

The roc-blueprint manual

THE ROC-BLUEPRINT CONTRIBUTORS

Version main

Table of Contents

1. What Blueprint is	2
1.1. What you write and what checks it	3
1.2. How it works	3
1.3. Where it runs	3
1.4. What it does not do	4
2. Get started	5
2.1. Install blueprint	5
2.2. Write a first Blueprint.roc	5
2.3. Lock the downloads and run something	6
2.4. Add a tool from Nix	7
2.5. Commit the lock, not the generated files	7
2.6. Where to go next	7
3. Write Blueprint.roc	8
3.1. Top-level settings	8
3.2. Environment settings	9
3.3. Fetch a program	11
3.4. Expect a tool on the machine	14
3.5. Choose where tools come from	15
3.6. Add tools for one system	16
3.7. Give a tool another command name	16
3.8. Configure a tool	16
3.9. Pin an exact version of a tool	17
3.10. Lock the Roc packages scripts use	18
3.11. Compose settings from modules	20
3.12. Pass data to the provider	20
4. Build artifacts and run workflows	21
4.1. Declare a build	21
4.2. What a build sees and produces	21
4.3. Order tasks and builds in a workflow	22
5. Run blueprint	23
5.1. What a run does, in order	24
5.2. What each command validates	24
5.3. The lock and generated files	25
5.4. What tasks need	25
5.5. What update needs and records	26

5.6. Environment variables	28
6. Work without Nix	30
6.1. A project that never starts Nix	30
6.2. What each command does without Nix	31
6.3. The store of downloads	32
6.4. How a download is made	33
6.5. What is on PATH	33
6.6. Programs blueprint still starts	34
6.7. What still needs Nix	34
6.8. Windows	34
7. Learn from the examples	35
7.1. Run an example	36
8. Understand the architecture	37
8.1. Why Blueprint has a core model	37
8.2. Terms	37
8.3. Layers	39
8.4. Flow	40
8.5. Invariants	42
8.6. The coupling test	44
9. The Spec, the Lock and providers	45
9.1. What the Spec holds	45
9.2. What the Lock holds	47
9.3. Compatibility of the formats	48
9.4. What a provider does	48
10. Contribute	51
10.1. Find your way around	51
10.2. Set up	52
10.3. Build and test	52
10.4. The job with no Nix	55
10.5. Write the manual	56
11. Maintain and release	58
11.1. Move a compiler	58
11.2. Release	59
11.3. What remains for Windows	64
11.4. Upstream workarounds still in place	64
12. Release notes	66
12.1. roc-blueprint 0.6.0	66
12.2. roc-blueprint 0.5.0	73

roc-blueprint lets a project describe its development environments, the commands that run in them, its sandboxed builds and its ordered workflows in one typed file, `Blueprint.roc`. The file says where every program comes from: a download checked against a digest the project commits, the machine itself, or a pinned Nix package source. The `blueprint` command carries it out, and starts Nix only for what a project takes from Nix.

How to use this manual

Deciding whether to use Blueprint

Start with [What Blueprint is](#).

Using it for the first time

Follow [Get started](#).

Writing a configuration

[Write Blueprint.roc](#) describes every setting, and [Build artifacts and run workflows](#) the sandboxed builds.

Looking up a command or variable

See [Run blueprint](#).

Running a project where Nix is not installed

Read [Work without Nix](#).

Looking for a project to adapt

Browse [Learn from the examples](#).

Writing a provider or judging a design change

Read [Understand the architecture](#) and [The Spec, the Lock and providers](#).

Changing roc-blueprint itself

Read [Contribute](#).

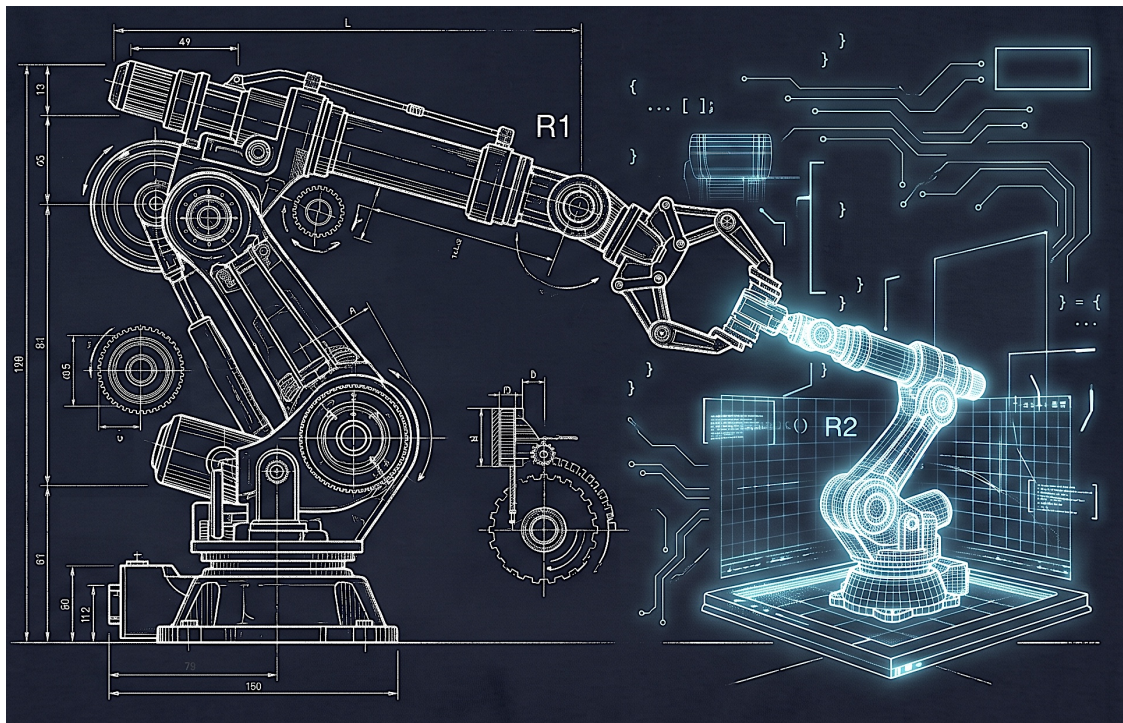
Releasing or looking after the repository

Read [Maintain and release](#).

Upgrading

See [Release notes](#).

Chapter 1. What Blueprint is



A project's `Blueprint.roc` is a Roc program that describes:

- **environments**: named sets of programs and tools, which may extend one another;
- **shells** and **tasks**: an environment to enter, or an exact command to run in one;
- **builds**: sandboxed commands that produce an artifact from declared inputs;
- **workflows**: ordered sequences of tasks and builds.

The `blueprint` command reads that file and carries it out. Everything an environment has comes from a place the file names:

Written as	Where it comes from	What fixes it
<code>Programs([Fetch(name, recipe)])</code>	A download, from the URL the recipe gives for each system.	The SHA-256 of the download, in <code>Blueprint.lock</code> .
<code>Expect([name])</code>	The machine. <code>blueprint</code> checks that it is on <code>PATH</code> and installs nothing.	Nothing: its version is the machine's.

Written as	Where it comes from	What fixes it
<code>Tools(["<source>#<name>"])</code>	A Nix package source the project declares with <code>Packages</code> .	The source's revision and digest, in <code>Blueprint.lock</code> .

The project commits `Blueprint.lock`, so a contributor's machine and a CI runner get the same downloads and the same packages. Nix is started only for an environment that has a tool from a Nix source, and for a build. A project that fetches its programs and expects the rest runs with no Nix installed; see [Work without Nix](#).

1.1. What you write and what checks it

`Blueprint.roc` is an ordinary Roc application on the roc-blueprint platform. Its one value, `config`, is a list of settings, so configurations compose with ordinary Roc lists, functions and imports.

Roc checks quoted values and whole-configuration rules during `roc check Blueprint.roc`: missing names, duplicate declarations, unknown references and inheritance cycles are compile-time errors. An editor shows them as you type. Checking needs a Roc nightly the platform release was built for; each release's notes name it. A download's URL, the path of a program inside an archive and the name of a system are each checked where they are written. Every tool names the package source it comes from, and its name is checked against that source's grammar. Whether a package exists is decided by Nix, and whether a URL serves anything by `blueprint update`, not by Roc.

1.2. How it works

The platform lowers and validates the whole configuration, then prints it as a **Spec**, a provider-neutral S-expression (format 2.7). The `blueprint` command runs Roc to get that Spec, validates it again through the same shared rules, plans the request, and owns every download, file write, the lock and execution. It fetches programs into a per-user store itself, and asks the Nix provider only for the environments that have a tool from a Nix source. The core package and the Nix renderer are pure: they discover nothing about the host and perform no effects. [The architecture chapter](#) explains the model and the rules that keep it independent of any one package ecosystem.

1.3. Where it runs

System	Fetches programs, expected tools, tasks and shells	Tools from a Nix source	Sandboxed builds
x86_64 Linux	Yes	Yes	Yes
arm64 Linux	Yes	Yes	No
Apple Silicon macOS	Yes	Yes	No
x86_64 Windows	No binary is released	No	No
Intel macOS	Not supported	Not supported	Not supported

A `blueprint` binary is released for x86_64 Linux, arm64 Linux and Apple Silicon macOS. It needs no Roc and no Python: it downloads the Roc compiler that reads `Blueprint.roc`. It needs [Nix](#) with flakes enabled where the table's last two columns apply, and `tar`, `ln` and `chmod` to unpack and link what it fetches.

On Intel macOS `blueprint` stops with a message naming the supported systems. On Windows the code that runs a task without Nix is present, and nothing of it can be used; see [Windows](#).

`Systems` in a configuration declares the systems a project is for: a fetched program gets a digest for each, and the generated Nix outputs have each. It installs nothing and is not evidence that a target runs: `x86_64-darwin` may still be listed there.

1.4. What it does not do

- Nix is the only package provider. A source may state Guix intent, and its compatibility is validated, but there is no Guix renderer or executor.
- An expected tool is looked for by name. No version is asked of it.
- A fetched program is one file, or one file inside an archive that is unpacked whole. An archive gives one command name.
- There is no parallel workflow scheduler: steps run in order.
- Tasks and the evaluation of `Blueprint.roc` are not sandboxed; only builds are, and a build cannot use fetched programs or expected tools.

Chapter 2. Get started

This chapter installs `blueprint`, writes a first `Blueprint.roc` that needs nothing but the binary, runs a task in the environment it describes, and then adds a tool from Nix.

2.1. Install blueprint

Each [release](#) attaches a `blueprint` binary for x86_64 Linux, arm64 Linux and Apple Silicon macOS, named `blueprint-x86_64-linux`, `blueprint-aarch64-linux` and `blueprint-aarch64-darwin`, with their SHA-256 sums in `blueprint.sha256`. Download the one for your system from the [latest release](#), check it and put it on your `PATH`:

```
sha256sum --check --ignore-missing blueprint.sha256
install -m 755 blueprint-x86_64-linux ~/.local/bin/blueprint
```

That is all the first project below needs. [Nix](#) with flakes enabled is needed for the tools a project takes from a Nix package source and for its sandboxed builds, and for nothing else: `blueprint requirements` says which of a project's tasks need it. Where Nix is installed, the same version also runs from the flake:

```
nix run github:lukewilliamboswell/roc-blueprint/<version> -- --help
nix profile install github:lukewilliamboswell/roc-blueprint/<version>
```

In GitHub Actions, the `setup-blueprint` action installs Nix and a release's binary and checks its sum.

Loading a configuration runs one exact Roc nightly. The binary uses a `roc` on `PATH` when it is that nightly, and otherwise downloads it from its release, checks it against a digest the binary records, and keeps it in its store of downloads. Set `ROC` to choose the executable yourself; it must be that same nightly.

2.2. Write a first Blueprint.roc

Put this in `Blueprint.roc` at the root of your repository. The platform URL is the latest release's:

```
app [config] { pf: platform "https://github.com/lukewilliamboswell/roc-
```

```

blueprint/releases/download/0.5.0/CupqPH9snWf11g7tWgcJf4Yg5cJ3K9fhL7fSgqDcqJQH.tar.zst" }

import pf.Recipes

config = [
  Name("my-project"),
  Environment("dev", [
    Programs([Fetch("roc-stable", Recipes.roc_nightly("nightly-2026-10-13-b17d287"))]),
    Expect(["git"]),
  ]),
  Shell("default", [Use("dev")]),
  Task("version", [Use("dev"), Run(["roc-stable", "version"])]),
  Task("status", [Use("dev"), Run(["git", "status", "--short"])]),
]

```

Use each release's `blueprint` binary with that release's platform URL. The two are built and tested together.

The file declares one environment, `dev`. It has two commands:

- `roc-stable` is **fetched**: `Recipes.roc_nightly` says where that Roc nightly is downloaded from on each system, and `Blueprint.lock` will hold the digest of each download.
- `git` is **expected**: the machine must already have it. `blueprint` checks for it and installs nothing.

Each task names its environment and an exact argument list.

2.3. Lock the downloads and run something

```

blueprint check      # validate Blueprint.roc; needs no lock
blueprint update     # record every download's digest in Blueprint.lock
blueprint run version # run the task's command in dev
blueprint shell      # enter dev: your own shell, with roc-stable on PATH
blueprint requirements # what each task needs of the machine
blueprint --help     # list this project's shells and tasks

```

`blueprint update` is the only command that decides anything: it downloads the compiler for your system, hashes it, and writes the digest for every system to `Blueprint.lock`. Every other command uses that file and fails if it no longer matches `Blueprint.roc`. `blueprint run` then downloads nothing: the compiler is kept in a per-user store and checked against the lock on every use.

None of this starts Nix, and `blueprint requirements` says so:

```

system x86_64-linux
providers: none
task version [dev]: programs roc-stable; expects git
task status [dev]: programs roc-stable; expects git

```

2.4. Add a tool from Nix

A tool that should be installed, rather than fetched as one program or expected of the machine, comes from a Nix package source. Declare the source, and write each tool with the source's name. Inside `config`:

```
Packages("nix", From(NixPackages("github:NixOS/nixpkgs/nixos-unstable"))),
Environment("native", [Tools(["nix#zig", "nix#pkg-config"])]),
Task("zig-version", [Use("native"), Run(["zig", "version"])]),
```

To give `native` the fetched compiler and `git` as well, start its settings with `Extend("dev")`.

The project now declares something Nix resolves, so `blueprint update` needs Nix: it pins the source's revision in the same `Blueprint.lock`. After that, only what uses the source starts Nix. `blueprint run zig-version` does; `blueprint run version` still does not, on a machine with Nix or without. `blueprint update nix` moves that one source and keeps every other pin and digest.

2.5. Commit the lock, not the generated files

Commit `Blueprint.roc` and `Blueprint.lock`. The lock is the authority: a versioned S-expression listing each download with its digest and each pinned source with its revision.

`blueprint` keeps what it generates under `.blueprint/` and writes a `.gitignore` there, so that directory never reaches a commit. If your repository already tracks `.blueprint/`, untrack it once:

```
git rm -r --cached .blueprint
```

2.6. Where to go next

- [Write Blueprint.roc](#) describes every setting.
- [Work without Nix](#) says where downloads are kept and what a project without Nix can do.
- [Build artifacts and run workflows](#) covers sandboxed builds.
- [The examples](#) are complete projects, attached to each release with that release's platform URL.

Chapter 3. Write Blueprint.roc

Blueprint.roc is a Roc application on the roc-blueprint platform. It provides `config`, a list of settings:

```
app [config] { pf: platform "<the platform URL of a release>" }

import pf.Recipes

config = [
  Name("my-project"),
  Environment("dev", [
    Programs([Fetch("roc-stable", Recipes.roc_nightly("nightly-2026-10-13-b17d287"))]),
    Expect(["git"]),
  ]),
  Shell("default", [Use("dev")]),
  Task("check", [Use("dev"), Run(["roc-stable", "check", "main.roc"])]),
]
```

An environment gets what it has from three places, and each is written differently: a program that is downloaded ([Fetch a program](#)), a tool the machine already has ([Expect a tool on the machine](#)), and a tool a package source installs ([Choose where tools come from](#)).

[Get started](#) has a complete file with the current URL.

3.1. Top-level settings

Setting	Meaning
<code>Name(Str)</code>	Project name. Required, once.
<code>Systems(List(System))</code>	The systems the project is for, each <code><arch>-<os></code> of a known architecture (<code>x86_64</code> , <code>aarch64</code> , <code>riscv64</code> , <code>i686</code> , <code>armv7l</code> , <code>powerpc64le</code>) and operating system (<code>linux</code> , <code>darwin</code> , <code>windows</code> , <code>freebsd</code>). Defaults to <code>x86_64</code> and <code>aarch64</code> on Linux and macOS; an empty list is invalid. Every fetched program is resolved for each of them.

Setting	Meaning
<code>Packages(InputName, From(Provider))</code>	A package source and the set it is: <code>NixPackages(FlakeRef)</code> or <code>GuixPackages(Str)</code> . Every source is declared; none is implied, and a project that takes no tool from a package source declares none.
<code>Overlay(InputName, FlakeRef)</code>	Named overlay input; only environments selecting its name apply it.
<code>Input(InputName, FlakeRef)</code>	Other flake input; not a tool source or overlay.
<code>Environment(EnvName, List(EnvironmentSetting))</code>	Named tools and scoped overlays, optionally inherited.
<code>Shell(EnvName, List(ShellSetting))</code>	Named alias with exactly one <code>Use(environment)</code> . <code>blueprint shell</code> selects alias <code>"default"</code> .
<code>Task(TaskName, List(TaskSetting))</code>	Named argv command with exactly one <code>Use(environment)</code> and one <code>Run(argv)</code> . No shell alias is required.
<code>Source(InputName, FlakeRef)</code>	Locked non-flake input, e.g. <code>Source("assets", "path:./assets")</code> .
<code>Build(InputName, List(BuildSetting))</code>	Sandboxed artifact with required <code>Use</code> , exact argv <code>Run</code> , relative <code>Output</code> ; optional <code>Inputs</code> and <code>Needs</code> .
<code>Workflow(WorkflowName, List(WorkflowStep))</code>	Ordered <code>RunTask(name, extra_argv)</code> , <code>BuildArtifact(name)</code> and <code>RunWorkflow(name)</code> steps.
<code>Raw(backend, target, Val)</code>	Provider-specific data; see Pass data to the provider .
<code>Custom(kind, name, Val)</code>	Extension data. The current CLI rejects unsupported extensions.

3.2. Environment settings

Inside an `Environment`, `Tools` and `Overlays` occur at most once, `ToolsFor` occurs at most once per `System`, `Command` occurs at most once per command name, `ToolWith` occurs at most once per tool, and `RocPackages`, `Programs` and `Expect` occur at most once each:

Setting	Meaning
<code>Tools(List(Tool))</code>	Tools a package source installs, each written <code>"<source>#<name>": "nix#git"</code> is <code>git</code> from the source declared as <code>nix</code> . A Nix name is an attribute path; quote a segment that contains dots, as in <code>"nix#zigpkgs.\\"0.16.0\\""</code> .
<code>ToolsFor(System, List(Tool))</code>	Add tools only for a declared target System. Each System occurs at most once per Environment.
<code>Command(Str, Tool)</code>	Expose one tool's main program under another command name, without adding the tool's own executables.
<code>ToolWith(Tool, List Str, Val)</code>	A tool built with options its package takes; see Configure a tool .
<code>RocPackages(List(Str))</code>	Released Roc bundle URLs to have in place before anything runs, so Roc programs in the environment resolve them without downloading; see Lock the Roc packages scripts use .
<code>Programs(List(Program))</code>	The programs the environment has on PATH that are downloaded, each a <code>Fetch(ProgramName, recipe)</code> ; see Fetch a program .
<code>Expect(List(ProgramName))</code>	Programs the machine must already have on PATH; see Expect a tool on the machine .
<code>Overlays(List(InputName))</code>	Ordered selection of declared overlay names.
<code>Extend(EnvName)</code>	Inherit one environment's tools, overlays, commands, configured tools, Roc packages, programs and expected tools before appending this environment's selections.

Inheritance deduplicates by first occurrence, parent first. Omitted or empty `Tools/ToolsFor/Overlays` lists do not clear inherited values. A standalone environment has no overlays unless selected. `Run` must contain a nonempty executable; arguments remain separate strings, including extra CLI arguments after `--`. There is no `In` setting or implicit task environment. One name is only one of a program, an expected tool and a `Command` in an environment, including what it inherits.

3.3. Fetch a program

A program that is published as a file or an archive is **fetched**: downloaded from a URL `Blueprint.roc` gives, checked against a digest in `Blueprint.lock`, and put on the `PATH` of the environments that list it. No package source is involved and Nix is not started.

```
Environment("dev", [
  Programs([
    Fetch("roc-stable", Recipes.roc_nightly("nightly-2026-10-13-b17d287")),
    Fetch("blueprint", Recipes.file("https://github.com/lukewilliamboswell/roc-
blueprint/releases/download/0.5.0/blueprint-{system}")),
  ]),
]),
```

`Fetch(name, recipe)` is a command name and a **recipe**: a function from a system to where the program is downloaded from on that system. The environment gains the command `name`. A recipe downloads nothing. It is asked once for each of the project's `Systems` while `Blueprint.roc` is checked, and its answers are what `blueprint` works from. The `Recipes` module is the platform's; import it with `import pf.Recipes`.

3.3.1. What a recipe answers

Answer	Meaning
<code>File(url)</code>	The program is that one file. It is made executable.
<code>Archive(url, format, member)</code>	An archive in the format <code>TarGz</code> , <code>TarXz</code> or <code>Zip</code> , with the program at the path <code>member</code> inside it. The archive is unpacked whole, and the command is a link to the program where it lies, so it finds what the archive put beside it.
<code>Mirrored(download, urls)</code>	A <code>File</code> or an <code>Archive</code> with alternate URLs for the same bytes, tried in order after its own. <code>Blueprint.lock</code> holds one digest for all of them, so an alternate need not be trusted.
<code>Unavailable</code>	The program does not exist for that system. This is an error only where a task on that system needs it; a program unavailable on every declared system is refused when <code>Blueprint.roc</code> is checked.

A recipe that cannot answer at all, for a version it does not know, stops with `crash` and a message; the compiler reports it with the recipe's file and line.

3.3.2. The recipes the platform provides

Recipes.file(template)

A program that is one file at a URL which names the system. `{system}` in the template stands for the system asked about, such as `x86_64-linux`, and `{arch}` and `{os}` for its two parts. No other placeholder is accepted, and none may stand in the host name. The program is taken to exist for every declared system: when its publisher has no file for one of them, `blueprint update` fails on that file and says so. State it by listing the systems the project supports in `Systems`, or write a recipe that answers `Unavailable`.

Recipes.roc_nightly(tag)

The Roc compiler of one release of [roc-lang/nightlies](https://roc-lang.github.io/nightlies/), by its tag, `nightly-YYYY-MM-DD-<commit>`. It answers for `x86_64` and `arm64` Linux, Apple Silicon and Intel macOS and `x86_64` Windows, and `Unavailable` for any other system. Fetched as `roc-stable`, it is the compiler a script that starts `#!/usr/bin/env roc-stable` runs on.

3.3.3. Write a recipe

A recipe is an ordinary function, so a project writes its own for a program whose downloads do not follow one pattern:

```
app [config] { pf: platform "<the platform URL of a release>" }

import pf.Config
import pf.Recipes
import pf.System

config = [
  Name("my-project"),
  Systems(["x86_64-linux", "aarch64-linux", "aarch64-darwin"]),
  Environment("dev", [
    Programs([
      Fetch("roc-stable", Recipes.roc_nightly("nightly-2026-10-13-b17d287")),
      Fetch("wasm-tools", wasm_tools),
    ]),
  ]),
  Task("validate", [Use("dev"), Run(["wasm-tools", "validate", "app.wasm"])]),
]

wasm_tools : System -> Config.Download
wasm_tools = |system| {
  base = "https://github.com/bytecodealliance/wasm-tools/releases/download/v1.240.0"
  match system {
    "x86_64-linux" => Archive("${base}/wasm-tools-1.240.0-x86_64-linux.tar.gz", TarGz, "wasm-
tools-1.240.0-x86_64-linux/wasm-tools")
    "aarch64-linux" => Archive("${base}/wasm-tools-1.240.0-aarch64-linux.tar.gz", TarGz, "wasm-
tools-1.240.0-aarch64-linux/wasm-tools")
    "aarch64-darwin" =>
      Mirrored(
        Archive("${base}/wasm-tools-1.240.0-aarch64-macos.tar.gz", TarGz, "wasm-tools-
```

```
1.240.0-aarch64-macos/wasm-tools"),
    ["https://mirror.example.net/wasm-tools-1.240.0-aarch64-macos.tar.gz"],
    )

    _ => Unavailable
}
}
```

Give a recipe of more than a line its type, `System` → `Config.Download`, as here: a mistake in it is then reported on its own line. A recipe written inline, as `Fetch("tool", |system| ...)`, is checked just the same, and the compiler may report a mistake in it against the platform's file with the line and column of the literal in words.

A `System` is not text. To write it into a URL, convert it:

```
Environment("dev", [
    Programs([Fetch("tool", |system| File("https://example.org/releases/1.0/tool- $\{system.to\_str()\}$ "))]),
]),
```

With `$\{system\}$` alone the compiler reports a type mismatch of `from_interpolation` on `Url`, at the first line of the file and not at the recipe. `Recipes.file` avoids the interpolation altogether:

```
Environment("dev", [
    Programs([Fetch("tool", |system| File("https://example.org/releases/1.0/tool- $\{system\}$ "))]),
]),
```

3.3.4. Share a program between environments

`Fetch` is written inside `Programs`, so a program cannot be listed without being declared. An environment that extends another inherits its programs. Two environments that do not extend one another share a program through an ordinary binding:

```
app [config] { pf: platform "<the platform URL of a release>" }

import pf.Recipes

roc_stable = Fetch("roc-stable", Recipes.roc_nightly("nightly-2026-10-13-b17d287"))

config = [
    Name("my-project"),
    Environment("scripts", [Programs([roc_stable]), Expect(["git"])]),
    Environment("release", [Programs([roc_stable]), Expect(["gh"])]),
    Environment("ci", [Extend("scripts")]),
]
```

One name always means the same downloads. Two `Fetch` values with one name whose recipes answer differently for a declared system are refused as two different programs are named `roc-`

stable.

3.3.5. What is checked, and when

When	What
<code>roc check</code> , in an editor or on the command line	Each text, as what it is where it is written: a system is <code><arch>-<os></code> of a known architecture and operating system, in a <code>match</code> pattern and beside <code>==</code> too; a URL is <code>https://<host>/<path></code> with no credentials, query or fragment; the path of a program in an archive is relative and not a URL, so the two cannot change places; a nightly tag has its form; a template has only its three placeholders. Then the whole configuration: a program name is a plain file name, one name is only one of a program, an expected tool and a <code>Command</code> in an environment, no URL is listed twice for one download, and every program exists for at least one declared system.
<code>blueprint update</code>	That every download exists, by downloading it or by taking the digest its release publishes; and, where both are to be had, that they agree. See What update needs and records .
<code>blueprint run</code> , <code>shell</code> and <code>workflow</code>	That <code>Blueprint.lock</code> holds the program for this system under the URLs <code>Blueprint.roc</code> now gives; that every expected tool is on <code>PATH</code> ; then that the download has the recorded length and SHA-256, on every use; that an archive holds nothing that leaves the directory it is unpacked in; and that the program is in it.

A program whose URLs changed since the lock was written is stale by itself: the message names it, and `blueprint update <name>` resolves it alone.

3.4. Expect a tool on the machine

Some tools are better taken from the machine than installed: `git`, which every developer and CI runner has, or `gh`, which must be the one that is logged in. `Expect` declares them:

```
Environment("dev", [
    Expect(["git", "gh"]),
]),
```

Before a task or shell of that environment starts, and before anything is downloaded for it, `blueprint` looks for each name on `PATH`. One message names everything that is missing:

```
blueprint: this machine lacks what the task needs, and blueprint installs none of it:
gh: environment dev expects it on PATH (Expect)
```

An expected tool is not installed, not put on `PATH` and not pinned, and no version is asked of it: the environment runs with whichever one the machine has. Where a version matters, fetch the program or take it from a package source instead, or have a task check `tool --version`. `blueprint requirements` lists what a project's tasks expect, so a CI job can install it first.

A sandboxed build has nothing of the machine, so an environment a build uses may not have `Expect` or `Programs`.

3.5. Choose where tools come from

A tool that a package source installs is written `"<source>#<name>"`, for a source declared with `Packages`:

```
Packages("nix", From(NixPackages("github:NixOS/nixpkgs/nixos-unstable"))),
Packages("stable", From(NixPackages("github:NixOS/nixpkgs/nixos-24.05"))),
Environment("dev", [Tools(["nix#git", "stable#jq"])]),
```

The source's name is the project's choice; `nix` and `stable` mean nothing to `blueprint`. The source is always written, in `Tools`, `ToolsFor`, `Command` and `ToolWith` alike, and there is no default. `"git"` alone is refused where it is written:

```
"git" names no package source. Write "<source>#git" for a declared source, as
in Packages("nix", From(NixPackages("github:NixOS/nixpkgs/nixos-unstable")))
and "nix#git"; or, if git is already on the machine, list it in Expect.
```

A tool that names a source the project does not declare is refused with the sources it does declare, as `unknown source: nix`; `Blueprint.roc` declares `stable`. A project whose environments have no such tool declares no source, and Nix is not started for it.

Nix is started only for an environment that has a tool from a Nix source. An environment Nix must build although it has no tool of its own, such as the sandbox of a build whose environment is empty, is built with the first Nix source declared, in the order of `Blueprint.roc`. The bash that `nix develop` runs a shell or task with comes from that first source too.

The core does not autodetect a provider or translate package names. Guix source intent and pure capability validation exist, but there is **no Guix renderer or executor**. An incompatible requested environment fails; it never retries another provider. Missing native packages and unavailable target packages fail in Nix rather than being silently filtered out. Unsupported Nix target declarations fail before file writes or provider execution.

3.6. Add tools for one system

For a shell shared by Linux and macOS, put common tools in `Tools` and declare platform libraries explicitly:

```
Environment("dev", [
  Tools(["nix#git", "nix#python3"]),
  ToolsFor("x86_64-linux", ["nix#wayland", "nix#alsa-lib"]),
]),
```

`ToolsFor` is provider-neutral Spec intent. It applies to inherited environments, shells, tasks and builds on that System; undeclared systems and duplicate declarations are rejected during evaluation.

3.7. Give a tool another command name

`Command` exposes one tool of a package source under another command name, without adding the tool's own executables. It keeps a pinned tool beside another of the same name: a second Python, say, for scripts that must not follow the project's own:

```
Packages("nix", From(NixPackages("github:NixOS/nixpkgs/nixos-unstable"))),
Packages("stable", From(NixPackages("github:NixOS/nixpkgs/nixos-24.05"))),
Environment("dev", [
  Tools(["nix#python3"]),
  Command("python-stable", "stable#python3"),
]),
```

The environment gains `python-stable` beside the `python3` of `nix`. The command runs the tool's main program as its package declares it; a package that declares none fails in Nix. Command names are plain file names. An extending environment inherits commands and replaces one by declaring the same name.

A `Command` is a tool of a package source, so its environment starts Nix. A program that is published as a download needs neither: `Fetch` names its command itself. A Roc compiler for scripts is written `Fetch("roc-stable", Recipes.roc_nightly(tag))`, as in [Fetch a program](#); `Command("roc-stable", "nix#rocpkgs.<tag>")` with the roc-overlay selects the same compiler through Nix, for an environment a sandboxed build uses.

3.8. Configure a tool

Some packages are built to order. A Rust toolchain from `rust-overlay` comes with the host's standard library only; the targets it can compile for and its extra components are options of the package. `ToolWith` names the tool and those options:

```
Overlay("rust", "github:oxalica/rust-overlay"),
Overlay("zig", "github:mitchellh/zig-overlay"),
Environment("host", [
  Overlays(["rust", "zig"]),
  Tools(["nix#zigpkgs.\"0.16.0\""]),
  ToolWith("nix#rust-bin.stable.\"1.95.0\".default", [
    ("targets", List([Str("x86_64-unknown-linux-musl"), Str("aarch64-unknown-linux-musl")])),
    ("extensions", List([Str("rust-src")])),
  ]),
]),
```

The environment gains `cargo` and `rustc` from Rust 1.95.0 with both musl targets and the standard library's source. Zig needs no options: `zig-overlay` names each release as an attribute, which `Tools` selects.

The options are the names and values the package's own `override` takes, written as `Val` data: strings, integers, booleans, lists and attribute sets. They are passed as data and are never an expression, so they select among what the package offers and cannot say how to build it. A package that takes no options, or not the ones named, fails in Nix.

A tool is either plain or configured in an environment, not both. An extending environment inherits configured tools; configuring an inherited tool replaces all of its options. Three things are not supported:

- options on a `ToolsFor` or `Command` tool;
- `rust-overlay`'s `fromRustupToolchainFile`, which reads a file of the project: the generated flake does not see the project's files;
- configured tools from a Guix source.

3.9. Pin an exact version of a tool

A tool's version is whatever its source holds at the pinned revision. `Blueprint.roc` names the tool, not its version, so to hold one tool at an exact version, give it a package source of its own, pinned to a `nixpkgs` revision that ships that version. For CPython 3.14.3:

```
Packages("py", From(NixPackages("github:NixOS/nixpkgs/<revision>"))),
Environment("dev", [Tools(["nix#git", "py#python314"])]),
```

`<revision>` is a full `nixpkgs` commit whose `python314` is 3.14.3. To find one, look at the history of the package's file in `nixpkgs`, or evaluate a candidate:

```
nix eval --raw github:NixOS/nixpkgs/<revision>#python314.version
```

How the source is written decides whether the version can move:

- A source that names a **revision**, as above, never moves. `blueprint update` resolves it to the same commit every time. Change the version by editing the revision.
- A source that names a **branch**, such as `github:NixOS/nixpkgs/nixos-25.05`, is moved by `blueprint update` and by `blueprint update py`, and takes whatever version the branch then has. `blueprint update nix` leaves it alone.

Nothing in `Blueprint.roc` states the version, so nothing fails if it changes. If a version matters, assert it where the tool is used, for example by having the project's check task run `python3.14 --version` and compare.

An overlay that publishes each version under its own attribute needs no second source: select the attribute, quoting a segment that contains dots.

```
Overlay("zig", "github:mitchellh/zig-overlay"),
Environment("dev", [Overlays(["zig"]), Tools(["nix#zigpkgs.\"0.16.0\""])]),
```

3.10. Lock the Roc packages scripts use

A Roc script names its platform and packages by release URL, and Roc downloads them on first use. `RocPackages` names those bundles for an environment instead, so Blueprint has them in place before its scripts start, and they run at all in a sandboxed build, which has no network:

```
Environment("scripts", [
  Programs([Fetch("roc-stable", Recipes.roc_nightly("nightly-2026-10-13-b17d287"))]),
  RocPackages([
    "https://github.com/roc-lang/basic-cli/releases/download/0.26.0-rc2/6j3zVDr3qrQa6aTh7Z1zgUfe9NeBE6R76T5ho3xrc8af.tar.zst",
    "https://github.com/roc-lang/http/releases/download/1.0.0/6ZUwqYhCS8PU9Mo6MF7oV82ET2o7KYb57CLKDq4cq4sS.tar.zst",
  ]),
],
Task("release", [Use("scripts"), Run(["scripts/release.roc"])]),
```

Each URL must be `https://` and end in `<hash>.tar.zst`. The hash is that of the bundle's bytes, so the URL itself says what the download must be.

List every bundle, including dependencies of dependencies. `basic-cli` above depends on `http`, so both are listed. `roc deps <file>` prints the URLs a program depends on. Blueprint does not read package headers, so it cannot add a missing one for you. A bundle that is not listed is downloaded by Roc as usual in a shell or task, and fails to resolve in a sandboxed build.

3.10.1. What the lock pins

For a shell or a task, nothing: `blueprint` downloads a missing bundle itself and has the compiler unpack it with `roc unbundle`, which fails unless the bytes are those the name is the hash of. That check is the pin, and it needs no Nix and no entry in `Blueprint.lock`. A project whose Roc packages are used by tasks and shells alone has none of them in its lock.

A sandboxed build cannot download. The bundles of an environment that a build runs in, or that it extends, are therefore fetched by Nix as well: `blueprint update` records those in `Blueprint.lock` like any other source, and Nix checks the bytes against that pin. An environment a build uses takes its compiler from a Nix source, since a build cannot use a fetched program:

```
Packages("nix", From(NixPackages("github:NixOS/nixpkgs/nixos-unstable"))),
Overlay("roc", "github:roc-lang/roc-overlay"),
Environment("build-scripts", [
  Overlays(["roc"]),
  Command("roc-stable", "nix#rocpkgs.nightly-2026-10-13-b17d287"),
  RocPackages([
    "https://github.com/roc-lang/basic-cli/releases/download/0.26.0-rc2/Gj3zVDr3qrQa6aTh7Z1zgUfe9NeBE6R76T5ho3xrc8af.tar.zst",
    "https://github.com/roc-lang/http/releases/download/1.0.0/6ZUwqYhCS8PU9Mo6MF7oV82ET2o7KYb57CLKDq4cq4sS.tar.zst",
  ]),
]),
Build("notes", [Use("build-scripts"), Run(["roc-stable", "scripts/notes.roc"]), Output("notes.txt")]),
```

An extending environment inherits its parent's bundles.

3.10.2. What is written, and where

- `blueprint shell` and `blueprint run` publish the environment's bundles into Roc's own package cache before entering it: the directory `roc/packages` under `$XDG_CACHE_HOME`, or under `~/.cache` when that is unset. Each bundle becomes the directory `<hash>` there, the files the compiler unpacked from the download, which is where Roc looks before downloading. A `<hash>` directory that already has a `main.roc` is never touched, whether Roc downloaded it or an earlier run published it; one without a `main.roc` is incomplete and is replaced, as Roc itself would. The copy is made in a temporary directory beside it, named `blueprint-<hash>.<random>.tmp`, and renamed into place, so another Roc process never sees half a package. Nothing else is written, and nothing is downloaded or written when every bundle is already there or the environment has no Roc packages. If the cache is deleted, the next run downloads and publishes again.
- `blueprint build` does not use that cache. The build gets a package cache of its own inside its build directory, holding only its environment's bundles, and `XDG_CACHE_HOME` points the build's command at it.

A build sees only the tools its environment declares, so a build that runs Roc needs a Roc tool, such

as the `roc-stable` command above.

3.11. Compose settings from modules

Use ordinary Roc lists and functions for composition, not a plugin registry. `ProjectTasks.roc` returns `List(Config.Setting)`; its `app` concatenates those settings into `config`. Imports do not install tools or add runtime operations to the CLI.

3.12. Pass data to the provider

The Nix provider accepts attribute data at two targets:

```
Raw("nix", "shell:default", Attrs([
  ("shellHook", Str("echo ready")),
  ("RUST_LOG", Str("debug")),
])),
Raw("nix", "flake", Attrs([("formatter", Str("nixpkgs-fmt"))])),
```

- `shell:<alias>` adds attributes to that alias's `mkShell` call, not other aliases or tasks using the same environment.
- `flake` adds attributes to flake outputs.

Values are `Str`, `Int`, `Bool`, `List([...])` and `Attrs([(name, value), ...])`. They are data, not Nix expressions. Duplicate attributes and overrides of managed `packages/devShells` are rejected. Raw for other providers is ignored.

Chapter 4. Build artifacts and run workflows

A `Build` runs one command in a sandbox and keeps one declared output. A `Workflow` runs tasks and builds in order. Sandboxed builds run in Nix, on x86_64 Linux only.

4.1. Declare a build

Inside `config`, with `scripts/build.roc` in the project and the `build-scripts` environment of [Lock the Roc packages scripts](#) use:

```
Source("assets", "path:./assets"),
Build("app", [
  Use("build-scripts"),
  Inputs(["assets"]),
  Run(["roc-stable", "scripts/build.roc"]),
  Output("dist/app"),
]),
```

`Run` is any argv; nothing about a build requires Roc. Roc takes the first `--` of its arguments for itself, so pass a script its own arguments after one: `Run(["roc-stable", "scripts/build.roc", "--", "--release"])`.

4.2. What a build sees and produces

Run `blueprint update`, then `blueprint build app`. The printed store path is resolved by Nix and contains exactly the declared file or directory. Missing outputs fail. `Needs(["library"])` adds build dependencies; cycles fail during configuration validation. Sources live under `$BLUEPRINT_INPUTS/<name>` and needed artifacts under `$BLUEPRINT_ARTIFACTS/<name>`, both read-only and separate from the writable project copy.

Each build snapshots current project files, including untracked/task-generated files, excluding VCS metadata, caller-generated roots, authority and all local input trees. Nix makes that copy directly from the project, so a file is executable in a build only when its owner may execute it. Changing a locked local source requires explicit update. Initial source/output policy rejects symlinks and special files. Only local x86_64 Linux sandboxed execution is verified; tasks/config compilation are not sandboxed. A build's `PATH` holds exactly the tools its environment takes from

its package sources, and nothing of the machine: no `sh`, `cp`, `mkdir` or `python3` unless the environment's `Tools` name `bash`, `coreutils` or `python3`. A command that is not there fails with `build command not found`, or inside the build's own script when that script calls one. A build does not see the variables Nix's `stdenv` exports either. A tool may give what it runs more than the build was given: the Roc compiler's package puts `coreutils` and a C compiler on the `PATH` of a script it runs. An environment with fetched programs or expected tools cannot be a build's: a sandbox holds only what its package sources supply, and the build is refused when it is planned. The `blueprint` executable is itself each build's builder inside the sandbox, so `blueprint build` refuses to run unless that executable is an `x86_64` Linux one. See the [complete runnable example](#), [real build fixture](#).

4.3. Order tasks and builds in a workflow

Inside `config`, referring to existing tasks and builds:

```
Workflow("ci", [RunTask("check", ["--verbose"]), BuildArtifact("app"), RunWorkflow("verify")]),
Workflow("verify", [RunTask("version", [])]),
```

`blueprint workflow ci` plans the complete dependency/capability/layout/lock closure before executing any task. Steps run in order and stop on the first failure. Repeated explicit tasks and builds repeat; every build snapshots current project files and rebuilds its dependency graph from that snapshot. Nix may reuse unchanged inputs, but Blueprint never caches artifact results across tasks. Locked sources are verified again; task edits to them require explicit update. Cycles and excessive depth/expansion fail during configuration validation. See the [real workflow fixture](#).

Chapter 5. Run blueprint

Run these in the directory containing `Blueprint.roc`, or set `BLUEPRINT_ROOT`.

Command	Meaning
<code>blueprint / blueprint gen</code>	Generate the Nix files under <code>.blueprint/</code> from the lock. A project with no Nix package source has none, and nothing is generated
<code>blueprint shell [NAME]</code>	Enter the environment of the shell alias <code>NAME</code> (default <code>default</code>): through <code>nix develop</code> when it has a tool from a Nix source, and otherwise in the caller's own shell
<code>blueprint run TASK [-- ARGS...]</code>	Run the task's argv in its environment, with the extra arguments appended
<code>blueprint build NAME</code>	Snapshot and build an artifact plus dependencies in Nix's sandbox; print its actual store path
<code>blueprint workflow NAME</code>	Execute an ordered task/build workflow, stopping on failure
<code>blueprint tasks</code>	List tasks and their environments
<code>blueprint update</code>	Write <code>Blueprint.lock</code> : the digest of every fetched program on every declared system, and the pins of package sources, inputs and build sources. See What update needs and records
<code>blueprint update NAME...</code>	Update only the named package sources, inputs, build sources and fetched programs, keeping everything else of a lock that is current for <code>Blueprint.roc</code> . Local path: inputs are read again either way. A named program is resolved again whether or not its URLs changed
<code>blueprint requirements [--json] [TASK...]</code>	Say what the named tasks need, or every task: a provider, fetched programs, tools of the machine. See What tasks need

Command	Meaning
<code>blueprint check</code>	Run the compiler check and validate all shell/task/build environments without requiring a lock
<code>blueprint spec / blueprint flake</code>	Print the Spec or the generated flake
<code>blueprint --help</code>	Project help; subcommand help lists tasks, aliases and builds

5.1. What a run does, in order

`blueprint run`, `shell`, `workflow` and `build` do the same things in the same order, and each stops the command before the next begins:

1. Read `Blueprint.roc` with the Roc compiler, and refuse what the configuration alone rules out, such as a task that does not exist or a program its recipe says is unavailable on this system.
2. Read `Blueprint.lock` and require that it matches `Blueprint.roc`.
3. Plan every step of the request. All problems are reported together: a program the lock does not hold, one whose URLs changed, a build this machine cannot run.
4. Look on `PATH` for every program the steps need of the machine: the tools their environments `Expect`, and `nix` when a step has a tool from a Nix source or is a build. Everything missing is named in one message.
5. Make every fetched program ready in the store, downloading what is not there and checking each against the lock.
6. Run the steps. Each gets its Roc packages published and its fetched programs first on `PATH`.

Nothing is downloaded, generated or run for a request that fails an earlier check. [Work without Nix](#) describes the store and the `PATH` of a step.

5.2. What each command validates

Provider capability checks follow the requested environment closure; unrelated valid provider declarations do not block selected operations. Whole-project structural validation and required-feature checks still apply. Full rendering (`gen`, `update`, `check`, `flake`) checks every shell, task and build environment that Nix builds; one with no tool from a Nix source is entered without Nix and is left out of the flake.

5.3. The lock and generated files

- **Commit** `Blueprint.roc` and the authority (default `Blueprint.lock`, a versioned S-expression listing each fetched program with its digest and each pinned source with its revision and digest); generated state (default `.blueprint/`) is not committed: `blueprint` writes a `.gitignore` there that ignores everything in it.
- Normal `gen`, `shell`, `run`, `build` and `workflow` require a lock that matches `Blueprint.roc` and never rewrite it. A lock that does not match says what changed, and one in a format this `blueprint` does not read is refused; both name `blueprint update`.
- A fetched program is stale by itself. When its URLs in `Blueprint.roc` are not those in the lock, only requests that use it fail, and `blueprint update <name>` resolves it alone.
- `nix develop` takes the bash it runs a shell or task with from a flake input named `nixpkgs`, so the generated flake gives that name to the first Nix package source declared and that bash is pinned by `Blueprint.lock` like every tool, unless the project declares an input named `nixpkgs` itself.

5.4. What tasks need

`blueprint requirements` reads `Blueprint.roc` and nothing else: not `PATH`, not the store and not `Blueprint.lock`. It answers before anything is installed or resolved, and gives the same answer on every machine. The system is `BLUEPRINT_TARGET`, or the one it runs on.

```
$ blueprint requirements
system x86_64-linux
providers: nix
task check [dev]: programs roc-stable; expects gh, git; roc packages 2
task host [zig]: provider nix (nix#zigpkgs."0.16.0"); programs roc-stable; expects gh, git; roc
packages 2
```

With `--json` the report is one JSON object for a program to read:

```
{"format":1,"system":"x86_64-linux","providers":["nix"],"expects":["gh","git"],"tasks":[
  {"name":"check","environment":"dev","providers":[],"tools":[],"programs":["roc-stable"],
  "unavailable":[],"expects":["gh","git"],"roc_packages":2},
  {"name":"host","environment":"zig","providers":["nix"],"tools":["nix#zigpkgs.\"0.16.0\""],"programs":
  :["roc-stable"],"unavailable":[],"expects":["gh","git"],"roc_packages":2}]}
```

Field	Meaning
<code>format</code>	The number of this shape, 1. Every field below is always present. A change a reader of format 1 could misread gets another number.

Field	Meaning
<code>system</code>	The system the report is for.
<code>providers</code>	The providers that must be on the machine to run the reported tasks on that system, each once. <code>"nix"</code> is there exactly when one of those tasks has, in its environment on that system, a tool from a Nix package source. A Nix source that is declared and that no reported task uses does not put it there. Neither does a shell, a build or <code>blueprint update</code> : they are not tasks, a build always needs Nix, and an update needs it whenever the project declares something Nix resolves.
<code>expects</code>	The programs those tasks expect the machine to have already, each once.
<code>tasks</code>	One object for each task, in the order asked, or in the order of <code>Blueprint.roc</code> when none is named.
<code>tasks[].providers</code> <code>, tasks[].tools</code>	The providers that task needs, by the same rule, and the tools they supply, as <code>Blueprint.roc</code> writes them.
<code>tasks[].programs</code> , <code>tasks[].unavailable</code>	The programs its environment fetches on that system, and those it lists that do not exist there. A task with an unavailable program cannot run on that system.
<code>tasks[].expects</code> , <code>tasks[].roc_packages</code>	The programs the task expects of the machine, and how many Roc packages its environment has.

A task that is not declared is an error, and nothing is printed for the others. To install Nix only where it is needed, test whether `providers` holds `"nix"` for the tasks a job runs:

```
blueprint requirements --json check | jq -e '.providers | index("nix")'
```

5.5. What update needs and records

`blueprint update` is the one command that decides what a project uses. It writes the whole lock or nothing.

5.5.1. Without Nix, and with it

`blueprint update` starts Nix only when `Blueprint.roc` declares something Nix resolves: a Nix package source, an input or overlay, or a build source, including the bundle of a Roc package that a build uses. It looks for `nix` before doing anything else and says why it needs it:

```
blueprint: this machine lacks what blueprint update needs, and blueprint installs none of it:
nix: Blueprint.roc declares the Nix source nix, which Nix resolves
```

A project that declares none of these is updated with no Nix on the machine. In a project that does, the pins Nix resolves and the digests of fetched programs share the one lock, and `blueprint update PROGRAM...`, naming fetched programs alone, asks Nix for nothing: it needs a lock that is already current for everything else. A name that is neither an input nor a program is an error that lists both kinds.

5.5.2. The digests of fetched programs

For each fetched program and each declared system, the lock holds the URLs the program is declared with, the SHA-256 and the length of the download, and how the digest was obtained:

obtained	Meaning
Computed	<code>blueprint update</code> downloaded the bytes and hashed them. Nothing else was to be had.
Published	The release that serves the download reports this digest, and the bytes were not downloaded. A run on that system checks its download against it.
Both	It hashed the bytes, and the release reports the same digest and length.

A reviewer of a lock change reads these: `Computed` alone rests on the one download `blueprint update` made, and `Published` on what the publisher's release says.

An entry whose URLs are unchanged is kept, and nothing is asked or downloaded for it; a program named on the command line is resolved again regardless. Otherwise the download for the system the update runs on is fetched and hashed, and is kept in the store for the run that follows. A program that is the very download `blueprint` fetched as its own compiler, as a `Recipes.roc_nightly` of that nightly is, is hashed from the store and not fetched a second time.

When the first URL is a GitHub release asset,

`https://github.com/<owner>/<repo>/releases/download/<tag>/<asset>`, the release is asked once what it publishes. A digest computed here that differs from the published one stops the update, and nothing is written. For another system the published digest is recorded and nothing is downloaded; an asset with no published digest, or a URL that is not a release asset, is downloaded and hashed instead.

5.5.3. When the release cannot be asked

The question to GitHub carries no token, and GitHub answers 60 such requests an hour from one address. A question that is refused, that cannot be put, or whose answer does not describe a release does not stop the update: `blueprint update` says so once for the release, downloads and hashes that release's assets for every system, and records each as `Computed`. A digest that is published and differs from the computed one always stops it. A run never asks GitHub anything.

```
blueprint: could not ask for the digests roc-lang/nightlies publishes for its release nightly-2026-10-13-b17d287: HTTP 403 from https://api.github.com/repos/roc-lang/nightlies/releases/tags/nightly-2026-10-13-b17d287 (1 attempt). GitHub answers 60 such requests an hour from one address when no token is sent, and blueprint sends none. The digests of its downloads are computed by downloading each of them instead, and Blueprint.lock records them as computed
```

Such an update downloads the program for every declared system, where one that is answered downloads it for this machine alone.

5.5.4. New download hosts

After writing the lock, `blueprint update` prints every download host the new lock names and the lock before it did not, one to a line:

```
Blueprint.lock now names download hosts it did not name before:
github.com
```

It prints nothing when there is none. A new host in a lock update is worth a reviewer's look.

5.6. Environment variables

- `BLUEPRINT_ROOT` names the directory containing `Blueprint.roc` (default: the current directory).
- `BLUEPRINT_WORKSPACE`, `BLUEPRINT_GENERATED_ROOT`, `BLUEPRINT_LOCK` choose caller paths; relative values resolve against the selected project root, not the invocation directory. Out-of-tree generated roots are supported.
- `BLUEPRINT_TARGET` selects a declared target (default: this machine's System).
- `ROC` selects the compiler that reads `Blueprint.roc`, which must be the nightly `blueprint` was built for; unset, a matching `roc` on `PATH` is used, or the nightly is downloaded into the store.
- `XDG_CACHE_HOME` names the directory that holds the store of downloads, `blueprint/`, and Roc's package cache, `roc/packages` (default: `.cache` in `HOME`).
- `SHELL` names the program `blueprint shell` starts for an environment with no tool from a Nix source (default: `/bin/sh`).

- `PATH` is where expected tools, `nix`, `tar`, `ln` and `chmod` are found.

No variable selects a provider or turns Nix on or off: `Blueprint.roc` alone decides what a request needs. No request `blueprint` makes carries a token or any other credential, and it reads none from the environment.

Chapter 6. Work without Nix

Nix is one place a project's tools can come from, and `blueprint` starts it only for what a project takes from it. A project that fetches its programs, expects the rest of the machine and declares no build runs every task with no Nix installed. This chapter says what such a project looks like, what each command then does, and where the downloads are kept.

6.1. A project that never starts Nix

```
app [config] { pf: platform "<the platform URL of a release>" }

import pf.Recipes

config = [
  Name("my-package"),
  Environment("dev", [
    Programs([Fetch("roc-stable", Recipes.roc_nightly("nightly-2026-10-13-b17d287"))]),
    Expect(["git"]),
    RocPackages([
      "https://github.com/roc-lang/basic-cli/releases/download/0.26.0-rc2/Gj3zVDr3qrQa6aTh7Z1zgUfe9NeBE6R76T5ho3xrc8af.tar.zst",
      "https://github.com/roc-lang/http/releases/download/1.0.0/6ZUwqYhCS8PU9Mo6MF7oV82ET2o7KYb57CLKDq4cq4sS.tar.zst",
    ]),
  ]),
  Shell("default", [Use("dev")]),
  Task("check", [Use("dev"), Run(["scripts/check.roc"])]),
]
```

Nothing in this file is Nix's: there is no `Packages`, `Overlay`, `Input`, `Source` or `Build`, and no tool written `"<source>#<name>"`. The compiler comes from a release by a digest in `Blueprint.lock`, `git` is the machine's, and the Roc packages are downloaded by the URLs that name their hashes.

What starts Nix is decided by `Blueprint.roc` alone, never by a variable, a flag or what happens to be installed:

Request	Starts Nix when
<code>blueprint run TASK</code> , <code>blueprint shell</code>	The environment has, on this system, a tool from a Nix package source: <code>Tools</code> , <code>ToolsFor this system</code> , <code>Command</code> or <code>ToolWith</code> . An overlay the environment selects does not start Nix by itself.
<code>blueprint workflow NAME</code>	One of its tasks does, or it has a build.
<code>blueprint build NAME</code>	Always. A build runs in Nix's sandbox.
<code>blueprint update</code>	<code>Blueprint.roc</code> declares something Nix resolves: a Nix package source, an <code>Input</code> or <code>Overlay</code> , a <code>Source</code> , or a Roc package that a build's environment has. An update that names only fetched programs never does.
<code>blueprint check</code> , <code>spec</code> , <code>tasks</code> , <code>requirements</code> , <code>gen</code> , <code>flake</code>	Never.

`blueprint requirements` gives the first row's answer for a project's tasks without running anything; see [What tasks need](#). Where Nix is needed and `nix` is not on `PATH`, the command stops before it downloads or runs anything and names the tool that asked for it:

```
blueprint: this machine lacks what the task needs, and blueprint installs none of it:
nix: environment host takes nix#zigpkgs."0.16.0" from the Nix source nix
```

6.2. What each command does without Nix

Command	What happens
<code>blueprint update</code>	Records in <code>Blueprint.lock</code> the SHA-256 and length of every fetched program on every declared system. It downloads this machine's programs and keeps them in the store; see What update needs and records .
<code>blueprint run TASK</code>	Looks on <code>PATH</code> for every program the environment expects, and stops with all that are missing in one message. Then makes each fetched program ready in the store, puts the environment's Roc packages where Roc looks for them, and runs the task's command from the project root with the fetched programs first on <code>PATH</code> .
<code>blueprint workflow NAME</code>	The same for each task in order. Everything every step needs is checked before the first step runs.

Command	What happens
<code>blueprint shell [NAME]</code>	Does the same and then starts the caller's own shell, the program <code>SHELL</code> names, or <code>/bin/sh</code> when it is unset, with the fetched programs first on <code>PATH</code> . Leaving that shell ends the command.
<code>blueprint / blueprint gen,</code> <code>blueprint flake</code>	Generate nothing: the project has no flake. <code>blueprint flake</code> prints nothing on standard output and says so on standard error.
<code>blueprint check, spec, tasks,</code> <code>requirements</code>	Work as everywhere. They read <code>Blueprint.roc</code> and nothing else.

A task still needs a `Blueprint.lock` that matches `Blueprint.roc`: the digests are in it. Evaluating `Blueprint.roc` needs the Roc compiler `blueprint` was built for, which it downloads into the same store when `ROC` is unset and no matching `roc` is on `PATH`.

6.3. The store of downloads

Everything `blueprint` downloads for itself is kept in one per-user directory, named by content:

```
<store>/v1/artifacts/sha256/<hex>      the downloaded bytes
<store>/v1/trees/sha256/<hex>.<kind>/  those bytes, made usable
```

`<store>` is `blueprint` under `XDG_CACHE_HOME`, or under `.cache` in `HOME` when that is unset; a relative value is refused. `<kind>` is `file`, `tar.gz`, `tar.xz` or `zip`: a single file made executable, or an archive unpacked whole, because a program may need what lies beside it. On macOS `roc` links with the `darwin` directory of its archive.

An artifact is never trusted

It is hashed again before every use, wherever it came from: a download, an earlier run, a restored cache or a hand that put it there. One whose bytes are not those its name says is removed, with every tree made from it, and downloaded again. A download that does not have the length and the SHA-256 in `Blueprint.lock` is not kept and nothing is unpacked from it.

A tree is derived state

It is checked once, when it is made from an artifact that has just been verified, and is not verified again. Whoever can write to the store can change a tree. In CI, cache `v1/artifacts` alone, keyed on `Blueprint.lock`: trees are rebuilt from it, and a cached artifact is re-hashed like any other.

An archive is listed before it is unpacked

An archive that holds an absolute name, a `..` segment, a name that is not plain text, a link that

leaves the tree, or anything that is not a file, a directory or a link, is refused whole, and so is one that lacks the program the recipe names.

Two runs at once are safe

Each file and tree is written under a fresh name beside its final one and renamed into place, so no half-written file is ever seen and two processes that fetch the same thing both succeed.

There is one gap: a run that finds an artifact with the wrong bytes removes its trees, and another process that is running a program from such a tree at that moment loses it.

The store may be deleted at any time; the next run fetches what it needs again. The files of a tree are read-only and its directories are not, so an ordinary `rm -r` removes it. A Roc nightly is about 67 MB as an artifact and about 220 MB unpacked.

6.4. How a download is made

`blueprint` downloads over HTTPS itself. It sends no token, cookie or other credential, and follows at most five redirects, each of which must be HTTPS. A request that fails in transit or is answered 408, 429 or a 5xx status is tried three times; a 404 is final at once. When a download has alternate URLs (`Mirrored`), they are tried in order, and the message for a program no URL served lists what each one answered. A whole download is held in memory while it is hashed.

Roc packages are not kept in this store. They go into Roc's own package cache; see [Lock the Roc packages scripts use](#).

6.5. What is on PATH

For each step, `blueprint` makes a directory `run/run-<random>/bin` under the generated directory (`.blueprint/` unless you moved it), holding one symbolic link for each fetched program, named as `Fetch` names it and pointing at the program inside its tree. That directory is put first on the step's `PATH`, ahead of everything the caller had, and is removed when the step ends, however it ends. A task whose command is the name of a fetched program is started from that link whatever else is on `PATH`.

Expected programs are not put anywhere: they are found on the caller's `PATH`, which the step inherits after the links. In an environment that also has Nix tools, `nix develop` is started with that `PATH` and puts its own tools ahead of it.

6.6. Programs blueprint still starts

Three programs of the machine are used on this path, besides the compiler and whatever the project expects:

Program	Used for
<code>tar</code>	Listing and unpacking an archive. A <code>Zip</code> unpacks only where <code>tar</code> is <code>bsdtar</code> , as on macOS; GNU tar does not read a zip.
<code>ln</code>	The symbolic links of a step's <code>bin</code> directory on Linux and macOS.
<code>chmod</code>	Making a downloaded <code>File</code> executable and a tree's files read-only on Linux and macOS.

6.7. What still needs Nix

- A sandboxed build, always, and only on `x86_64` Linux. A build's environment may not have fetched programs or expected tools: a sandbox holds nothing of the machine.
- Any tool from a Nix package source, in the environments that have one. Other tasks of the same project run without Nix.
- `blueprint update` in a project that declares anything Nix resolves. Adding the first Nix source to a project therefore makes its lock stale for every task until someone with Nix runs `blueprint update` and commits the result; after that, the tasks that use no Nix tool again run without it.

6.8. Windows

The release includes `blueprint-x86_64-windows.exe` and the platform's `x64win` target. The `no-nix` Cl job checks fetching the compiler, writing a lock, reporting requirements, and building and running a `basic-cli` program.

The store uses `LOCALAPPDATA` when `XDG_CACHE_HOME` is unset. `Recipes.roc_nightly` selects the nightly's zip and `roc.exe`; the Windows `tar` reads zip archives. Fetched programs are exposed with hard links, or copies when the store is on another drive, and need no `ln` or `chmod`.

Include `"x86_64-windows"` in the project's `Systems`: the default list contains Linux and macOS only, and fetched programs are locked for the declared systems.

Windows does not execute a script's shebang. Name its interpreter in the task: `Run(["roc-stable", "hello.roc"])`. Nix tools and sandboxed builds remain unavailable. CI checks Windows Server; Windows 10 and 11 desktop installations have not been verified here.

Chapter 7. Learn from the examples

Four complete projects live in `examples/`. In the repository they use the platform in the working tree, so they follow `main`. To start a project of your own, download `roc-blueprint-examples-<version>.zip` from a [release](#): the same projects with that release’s platform URL, as they were tested for it, and the Spec each one prints.

all-settings

Two Nix package sources, each tool written with its source, a named overlay scoped to `dev`, parent-first environment inheritance, shell aliases, argument-list tasks, a Roc package, two fetched programs (`Recipes.roc_nightly` and `Recipes.file`), an expected tool and raw Nix settings. `ci` uses `base` without the overlay.

composition

Imports `ProjectTasks.roc`, whose pure `settings : EnvName → List(Config.Setting)` function supplies `fmt`, `test` and `args` tasks. The application concatenates them into `config`; all use `dev` directly. Inherited `git` is deduplicated before the child’s `python3`. The tasks print tool versions or their arguments as JSON; they do not format files or run a test suite. No plugin registration, generated command line, implicit tool installation or runtime code loading is involved. It also imports `Releases.roc`, whose `blueprint : Str → (System → Config.Download)` is a recipe written by hand: given a release, a function from the System to where that release’s `blueprint` is downloaded, or `Unavailable`, which a `Fetch` in `Programs` takes.

artifacts

A complete local locked source, a sandboxed library and application that depends on it, and a typed workflow of tasks and builds. Its [README](#) shows usage and output, including explicit update and the read-only source and artifact locations.

extensions

Custom extension blocks and raw values. The platform emits them, and `blueprint check` refuses the project with `needs features: extensions`, because no provider implements them. That refusal is what the example shows.

7.1. Run an example

In an unpacked examples archive, with the `blueprint` binary of the same release:

```
cd composition
blueprint check
blueprint update
blueprint run fmt
blueprint run args -- 'two words' '' '--literal'
```

Only an explicit `update` creates or changes `Blueprint.lock`. `gen`, `shell`, `run`, `build` and `workflow` require a lock that matches and leave it as it is; the generated `.blueprint/flake.lock` is derived from it. Every example declares a Nix package source, so `blueprint update` needs Nix in each; [Work without Nix](#) has a project that needs none.

In a checkout of the repository the examples need the platform host built first; see [Build and test](#).

Chapter 8. Understand the architecture

This page explains how Blueprint is designed and why. It is written for contributors and provider authors who already know what a development environment is and want to understand the shared vocabulary, the layers, the stages between them, and the rules that keep Blueprint independent of any one package ecosystem or use case. After reading it, you should be able to judge whether a proposed change fits the architecture.

This is an explanation of the intended design. It does not describe the progress of any particular implementation.

8.1. Why Blueprint has a core model

Blueprint turns a typed description of a development environment into a working, reproducible environment. Many tools do this for one ecosystem, and in doing so they tie the user's intent to that ecosystem's formats and commands. Blueprint separates the two.

It adopts proven ideas from reproducible build systems: pinned inputs, content addressing, fetches checked against a known digest, and substitution from caches. Every ecosystem-specific concern belongs to a replaceable **provider**. The same Blueprint can therefore be realised through different package ecosystems, or through Blueprint's own store, on macOS, Linux and Windows.

The protocol is the product; providers are adapters.

8.2. Terms

The vocabulary has three parts: **data** (the things Blueprint records and produces), **stages** (the steps that turn one kind of data into another) and **roles** (who carries out each step).

8.2.1. Data

Term	Definition
Blueprint	The authored source: what the user <i>intends</i> .

Term	Definition
Spec	The normalised, validated, provider-neutral form of a Blueprint. Every stage after evaluation consumes it.
Component	One thing a Spec asks for: a package identity plus a version constraint.
Source	A pinned universe that components are chosen from, such as a revision of a package collection, a channel snapshot or a registry commit.
System	A target platform: operating system, architecture and C library.
Lock	The <i>decisions</i> : the exact pins chosen for each Source and Component, with their digests and how they were verified.
Artifact	Fetches bytes, identified by digest and never modified.
Tree	An unpacked, content-addressed artifact in a Store.
Store	The local collection of artifacts and trees on one machine.
Host facts	What is observed about the machine rather than provided by Blueprint, such as the system C library version, an installed platform toolchain or a program a Blueprint says it expects to find there.
Environment	The realised set of trees and host facts for one shell on one System.
Activation	Data that describes how to enter an Environment: search-path and environment-variable operations. It contains no shell text.
Report	A derived, read-only view of committed data, such as a change summary, a software bill of materials or an attestation.

Tool selections in a Spec may name a System. Evaluate records those selections as intent, so Realise includes them only for that System without inferring availability from a provider or the host.

8.2.2. Stages

Stage	Responsibility
Evaluate	Turn a Blueprint into a Spec, rejecting invalid intent.
Resolve	Decide exactly what to use and whether to trust it. This is the <i>only</i> stage that makes choices.
Fetch	Obtain the exact bytes the Lock names, and prove that they match.

Stage	Responsibility
Realise	Turn verified artifacts into trees and an Environment.
Activate	Describe how to enter the Environment, as data.
Present	Render an Activation for one consumer: a shell, CI, an editor, an agent or a container.
Inspect	Derive Reports from Specs and Locks without touching a Store.

8.2.3. Roles

Core

Owns the data types, the invariants, the **plan** of every request and the **effect executor**. The Core is the only part that performs I/O. What a Blueprint declares as a download with a known digest needs no ecosystem at all: the Core plans it, fetches it, keeps it in the Store and puts it in the Environment itself, and it checks the host facts a Blueprint declares.

Provider

A pure plug-in that implements one or more stages for one ecosystem. The Core asks a provider only for what comes from that provider's Sources, one Environment at a time; a request that needs nothing of an ecosystem involves no provider. The kinds are:

- *resolver provider*, which chooses pins from an ecosystem's Sources;
- *realiser provider*, which builds trees locally or through remote execution;
- *presenter*, which renders an Activation for a shell, a CI system, a machine-readable format or a container definition;
- *inspector*, which derives Reports such as change summaries and audits.

Blueprint says **provider** rather than **backend** on purpose. A backend suggests one component that owns the whole pipeline from end to end, which is exactly the coupling this model is designed to avoid.

8.3. Layers

Each kind of data belongs to one layer. A layer is defined by the question its data answers, how long that data lives, and which stages may change it.

Layer	Data	Question it answers	Lifetime	Changed by
Intent	Blueprint, Spec	What do we want?	committed	people
Decision	Lock, Sources	What exactly did we choose, and why do we trust it?	committed, reviewed	Resolve only
Materialisation	Artifact, Tree, Store, Environment	What exists on this machine?	disposable cache	Fetch, Realise
Interface	Activation, renderings	How is it used?	ephemeral	Activate, Present
Cross-cutting	Reports	What changed, and what is in it?	derived	Inspect

The two committed layers, Intent and Decision, **are the protocol**. Everything below the Lock can be replaced by another provider and rebuilt from the Lock.

8.4. Flow

The diagram shows how data moves through the layers. Solid arrows are stages that produce new data; each arrow is labelled with its stage name. Dotted arrows are read-only: Inspect reads committed data, and a remote cache can supply or receive trees. Each box is labelled with its layer, so the diagram does not depend on color.

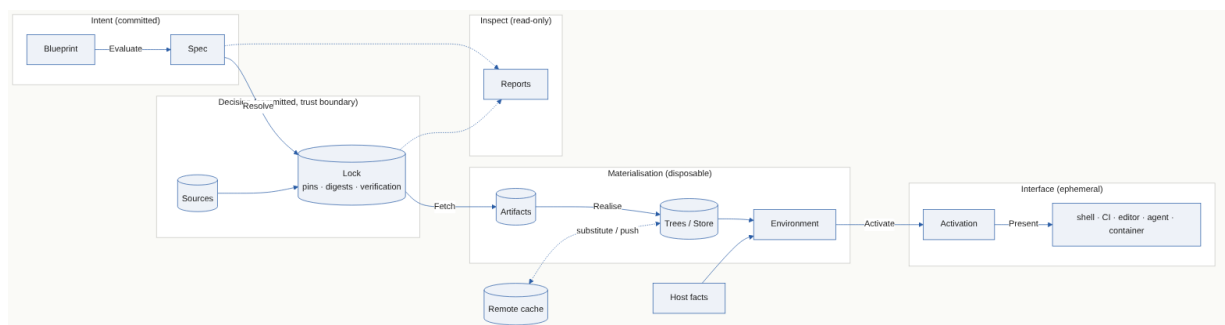


Figure 1. Data flow through the Blueprint layers

The Lock is the trust boundary. Everything to its left expresses and records decisions; everything to its right follows those decisions without making new ones.

8.4.1. Stage responsibilities

Stage	Input	Output	Effects	Performed by	Must hold
Evaluate	Blueprint	Spec	none	configuration layer	Deterministic. No provider concepts outside namespaced hints.
Resolve	Spec, Sources, previous Lock	Lock	network reads, signature checks	the Core for declared downloads; a resolver provider, planned by the Core, for its Sources	The only stage that makes choices. Trust is recorded, never assumed.
Fetch	Lock	Artifacts	network, Store writes	Core, with mirrors and caches	Each artifact's digest equals the digest in the Lock.
Realise	Artifacts, System, host facts	Trees, Environment	Store writes, possibly process execution	the Core for what it fetched; a realiser provider, planned by the Core, for an Environment that takes components from its Sources	A function of its inputs only. No network access and no new choices.
Activate	Environment	Activation	none	Core or realiser provider	Produces data only, never shell text.

Stage	Input	Output	Effects	Performed by	Must hold
Present	Activation	a rendering	none, except running a command when asked	presenter	Each presenter does its own escaping.
Inspect	one or more Specs and Locks	Report	none	inspector	Never reads a Store, so it works with every provider.

8.4.2. Commands as flows

Every user-facing command is a path through the same stages. No command needs a stage of its own.

Kind of command	Flow
Update the pins	Evaluate, then Resolve. The new Lock is committed and reviewed as a diff.
Enter, run in or build an environment	Evaluate, Fetch, Realise, Activate, then Present. With a warm Store this works offline.
Check or describe a project	Evaluate, then Inspect.
Review a proposed change	Evaluate and Inspect on two revisions, a base and a head.
Audit what is used	Inspect the Lock.

8.5. Invariants

These rules keep the model neutral and reproducible. A design that breaks one of them breaks the architecture.

8.5.1. 1. Intent is neutral

A Spec contains no provider concepts except inside namespaced hints.

8.5.2. 2. Only Resolve makes choices

Every later stage is a pure function of the Lock, the System and the host facts. This is Blueprint's reproducibility guarantee.

8.5.3. 3. The Lock is the single source of truth and the trust boundary

- No later stage changes a pin, URL or digest.
- Trust decisions happen during Resolve and are recorded in the Lock.
- An update is always a visible, reviewable diff.

8.5.4. 4. Content is immutable and verified before use

Caches and mirrors are never trusted. Everything they return is checked against the Lock.

8.5.5. 5. Materialisation is disposable

Deleting a Store or an Environment loses nothing that the Lock cannot rebuild.

8.5.6. 6. Host dependence is explicit

Anything Blueprint does not provide is a declared host fact that Reports show. Nothing about the host is assumed silently.

8.5.7. 7. Providers are pure

- Providers return steps, and the Core executes them.
- A provider can cause only the effects the Core grants it.
- The Core knows providers only through the provider interface.
- Which provider a request needs follows from the Spec alone. Nothing ambient, such as a variable or what is installed, selects one.

8.5.8. 8. Activation is data, and rendering happens last

No stage before Present produces shell text.

8.5.9. 9. Inspectors read only committed data

If a Report needs anything else, the Lock is incomplete, and the fix belongs in the Lock.

8.5.10. 10. One codec, one versioning scheme

- Spec, Lock and Activation share one codec and one set of major/minor compatibility rules.
- Other formats, such as JSON, TOML and shell scripts, are *renderings*, never protocol data.

8.6. The coupling test

A new use case should need only a new provider or a new rendering. It should never change the Core or the committed data.

Use case	What it needs	Core and Lock unchanged?
Developer shell	presenter	yes
CI system	presenter	yes
Editor or AI agent	presenter with machine-readable output	yes
Container or development container	presenter	yes
A new package ecosystem	resolver and realiser providers	yes
A machine with no package ecosystem installed	nothing: the Core fetches declared downloads itself	yes
Remote execution	realiser provider	yes
Bill of materials or change review	inspector	yes

A proposal breaks the architecture, and needs redesigning, if it requires any of these:

- a provider-specific field outside hints, which breaks [invariant 1](#);
- a choice made after Resolve, which breaks [invariant 2](#);
- an inspector that reads a Store, which breaks [invariant 9](#).

Chapter 9. The Spec, the Lock and providers

This chapter is for someone reading or producing a Spec, writing a provider, or changing how blueprint plans its work. [The architecture chapter](#) defines the terms.

9.1. What the Spec holds

`roc Blueprint.roc` prints the Spec as an S-expression: records are `((field value) ...)`, lists `(a b)`, tags `Tag` or `(Tag payload ...)`, and `;` starts a comment. `blueprint-core/Spec.roc` documents every field.

In outline:

- `format` — `major 2` (`minor 7`; see [Compatibility of the formats](#)).
- `name`, `systems` (strings such as `"x86_64-linux"`).
- `sources` — `{ name, provider }`, where `provider` is `NixPackages(Str)` or `GuixPackages(Str)`. Every source a tool names is declared: validation adds none, and a project whose environments have no tool of a package source has no source. `Auto` is still read, so that a Spec written before 0.6 is refused by name; no Spec is written with it.
- `inputs` — `{ name, url, kind }`, where `kind` is `Overlay` or `Flake`.
- `environments` — `{ name, parents, tools, overlays }`; tools are `{ source, name }`, where `source` is a declared source, and overlays are ordered input names. Validation resolves a single parent, deduplicates parent-first selections and clears `parents`.
- `system_tools` — optional `{ environment, system, tools }` selections. They inherit parent first and only enter the environment on their named `System`.
- `commands` — optional `{ environment, name, tool }` launchers. Validation flattens inheritance; a child replaces a same-named inherited command.
- `configured_tools` — optional `{ environment, tool, options }`: a tool and the options its package is built with, an `Attrs` value. Validation flattens inheritance; a child replaces an inherited tool's options, and a tool an environment also has plain is rejected.
- `roc_packages` — optional `{ environment, name, source }`: a Roc bundle's content hash and the `build_sources` entry that names its URL. The platform derives both from the bundle URL (`roc-<hash>, tarball+<url>`); environments sharing a bundle share one source. Validation flattens

inheritance, parent first. The Core fetches the bundles of a task or a shell itself. A provider resolves and locks such a source only when a build's environment has the package (`Project.resolved_build_sources`), so Roc packages that tasks and shells alone use need no provider.

- `programs` — optional `{ name, system, urls, unpack, path }`: a program downloaded for one System, under the command name `name`. `urls` is a nonempty list, the primary first and then alternates for the same bytes, each once. `unpack` is `File`, `TarGz`, `TarXz` or `Zip`; `path` is the program inside the unpacked archive and empty for a `File`. A program with no entry for a System is unavailable there. The Lock records the digest; the Spec never does.
- `environment_programs` — optional `{ environment, programs }`: the declared programs an environment has on `PATH`, by name.
- `expects` — optional `{ environment, programs }`: programs the machine must already have on `PATH`. They are checked for, never installed. Validation flattens the inheritance of both, parent first and each name once, and rejects a name an environment has from two of a program, an expected tool and a command.
- `shells` — `{ name, environment }` aliases.
- `tasks` — `{ name, environment, run }`, with nonempty executable argv.
- `build_sources` — `{ name, ref }`, locked non-flake sources, separate from package-provider sources.
- `builds` — `{ name, environment, inputs, needs, run, output }`; named source and build references, exact argv and contained relative file/directory output.
- `workflows` — `{ name, steps }`; typed `RunTask(Str, List(Str))`, `BuildArtifact(Str)` or `RunWorkflow(Str)` steps, with ordered bounded expansion.
- `raw` — `{ backend, target, value }`, passed through to one provider (the wire field keeps the name `backend`).
- `extensions` — `{ kind, name, value }`, blocks a provider may understand.
- `requires` — features the config uses beyond the core (`"raw"`, `"extensions"`, `"sources"`, `"builds"`, `"workflows"`, `"system-tools"`, `"commands"`, `"roc-packages"`, `"configured-tools"`, `"programs"`, `"expects"`), so an older blueprint can say what's missing. The platform always writes `"explicit-sources"`: a blueprint from before 0.6 would give the Spec a default source it does not declare, and refuses it by that name instead. A blueprint given a Spec without it says the platform is older than 0.6.0.

Value is `Str`, `Int`, `Bool`, `List` or `Attrs`. Its S-expression encoder and parser are hand-written to avoid recursive-codec derivation problems. `requires` is reserved in Roc; the Roc field is `requires_`, and `Sexpr` drops a trailing `_` on the wire. There is no major-1 shell `packages` field.

`Lower.lower` checks authoring-setting cardinality, asks the recipe of each `Fetch` in an environment's `Programs` once for every declared System and keeps its answers as `programs` (the function never

reaches the Spec; a program listed by several environments is there once, and two recipes under one name that answer differently for the declared Systems are refused), then calls `Project.validate` as a top-level constant. The CLI calls that same semantic validator after `Spec.parse` and required-feature checks, so external Spec does not bypass reference or graph rules. `Spec.parse` rejects other majors before decoding nested records; parsing alone does not establish semantic validity.

Static checks cover generic references, duplicates, inheritance cycles, `argv`, download URLs (`https://<host>/<path>` with no credentials, query or fragment; `http://127.0.0.1` is accepted so that a test can serve one) and each source's own grammar for the names of its tools. A tool that names no declared source is refused with the sources that are declared. Package existence and target availability are Nix runtime checks. Core validation never probes the host or installs/fetches anything.

9.2. What the Lock holds

`Blueprint.lock` is a `blueprint-core/Lock.roc` value in the same S-expression codec, with its own `format` (currently 1.1) and the same compatibility rules as the Spec. It holds `intent` (the parts of the Spec Resolve consumed: source and input declarations, the build sources a provider resolves and per-environment overlay order), which the CLI records on `update` and compares before every other command, failing with what changed; `sources` (one provider-neutral `{ name, provider, ref, rev, digest }` per declared input; `digest` is `sha256:<hex>`: the Nix provider converts its NAR hashes for remote inputs, and local inputs carry Blueprint's own tree digest from core `Tree`), `provider-namespaced hints`, and `fetches` (one `{ name, system, urls, sha256, size, obtained }` per declared program and System: the SHA-256, in hex, and the length of the download when it was resolved, the URLs it was declared with, and whether the digest was `Computed` from the bytes, `Published` by the release that serves them, or `Both`; an entry with no `obtained` is read as `Unstated`. Programs are not part of `intent`; each entry carries its URLs, so a program whose URLs changed is stale alone). A project that asks no provider for anything has a Lock with no `sources` and no `hints`. The Nix provider's hint carries its declared input identity and the complete native lock graph; decoding rejects a lock whose Sources disagree with those pins, so hand edits to either side fail. The graph's root holds one edge that is no declared input: `nixpkgs`, following the package source `Locks.shell_source` selects, which Nix writes for the alias in the generated flake (see Providers). Decoding accepts that edge only as the declarations determine it. A Lock written before the alias has no such edge and is still valid: `Locks.derive` adds it to the working lock, which changes no pin, so nothing needs `blueprint update` and the next update records it. A Lock that blueprint 0.5 or older wrote records, in its `intent`, the `Auto` source every project had then: it is read, found stale for every Spec, and reported as old. Older JSON locks are not migrated. Either way, run `blueprint update`. `fuzz/lock-parse` fuzzes the parser.

9.3. Compatibility of the formats

- The decoder accepts the same `major`, whatever the `minor`; consumers must still reject unsupported required features and semantically invalid data.
- **Minor** (compatible): a new optional top-level field (missing fields default to empty, unknown ones are ignored), or a new `requires` feature.
- **Major** (breaking): anything else, such as a new required field, a new field inside a nested record, or a new `kind` tag.

9.4. What a provider does

`Realise.plan(spec, request, system, layout, lock, providers)` is the Core's own pure plan of a request. An environment's fetched programs, the tools it expects of the machine and its Roc packages are planned from the Spec and the Lock alone, as `Present`, `Programs` and `RocBundles` operations. A provider is asked only for an environment that has a tool from a source it `realises`, and its steps then follow what the Core supplies. A task whose environment has no such tool is its own `argv`, and a shell with none has no `argv`: the executor starts the caller's shell. Every problem of every step is collected before one is reported, and a request with a problem yields no step. `Realise.requirements` answers, from the Spec alone, which providers a task needs and which programs it fetches and expects.

The executor checks every `Present` program of every step on `PATH` and reports all that are missing together; then makes every `Programs` entry ready in its store, where a download is kept by its SHA-256 and hashed again on each use; then runs the steps, each with a directory of links to its programs first on `PATH`, removed when the step ends.

`Request`, `Steps` and `Layout` are importable pure core types. `NixProvider.plan(project, request, target, layout, locks)` derives an ordered `Steps.steps` sequence, each holding action, files, exact `argv`, artifact metadata and materialization operations. `Request.Workflow(name)` uses the same atomic planner as standalone tasks/builds. Execute each step's operations, stage its files, then invoke its `argv`; stop immediately on failure. The entire plan must succeed before effects. The caller owns all effects; no provider registry or serialized config recipes are involved. `blueprint-core/Provider.roc` is the Core/Provider contract: a `name`, the `features` the provider implements, `render`, `preflight`, `realises` (which package sources it supplies), `realise` (Steps from the Lock) and `resolve plus lock_from_native` (the programs the machine must have, and what to stage and run to produce new pins). Resolve asks a provider only when the Spec declares something it resolves (`Realise.resolvers`); the digests of fetched programs are the Core's own to obtain. The CLI uses only that record; `scripts/test.roc` fails if it reaches into a provider's modules directly. Nix is the only implemented provider. It:

- supplies the sources declared as `NixPackages` and no other; a project that declares none has no flake, and nothing is generated for it;

- builds an environment that has no tool of its own, such as the sandbox of a build whose environment is empty, with the first Nix source the project declares, in the order of `Blueprint.roc`;
- imports each selected environment's package sources per system with only that environment's ordered overlay stack; aliases and task entries share it;
- checks provider compatibility for the requested environment/build closure, not unrelated valid source/environment declarations. `render` selects every build's environment and every shell or task environment that has a tool of a Nix source, and leaves out those the Core enters without Nix; `render_environment` selects one. Whole-project structure, required features, declared targets and Nix Raw remain checked;
- declares `nixpkgs` as an input that `follows` the first Nix package source the flake declares, whatever it is called, and nothing when the project declares an input of that name. `nix develop` takes the bash it runs from the input called `nixpkgs` and otherwise from the registry's floating one, which no Lock pins. The alias is one name for the whole flake, so it cannot follow each environment's own source;
- emits native package attributes without translation, fallback or availability filtering: missing/unavailable packages fail in Nix. Explicit `system_tools` select packages for their named System before Nix evaluates them. Each `commands` entry becomes a package holding one launcher that execs the tool's main program. Each `configured_tools` entry is the tool's `override` applied to its options, rendered as data like `raw`, in shells and builds;
- appends `Present` for `nix` to every step that starts it, with the reason, and none to a step that only generates files;
- gives a build its environment's `roc_packages`, since a build has no network: the build's specification lists the locked bundles and a `1n`, and `blueprint __build-runner` links them into a cache inside the build directory and sets `XDG_CACHE_HOME` for the build's command. A shell or a task gets nothing of them from this provider: the Core's `RocBundles` operation downloads the missing ones into Roc's package cache, with Nix or without. There is no shell hook;
- rejects unsupported Nix target declarations and restricts build requests to x86_64 Linux. Other supported output shapes are not execution evidence;
- renders `raw` for backend `"nix"` at `shell:<alias>` and `flake` as data, rejecting invalid targets, duplicate attributes and managed-field overrides. Alias Raw does not affect other aliases or tasks; other providers' Raw is inert;
- refuses any `extensions` and advertises `"raw"`, `"sources"`, `"builds"`, `"workflows"`, `"system-tools"`, `"commands"`, `"roc-packages"` and `"configured-tools"`;
- builds ordinary derivations with exact argv, filtered project snapshots, read-only declared sources/artifacts and checked file/directory outputs. Each is a raw `derivation` whose builder is the CLI's own executable, run as `blueprint __build-runner`: no shell or interpreter stands

between Nix and the user's argv, and the build's PATH is its environment's tools alone.

`Project.check_environment(project, Nix | Guix, name)` is a pure compatibility check. Guix source intent, native tool grammar and overlay-capability rejection are modeled; no Guix renderer, task implementation or executor exists. The reference CLI lists Nix as its one provider and selects nothing: `Realise.plan` asks a provider only for an environment that has a tool from a source it supplies, so which requests reach Nix is decided by the Spec.

The source package exports `NixProvider` and `Locks`, without importing the CLI or an effectful platform. `Locks.decode` parses the Lock, checks that its Sources match the "nix" hint's pins, and validates the native Nix graph; `Locks.derive` validates declaration/ordered-overlay identity and translates local project-relative paths into a disposable working lock. `NixProvider.plan` uses that translation. The former opaque-text `render_files` seam was removed, not retained as a bypass. `fixtures/consumer/main.roc` is an independent app using these APIs, including caller-selected paths, decoded supplied authority and exact argv; `scripts/test.roc` checks it and runs its expect's. `scripts/test_builds.roc` separately proves actual relocated local-source translation.

`gen`, `shell`, `run`, `build` and `workflow` require existing matching authority. Only explicit `update` resolves new pins and publishes authority; normal commands stage derivatives and prohibit native lock updates. Named input declarations remain stable across selected closures; selected overlays remain scoped and ordered. Local authority contains relative identity and Blueprint tree digests (core `Tree`, computed and verified by the CLI without Nix), not checkout paths. Dirty local inputs fail until explicit update. Local verification, caller observation and Nix's filtered project copy repeat per explicit build, never reusing artifact results by name across tasks.

A new feature usually means: a setting in the platform (`Config.roc`, `Lower.roc`), then either an `extensions` kind or a new optional Spec field (a minor bump), then support in a provider. Prototyping it as `Raw` or `Custom` first needs no Spec change at all.

Chapter 10. Contribute

Use issues for bugs and design discussion, and pull requests for changes. Report a suspected vulnerability privately, as [SECURITY.md](#) describes.

10.1. Find your way around

blueprint-platform/ *.roc	the roc-blueprint platform that Blueprint.roc apps use setting types, checked values, lowering to the Spec
host/, build.zig	Zig host, built into targets/<target>/libhost.a
targets/ README)	linker inputs, none committed: libhost.a is built, the rest fetched (see its README)
blueprint-core/ Project.roc	roc-blueprint-core: Spec, Provider contract, validation, Steps, Value, codec pure normalization, references and provider capability checks
Realise.roc	pure planning of a request: what the Core supplies, and what it asks of a provider
fuzz/	roc-fuzz targets: spec-parse, spec-round-trip, lock-parse
blueprint-nix/ NixProvider.roc	importable pure Nix provider (depends only on blueprint-core) shared pure request planning and flake rendering
Locks.roc	Nix pins <-> the Lock: Sources plus a "nix" hint
tests/	Spec fixtures and golden flakes
blueprint-cli/	the blueprint CLI (basic-cli + weaver); as 'blueprint __build-runner' it is also each build's in-derivation argv/output/isolation check
Store.roc, Download.roc	the per-user store of downloads and the HTTPS client that fills it
Bootstrap.roc	the recorded download of the compiler that reads a Blueprint.roc
Published.roc	what a release publishes as the digest of an asset, asked by update
fixtures/consumer/	independent consumer of the Spec and Nix packages
fixtures/overlays/	overlay flakes for scripts/test_scenarios.roc
examples/all-settings/	environments, sources, scoped overlays, tasks and Raw
examples/composition/	imported pure modules: reusable task settings and a program's recipe
examples/artifacts/	runnable source/dependency/workflow example and its Roc scripts
examples/extensions/	Custom blocks; CI checks blueprint refuses them clearly
scripts/	Roc scripts, run from the repository root; their modules are in scripts/src/ everything CI checks, in groups; runs the test_*.roc suites
test.roc	fetch or check the platform's linker inputs
link_inputs.roc	bundle core and the platform into dist/ and smoke-test the pair
bundle.roc	cross-build the CLI and its smoke-test program for each released system
build_release.roc	run a built CLI with no Roc and no Python of its own; with --no-nix, with no
smoke_binary.roc	
Nix either	
fuzz.roc	replay or fuzz the roc-fuzz targets
release_notes.roc	render a release's notes from docs/releases/
package_examples.roc	archive the examples with a release's platform URL
test_release.roc	test that archive against the bundles, served from localhost
docs_snippet.roc	write the part of the manual that names the latest release
check_version.roc	check a release's version before building and against the public API after

```

package_docs.roc      archive the rendered site
release_assets.roc    gather a release's assets and check the names other repositories depend on
test_builds.roc, test_workflows.roc  real-Nix build and workflow suites on
scripts/src/BuildHarness.roc
docs/                 the manual: index.adoc includes one chapter per top-level .adoc file
  releases/           each release's notes, included by releases.adoc
link-inputs.lock.json the linker-input release the platform links, pinned by content
flake.nix             builds blueprint with the pinned Roc; the development shell; the release
cross-compiler

```

The local platform and CLI share `blueprint-core`, including `Project.validate`. The Spec wire format is major 2 (currently 2.7); published major-1 bundles are not compatible. The architecture, terminology (Spec, Provider, Lock, stages) and invariants are defined in [the architecture chapter](#); keep it in sync with changes.

10.2. Set up

Full development and CI require x86_64 Linux with Nix and flakes enabled.

```
nix develop
```

gives two Roc compilers from [roc-overlay](#), Zig 0.16 and the programs the scripts start: `coreutils`, `git`, `curl`, `tar`, `gzip`, `zip`, `unzip` and `gh`. It has no Python. `roc` is the product compiler, the nightly in `blueprint-cli/compiler-version`: it builds and tests the CLI, the packages and the platform, and a released `blueprint` fetches it. `roc-stable` is the nightly `rocStableTag` names in `flake.nix`: the scripts in `scripts/` start with `#!/usr/bin/env roc-stable` and run on it, so they keep working while the product moves to a newer nightly. The scripts reach the product compiler through `ROC`, or `roc` on `PATH`. This shell is the one definition of the toolchain: every CI job runs its scripts through `nix develop -c`, so what passes in the shell is what CI runs. `blueprint` is not in the shell, so entering it never compiles the CLI; build it as below, or use `nix run .`

`blueprint-cli/main.roc` uses the released `basic-cli` platform by URL. Roc downloads it on first use outside Nix; the Nix blueprint package fetches the same archive by hash, so its sandboxed build needs no network.

10.3. Build and test

```

scripts/link_inputs.roc fetch          # the musl runtime files the platform links
(cd blueprint-platform && zig build)    # libhost.a or host.lib for all five hosts
roc test blueprint-core/main.roc       # Spec round trips and format tests
roc test blueprint-cli/main.roc        # the CLI's, the Nix provider's and every core expect
roc build blueprint-cli/main.roc --output=./blueprint
(cd examples/all-settings && ../../blueprint run --help) # try the CLI
scripts/test.roc                       # everything CI runs

```

```

scripts/test.roc unit cli      # or only some groups: static, unit,
                                # cli, scenarios, builds, package, glue, fuzz
scripts/fuzz.roc 300           # fuzz each target for 5 minutes

```

The `.roc` files in `scripts/` are Roc programs; they start other programs by argument list, not through a shell. Run one from the repository root with the pinned `roc` on `PATH`, or as `"$ROC" scripts/test.roc`.

The platform's linker inputs are not committed. `scripts/link_inputs.roc` `fetch` downloads the release pinned in `link-inputs.lock.json` into `.cache/link-inputs/`, verifies it and installs the files beside `libhost.a`; nothing that evaluates a `Blueprint.roc` against the local platform links without them. `scripts/link_inputs.roc` `check` verifies what is installed without the network. See [blueprint-platform/targets/README.md](#).

`scripts/test.roc` is the full CI entry point. Its groups are `static` (formatting; no tracked object file, archive or import library; the CLI's use of the Provider contract; the compiler downloads `Bootstrap.roc` records, one for each system the CLI runs on; each workflow's permissions and checkouts; the release notes chapter against the notes files; the flake's list of core modules and its version), `unit` (the platform's, the scripts', the CLI's and the consumer's `expect`s`, the configuration fixtures and the Roc the manual shows), ``cli`, `scenarios` (`test_scenarios.roc`, `test_roc_packages.roc` and the golden flakes parsed by Nix), `builds` (`test_builds.roc` and `test_workflows.roc`), `package` (the flake's package and development shell, the packaged CLI building an artifact with only Nix on `PATH`, and a dry run of what a release ships: core and the platform bundled and smoke-tested, the examples archived and the archive tested; see [The two bundles](#)), `glue` (`scripts/check_glue.roc`: the host's committed `roc_platform_abi.zig` is what `roc` `glue` writes for the platform with the product compiler; `scripts/check_glue.roc --update` rewrites it) and `fuzz` (each target's committed corpus, replayed, which takes seconds; `.github/workflows/fuzz.yml` fuzzes each target for five minutes every week, starting from the inputs earlier runs found, which it keeps in the Actions cache through `FUZZ_CORPUS`, and uploads any crashing input). `scenarios` and `builds` are the real-Nix suites, split into two groups of about the same length so that CI runs them as two jobs at the same time. `scripts/test_config.roc` runs the compiler on generated `Blueprint.roc` fixtures: the platform's own rules, which only a compiler run can show (a setting given twice or not at all, in `Lower.roc`, and each checked name's `from_quote`), a few core rules and bounds as witnesses that shared validation surfaces at compile time, and that settings composed from an imported module emit the same Spec as inline ones. Every other rule of `Project.validate` has an `expect` in `blueprint-core/Project.roc`. The bundle gate repeats a few of the fixtures against the served platform, plus the all-settings example. It then compiles the manual's Roc (`scripts/src/DocSnippets.roc`): every `[source, roc]` block of `docs/` and `docs/releases/`, and the first program of the Get started chapter, against the platform in the working tree; see [Write the manual](#). `scripts/test_cli.roc` exercises the built CLI with and without a configuration, checks validation and help/version handling, and records Nix `argv` to verify shell selection and task arguments without entering a shell. It holds what only a run can show: the order of checks before any effect, how often the configuration is evaluated, exact provider `argv` and error text. Rules that

a pure expect in `blueprint-core` or `blueprint-nix` already holds are not repeated there. The all-settings integration tests separately run tasks through real Nix. `scripts/test_scenarios.roc` writes projects and runs them through the real CLI and real Nix: the all-settings and extensions examples, noncommutative overlays in both orders with inheritance, an unselected overlay's native missing-package failure and a declared overlay that throws if evaluated, system-scoped tools on Linux and macOS with the package-target rejection of an unscoped one, renamed commands, and a task that names a script by path, as `scripts/check_all.roc`, which starts by its `roc-stable` shebang on a platform `RocPackages` supplies and imports a module beside it, and `zig-overlay` and `rust-overlay` as overlays, each supplying an exact release named in a quoted attribute, and configured tools: a fixture overlay's package says which options it was built with, and `rust-overlay` evaluates a toolchain with two targets and a component. It also runs `nix develop --debug` on a staged flake and requires Nix's own report that its bash comes from the locked package source, with no registry lookup, and runs a task with the `nixpkgs` edge removed from `Blueprint.lock`, as an earlier blueprint wrote it. A project it writes pins its inputs to the revisions in `fixtures/consumer/inputs.lock` and `flake.lock`, and its `Blueprint.lock` is read-only and compared after every step.

`fixtures/consumer/main.roc` holds the staged bytes and supplied-lock preservation as `expect`s`.

Parse fuzzing also checks successful semantic normalization for idempotence.

`scripts/test_builds.roc` runs real sandboxed builds of artifacts, dependencies and sources, including host-file and TCP isolation with positive host controls, fail-closed runner checks, exact argv, immutable locks, freshness and relocation. It also checks what Nix copies from the project (bytes, the owner's execute bit, raw names, exclusions, refused symlinks and special files) and that an undeclared tool is not found. It runs `blueprint _build-runner` directly on the host for the checks that need no sandbox, with a hand-written witness naming namespaces the host does not have.

`scripts/test_workflows.roc` runs real ordered task/build workflows, nested repetitions, failure stops, snapshot/dependency freshness, repeated locked-source verification, whole-closure preflight and immutable authority, including out-of-tree layouts, and then a temporary copy of the complete artifacts example. Both are Roc scripts on the harness in `scripts/src/BuildHarness.roc`, and the tasks and builds of their fixtures are Roc scripts run as `roc-stable` through `RocPackages`. Their inputs are pinned in `fixtures/consumer/inputs.lock` and `fixtures/roc-inputs.lock.json`; one first step fetches them and every later Nix call is recorded and run with `--offline`. See

[fixtures/builds/README.md](#). `scripts/test_update.roc` checks local-source preflight, authority observation, concurrent publication, named updates and what a lock from blueprint 0.5 is told.

`scripts/test_isolation.roc` checks what a build stages about its caller: the namespace identities, the CLI's own path as the runner, and the refusals when either cannot be used. Both put a failing `python3` on `PATH`: the CLI itself must not use a host Python. `scripts/test_programs.roc` runs projects whose environments fetch programs and expect tools of the machine. Nothing selects how they run but their own `Blueprint.roc`: one with no Nix tool must never start the `nix` stub. The downloads come from a server the script runs on the loopback address, the store is a directory of the test's own, and the lock entries are written by the test. It checks what is kept in the store and under which names, that a second run makes no request, that a stored artifact with other bytes is removed with its tree and fetched again, two runs at once from an empty store, every refusal (a missing expected program, a program the lock does not hold or holds under other URLs, another length or digest, each URL's failure, an archive with a link out of its tree), the caller's own shell, and parity: the same task with one Nix tool added prints the same from the same store file, and only then starts Nix.

Its last cases let `blueprint update` write the lock: with no `nix` on `PATH` for a project that only fetches, with nothing asked when nothing changed, for a named program alone, after a URL list changed, and for release assets. A download URL must be `https://`, with one exception made for these tests: `http://127.0.0.1:<port>/`, which no other machine can serve. A loopback URL of the shape `<owner>/<repo>/releases/download/<tag>/<asset>` is a release asset, and the same server describes its release at `/repos/<owner>/<repo>/releases/tags/<tag>`, as GitHub's API does for a GitHub URL. That is the whole seam: no variable or flag redirects the question. Through it a digest computed here is compared with a published one, another system's is recorded unfetched, a published digest that differs stops the update, a question that is refused or answered with something else leaves every digest computed and says so once, and a download missing for one system says how to declare that. It ends with `blueprint requirements`, asked with no `nix`, no lock, an empty store and no server: the report is the same, a declared and unused Nix source requires no provider, and nothing is written. These are Roc scripts, run from the repository root with `./blueprint built`; `ROC` names the compiler. They replace every tool the CLI runs with one program, `scripts/stubs/tool.roc`, which each test builds once and installs under several names (`roc-record`, `roc-wire`, `roc-probe`, `nix`, `guix`, `python3`, `readlink`). A copy decides what to be from its own file name and how to behave from `STUB *` variables, and records each invocation as a file holding one line of JSON in `<name>-calls/` under its working directory. Its header lists what each name does. `scripts/src/CLiHarness.roc` holds what the three tests share. `scripts/test_roc_packages.roc` is a Roc script, like `link_inputs.roc`. It runs the real CLI and Nix with a Roc package cache of its own in a temporary directory, and refuses to run if that directory lies inside `~/.cache`. A task's bundles are downloaded by `blueprint` itself from the releases their URLs name. It checks publication into an empty cache, that a second run writes nothing, that an existing package is left alone and an incomplete one replaced, that failures leave nothing behind, that a project with no build locks no bundle and its task still downloads them, and that a sandboxed build resolves its bundles from the lock with no network while the same build without one of them does not. Normal execution tests explicitly initialize authority with `update` first. The complete artifacts example is executed in a temporary copy by the workflow suite (`scripts/test_workflows.roc`).

10.4. The job with no Nix

Every suite above runs where Nix is installed, and replaces it with a stub where it must not be started. One job of `.github/workflows/ci.yml` runs on machines that have no Nix at all: `no-nix`, on `ubuntu-latest`, `ubuntu-24.04-arm` and `macos-latest`, with the binaries and platform hosts the build job produced and no step that installs anything. It runs `dist/smoke-<system> --no-nix dist/blueprint-<system>`, the NoNix mode of `scripts/src/SmokeBinary.roc`, which:

- refuses to start when `nix` is on `PATH`, and leaves no `roc` there;
- writes a project whose compiler is `Programs([Fetch("roc-stable", Recipes.roc_nightly(...))])` of the nightly in `blueprint-cli/compiler-version`, with `Expect(["git"])` and the Roc packages of a `basic-cli` script, against this checkout's platform;

- gives the binary a store and a Roc package cache of its own, empty;
- runs `blueprint update` and requires that the compiler was fetched exactly once, for reading `Blueprint.roc` and as `roc-stable` together;
- requires that `blueprint requirements --json` names no provider;
- runs the script as a task, which the fetched compiler builds, links and runs. On macOS that is the test that `roc` finds the `darwin` directory of its archive through the link it is started by;
- runs the task again and requires that nothing is downloaded, and that no flake was generated.

It reaches GitHub for the nightly and the Roc packages, and asks GitHub's API once for the nightly's published digests; a refusal there is a notice, not a failure. To run it on a machine that has Nix, hide Nix from `PATH`:

```
roc build scripts/smoke_binary.roc --output=/tmp/smoke
env PATH=/usr/bin:/bin /tmp/smoke --no-nix ./blueprint
```

10.5. Write the manual

The manual is the AsciiDoc in `docs/`. `index.adoc` is a book that includes one chapter per top-level `.adoc` file; `.github/workflows/docs.yml` renders it as a site and a PDF with the shared `build-docs` action, on every pull request, and deploys the site from `main`. Every AsciiDoctor warning fails the build.

To render it as CI does, with Docker and a checkout of [roc-automation](#) beside this one:

```
scripts/docs_snippet.roc # docs/generated/, from the latest release; needs a logged-in gh
python3 ../roc-automation/actions/build-docs/build_docs.py --workspace . --pdf-filename roc-
blueprint.pdf
```

The site is `.docs-out/site/` and the PDF `.docs-out/roc-blueprint.pdf`; neither is committed.

The manual describes the tool as it is: one recommended way to do each thing, in the present tense. What changed between versions belongs in the release notes, and what is unfinished in an issue.

The Roc in the manual is compiled by `scripts/test.roc` `unit`, so a block must be one of four things:

- a whole `Blueprint.roc`, starting `app [`, whose platform is written as a placeholder;
- settings as they stand inside `config`, complete enough to check by themselves: an environment a task uses is in the block. A tool of the source `nix` may be written without the `Packages("nix", ...)` line;
- `[source, roc, snippet=refused]`, settings the compiler must refuse, for a mistake the text explains;

- `[source,roc,snippet=unchecked]`, which is not compiled: use it only for a line of some other program or for settings that cannot stand alone.
- Each chapter opens with one `= Title` and is also rendered as a page of its own, so it cannot rely on attributes set in `index.adoc`.
- Link to another chapter as `xref:chapter.adoc#section-id[text]`, always with a section's ID and text. Section IDs come from headings, lowercase with hyphens.
- Give tables explicit column widths.
- A release's notes are `docs/releases/<version>.adoc`, included from `releases.adoc`, newest first; see [Release](#).

Chapter 11. Maintain and release

11.1. Move a compiler

Nothing moves either compiler automatically. Each moves in one reviewed pull request, with whatever the new compiler breaks fixed in the same request.

To move the product compiler:

1. Put the nightly's tag in `blueprint-cli/compiler-version`.
2. Replace the entries of `compilers` in `blueprint-cli/Bootstrap.roc` (see [The bootstrap table](#)) with the nightly's.
3. If the locked `roc-overlay` predates that nightly, run `nix flake update roc-overlay`.
4. Set the compiler and overlay in `fixtures/roc-inputs.lock.json` to match.
5. Move the basic-cli release in `blueprint-cli/main.roc`, the fixtures, the examples, `scripts/src/SmokeBinary.roc` and `rocPackages` in `flake.nix` if the nightly needs a newer one.
6. Move the nightly the manual's examples and `Published.snippet` fetch as `roc-stable`, so that a reader's first project downloads one archive for both the compiler `blueprint` uses and the one its scripts run on.
7. Set the hash of `rocGlueSpec` in `flake.nix` to that of the nightly's source archive, which `nix build .#roc-glue-spec` reports, and run `scripts/check_glue.roc --update`. Commit the bindings it changes together with whatever the host needs to build against them.

11.1.1. The bootstrap table

A project has no lock until its `Blueprint.roc` has been read, so the digest of the compiler that reads it cannot come from one. `blueprint` carries it: `compilers` in `blueprint-cli/Bootstrap.roc` holds, for the nightly in `blueprint-cli/compiler-version` and each system `blueprint` runs on, the URL of the compiler archive in `roc-lang/nightlies`, its SHA-256, its size, its format and the path of `roc` inside it.

System	Archive of the nightly
<code>x86_64-linux</code>	<code>roc_nightly-linux_x86_64-<date>-<commit>.tar.gz</code>

System	Archive of the nightly
aarch64-linux	roc_nightly-linux_arm64-<date>-<commit>.tar.gz
aarch64-darwin	roc_nightly-macos_apple_silicon-<date>-<commit>.tar.gz
x86_64-darwin	roc_nightly-macos_x86_64-<date>-<commit>.tar.gz
x86_64-windows	roc_nightly-windows_x86_64-<date>-<commit>.zip

gh release view <tag> --repo roc-lang/nightlies --json assets reports each asset's digest and size. The entries are data and nothing else: no rule derives a URL. `scripts/test.roc static` fails until every system blueprint runs on has an entry of the nightly in `compiler-version`.

With `ROC` unset and no matching `roc` on `PATH`, blueprint downloads its system's archive into the per-user store and checks it against the entry before using it; it starts no provider to read a `Blueprint.roc`. The entry names the same download as `Recipes.roc_nightly` of that tag, so a project that fetches that nightly as `roc-stable` holds it once, and blueprint update records its digest from the store without a second download.

To move the scripts' compiler, change `rocStableTag` in `flake.nix` and the `roc: "<tag>"` field in the header of each `scripts/*.roc` to the same tag; `scripts/test.roc static` names any script that differs. Move the `basic-cli` release in those headers too if the nightly needs a newer one. A compiler other than the one a header names refuses the script, and `roc fmt` rewrites the field to the nightly that runs it, so the scripts are formatted with `roc-stable fmt scripts`. `scripts/smoke_binary.roc` and `scripts/stubs/tool.roc` are the exceptions: the product compiler builds them, so their headers must suit both compilers and carry no `roc` field.

The required checks are listed under [Repository settings the workflows rely on](#). Keep their names aligned with the workflows when changing CI.

`basic-cli 0.26.0-rc2` requires the October 12 nightly or later. This repository pins `nightly-2026-10-13-b17d287` for the product and scripts, with `roc-fuzz 0.4.3`.

The complete suite requires x86_64 Linux and a running Nix daemon; the development shell supplies the rest. `zig build` builds the platform host for all five targets, so on Apple Silicon macOS the CLI's tests, the native CLI and the evaluation of a `Blueprint.roc` work too. Sandboxed builds, and so the `scenarios`, `builds` and `package` groups, remain x86_64 Linux only.

11.2. Release

One release holds the platform bundle, the `roc-blueprint-core` bundle it depends on, the `blueprint` binaries with their checksums, the examples archive, the documentation archive and the manual. It is made by `.github/workflows/release.yml`, run by hand from `main` with a version. Every pull request

runs the same workflow as a dry run, under the version `0.0.1-dry-run`, which is never published.

1. In one pull request, set `version` in `blueprint-cli/main.roc` and `flake.nix` to the release, and write its notes (see [Write the release notes](#)). Merge it.
2. Run the **Release** workflow from `main` with that version, for example `0.5.0`, or `0.5.0-rc1` for a release candidate. A version with a `-` in it is published as a pre-release.
3. Approve the `release` environment if it asks.

A release candidate reports its own version, so `version` must be set to `X.Y.Z-rc1` for it and to `X.Y.Z` for the final release: the two are different commits. Both use the notes in `docs/releases/X.Y.Z.adoc`.

11.2.1. What the workflow does

validate

Requires the default branch, then `scripts/check_version.roc` request requires a well-formed version, its notes, that no tag or release of it exists, and that it is the version the CLI and the flake report.

build

Cross-builds the binaries (`scripts/build_release.roc`), runs the whole test suite on them, bundles both packages for the version (`scripts/bundle.roc`), checks the version against the public API (`scripts/check_version.roc bump`), archives the examples (`scripts/package_examples.roc`), renders the manual with the release's platform URL and archives the site (`scripts/docs_snippet.roc`, `scripts/package_docs.roc`), renders the notes (`scripts/release_notes.roc`) and gathers the assets (`scripts/release_assets.roc`). On a pull request the test suite is left to `ci.yml`.

test

Runs each binary on its own kind of machine, and `scripts/test_release.roc` tests the examples archive there against the two bundles.

publish

The only job that may write to the repository. It checks out nothing and runs no script of the repository. It creates a draft release at the tested commit, uploads every asset, downloads each back and compares it with the tested file, and only then publishes. It does not run on a pull request.

site

For a final release, rebuilds and deploys the manual, which then names the new release. A release candidate does not change the site.

The machines that run the arm64 Linux and macOS binaries have Nix and deliberately no Roc, so

`build_release.roc` also cross-builds `scripts/smoke_binary.roc` for each system and they run that (`dist/smoke-<system>` `dist/blueprint-<system>`). The project the smoke test writes pins what it resolves: packages at the `nixpkgs` revision in `fixtures/consumer/inputs.lock`, the overlay at the `roc-overlay` revision in `flake.lock` and the compiler at the nightly in `blueprint-cli/compiler-version`, all compiled into the smoke program. A release therefore does not depend on what moved upstream that day. `.github/workflows/floating.yml` runs the same test weekly with `--floating`, which resolves what a new user's project would: a branch of `nixpkgs`, the overlay's default branch and its newest nightly. With `--no-nix` the same program writes a project that takes nothing from Nix and requires a machine that has none; the `no-nix` job of `ci.yml` runs it on all four released systems (see [The job with no Nix](#)).

`build_release.roc` takes its compiler from the flake: `nix build .#roc-cross` is the x86_64 Linux compiler archive of the nightly in `blueprint-cli/compiler-version` with the `darwin/` `sysroot` of the Apple Silicon archive beside it, which Roc needs to link macOS programs. Both are fetched by the URL and hash the locked `roc-overlay` records. The package is not in the development shell, so only a release build downloads the macOS archive. The four CLI binaries are compiled at the same time, each about 2.3 GB at its peak.

A binary downloads the compiler named in `blueprint-cli/compiler-version` by the digest it records; see [The bootstrap table](#).

11.2.2. Write the release notes

Write the release's notes by hand in `docs/releases/X.Y.Z.adoc` and include the file in `docs/releases.adoc`, newest first; `scripts/test.roc` static fails when the two disagree. The notes say what changed for a user deciding whether and how to upgrade: what a user can or must now do, the symptom each fix removes, and before and after for anything breaking. They are not a list of commits.

The file opens with `= roc-blueprint X.Y.Z` and names the bundles as `{platform-ur1}` and `{core-ur1}`, because the URLs hold hashes that do not exist until the release builds them.

`scripts/release_notes.roc X.Y.Z dist/notes.md` renders the notes as the release body with the real URLs, and adds what the release was built from: the Roc nightly and the Zig version, the `roc-overlay` revision and the hashes of the two compiler archives it supplied, the flake's `nixpkgs` revision, the core bundle, the Roc packages compiled into the CLI as `roc deps blueprint-cli/main.roc` resolves them, and the linker-input release with the SHA-256 of `link-inputs.lock.json`. It fails on a placeholder that is unused or unknown. The committed file keeps its placeholders and is not rewritten after the release. A dry run renders the newest notes there are.

11.2.3. The two bundles

During development the platform uses `core: "../blueprint-core/main.roc"`. `roc bundle` only packs files below the entry point's directory, so `scripts/bundle.roc [VERSION]` bundles the core package first, then a staged copy of the platform whose `core:` is the URL that core bundle will have in the release of `VERSION`. With no version it is the dry run.

Roc identifies a package by its URL minus the version and hash, so two archives named only by their hashes under one tag would look like one package served with two hashes, and the compiler refuses that as conflicting package contents. The core archive is therefore published as `roc-blueprint-core-<hash>.tar.zst`: the prefix is part of its identity. The platform's archive keeps the name `roc bundle` gave it, `<hash>.tar.zst`.

The release does not exist while the bundles are tested, so the script serves both itself, from a port the operating system assigns, at the paths the release will give them. The compiler is told to load the core bundle from there (`--replace-dep`) while it checks a few of the configuration fixtures and runs `examples/all-settings/Blueprint.roc`. It gets an empty package cache for this, so it must download both bundles. The modules and linker inputs packed are those in the directories and in the `targets:` section of `blueprint-platform/main.roc`.

Before bundling the platform, `scripts/bundle.roc` makes the check of `scripts/link_inputs.roc check`, so only linker inputs matching `link-inputs.lock.json` are packed. Their licences and `dependency.json` inventory go into the bundle under `linker-inputs/`.

`scripts/test.roc` package runs the dry run on every commit, with the examples archive and its test. A change that adds a Spec field needs nothing more: the core bundle carrying the field is released with the platform that uses it.

11.2.4. The version check

`scripts/check_version.roc bump` runs `roc bump` for both packages against the latest release, and a release fails unless its version bumps at least as far as the public API changed. The core package is compared from the working tree. The platform is compared as bundled: in the working tree it names the core package by path, which `roc bump` refuses in a public type, so the script unpacks the platform bundle and puts the unpacked core bundle in a package cache of its own, where the compiler finds it by hash although its URL does not exist yet. A dry run reports the required bump and expects nothing.

The platform exposes a type of the core package, and a package's URL is part of that type's identity, so the two versions move together.

11.2.5. Asset names other repositories depend on

The names of the binaries and of their checksum file are a contract. `roc-automation's setup-blueprint` action downloads `blueprint-x86_64-linux`, `blueprint-aarch64-linux` and `blueprint-aarch64-darwin` from the release a caller names, exactly as published. This release also includes `blueprint-x86_64-windows.exe`; the setup action does not yet install it. Callers copy the SHA-256 values from `blueprint.sha256`. Renaming an asset, archiving the binaries or changing the checksum file breaks every caller. `scripts/release_assets.roc` fails the build unless exactly those five names are present, and the publish job checks again before it uploads. A binary for another system is an addition; it does not change these.

The platform bundle is the one `.tar.zst` asset named by its hash alone, and the core bundle the one that starts `roc-blueprint-core-`. The manual and the version check find the previous release's bundles that way.

11.2.6. Repository settings the workflows rely on

These are set by hand in the repository's settings.

- A `release` environment, restricted to `main`. The publish job names it.
- GitHub Pages with **GitHub Actions** as its source, and the `github-pages` environment restricted to `main`. `docs.yml` deploys to it.
- Immutable releases, and a tag ruleset under which a version tag cannot be moved or deleted.
- The `main` ruleset requires a pull request, signed commits, and these checks with the branch up to date: `test` (the aggregate job of `ci.yml`), `Release dry run` (that of `release.yml`) and `Build the manual and site` (`docs.yml`).
- The default workflow token is read-only, and GitHub Actions may not create or approve pull requests.
- Private vulnerability reporting is enabled; `SECURITY.md` links to it.

Every workflow sets `permissions: {}` and states permissions per job, every checkout sets `persist-credentials: false`, and every action is pinned to a full commit; `scripts/test.roc` `static` checks the first two.

To adopt a newer basic-cli release, change the URL in `blueprint-cli/main.roc` and `fixtures/consumer/main.roc`, then update the matching entry in `rocPackages` in `flake.nix` with the new URL and hash (`nix store prefetch-file <url>`). Add any new transitive dependency to the same list. Rerun `scripts/test.roc`. If you change the weaver URL, update `rocPackages` in `flake.nix` to match.

11.3. What remains for Windows

[Issue #40](#) asks for `blueprint run` on Windows. What needs nothing from upstream is in place: a task whose environment has no Nix tool runs with no provider, from fetched programs, expected tools and Roc packages, and the planner, the store's location and `Recipes.roc_nightly` have their Windows forms. The Windows target and release binary are built in CI; the Nix-free smoke test runs on Linux, macOS and Windows.

The upstream prerequisites are now released: `basic-cli 0.26.0-rc2` supplies its Windows MSVC host and runtime, the pinned Roc nightly links from listed inputs, and the locked `roc-platform-template-zig` release supplies this platform's startup and import libraries.

`zig build` builds `targets/x64win/host.lib`; the platform bundle carries it and the verified Windows linker inputs. `scripts/build_release.roc` builds `blueprint-x86_64-windows.exe` and its smoke program. The `no-nix` CI job runs that program on `windows-latest`, fetching the compiler, updating a lock, reporting requirements and building and running a `basic-cli` program twice. Windows skips `chmod` and exposes fetched programs with hard links, falling back to a copy across volumes. Archives use the machine's `bsdtar`.

The full Nix suites remain Linux-only. Windows 10 and 11 desktop execution has not been verified here; the CI runner checks Windows Server. Tasks must invoke their interpreter explicitly, for example `Run(["roc-stable", "hello.roc"])`.

11.4. Upstream workarounds still in place

Compile-time configuration lowering is restored with Roc `nightly-2026-09-23-c7852fd`: `roc check Blueprint.roc` now rejects whole-config errors, including a missing `Name` or duplicate shells. Older compilers crashed while bundling a top-level constant dependent on the app's `config`; the `local-Spec` and `released-Spec` bundle tests guard against that regression. `blueprint check` retains a compiler check for diagnostics and reuses the `Spec` already loaded for CLI parsing, rather than running the configuration again. The loaded `Spec` is still needed for shared semantic validation and provider compatibility checks. Runtime validation also protects consumers from external `Spec`; it does not make major-1 platforms compatible with the current CLI. Loading the `Spec` still executes Roc, so retain compiler provisioning in the Nix wrapper and support for the `ROC` override.

These remaining dependencies and workarounds still apply:

- **Pinned Roc.** `blueprint-cli/compiler-version` selects the compiler, and the `basic-cli` release in `blueprint-cli/main.roc` must be compatible with it.
- `roc bundle --output-dir` must be on the same filesystem as the working directory. Otherwise the bundler fails with `CrossDevice`.
- **Error unions in `blueprint-core/osexpr.roc` use a named extension (`..others`).** These preserve open

error types without the compiler warning about a redundant bare `...`.

Chapter 12. Release notes

Each release has its notes in `docs/releases/`, newest first. Releases before 0.5.0 have theirs on the [releases page](#).

12.1. roc-blueprint 0.6.0

This release makes Nix one source among several. A `Blueprint.roc` now says where each thing comes from: a program downloaded from a URL and checked against a digest in `Blueprint.lock`, a tool the machine already has, or a tool from a Nix package source the project declares. `blueprint` starts Nix only for an environment that has a tool from a Nix source, and for a build. A project that fetches its compiler and expects `git` runs its tasks with no Nix installed.

That matters most in CI. A job that ran one Roc script through `blueprint` spent about 40 seconds on Nix before the script started. The same task in a project with no Nix source takes 3 to 7 seconds on an empty cache and under a second after that, measured on x86_64 Linux.

Every `Blueprint.roc` written for 0.5 must be edited, and its lock resolved again. "If you used 0.5", below, has the steps.

12.1.1. Use this release

```
app [config] { pf: platform "{platform-url}" }
```

The `blueprint` binary is attached for x86_64 Linux, arm64 Linux and Apple Silicon macOS, with SHA-256 sums in `blueprint.sha256`. It needs no Roc: it downloads the compiler that reads `Blueprint.roc` into its own store and checks it against a digest it records. It needs Nix only for what a project takes from Nix. Use this release's binary with this release's platform URL.

12.1.2. Fetch a program

`Fetch` inside `Programs` puts a downloaded program on an environment's `PATH`. A recipe, a function from the system to a download, says where it comes from; `Blueprint.lock` holds its SHA-256 and length for every declared system.

```

app [config] { pf: platform "{platform-url}" }

import pf.Recipes

config = [
  Name("my-package"),
  Environment("dev", [
    Programs([
      Fetch("roc-stable", Recipes.roc_nightly("nightly-2026-10-13-b17d287")),
      Fetch("blueprint-0-5", Recipes.file("https://github.com/lukewilliamboswell/roc-
blueprint/releases/download/0.5.0/blueprint-{system}")),
    ]),
  ]),
  Task("check", [Use("dev"), Run(["roc-stable", "check", "main.roc"])]),
]

```

- `Recipes.roc_nightly(tag)` is the Roc compiler of one release of roc-lang/nightlies. When the tag is the nightly `blueprint` itself runs on, one download serves both.
- `Recipes.file(template)` is a program that is one file, at a URL in which `{system}`, `{arch}` and `{os}` stand for the system.
- A recipe of your own is an ordinary function, `System → Config.Download`, that answers `File(url)`, `Archive(url, TarGz | TarXz | Zip, path)`, `Mirrored(download, alternate_urls)` or `Unavailable`.
- A URL, the path inside an archive, a system and a nightly tag are each checked where they are written, by `roc check`.

A download is kept in a per-user store, `blueprint/` under `XDG_CACHE_HOME` or `~/.cache`, and hashed again before every use. An archive is unpacked whole and the command is a link into it.

12.1.3. Expect a tool on the machine

`Expect` names programs the machine must already have. `blueprint` looks for them on `PATH` before it downloads or runs anything, names every one that is missing in one message, and installs nothing.

```

Environment("dev", [
  Expect(["git", "gh"]),
]),

```

12.1.4. Name the source of every tool

A tool from a package source is written `"<source>#<name>"`, and the source is declared. There is no default source:

```

Packages("nix", From(NixPackages("github:NixOS/nixpkgs/nixos-unstable"))),
Environment("host", [Tools(["nix#zig", "nix#pkg-config"])]),

```

Only an environment with such a tool starts Nix. Other tasks of the same project run without it.

12.1.5. Update without Nix

`blueprint update` starts Nix only when `Blueprint.roc` declares something Nix resolves: a Nix package source, an input, an overlay or a build source. A project that declares none is updated on a machine with no Nix.

- For each fetched program it downloads and hashes this machine's file. When the URL is a GitHub release asset it also asks the release, once and with no token, what digest it publishes. The two must agree, and the published digest is recorded for the other systems without downloading them. The lock says how each digest was obtained: `Computed`, `Published` or `Both`.
- When GitHub cannot be asked, which its limit of 60 requests an hour from one address makes common on shared CI runners, the update says so and downloads and hashes the file for every system instead.
- `blueprint update PROGRAM` resolves one fetched program again and nothing else, and never needs Nix.
- After writing the lock it prints every download host the new lock names and the old one did not.

12.1.6. Ask what tasks need

`blueprint requirements [--json] [TASK...]` reads `Blueprint.roc` alone and says, for each task, whether it needs Nix, which programs it fetches and which it expects of the machine. A CI job can install Nix exactly when it is needed:

```
blueprint requirements --json check | jq -e '.providers | index("nix")'
```

`blueprint shell` works without Nix too: for an environment with no Nix tool it starts your own `$SHELL` with the fetched programs on `PATH`.

12.1.7. If you used 0.5

1. Replace the platform URL in the `app` header with this release's, and use this release's `blueprint` binary.
2. Declare a package source, and write its name before every tool: `"git"` becomes `"nix#git"`, in `Tools`, `ToolsFor`, `Command` and `ToolWith`. A 0.5 project that declared no source used `github:NixOS/nixpkgs/nixos-unstable`; declare that to keep the same packages. Remove `Packages(name, Auto)`.
3. Or take the tool off Nix: move a program that is published as a download to `Programs([Fetch(...`

]], and a tool every machine has to Expect. `roc check Blueprint.roc` reports each remaining tool with no source and shows both ways out.

4. Run `blueprint update` and commit the new `Blueprint.lock`. A 0.5 lock is refused until then. Read the diff: it gains a `fetch` entry for each program and system.
5. In CI, install Nix only where `blueprint requirements` says a task needs it, and cache the `v1/artifacts` directory of the store, keyed on `Blueprint.lock`.

A pure package

Before, for 0.5:

```
config = [
  Name("my-package"),
  Overlay("roc", "github:roc-lang/roc-overlay"),
  Environment("dev", [
    Overlays(["roc"]),
    Command("roc-stable", "rocpkgs.nightly-2026-10-13-b17d287"),
    Tools(["git", "gh"]),
    RocPackages([
      "https://github.com/roc-lang/basic-cli/releases/download/0.26.0-
rc2/6j3zVDr3qrQa6aTh7Z1zgUfe9NeBE6R76T5ho3xrc8af.tar.zst",
      "https://github.com/roc-
lang/http/releases/download/1.0.0/6ZUwqYhCS8PU9Mo6MF7oV82ET2o7KYb57CLKDq4cq4sS.tar.zst",
    ]),
  ]),
  Task("check", [Use("dev"), Run(["scripts/check_all.roc"])]),
]
```

After. Nothing in it is Nix's, so no command starts Nix:

```
app [config] { pf: platform "{platform-url}" }

import pf.Recipes

config = [
  Name("my-package"),
  Environment("dev", [
    Programs([Fetch("roc-stable", Recipes.roc_nightly("nightly-2026-10-13-b17d287"))]),
    Expect(["git", "gh"]),
    RocPackages([
      "https://github.com/roc-lang/basic-cli/releases/download/0.26.0-
rc2/6j3zVDr3qrQa6aTh7Z1zgUfe9NeBE6R76T5ho3xrc8af.tar.zst",
      "https://github.com/roc-
lang/http/releases/download/1.0.0/6ZUwqYhCS8PU9Mo6MF7oV82ET2o7KYb57CLKDq4cq4sS.tar.zst",
    ]),
  ]),
  Task("check", [Use("dev"), Run(["scripts/check_all.roc"])]),
]
```

A platform with a Zig toolchain from Nix

Before, for 0.5:

```
config = [
  Name("my-platform"),
  Overlay("roc", "github:roc-lang/roc-overlay"),
  Overlay("zig", "github:mitchellh/zig-overlay"),
  Environment("dev", [
    Overlays(["roc", "zig"]),
    Command("roc-stable", "rocpkgs.nightly-2026-10-13-b17d287"),
    Tools(["git", "zigpkgs.\0.16.0\"]),
  ]),
  Task("check", [Use("dev"), Run(["scripts/check_all.roc"])]),
  Task("build-host", [Use("dev"), Run(["zig", "build"])]),
]
```

After. The scripts' environment needs no Nix; the environment that builds the host extends it and takes Zig from a declared source:

```
app [config] { pf: platform "{platform-url}" }

import pf.Recipes

config = [
  Name("my-platform"),
  Packages("nix", From(NixPackages("github:NixOS/nixpkgs/nixos-unstable"))),
  Overlay("zig", "github:mitchellh/zig-overlay"),
  Environment("dev", [
    Programs([Fetch("roc-stable", Recipes.roc_nightly("nightly-2026-10-13-b17d287"))]),
    Expect(["git"]),
  ]),
  Environment("host", [
    Extend("dev"),
    Overlays(["zig"]),
    Tools(["nix#zigpkgs.\0.16.0\"]),
  ]),
  Task("check", [Use("dev"), Run(["scripts/check_all.roc"])]),
  Task("build-host", [Use("host"), Run(["zig", "build"])]),
]
```

`blueprint update` needs Nix in this project, because it declares a Nix source. After it, `blueprint run check` starts no Nix and `blueprint run build-host` does.

12.1.8. Breaking changes

- **Every tool names its source.** A tool is "`<source>#<name>`" in `Tools`, `ToolsFor`, `Command` and `ToolWith`. `"git"` alone is an error at the literal, with both fixes in the message.
- **No default source and no `Auto`.** `Packages(name, Auto)` is refused, and a project that declares no source has none. A project with no tool from a package source declares none.

- **The first declared Nix source has the roles `default` and `had`.** An environment Nix must build although it has no tool of its own, such as the sandbox of a build with an empty environment, uses the first Nix source in the order of `Blueprint.roc`. So does the `nixpkgs` input the generated flake gives `nix develop` for its bash. A source called `default` is no longer preferred for either.
- **Systems accepts known systems only.** Each is `<arch>-<os>` of `x86_64`, `aarch64`, `riscv64`, `i686`, `armv7l` or `powerpc64le` and `linux`, `darwin`, `windows` or `freebsd`. Another name is an error where it is written.
- **The lock is format 1.1 and a 0.5 lock is refused.** It gains `fetches`, the digest of each program on each system with how it was obtained. Run `blueprint update`.
- **Roc packages that only tasks and shells use leave the lock.** `blueprint` downloads them itself, and the compiler refuses bytes that are not those the URL's hash names. A bundle stays in the lock when a build's environment has it, since Nix fetches it for the sandbox.
- **BLUEPRINT_PROVIDER is gone**, with the host provider it selected. `Blueprint.roc` alone decides whether a request needs Nix. The fixed table that mapped Nix tool names to programs of the machine, and its exact-version check of Zig and Rust, went with it: write `Expect`, which asks no version.
- **GH_TOKEN and GITHUB_TOKEN are not read.** No request carries a token. Only `blueprint update` asks GitHub anything, and it falls back to downloading when refused.
- **A build cannot use fetched programs or expected tools.** An environment that a build uses takes everything from its package sources.
- **The generated flake has no `blueprintRocPackages` output**, and it leaves out every shell and task environment that has no tool from a Nix source. A project with no Nix source has no flake at all: `blueprint gen` writes none and `blueprint flake` prints nothing.
- **The "no input named X" message** of `blueprint update NAME` now reads `no input or program named X` and lists both.

12.1.9. When versions are mixed

The Spec is format 2.7 and always requires the feature `explicit-sources`.

- A 0.5 `blueprint` given a 0.6 `Blueprint.roc` says `Blueprint.roc` needs `features: explicit-sources; upgrade blueprint`.
- A 0.6 `blueprint` given a `Blueprint.roc` on an older platform says `Blueprint.roc` uses a `roc-blueprint` platform older than `0.6.0`; update the platform URL in its app header and see `docs/releases/0.6.0`.
- A 0.6 `blueprint` given a 0.5 lock says `Blueprint.lock` was written by `blueprint 0.5` or older, which gave every project a package source it did not declare; run `blueprint update` to resolve it again with this `blueprint`, and review what changes.

12.1.10. For programs that import the core package

```
core: "{core-url}"
```

- Provider gains `realises`, which says which package sources a provider supplies, and loses `compiler` and the `Compiler` type: the Core fetches its own compiler.
- `Provider.Resolution` gains `present`, the programs the machine must have for an update.
- The new module `Realise` plans a request (`plan`, `preflight`, `requirements`, `resolvers`). A consumer that called a provider's `plan` directly should call `Realise.plan`, which asks the provider only for environments that need it.
- `Steps.Operation` loses `RocPackages`, `Fetch`, `Search` and `Require` and gains `Present` and `Programs`; `RocBundles` is a list of names and URLs. `Steps.Program` is new.
- `Spec` gains `programs`, `environment_programs` and `expects`, and `Lock` gains `fetches`. `Project.tool` accepts only `<source>#<name>`, and `Tool.to_str` always writes the source.

12.1.11. Known limits

- The Nix-free path is run in CI on x86_64 Linux, arm64 Linux and Apple Silicon macOS. It was developed and measured on x86_64 Linux.
- Windows x86_64 has a binary and platform target. The Nix-free CI smoke test fetches Roc, updates a lock and runs a basic-cli program. Invoke script interpreters explicitly; Nix tools and builds remain unavailable.
- `tar`, `ln` and `chmod` must be on `PATH` to unpack and link a fetched program. A `Zip` archive unpacks only where `tar` is `bsdtar`.
- A `Fetch` gives one command. An archive that holds several programs needs a `Fetch` for each.
- An expected tool is not version-checked.
- `blueprint requirements` reports tasks. It does not say that a build, a workflow with a build, or `blueprint update` needs Nix.
- `Recipes.file` answers for every declared system. Where a release has no file for one of them, `blueprint update` fails on it and says to list the supported systems in `Systems` or to write a recipe that answers `Unavailable`.
- Adding the first Nix source to a project makes its lock stale for every task until someone with Nix runs `blueprint update`.
- The digests in the lock are those `blueprint update` saw. Review a lock change as you review code.

12.2. roc-blueprint 0.5.0

This release puts everything a project needs in one place: the platform, the core package it depends on and the `blueprint` binaries are now published together under one version. It is also the first release with a manual and an archive of the examples.

A `Blueprint.roc` written for 0.4 needs only the new platform URL.

12.2.1. Use this release

```
app [config] { pf: platform "{platform-url}" }
```

The `blueprint` binary is attached for x86_64 Linux, arm64 Linux and Apple Silicon macOS, with SHA-256 sums in `blueprint.sha256`. It needs Nix, and fetches its own Roc compiler through Nix. Use this release's binary with this release's platform URL.

12.2.2. The core package moves into this release

`roc-blueprint-core`, the package that defines the Spec and its S-expression format, used to have releases of its own under `core-X.Y.Z` tags. It is now an asset of each `roc-blueprint` release:

```
core: "{core-url}"
```

A program that imports the core package must change its URL to this form. Roc identifies a package by its URL, so this is a different package from the one under the `core-` tags, and the two cannot be mixed in one program. There will be no further `core-X.Y.Z` releases.

12.2.3. Configure a tool

`ToolWith` adds a tool built with options its package takes. A platform host that cross-compiles Rust can now declare its toolchain's targets and components:

```
Overlay("rust", "github:oxalica/rust-overlay"),
Environment("host", [
  Overlays(["rust"]),
  ToolWith("rust-bin.stable.\\"1.95.0\\".default", [
    ("targets", List([Str("x86_64-unknown-linux-musl"), Str("aarch64-unknown-linux-musl")])),
    ("extensions", List([Str("rust-src")])),
  ]),
]),
```

The options are data passed to the package's `override`, never an expression. The Spec is now format 2.6, and a configuration that uses `ToolWith` requires the feature `configured-tools`, which a

`blueprint` older than this release refuses by name.

12.2.4. Changes to the tool

- `blueprint` writes a `.gitignore` into the directory it generates (`.blueprint/` unless you moved it), so generated state stays out of a commit even when the repository's own `.gitignore` does not list it. A directory that is already tracked stays tracked until you run `git rm -r --cached .blueprint`.
- On Intel macOS `blueprint` now stops with a message that names the supported systems, where it used to fail inside Nix. `x86_64-darwin` remains a system a configuration may list.
- `nix run github:lukewilliams/roc-blueprint/0.5.0` works on arm64 Linux.
- The code that will run tasks on Windows without Nix is included, but this release has no Windows binary and its platform has no Windows target, so Windows cannot use it yet.

12.2.5. Documentation and examples

- The project has a manual, published as a site and attached to this release as `roc-blueprint-0.5.0.pdf`. The reference that was in the README is now there.
- `roc-blueprint-examples-0.5.0.zip` holds the example projects with their platform set to this release, and the Spec each one printed when the release was made.
- `roc-blueprint-docs-0.5.0.zip` holds the rendered manual.