without contacting the attestation service.

After adoption, a pull request that changes any URL or digest is a supply-chain change and receives the same review as a dependency update.

A same-repository pull-request release is admissible when a trusted default-branch publisher binds it to the exact branch SHA and producer workflow, verifies its attestations, and commits only the resulting content lock. Merging that reviewed lock is the adoption event. Routine consumers then use the same local hash checks as for any other release; they do not query provenance again.

Chapter 15. Automation trust model

This chapter states who may do what in the workflows and shared actions this manual describes, and why each boundary is drawn where it is. It is normative for the shared actions in roc-automation: a change to their behavior or permissions must keep these boundaries and arrive with regression tests for them.

Two rules carry most of the weight:

- A job that runs repository or candidate code has no write access.
- A job with write access runs only reviewed code at a full commit SHA, and handles candidate output as inert files.

15.1. The three workflows

Workflow and job	Token access	Runs repository code
<code>ci.yml test</code>	<code>contents: read</code>	Yes, including pull-request code
<code>release.yml build and test</code>	<code>contents: read</code>	Yes
<code>release.yml publish</code>	<code>contents: write</code> , behind a protected environment	No: it runs <code>actions/publish-release</code> on artifacts produced by earlier jobs
<code>release.yml site deploy</code>	<code>pages: write</code> , <code>id-token: write</code>	No: it deploys an artifact built by a read-only job
<code>nightly.yml test</code>	<code>contents: read</code>	Yes: default-branch scripts and the published examples archive
<code>nightly.yml report</code>	<code>issues: write</code>	No: it runs <code>actions/nightly-report</code>

The publish job accepts only a manual dispatch on the default branch, uses the event commit throughout, and consumes the artifacts of its own run. It never checks out the repository, so a script changed in the tested commit cannot run with release credentials.

The nightly compatibility workflow has no contents, pull-request, actions or status write access in any job. It cannot commit, merge, dispatch or publish. Its report job runs `actions/nightly-report` at a

full commit SHA and checks out nothing. It receives only the outcome of the examples step, the nightly tag and the release tag from the test job, never text produced by the examples under test. It edits a single tracking issue, identified by its exact title and a bot author. Every workflow in a repository shares the GitHub Actions bot identity, so that test excludes issues opened by people and does not prove which workflow opened the issue.

No workflow in the model opens a pull request, approves one, or merges one. Every change to the repository is a maintainer's pull request under the repository's own rules.

15.1.1. What the jobs with write access run

The publish job and the report job hold the tokens that can change a repository's releases and issues. Each runs one shared action, `actions/publish-release` or `actions/nightly-report`, and each action is a Roc script that runs from the action's own copy of roc-automation, at the full commit SHA the workflow names. What runs in those two jobs is therefore:

- Nix, installed by an install action that the shared action references by full commit SHA; that action fixes the Nix version and downloads its installer from the Nix project over HTTPS;
- the `blueprint` binary, for the release that roc-automation's own `Blueprint.roc` names, admitted by its reviewed SHA-256 as [The blueprint binary](#) describes;
- the environment that roc-automation's `Blueprint.lock` pins: `roc-stable`, which runs the script, the basic-cli platform the script is built on, the `gh` CLI, which makes every request, and `coreutils`, which provides `sha256sum`; and
- the script and its modules under `scripts/` in roc-automation.

The one full SHA in the workflow's `uses:` line fixes all of these. The install action's SHA, the recorded hashes of the binary, the lock and the script are all files of that commit, and the lock pins each input by revision and content digest. Nix takes the compiler and the platform by the hashes those inputs record, and takes the tools from nixpkgs at the pinned revision, as built by its binary cache, whose signatures it checks.

Neither job checks out the calling repository, and neither action runs any of its code. The calling repository's `Blueprint.roc`, lock and scripts play no part. The report action reads no file from the job's workspace. The publish action reads the notes file and the asset files it is given, which an earlier job built: it sends the notes as text, and it hashes, uploads and compares the assets as bytes, and executes none of them.

The token is in the environment of the one step that runs the script, where `gh` reads it. It is never a command-line argument and is never printed, and the scripts check every input before the first request.

Nix does not hold the token. The install action would otherwise write the job's default token into

Nix's configuration, for Nix to present to GitHub on its own fetches. These two actions clear that setting, and stop before installing anything further if Nix still reports a token. Every input Nix fetches is public and pinned by the lock, so it needs none.

15.2. Compilers

`Blueprint.roc` names the release tag used for repository tooling, and `Blueprint.lock` pins the overlay by revision and content digest. Changing the tooling compiler is therefore a reviewed source change. Nix verifies the compiler archive against the hash the pinned overlay revision records for that tag. This is content integrity under a reviewed hash, as in [Released artifact verification](#); it is not provenance for the upstream compiler build.

Bundles listed under `RocPackages` are locked the same way. A bundle a script names and the environment does not list is downloaded by Roc, which checks it against the content hash in its URL.

The package compiler is the latest published nightly and is not pinned. It is installed by the official `roc-lang/setup-roc` action, referenced by full commit SHA, from a `roc-lang/nightlies` release over HTTPS. That action does not check a nightly download against a digest, so the compiler's integrity rests on GitHub's release hosting and the action's pinned revision. This is accepted because the nightly only ever runs in jobs that hold no write access. A workflow resolves the moving release once and passes the tag to every job. Release notes record the version used.

15.3. The blueprint binary

`actions/setup-blueprint` installs Nix, through an action referenced by full commit SHA, and downloads the prebuilt `blueprint` binary from a `roc-blueprint` release. It builds nothing. Before the binary runs, the action checks it against a reviewed SHA-256 and fails on a mismatch. Routine CI verifies that reviewed hash locally and does not call an attestation service. The hash establishes which bytes run; it is not provenance for the `roc-blueprint` build.

The reviewed hashes are recorded in the action, one set for each `roc-blueprint` release, and are reviewed in the pull request that adds them. A caller references the action by full commit SHA, so the hashes it trusts are fixed by the same reference as the code that checks them. A caller may supply a hash of its own for its runner, which then takes the place of the recorded one.

A release's binary must be used with that release's platform URL in `Blueprint.roc`, so the action takes the release from that URL. The file is candidate content in a pull request, and that is safe: the version it names can only choose among releases whose hashes are already reviewed. A version the action does not record, with no hash from the caller, fails the step.

The action accepts only a release version, from its `version` input or from `Blueprint.roc`, and builds

the download URL itself, under the roc-blueprint releases of this project, so neither can redirect the download. The binary is written to a private temporary file and becomes executable and visible on `PATH` only after its hash matches. A mismatch, a release with no reviewed hash for the runner, or a system with no released binary fails the step and installs nothing.

The binary evaluates `Blueprint.int.roc` with a Roc compiler of its own. It uses a `roc` on `PATH` only when that is exactly the nightly the binary was built with. Otherwise it fetches that nightly through Nix from a roc-overlay revision pinned inside the binary, and Nix verifies the archive against the hash that revision records. The `ROC` environment variable overrides this selection.

The action takes no token as an input. The Nix installer it runs reads the job's default token and configures Nix to present it to GitHub when it fetches from there. A cache that requires `id-token: write` widens a test job's token and must be an explicit caller choice, never a default.

15.4. Documentation deployment

The build job of `docs.yml` runs repository sources through `actions/build-docs` with `contents: read`. The deploy job holds `pages: write` and `id-token: write`, runs only for the default branch, checks out nothing, and deploys the artifact the build job uploaded. A pull request can build the site and cannot deploy it.

15.5. Linker-input release publisher

The pull-request linker-input publisher is a separate controller action for inert, fully declared files produced by a caller's unprivileged build jobs. A privileged consumer workflow invokes the controller at an exact reviewed SHA. Caller source is never checked out, imported, sourced, or executed in its privileged job. Publication requires manual dispatch on the caller's live default-branch commit. The controller validates the complete regular-file set and content identities, verifies the producer's attestations, refuses mismatched existing tags/releases, re-downloads and hashes the complete draft asset inventory, and requires immutable-release enforcement after publication. See [Trusted publisher contract](#) for the caller contract.

15.6. Released artifact verification

Consumers use reviewed content hashes as the runtime identity of released binary inputs. Routine CI verifies downloaded and cached bytes locally before extraction or execution. It must not depend on a remote signing, transparency, or attestation service when a reviewed hash is available. Provenance attestations are verified when a new release or lock is adopted, during an explicit audit, or by a downstream verifier. See [Verify released artifacts](#) for the complete admission and consumption policy.

15.7. The pin-bump controller and the release guard

roc-automation also contains `actions/nightly`, the reusable `update-roc-nightly.yml` and `check-config.yml` workflows, and `actions/check-release`. They are not part of the model this manual describes. This section is the trust reference for that code: read it before changing their behavior or permissions.

Consumers pin both reusable workflows to a reviewed full SHA. Each shared workflow pins the controller action and checkout action to full SHAs. The controller source runs from `github.action_path`; configuration and git operations use the caller's `GITHUB_WORKSPACE`. Consumer Python files are never imported by the controller.

The scheduled/manual caller must run on the default branch. Prepare records that trusted checkout's commit and creates one pin-only commit on `automation/roc-nightly`. It refuses an existing branch tip with other changed files and uses an exact git lease. GitHub's commit API signs the pin commit, and the controller verifies that GitHub reports the signature as verified.

The caller grants a permission ceiling. Actual jobs reduce it:

Job	Token access
prepare	contents: write; pull-requests: write
validate	contents: read; actions: write; statuses: write
report	contents: read; pull-requests: write
configuration check	contents: read

The controller uses `GH_TOKEN` from `github.token`, `DEFAULT_BRANCH`, and GitHub's standard repository/workspace/run environment. Prepare additionally accepts `FORCE` for an explicit retry. Its outputs are `changed`, `sha`, and `nightly`. Validate accepts `CANDIDATE_SHA` and emits JSON `runs` with workflow, ID, URL, and conclusion, plus `passed`, which is true only when every configured run succeeds. Report accepts that SHA, `NIGHTLY_TAG`, `TEST_RESULT`, `TEST_PASSED`, and `VALIDATION_RUNS`.

Validation dispatches workflows on the candidate branch with `nightly_validation: true`. GitHub API version 2026-03-10 returns their exact run IDs. The controller verifies run commit, branch, and event, waits for all results, and refuses stale branch heads. A normally completed failed run becomes a handled, non-passing candidate: it is reported without failing the controller itself. Missing, cancelled, skipped, timed-out, or mismatched evidence fails the controller. None of these states becomes a passing report. Previous success is cleared before a new candidate is prepared. A timeout links to runs for diagnosis.

A workflow dispatch executes project code under that validation workflow's own permissions. Each consumer must keep test jobs read-only and guard every release or deployment job. This is

a required review boundary, not something the controller can infer from a green workflow result. PR review and branch protection remain maintainer responsibilities; this automation grants no bypass or self-approval.

No credentials are persisted by checkout. Authenticated git pushes receive a short-lived token in the child process environment, never command-line arguments. By default the action merges only its validated pin-only PR. It never approves PRs, bypasses repository rules, publishes releases, or deploys sites. The merge policy is described below.

15.7.1. Merge authority

The boolean `auto_merge` in trusted configuration defaults to true. Consumers may set it to false to opt out. The merge job still starts after successful validation and reporting; its merge phase reads that policy at the trusted event SHA and exits without merging. The workflow does not gate the job on a value produced by an earlier job. It performs no consumer checkout: it reads configuration through the API at the original default-branch event SHA and executes only the pinned shared controller. It has contents write, pull-requests read, and actions read permissions. Test jobs never receive this merge token.

Immediately before merging, the controller independently rechecks the PR, signed bot commit, published upstream tag, configured workflow paths and live run results, branch heads, and active pull-request/strict status-check rulesets. It passes the expected head SHA to GitHub's normal merge endpoint. The merge commit preserves the exact verified, GitHub-signed candidate commit in history. No bot approval or protection bypass is used. Repository rules must have no bot bypass; administrators remain responsible for protecting those rules and the trusted workflow/config.

Token scopes cannot restrict contents write to one file or one operation. The pin-only restriction is controller policy, backed by required PRs and checks; it is not a native GitHub file-scoped merge permission. All consumers share the GitHub Actions bot identity, so the identity check alone is not proof that a particular workflow created a commit. The shape, trusted base, signature, upstream release, and independently checked validation are also required.

All controller jobs explicitly restrict execution to schedule/manual events on the default branch (and reject tags). Every non-check controller invocation also validates that event/ref boundary before performing any API or git operation. Prepare, validate, and report use checkout's original event commit under these explicit event/ref guards and verify it equals the event SHA before doing work. The merge job reads that immutable SHA through the API and checks the live default branch before merging. If the branch moves, retry the updater on its current commit.

Opted-in validation publishes pending commit statuses for the active ruleset's required Actions contexts, then publishes success only after the exact dispatched runs and all corresponding jobs have succeeded. Missing, skipped, or failed jobs cannot produce success. This is needed

because GitHub can leave dispatched check runs unattached to an Actions-bot PR. Statuses use the same check names and Actions identity; they do not replace or bypass the required checks. The merge job has no status-write permission and independently rechecks the run and job evidence.

15.7.2. Read-only release policy

`actions/check-release` is separate from the nightly controller. It requires an explicit dispatch and exact event checkout, reads the compiler pin from that checkout, and defaults to a final compiler version for publication. It uses `github.token` to read `roc-lang/roc` release metadata for the exact pin spelling; missing releases, drafts, prereleases and releases without assets are rejected. Compiler compatibility branches must match the compiler line. Optional default-branch permission alone does not permit nightly publication. A separate default-off `allow-nightly-bootstrap` input permits an exact nightly only on the actual default branch; metadata then comes from `roc-lang/nightlies` with the same published release and asset checks. Compatibility branches cannot use this exception. Remove the bootstrap opt-in when the consumer adopts the intended stable-only policy.

This action has no write operations, consumer Python imports, approval, or bypass. The official release metadata is upstream evidence, not verification of compiler binary integrity. Consumers retain compiler installation, actual version checks, tests, artifact identity, tag uniqueness, and publication permissions.

15.7.3. Header compiler roots

Consumers may set `compiler_roots` in trusted `.github/roc-nightly.json` to select exact source-root paths instead of legacy `.roc-version`. This is an edit-authority list, not a version registry; versions come from literal `roc` fields in the selected app/package/platform headers. Public-example roots belong in the list when testing released dependencies with each new nightly; their dependency URLs remain immutable. Independent-compiler example roots stay outside the list. Paths reject escapes, globs, duplicates and non-Roc files; local source reads reject symlinks and paths outside the checkout.

Prepare preserves bytes outside each selected pin literal. Before replacing an existing candidate, and independently before merging, the controller reads immutable parent/candidate blobs via the API and compares the full contents against the exact expected literal replacements. The complete changed-file set must match those roots, with every file modified in place. Extra files or a body change cannot pass as a compiler update. A stale candidate is interpreted using the immutable configuration from its own parent, so a reviewed change to selected roots or migration between `.roc-version` and header pins cannot make that branch unverifiable or broaden its historical authority. Merge uses trusted event configuration and does not check out or execute

candidate source. Tests cover body mutations, config evolution, extra files, selected-root limits and preservation of unselected examples.

The read-only release guard accepts `compiler-root` to read the source header; empty retains legacy `.roc-version`. An optional paired `simulated-stable-pin` and `simulated-stable-line` permits a clearly labeled simulation only on the exact `roc-<line>.x` branch and only for that exact nightly. It still verifies the published nightly asset release and emits `compiler-channel: simulated-stable`. It never reports stable upstream availability. Remove the mapping after rehearsal.

15.7.4. Manifest compiler authority

A consumer may instead set `compiler_manifest` to an object with a repository-relative JSON `path` and a top-level string `key`. This mode is mutually exclusive with `compiler_roots`; omitting both retains the legacy mode. The controller validates the whole JSON document, rejecting duplicate keys, and replaces only the selected literal's contents. It preserves whitespace, line endings, and all other fields. Paths and symlinked parent directories are checked before local reads. The API candidate verifier compares the full immutable blobs and exact changed-file set, including when validating an older candidate under its parent's configuration. The manifest does not give the controller permission to edit arbitrary metadata.

The release guard accepts paired `compiler-manifest` and `compiler-key` inputs, mutually exclusive with `compiler-root`. This records a tested toolchain without embedding a compiler constraint in a reusable platform's header. Consumer workflows must install the selected compiler and verify its actual version. A moving latest release must be resolved once, not independently by each matrix job.

Chapter 16. Write project documentation

Use this guide when creating or reviewing documentation for a Roc package, platform, application, or shared tool. It separates the reader's needs from the project evidence that maintainers must preserve.

16.1. Give each page one purpose

Choose the page's dominant purpose before writing it. The four Diátaxis forms provide a useful test:

Form	Reader need	Shape
Tutorial	Learn by completing a guided first experience	A safe, linear path with a visible result
How-to guide	Accomplish a real task	Goal-oriented steps that assume relevant knowledge
Reference	Look up exact facts	Complete, neutral descriptions organized like the interface
Explanation	Understand context, design, or tradeoffs	Connected reasoning that answers why

A page can link to another form, but should not interrupt a tutorial with an exhaustive option catalogue or turn reference material into a narrated lesson. Use one topic for one subject or reader question. Split a page when its sections serve different audiences or outcomes.

The manual's reading order may still follow a natural journey: context, first success, goal-oriented guides, reference, contributor material, then appendices. The four forms are an editing discipline rather than required top-level folders.

16.2. Define the page before drafting

Record these answers in the issue, pull request, or draft notes:

- Who is the intended reader?
- What do they already know and have installed?
- What question will this page answer?
- Which documentation form fits that question?
- What should the reader be able to do or understand afterward?
- Which commands, interfaces, versions, and platforms constrain the answer?

Lead with the outcome. Keep prerequisites close to the first step. Use complete, copyable examples and explain the observable result. Link to detailed reference or background when it would distract from the page's purpose.

16.3. Start from the reader's knowledge

Name the primary audience and assume only the knowledge that audience normally has. A package user may understand Roc while knowing nothing about the native engine, protocol, algorithm, or build system behind the package. Introduce a concept in ordinary language before using its specialist name.

For an unfamiliar tool, answer these questions before presenting its API:

- What problem does this solve in the reader's application?
- What must the reader provide?
- What does the tool provide in return?
- How does it differ from the approach the reader probably uses today?
- What does a first successful result look like?
- When is the tool a good fit, and when should the reader choose something else?

Use progressive disclosure. Keep implementation names, escape hatches, internal architecture, and contributor concerns in explicitly advanced or internal material until a user task requires them. Do not use the introduction as a feature inventory.

Before publication, ask a reviewer who did not design the feature to read from the primary audience's perspective. Have them list unanswered questions, undefined terms, assumed experience, and places where they could copy commands without understanding the purpose. Resolve adoption and first-use gaps before polishing expert reference material.

16.4. Show structure and behavior

Use a Mermaid diagram when relationships, state changes, trust boundaries, data flow, or a multi-stage process would take several paragraphs to reconstruct. A diagram should make one point and sit beside the prose that explains its meaning. Prefer stable concepts over incidental implementation details so it remains useful as the code changes.

In AsciiDoc sources built by `actions/build-docs`, embed Mermaid source so the same diagram is rendered into the HTML site and PDF:

```
[mermaid]
....
flowchart LR
    Source[Roc target] --> Build[roc build --fuzz]
    Build --> Runner[Self-contained runner]
    Runner --> Corpus[Coverage corpus]
    Runner --> Failure[Reproducing failure]
....
```

Give every diagram a descriptive title or nearby textual explanation. Do not put information only in color, and make labels meaningful when printed in grayscale. Keep the Mermaid source readable because it is also the maintainable form of the diagram.

Use syntax-highlighted source blocks for commands, configuration, source code, schemas, reports, and output. Name the language explicitly and keep examples focused enough to scan:

```
[source,roc]
----
target = Fuzz.target({ name: "parser", test, show })
----
```

Prefer examples that CI compiles or exercises. When a snippet is intentionally incomplete, say what was omitted. Separate commands from their output, label both, and never rely on highlighting alone to communicate an error or required change.

16.5. Write for the shared renderer

A manual built by `actions/build-docs` is a book with one entrypoint and one file per chapter. The renderer produces the whole book as `index.html`, a separate HTML page for every chapter, and a PDF. A few rules keep all three outputs correct.

- Keep every chapter directly under `docs/`. Only top-level `.adoc` files are rendered as pages; a subdirectory is copied to the site unrendered.
- Keep release notes in `docs/releases/<version>.adoc`, and pull them into the manual from a top-level `releases.adoc` chapter with one `include::releases/<version>.adoc[leveloffset=+1]` line for

each, newest first. They are then rendered in the book, on their own page and in the PDF.

- Make `index.adoc` a `:doctype: book` that pulls in each chapter with `include::<chapter>.adoc[leveloffset=+1]`.
- Open every chapter with a single `Title`. Prefer a title that names what the reader will do.
- Let section IDs be generated from headings. The renderer writes them in lowercase with hyphens, so `Host bindings` becomes `host-bindings`. Give an explicit `[#id]` only where a heading starts with markup or is likely to be reworded.
- Link between chapters with `xref:chapter.adoc#section-id[link text]`, always with a section ID and link text. That form resolves in the book, in the chapter page and in the PDF. A link to a chapter with no section ID is dead in the book, and `<<section-id>>` resolves only where both ends are in the same output, so keep it to `index.adoc`.
- Attributes set in `index.adoc` do not reach a chapter rendered as its own page. Do not rely on a custom attribute inside a chapter.
- Give tables explicit column widths, such as `[cols="1,2",options="header"]`.
- Use `{docs-version}` for the version label; the workflow supplies it.
- Treat every renderer warning as an error. The build fails on a missing include or a malformed block.
- Ignore the output directory, `.docs-out`, in `.gitignore`.

Build the manual in a pull request, and deploy it from the default branch, as described in [Documentation site](#).

16.6. One manual, several audiences

A project has more than one kind of reader: someone deciding whether to use the package, someone using it for the first time, an experienced user looking up a detail, a contributor, a maintainer cutting a release, a person verifying a download. Their needs differ, and they should all be served by one unified document, the manual, and not by notes scattered across the repository.

- Put contributor notes, maintainer procedures, architecture and release history in the manual, as chapters beside the user documentation. A reader then has one place to look and one table of contents to scan, and a single build keeps every part current.
- Decide what each section is for before writing it. Name the reader it serves and what that reader should be able to do afterwards. A section that cannot answer both is not ready to be written.
- Introduce information gradually. Order the manual from the reader who knows least to the

reader who knows most: what the project is, a first success, everyday tasks, reference, then contributing and maintenance. No chapter should require something a later chapter explains.

- Keep audiences apart. A getting-started chapter does not stop to explain the release process, and a maintainer chapter does not re-teach the basics. Link across instead.
- Tell each reader where to start. Open the manual with a short guide that sends each kind of reader to their chapter.

16.7. Keep the README short

A README is the front door, and it faces users. It answers three questions and then hands over to the manual: what is this, what does it look like, and where do I go next.

- One or two sentences saying what the project is and who it is for.
- A prominent link to the documentation site, and to the latest release.
- A grid of screenshots or short examples. A picture of the output, or a few lines of code beside what they produce, tells a visitor more in five seconds than a page of prose. Choose a handful that show the range of the project.
- A minimal install or usage snippet, when one fits in a few lines.

A README must never become a dumping ground. Build instructions, contributor setup, design notes, troubleshooting and release procedure all belong in the manual. When something seems to need adding to the README, add it to the manual and link to it. A README that has grown past a screen or two has usually started doing the manual's job, and the two then drift apart.

The same holds for the other root files. `CONTRIBUTING.md` is a short pointer to the contributing chapter. `SECURITY.md` states how to report a vulnerability. Neither carries guidance that is not also in the manual.

16.8. Cover the public project experience

Review the documentation as a connected entry path, not only as individual pages. A user should be able to find:

- a concise statement of the problem the project solves and its supported scope;
- how to obtain a released version and verify any required toolchain;
- a minimal working example with expected output or behavior;
- task-oriented guidance for normal operation and common failures;
- reference for every public input, output, command, option, configuration value, file format,

and API;

- compatibility, support, and known limitation information;
- where to report bugs, request enhancements, discuss changes, and privately report vulnerabilities;
- how to contribute, including the change process, standards, tests, and checks;
- human-readable release notes and upgrade impact; and
- license, third-party notices, provenance, and acknowledgments where applicable.

All of these live in the manual, as one document. The files that hosting services and tools expect at the repository root, such as `README.md`, `CONTRIBUTING.md`, `SECURITY.md` and `LICENSE`, stay there and stay short; see [Keep the README short](#).

16.9. Keep claims testable

Documentation is evidence of a declared process, not proof that the process is followed. For every operational claim, identify how it is checked:

Claim	Useful evidence
The documented build works	A clean CI job runs the same command
Published examples work	The release tests its examples archive before upload, and a nightly run tests the published archive unchanged
The test suite is the contribution gate	Contributor instructions and required CI run the stated command
The external interface is documented	An API or schema audit fails when public entries are missing
Releases are reproducible and identifiable	Immutable tags, artifact digests, provenance, and release notes
Vulnerability reports receive timely responses	A working private channel and periodically reviewed response history

Prefer generated reference material only when generation is complete, readable, and checked. Handwritten explanations and examples still need review when the interface changes.

16.10. Review each change

Before merging documentation:

- verify links, commands, examples, and expected results;
- build every published format, including HTML, PDF, and generated API pages;
- inspect Mermaid diagrams and highlighted code in both HTML and PDF output;
- check headings and links without relying on generated numbering;
- confirm the navigation names describe reader goals or subjects;
- check that warnings, limitations, and destructive steps appear before the action they qualify;
- review accessibility of diagrams, tables, link text, and color use;
- remove stale version claims and resolved temporary limitations; and
- update the project evidence inventory when a URL or process changes.

[Project-practice evidence](#) supplies the companion checklist for repository and operational evidence. The [Diátaxis framework](#) and [DITA topic guidance](#) are the sources for the content-shape principles used here.

Chapter 17. Project-practice evidence

The OpenSSF Best Practices [passing criteria](#) cover project practices as well as tooling. The model in this manual supplies some of the evidence. Following it does not establish a project's badge status: a criterion is met when the project can point at the behavior, not at a configuration file.

17.1. What the model supplies and what a project still owes

Area	Supplied by the model	Per-project evidence to review
Change control	Actions pinned by full SHA, signed commits, pull requests for every tooling change	Public history, effective review rules, maintained releases
Testing	A check task on every pull request, exact-artifact release tests, a daily compatibility run	Meaningful project tests, a documented invocation, tests for new behavior
Analysis	A named place for code scanning workflows	Analysis suited to the project's own languages; findings resolved
Dependencies	Dependabot pull requests for actions; locked tools and bundles	Dependency inventory, reviewed upgrades, timely vulnerability fixes
Delivery	Immutable releases, content hashes verified locally, provenance reviewed at adoption	The actual release history and its notes
Reporting	A <code>SECURITY.md</code> and private reporting enabled	Working report channels and the actual response history
Documentation	A manual rendered and deployed by a shared action	Project purpose, interface documentation, contribution requirements, license

17.2. Keep an evidence inventory

For each project, maintain an evidence inventory. It is an internal review aid; the public documentation should explain what readers need without discussing a badge application.

Field	What to record
Criterion or practice	A short statement of the behavior being assessed
Applicability	Applicable, not applicable with a reason, or unresolved
Public evidence	A stable URL to documentation, releases, repository settings, or history
Verification	The CI check, manual review, or operational observation that tests the claim
Status	Met, incomplete, or needs investigation; never infer this from a template
Owner and review date	Who maintains the evidence and when it was last checked
Follow-up	A linked issue or pull request for missing work

At minimum, review this inventory:

Area	Evidence to locate and verify
Project entry points	Purpose, supported scope, obtaining a release, feedback, discussion, and contribution links
Legal	Project license in a standard location and applicable third-party notices
Documentation	Basic use, a working first example, public inputs and outputs, commands, configuration, formats, and APIs
Change control	Public version history, reviewable interim changes, unique release versions, immutable tags, and human-readable release notes
Reporting	Searchable issue history, enhancement path, private vulnerability channel, and actual response history
Build and tests	Clean source build, documented test command, CI execution, and a policy to test significant new behavior
Analysis	Appropriate warnings, linters, static analysis, dynamic analysis, and timely treatment of findings

Area	Evidence to locate and verify
Dependencies and delivery	Reviewed updates, dependency inventory, HTTPS delivery, locally verified content hashes, provenance review at adoption, and credential checks
Maintenance	Recent activity, supported versions, vulnerability handling, and periodic review of stale claims

[Cover the public project experience](#) expands the documentation row into an authoring and review process. Verify that the configured reporting channel works. Review repository permissions and branch rules against actual successful checks, including signed commits where required.

17.3. Review and bots

Gold requires at least half of proposed modifications to be reviewed by another person before release. No workflow in this model approves or merges a change, so every tooling change is a maintainer's pull request and counts or fails to count on its own review. Do not manufacture bot approvals.

OpenSSF Scorecard is a separate measurement. Its automated checks are not the Best Practices badge, and it scores review independently. Revisit practice evidence and response history periodically.

Sources: [passing](#), [gold](#), [Scorecard Code-Review](#).

Chapter 18. Contribute to roc-automation

This chapter is for someone changing roc-automation itself: its manual or its shared actions. It assumes the rest of the manual.

18.1. Propose a change

Use issues for bug reports and design discussion, and pull requests for changes. Explain the affected behavior, the change, and how it was validated. Report vulnerabilities privately, as `SECURITY.md` describes. Commits are signed, and changes reach the default branch through pull requests.

18.2. Change the manual

The sources are the AsciiDoc files under `docs/`. [Write for the shared renderer](#) lists the conventions they follow.

Build the site and PDF locally with Docker:

```
python3 actions/build-docs/build_docs.py --workspace . --pdf-filename roc-automation.pdf --docs
-version 3
```

The output is written to `.docs-out/`. The `Documentation` workflow runs the same build on every pull request and deploys from the default branch.

The manual describes the model as it should be practised. A repository or tool that does not yet follow it is recorded in a GitHub issue, not in the manual. Record a change in the release notes under `docs/releases/`, in the file for the version it will ship in. `docs/releases.adoc` includes each of those files.

18.3. Change a shared action

Read [the trust model](#) first. It is normative: a change to an action's behavior or permissions keeps those boundaries and arrives with regression tests for them.

`actions/nightly-report` and `actions/publish-release` are Roc scripts, `scripts/nightly_report.roc` and `scripts/publish_release.roc`, and their tests are `expect` tests that `blueprint run check` runs; [Change an action that is a Roc script](#) says how they are built and tested. The other actions are Python. Python 3.10 or later and the standard library are sufficient for their tests:

```
python3 -m unittest discover -s tests -v
```

- Add regression tests for fixes and tests for new behavior, especially for stale commit or run evidence, branch ownership, permissions and malformed input.
- Keep each token permission on the job that needs it.
- Never execute candidate code in a job that can write.
- Pin remote actions and workflows to full commit SHAs.
- Use the standard library unless an additional runtime dependency has an explicit justification.
- Review dependency changes and address actionable static-analysis findings.

18.4. Change a script

The repository's own automation is Roc scripts under `scripts/`, written as [Write automation as Roc scripts](#) describes. With `blueprint` installed as [Get blueprint](#) describes:

```
blueprint run check
```

It runs the `expect` tests of the modules under `scripts/src/`, checks each script with `roc-stable`, checks their formatting, and holds every script's `roc` field to the release `Blueprint.roc` names. CI runs the same task.

`scripts/check_settings.roc` must stay read-only. Its one request function states the `GET` method, and what it decides from GitHub's answers is pure logic in `scripts/src/Settings.roc`. A change to a check arrives with `expect` tests for the passing and the failing answer, and a setting that cannot be read is reported as unknown. CI does not run it against a repository, because that needs administration access; see [Confirm the settings](#).

18.4.1. Change an action that is a Roc script

`scripts/nightly_report.roc` and `scripts/publish_release.roc` run with write tokens, so they are held to [What the jobs with write access run](#) and to the rules for every shared action.

Every decision is in a module without effects: `scripts/src/NightlyReport.roc`, `scripts/src/PublishRelease.roc`, and `scripts/src/GitHub.roc` for the `gh` invocation and the checks the

two share. A module checks the inputs into a plan, and then decides one step at a time: given the answer to the last request, it names the next request or the end. The script only carries each request to `gh`, or to the runner's disk, and brings the answer back.

- A change to a validation rule, to the order of the requests, to what a failure leaves behind or to an error message arrives with an `expect` test. The tests drive the same step functions against a pretend repository and compare the whole sequence of requests, so none of them reaches GitHub.
- Keep the method of every request stated in `GitHub.args`, and the token out of every plan, argument and message. `gh` reads it from `GH_TOKEN`.
- A local check comes before the first request. In `PublishRelease` the local steps and the requests are different types, so a step that reads GitHub cannot be placed among the local checks.
- The composite `action.yml` repeats the two steps of `actions/setup-blueprint`, because an action cannot reference a sibling at its own commit. Change the install action's SHA in all three files together.

To run a script by hand, give it the environment the action gives it: the inputs as `INPUT_*`, the token as `GH_TOKEN`, and the runner's `GITHUB_*` variables. A pass with a title that no issue has only lists issues:

```
GH_TOKEN="$(gh auth token)" GITHUB_REPOSITORY=<owner>/<name> \
  GITHUB_SERVER_URL=https://github.com GITHUB_RUN_ID=1 GITHUB_OUTPUT=/tmp/outputs \
  INPUT_OUTCOME=success INPUT_NIGHTLY_TAG=probe INPUT_RELEASE_TAG=0.0.0 \
  INPUT_TITLE='nightly-report probe, never opened' \
  blueprint run nightly-report
```

`ci.yml` runs `actions/nightly-report` itself with those inputs, in a job whose token can only read issues. No job there holds a token that can write issues or releases, so the requests that write are covered by the `expect` tests and not by a run against GitHub.

18.5. Release the tools

The scripts behind the shared actions are also built as static Linux executables, so that a calling job can run one without Nix or a compiler.

```
blueprint run build-tools
```

It writes `dist/<tool>-<system>` for `x86_64-linux` and `aarch64-linux`. Both are built on one machine, and two builds of the same commit give the same bytes.

The `Release tools` workflow publishes them under the tag `tools-<version>`. It follows [The order of a release](#):

1. The build job runs `blueprint run check`, builds the executables and lists their SHA-256 digests in `sha256sums.txt`.
2. A test job on each runner architecture downloads those exact files and runs them with no checkout, no Nix and no compiler.
3. The publish job, on dispatch only, hands the tested files to `actions/publish-release`.

A pull request runs the first two steps as a dry run. Before dispatching a release, commit its notes as `docs/releases/tools-<version>.md`.

18.6. Release a shared action

Repositories reference an action by full commit SHA and adopt a new one through a Dependabot or maintainer pull request. Nothing updates them automatically.

For a change to an action that a reusable workflow in this repository calls, commit the action and its tests first, then update the workflow to that exact commit. This gives the nested action an immutable revision without a self-referential commit hash.

Use a unique version tag for a published release, record any assigned vulnerability identifiers for fixes, and never move an existing tag.

Chapter 19. Release notes

Each version of this manual has its notes in `docs/releases/`, newest first.

19.1. roc-automation 3

This version applies what the first repository to adopt the model found. It removes repeated values from every workflow, adds two shared actions for steps that each repository had written differently, and says what the model asks of the simplest kind of repository.

19.1.1. `actions/setup-blueprint` needs no inputs

The action reads the roc-blueprint release from the platform URL in the checked-out `Blueprint.roc`, and records the reviewed SHA-256 of each release's binaries itself. See `actions/setup-blueprint`.

If you followed version 2:

- move the reference to a revision of the action that records your Blueprint release;
- delete the `version` and `sha256-*` lines from every call; and
- make sure the checkout step comes before the action.

```
- uses: actions/checkout@REVIEWED_FULL_SHA
  with:
    persist-credentials: false
- uses: lukewilliamboswell/roc-automation/actions/setup-blueprint@REVIEWED_FULL_SHA
```

The inputs still work, so an existing call keeps running unchanged. Keep `version` only in a job that has no checkout, and a `sha256-*` input only to install a release the action does not record. [Upgrade Blueprint](#) is now a change to `Blueprint.roc` alone.

19.1.2. `actions/publish-release`

The publish job of `release.yml` calls `actions/publish-release` in place of hand-written steps. It stages a draft at the tested commit, uploads the assets, downloads each one back and compares it with the tested file, and only then publishes. It refuses a version that already has a tag, a release or a

draft, and it never asks GitHub to generate notes.

[Recover an interrupted publication](#) is rewritten to match: a draft left by a failed run is deleted, and the publish job of the same run is run again.

19.1.3. `actions/nightly-report`

The report job of `nightly.yml` calls `actions/nightly-report` in place of inline steps. It keeps the one tracking issue, which is the open issue with the exact title whose author is a bot account, and it treats any outcome other than `success` or `failure` as a setup failure that leaves the issue alone.

Pass it the outcome of the step that ran the examples, not the result of the test job; [Reporting](#) explains the difference.

19.1.4. Repository settings are checked by a task

`blueprint run check-settings -- <owner>/<name>`, run from a checkout of `roc-automation`, reads a repository's settings from GitHub and reports each line of [Repository settings](#) as passing or failing, with the fix for each failure. It only reads, and it runs on a maintainer's machine. [Confirm the settings](#) describes it.

The chapter is more exact in three places so that the task can check them:

- neither an app nor a deploy key is on a ruleset's bypass list;
- the `release` and `github-pages` environments name the default branch, and do not use "Protected branches only"; and
- the tag ruleset targets all tags and restricts both updates and deletions.

This repository now has a `Blueprint.roc` and Roc scripts of its own, checked by `blueprint run check`.

19.1.5. A pure Roc package tests on one runner

[A pure Roc package](#) names the profile most repositories follow: no host code, no native inputs, nothing that differs by target. Such a package runs each workflow on `ubuntu-latest` only, has no matrix and no aggregate job, and skips the platform chapters.

19.1.6. Three release rules are settled

- A dry run renders the release notes for the highest version under `docs/releases/`.
- The compiler versions follow the rendered notes under the heading "Compilers".
- Several bundles in one release each carry a filename prefix that names the package, because Roc identifies a package by its URL without the version and hash. The prefix is part of the

package's identity.

See [Write the release notes](#) and [Several bundles in one release](#).

19.2. roc-automation 2

This version adds the `actions/setup-blueprint` shared action and settles how the nightly compiler is installed.

19.2.1. `actions/setup-blueprint`

The action the manual describes now exists. It installs Nix and the released `blueprint` binary for x86_64 Linux, arm64 Linux and Apple Silicon macOS runners, and admits the binary only when it matches a SHA-256 the caller supplies. See `actions/setup-blueprint` for its inputs.

19.2.2. The nightly compiler is `roc`, installed by `roc-lang/setup-roc`

Version 1 described a planned `actions/setup-roc-nightly` that installed the compiler as `roc-nightly` and set `ROC_NIGHTLY`. That action is dropped. Workflows use the official `roc-lang/setup-roc` action, which installs the latest nightly as `roc`, and scripts run package work with `roc`. See [Install the nightly compiler](#).

If you followed version 1:

- replace `roc-nightly` with `roc` in workflows and scripts;
- remove any use of `ROC_NIGHTLY`; and
- pass the resolved tag to `roc-lang/setup-roc` through its `nightly-tag` input.

19.3. roc-automation 1

Initial version of the roc-automation manual.

It describes an opinionated approach for managing Roc packages and platforms: the tooling environment, repository layout, examples, testing, the pull-request, release and nightly workflows, platforms and their native build inputs, and the trust model behind the shared actions.