An introduction to rhiza-claude

What the plugin is for, before what it contains

Jebel Quant

Release v0.6.1

Abstract

rhiza-claude is a Claude Code plugin for repositories that share their scaffolding with a template repository instead of each carrying a private copy of it. This document is the part the reference pages cannot be: not a catalogue of the seven markdown files under `commands/`, but an account of the problem they exist to solve — the two-repository boundary a sync respects, what "rhiza-managed" actually means on disk, the order the commands are meant to be run in, and why a plugin whose selling point is determinism keeps half of itself in prose. It covers installation, compares the design against the obvious alternative of shipping an MCP server, and closes with a worked first run: empty directory to a managed, synced, scored repository. The figures throughout are captured output, not mock-ups.

1 The problem

Ten repositories need substantially the same scaffolding. Continuous integration workflows, a `Makefile`, lint and type and coverage gates, a documentation build, a licence, a release path. None of it is the interesting part of any of those ten projects, and all of it has to exist before the interesting part can be trusted.

The obvious approach — write it once, copy it into each repository — fails slowly rather than loudly. A workflow gets fixed in one repository and not the other nine. A coverage threshold gets lowered under deadline in a repository nobody revisits. Two years later there is no answer to the question *which repositories have the current gates*, because nothing anywhere records which vintage of the scaffolding each one received.

rhiza is that scaffolding kept once, in a *template repository*: `jebel-quant/rhiza` for Python, `rhiza-go` for Go. **rhiza-claude** — this plugin — is how an individual repository adopts that template, keeps up with its releases, and is told how well it is doing against the gates the template delivers.

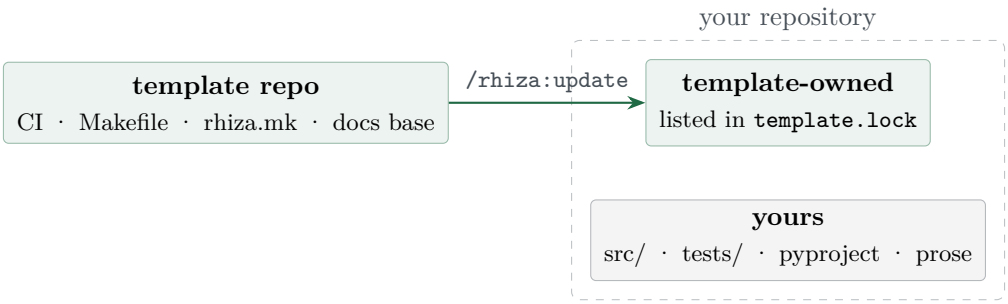
Two things follow from that framing, and they are the reason an introduction is needed at all. First, there are always *two* repositories in play, and almost every question a newcomer has is really a question about the boundary between them. Second, the interesting work is not the copying — it is knowing precisely which files are the template's to overwrite and which are yours to keep.

2 The two-repository model

	owns	**who changes it**
the template	CI stubs, `Makefile`, `.rhiza/rhiza.mk`, the docs base	upstream — you receive it
your repository	source, tests, `pyproject.toml`, README prose	you — the sync never touches it

A sync is therefore *not* a copy of the template over your repository. `/rhiza:update` writes only template-owned paths, and it knows which paths those are because the previous sync wrote them

down. Your `src/` is not in that set, so it cannot be swept in — there is no blanket `git add --all` anywhere in the flow, which is a deliberate design constraint rather than an implementation detail.



The dashed boundary is your repository; the arrow is the only thing a sync does. Nothing crosses into the lower box.

3 What "rhiza-managed" means

It means two files under `.rhiza/`, and they answer different questions.

file	it is	question it answers
`template.yml`	a **pointer**: template repository and pinned `ref`	what *would* we sync from?
`template.lock`	a **record**: repository, ref, commit SHA, timestamp, strategy, every managed file	what *was* delivered, and when?

Intent versus outcome. The two disagree in both directions, which is the whole reason both exist. A freshly initialised repository has a valid pointer and no lock at all (Figure 1). A long-synced repository can carry a lock beside a pointer somebody has since broken by hand. Reporting either one alone is misleading, so `/rhiza:status` always reports both halves.

The same distinction is load-bearing for `/rhiza:quality`, which refuses to run unless it finds *both* `.rhiza/template.yml` and `.rhiza/rhiza.mk`. Every gate it scores is a `make` target that the sync delivers; scoring a managed-but-unsynced repository would report it as *broken* when it is merely *unsynced*, and that is a wrong answer rather than a missing one. By contrast `/rhiza:release` requires neither file: it reads the repository's own `[tool.bumpversion]` configuration, so it works on any git repository, including this plugin.

4 Installing the plugin

Nothing above is reachable until Claude Code knows the plugin exists. Installation is two lines, typed inside Claude Code:

```
/plugin marketplace add Jebel-Quant/rhiza-claude
/plugin install rhiza@rhiza-claude
```

The first line registers the *marketplace* — this repository, read straight from git; the second installs the *plugin* it publishes. From a shell, `make install` runs the equivalent `claude plugin` commands, which is the form to prefer in a setup script. Either way the seven commands then appear namespaced under `/rhiza:`, and typing `/rhiza` is enough for Claude Code to autocomplete them.

managed, but never synced — the pointer is valid, the lock is absent

```
$ python3 scripts/validate.py …/my-lib
  Validating template configuration in: …/my-lib
✓ Template file exists: .rhiza/template.yml
✓ YAML syntax is valid
  Project language: python
✓ pyproject.toml exists: …/my-lib/pyproject.toml
✓ 'src' folder exists: …/my-lib/src
! Standard 'tests' folder not found: …/my-lib/tests
✓ Using profile mode (profiles: ['github-project'])
✓ repository format is valid: jebel-quant/rhiza
✓ ref is valid: main
✓ Validation passed: template.yml is valid

$ python3 scripts/status.py …/my-lib
No template.lock found — run /rhiza:update to perform the first sync
```

Figure 1: *The two halves disagreeing, which is the normal state after* `/rhiza:init`*. Validation passes — the pointer is well-formed and names a real template — while* `template.lock` *does not exist, because nothing has been synced yet. A tool that reported only the first line would call this repository healthy; one that reported only the second would call it broken. It is neither: it is* managed but not yet synced*.*

Pinning a version

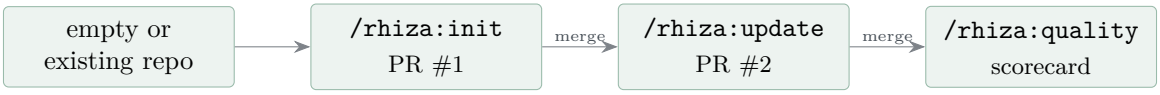
By default the marketplace tracks this repository's default branch, so `/plugin install` gives you the latest release. Pinning happens at the *marketplace* layer rather than per plugin — append the version's git tag as a `#<ref>` suffix when adding the marketplace:

```
/plugin marketplace add Jebel-Quant/rhiza-claude#v0.6.1
/plugin install rhiza@rhiza-claude
```

Once the marketplace is added, `/plugin install` uses whatever ref it points at, so switching versions means removing the marketplace and re-adding it at the tag you want. This is worth knowing because it mirrors the model the plugin itself applies to your repositories: a pinned ref, changed deliberately, rather than a floating `latest`.

The one thing to install *outside* Claude Code is `uv`, and even that is optional up front — both `/rhiza:init` and `/rhiza:update` check for it as their first step and offer to install it. There is no `rhiza` CLI; see §11.

5 The intended path



Three commands, two pull requests you merge yourself, in that order:

1. `/rhiza:init` writes exactly one file of its own — `.rhiza/template.yml` — and **deliberately syncs nothing**. No CI, no `Makefile`, no gates arrive at this step.
2. `/rhiza:update` performs the first sync. This is what brings the template content in and what writes `.rhiza/template.lock`.

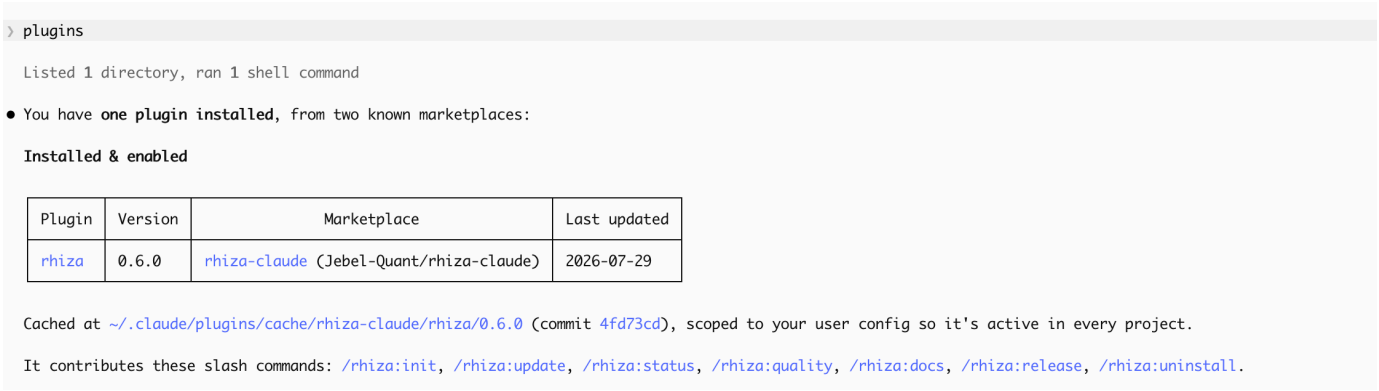


Figure 2: *The installed plugin as Claude Code reports it. One plugin, from the* `rhiza-claude` *marketplace, cached under* `~/.claude/plugins/` *at a specific version and commit, contributing exactly the seven slash commands this document describes. It is scoped to the user configuration rather than to one project, which is why* `/rhiza:init` *is available in a directory that is not yet rhiza-managed — the case it exists to handle. Cropped from a full-window screenshot to the region that carries those facts; captured at v0.6.0, one patch release before the v0.6.1 described here.*

3. `/rhiza:quality` needs that content to exist as real `make` targets, so it comes third.

Running them out of order is not dangerous — each stops early and says why — but the sequence is worth knowing in advance, because *"`/rhiza:init` ran and no CI appeared"* is the expected outcome of step one rather than a failure. Separating the two pull requests is the point: the file that declares which template you follow is reviewable on its own, apart from the several dozen files that template then delivers.

6 A worked first run

0. A git repository with a remote

```
mkdir my-lib && cd my-lib && git init
gh repo create my-lib --private --source=. --push   # or create it in the UI
```

`/rhiza:init` never pushes to your default branch. If the repository does not exist upstream yet it asks *you* to create it — an empty README is enough — rather than pushing one itself. Then start Claude Code in that directory.

1. `/rhiza:init` — become rhiza-managed

It detects platform, owner and name from `origin` (asking when it cannot), picks the language and template repository, resolves the template's latest release as the initial pin, and opens a pull request on a `rhiza_init_<date>` branch:

written by	file
the command itself	`.rhiza/template.yml` — the pointer and pinned ref
the skeleton procedure	`pyproject.toml` (via `uv init --lib`), `src/`, `.python-version`
the license procedure	`LICENSE` plus the SPDX metadata in `pyproject.toml`

There is no `.rhiza/template.lock` yet, and no workflows (Figure 3). Merge the pull request.

```
/rhiza:init — the pointer, and nothing else

$ python3 scripts/init_scaffold.py --host github --language python .
created  .rhiza/template.yml

$ python3 scripts/init_skeleton.py --owner jebel-quant --repo my-lib .
note     normalised uv's placeholder hello() to a package docstring
note     seeded the empty README.md uv left behind — /rhiza:docs owns the real one
note     pyproject.toml: description, project.urls, dependency-groups
modified src/my_lib/__init__.py
modified README.md
modified pyproject.toml

$ cat .rhiza/template.yml
repository: "jebel-quant/rhiza"
ref: "main"
profiles:
  - github-project

$ git status --short  &&  ls .rhiza/
A  .python-version
A  .rhiza/template.yml
A  README.md
A  pyproject.toml
A  src/my_lib/__init__.py
A  src/my_lib/py.typed
template.yml
# no template.lock, no .github/workflows, no Makefile — by design
```

Figure 3: *Step one, driven through the bundled scripts the command delegates to. Six files are staged and `.rhiza/` contains exactly one of them — the pointer. No lock, no `Makefile`, no `.github/workflows/`: this is the expected outcome of `/rhiza:init`, not a failure of it.*

2. `/rhiza:update` — the first sync

It bumps the `ref` in `template.yml` to the newest template release, syncs, resolves any conflict by taking the upstream side, and opens a second pull request carrying the template-owned files — `.github/workflows/`, `Makefile`, `.rhiza/rhiza.mk`, the docs base — together with `.rhiza/template.lock` recording exactly what was delivered. Only paths the lock names are staged, so nothing of yours is included even if it changed in your working tree. Merge that too.

3. `/rhiza:status` — confirm both halves

```
/rhiza:status --files
```

Read-only. It validates the pointer *and* prints what the lock records: repository, ref, commit SHA, timestamp, strategy, and with `--files` the managed files as a tree (Figure 4). This is how you distinguish *managed and synced* from *managed but never synced* — compare it against Figure 1, which is the same command on a repository that has only been initialised.

4. `/rhiza:quality` — get scored

Now that the gates exist, this runs them — lint, types, documentation, dependencies, security, tests, test layout, complexity, architecture — and produces a 1–10 scorecard with findings and an optional menu to file them as issues. It proposes fixes and applies none.

Figure 4: *A synced repository reporting both halves. The six header lines come from the lock, so they describe what actually arrived — including the commit SHA, which pins the answer to* which vintage of the scaffolding does this repository have. *The tree below is the set of managed files: every path the sync owns and may overwrite, and by omission, everything that is yours.*

Afterwards: `/rhiza:update` again whenever the template cuts a release (`--check` compares your pin against the latest, as in Figure 5), `/rhiza:docs` to create or refresh `README.md`, `CLAUDE.md` and `mkdocs.yml` (preserving hand-written prose), and `/rhiza:release` when you want to cut a version.

7 Commands and procedures

The plugin ships two kinds of markdown, and the distinction is enforced rather than conventional:

- `commands/*.md` are **slash commands** you invoke: seven of them, each with a reference page.
- `prompts/*.md` are **internal procedures**: seven shared steps a command reaches with the `Read` tool, kept deliberately *outside* `commands/` so they cannot be invoked directly.

Procedures are not implementation trivia, and filing them under "internals" undersells them: they are where the shared behaviour lives, which is why `init.md` alone does not explain what `/rhiza:init` does. Installing `uv`, choosing a work branch off an untouched default, scaffolding

Figure 5: *`--check` adds one line to the report: the pinned ref, the latest upstream release, and the distance between them. This is the question the copy-paste approach cannot answer for any repository, and the reason the lock records a ref at all.*

a `pyproject.toml`, writing a licence, gathering design evidence, applying the scoring rubric — each is a procedure, and that is exactly why `/rhiza:init` and `/rhiza:update` behave identically everywhere they overlap. A check in CI enforces the arrangement: no procedure may carry command frontmatter, none may collide with a command name, every referenced procedure path must resolve, and no procedure may be orphaned.

8 The seven commands, and what each writes

An introduction should say which of these change your repository, because that is the first thing anyone actually wants to know.

command	what it does	what it writes
`/rhiza:init`	make the repository rhiza-managed; no sync, no gates	one file of its own, then a pull request
`/rhiza:update`	sync to a template release, upstream side wins conflicts	template-owned paths only, then a pull request
`/rhiza:quality`	run the gates and score 1–10 with findings	nothing — proposes fixes, applies none
`/rhiza:docs`	create or refresh `README.md`, `CLAUDE.md`, `mkdocs.yml`	those files; no commit, no pull request
`/rhiza:release`	pick a next version from a table, bump, changelog	one local commit and one tag; never pushes
`/rhiza:status`	report the pointer *and* the lock	nothing — read-only
`/rhiza:uninstall`	delete every file the lock records, prune empty dirs	removes files; confirms first

Two patterns are worth reading off that table. The commands that change a *shared* repository do it through a pull request you review — neither `/rhiza:init` nor `/rhiza:update` pushes to your default branch. And the ones that assess rather than act (`/rhiza:quality`, `/rhiza:status`) write nothing at all, which is what makes them safe to run on a repository you have just inherited and do not yet understand.

Common confusions

"`/rhiza:init` finished but no CI appeared."
Correct behaviour. `/rhiza:init` writes the pointer; `/rhiza:update` performs the first sync. The

```
/rhiza:release — the legal next versions, with no recommendation
$ python3 scripts/check_version_bump.py --current 0.6.1
current  : 0.6.1
floor    : v0.6.1
patch    : v0.6.2
minor    : v0.7.0
major    : v1.0.0
ok       : no target given — listing suggestions only
```

Figure 6: *The division of labour in one screen. Computing the candidates is arithmetic over semver and belongs in tested code: the script derives them, and enforces the floor that keeps a release from going backwards. Choosing among them is a judgement about API stability, so the script stops — `listing suggestions only` — and the choice is the user's.*

template content arrives in the second pull request.

"Which CLI do I have to install?"
None. `uv` is the only hard dependency; the plugin's own scripts are stdlib-only Python. There is no `rhiza` CLI in this picture.

"Can I edit a synced file?"
You can, but `/rhiza:update` resolves conflicts by taking the upstream side, so your edit is expected to lose. Template-owned files are changed *upstream*; that is the point of having one original.

"Will a sync overwrite my README?"
No. `README.md` is yours, not template-owned. `/rhiza:docs` is the only command that writes it, and it preserves hand-written prose — it replaces the generated badge block and corrects stale facts, but never deletes what a human wrote.

"`/rhiza:quality` refuses to run."
It found the pointer but no `.rhiza/rhiza.mk` — managed, never synced. Run `/rhiza:update` first. The refusal is deliberate — scoring at that point would report every missing gate as a failure of your code.

9 Why the commands are prose

The split is deliberate: **deterministic work belongs in tested code, judgement belongs in markdown.**

The bundled Python under `scripts/` does the parts with exactly one right answer — parsing the lock, merging synced files, comparing versions as semver, computing the legal candidate bumps — and it is stdlib-only, strictly type-checked, and held at 100 % test coverage. The markdown does the parts that require a reading of your repository: which findings matter, whether a breaking change should spend the 1.0 signal, what to preserve in a README a human wrote.

Two consequences are worth stating, because they are what make the arrangement more than a slogan. First, *the prose is gated too*: a command that names a script, a flag or another command which no longer exists fails the build rather than failing in front of a user mid-task. Second, where a model would reliably guess badly, the guess is removed rather than improved — `/rhiza:release` lays the legal next versions out as a table and declines to recommend one (Figure 6), because whether `0.x` should become `1.0.0` is an intention about API stability that no commit log records.

10 Compared with an MCP server

The obvious alternative design is not a plugin at all. The Model Context Protocol is the standard way to give a model capabilities it does not otherwise have: you run a *server* — a local subprocess speaking over stdio, or a remote endpoint behind a URL — and it advertises typed tools that any MCP-capable host can discover and call. Shipping rhiza as `rhiza-mcp` exposing `rhiza_sync`, `rhiza_status` and `rhiza_score` would be a perfectly conventional choice. It is worth being explicit about why this is a plugin instead, because the reasoning generalises past this project.

	an MCP server	**this plugin**
what ships	a program, plus a transport	markdown and stdlib scripts
at rest	a process or endpoint to run, version and reach	files in a git repository
interface	a tool schema, fixed at the boundary	prose the model reads
reach	whatever the server can talk to	whatever the host already can
portability	any MCP host	Claude Code only

The deciding question is reach. An MCP server earns its keep when it can touch something the host cannot: a hosted service, a database, an internal API whose credentials have no business being in a shell. Everything rhiza operates on is already under the host's hand — `git`, `make`, the filesystem, `gh`/`glab`. A server wrapping those would add a process, a lifecycle, and a schema without adding a single capability; the same `git` commands would run, one indirection further away. Worse, the interesting half of this plugin does not fit through a tool schema at all. `/rhiza:quality` is a reading of your repository and `/rhiza:docs` preserves prose a human wrote — neither is a function call with arguments, and a typed boundary is the wrong shape for judgement.

Where an MCP server would be the better answer. Three cases, none of which this plugin is:

- **A boundary worth enforcing.** If the operation needed authorisation, auditing or rate limiting, a narrow typed surface beats handing the model a shell. rhiza needs none of that: the destructive commands go through a pull request you review, which is the boundary.
- **State that outlives one repository.** *Which of our thirty repositories are behind, and by how many releases?* is a fleet question. Answering it means reading thirty locks and holding the result somewhere — genuinely server-shaped work, and a plausible companion to this plugin rather than a replacement for it. Notice it would consume exactly the same `template.lock` files; the per-repository commands would not change.
- **Other clients.** MCP is client-agnostic and a plugin is not. If these workflows had to run from several editors, or from CI with no Claude Code present, the protocol would be the reason to pay for a server.

That last point is the honest cost of the choice made here, and it should not be glossed: **this plugin works in Claude Code and nowhere else.** What buys back the price is that the plugin is content rather than infrastructure. There is nothing to run, nothing to keep alive, and no version skew between a server and the repository it acts on — the behaviour *is* a set of files, reviewable as a diff, pinned by a git ref, and gated by the same CI that checks the scripts. For a tool whose entire subject is keeping many repositories honest about which version of something they have, that symmetry is not an accident.

11 What it needs, and what it is not

`uv` is the one hard dependency, and both `/rhiza:init` and `/rhiza:update` offer to install it as their first step. `git` and `make` are used and are near-universal. The plugin's own scripts are stdlib-only Python: **there is no `rhiza` CLI to install**, which is the most common point of confusion about what this actually requires.

Honest scope, at v0.6.1: seven user-facing commands, seven internal procedures. It is not a build system, a package manager, or a replacement for your CI — the template delivers workflows, and the plugin's job is to get them to you intact and tell you how the result scores. Nothing here runs on a schedule or in the background; every command is something you invoke, and the two commands that change a shared repository (`/rhiza:init`, `/rhiza:update`) do it by opening a pull request you review.

12 Glossary

The words that carry the model, in one place. Each is used above in exactly this sense.

template repository
The upstream repository holding the shared scaffolding — `jebel-quant/rhiza` for Python, `rhiza-go` for Go, or a fork of either. You read from it; you do not sync *to* it.

pointer — `template.yml`
Declared intent: which template repository, and which pinned `ref`. The one file `/rhiza:init` writes itself, and the file `/rhiza:update` bumps.

lock — `template.lock`
Recorded outcome: repository, ref, commit SHA, timestamp, strategy, and the list of managed files. Written by a sync, never by hand.

managed file
A path the lock names — template-owned, and therefore the sync's to overwrite. Everything not in that list is yours, and no command stages it.

rhiza-managed vs. synced
Managed means the pointer exists. *Synced* means the content arrived, which is observable as `.rhiza/rhiza.mk`. Both are required by `/rhiza:quality`, and the gap between the two is the state `/rhiza:init` deliberately leaves you in.

procedure — `prompts/`
A shared step a command reaches with the `Read` tool, kept outside `commands/` so it cannot be invoked as a slash command.

gate
A `make` target delivered by the sync that either passes or fails: lint, types, docstring coverage, tests, dependency and security checks, test layout.

scorecard
What `/rhiza:quality` produces: a 1–10 mark per category with findings, plus an optional menu to file them as issues. Assessment only — it applies no fixes.

Reference documentation for every command: jebel-quant.github.io/rhiza-claude. The short form of this introduction opens the repository's `README.md`; the full form opens `docs/index.md`.