`rhiza-task`: Replacing a Synced Make Layer with a Pinned Command-Line Interface

Jebel Quant Research

Abstract

Project templates distribute shared development infrastructure by copying it. For a `make`-based gate layer this is not a stylistic choice but a structural one: `make`'s `include` directive resolves paths on the local filesystem and cannot reach a remote file, so a template has no mechanism to *reference* its own recipes and must therefore duplicate them into every consumer. Every downstream difficulty — version skew, local edits that the next sync reverts, exclusion lists, target shadowing — follows from that one limitation rather than from anything about `make`'s syntax. We describe `rhiza-task`, which replaces the copied layer with an ordinary package dependency. The 62 gates of the `rhiza` template become a registry populated through Python entry points; `make`'s `?=` and `+=` become a six-layer configuration resolver; and each recipe's contract — previously expressed as shell inside a `make` rule — becomes an argument vector that can be asserted without provisioning the tool it names. The resulting suite of 301 tests executes in under a second, provisions nothing, and runs on Windows, which the shell recipes it replaces could not.

1 Introduction

A project template solves a real problem: a team with twenty repositories wants one definition of what "the tests pass" means. The usual implementation is a scaffolding tool that copies files into each consumer, plus a sync mechanism that re-copies them when the template changes.

Copying works acceptably for files a consumer is expected to own and edit — a `README` skeleton, a license header. It works badly for files that encode shared *behaviour*, because a copy has no identity. Once the same recipe exists in twenty working trees, there is no version to pin, no way to tell an intentional local change from a stale one, and no way to fix a bug for everyone without twenty merges.

The specific instance we address is `rhiza`, a template whose gate layer was approximately 200 lines of `.rhiza/rhiza.mk` plus roughly a thousand more spread across the `.rhiza/make.d/` fragments,[1] synced into each consumer at a template tag. This paper describes what replaced it and why the replacement is a package rather than a better `Makefile`.

Contributions.

1. An account of why `include` is the root cause, and why the fix therefore cannot live in `make` (Section 2).
2. A task model — registry, guards, and three outcomes rather than two — that recovers what `make`'s double-colon rules and exit statuses provided (Section 3).
3. A testing approach in which each gate's contract is its argument vector, making the layer assertable without a toolchain (Section 4).

[1] The repository's own documentation quotes two different figures for the fragments — 1023 lines across 15 files in the `README`, 823 across ten in the package docstring — most likely because one counts the bundle-owned fragments and the other only those ported. The argument does not turn on which is right.

4. Measurements from the implementation, and the limitations that approach accepts (Sections 5 and 6).

2 The problem is distribution, not syntax

It is tempting to read a 200-line `Makefile` full of `$$`-escaped shell as an argument against `make`. It is not. `make` is a capable dependency engine, and most of the layer's recipes were short. The difficulty is narrower and harder:

> `include` takes a path, not a URL. A `Makefile` cannot say "use version 1.0.0 of somebody else's rules."

Every consequence follows mechanically. Because the recipes must be copied, a consumer holds a snapshot rather than a reference, so "which version of the gates does this repository run" has no answer beyond a template tag recorded elsewhere. Because the copy is a real file in a real working tree, a consumer can edit it — and will, because no shared default fits twenty repositories — after which the next sync either reverts the edit or conflicts with it. The template accretes an exclusion mechanism so a consumer can opt out of a file; consumers accrete shadowing targets in their own `Makefile` so they can override a recipe they cannot edit. Both are workarounds for the absence of a reference.

A second, independent cost is the shell. `make` recipes are shell fragments, so a gate that needs an array, a retry loop, or a conditional argument list must express it in portable `sh`, with `$$` escaping throughout. This is why the layer carried roughly forty lines probing whether `make` had fallen back to `cmd.exe` on Windows. That probe is not incidental complexity; it is the price of the implementation language.

Replacing the layer with a published package addresses both at once. A dependency *is* a reference: `uvx rhiza-task@1.4.0 test` names a version, resolves it, and leaves nothing in the working tree to drift or exclude. And a package is written in a language with data structures, so the retry loop becomes a `for` statement.

3 Design

3.1 The task model

A gate has three parts, which `make` expressed positionally and this model names: preconditions, prerequisites, and a body. A task is a frozen record:

```
@task(
    "test",
    "run all tests",
    section="Python",
    layer="python",
    needs=("install",),
    guards=(Guard("tests_folder", glob="test_*.py",
                reason="no test files found"),),
)
def test(cfg: Config) -> None:
    ...
```

`needs` replaces `make`'s prerequisites, deduplicated within one invocation — which is what `make` gave for free, and the reason `install` can be named by eleven tasks without running eleven times. `guards` are evaluated before the body. `section` replaces the `##@` comment

convention that the previous layer parsed with `awk`, and the help string replaces `##`: the `list` output is generated from the registry rather than scraped from comments.

3.2 Three outcomes, not two

A process reports success or failure. A gate has a third state that matters: it ran and found nothing to measure. The previous layer signalled this by printing a warning and exiting zero, which is indistinguishable from success to any caller — and one downstream repository documented in its own `Makefile` that it had a gate "silently passing over nothing" for exactly this reason.

Here a guard failure raises `Skip`, which the runner records as a distinct outcome and reports in its own colour. A `--strict` flag promotes every skip to a failure, which is the correct setting in a consumer repository, where a skip usually means a gate lost its subject.

Failures propagate the child process's own exit status rather than collapsing to one, so a caller can still distinguish `pytest`'s 2 from `cargo`'s 101. A status outside a shell's usable 1–255 range — zero, or the negative signal number reported for a killed child — collapses to 1, since it cannot be passed to `exit` unchanged. A task blocked because a prerequisite failed is recorded as `BLOCKED` and contributes no status of its own: it never started a process, and the failure that blocked it is recorded earlier and speaks for the run.

3.3 Discovery through entry points

Task modules are not imported from a hardcoded list. Each registers itself under the `rhiza_task.tasks` entry-point group, and the CLI imports whatever it finds:

```
[project.entry-points."rhiza_task.tasks"]
python = "rhiza_task.tasks.python"
rust = "rhiza_task.tasks.rust"
go = "rhiza_task.tasks.go"
```

Two properties follow. First, a consumer adds a gate by publishing a module and declaring the same entry point — the replacement for `-include local.mk`, and a first-class citizen rather than an override, since built-ins arrive by the identical mechanism. Second, the top of the dependency graph inverts safely: the CLI never imports the task modules, so the edge that would otherwise need discipline to keep pointing the right way does not exist as an import at all. A module that fails to import is reported and skipped, so a broken third-party gate does not take the others down.

3.4 Configuration in six layers

`make`'s `?=` gives a default that the environment can override; `+=` appends. Reproducing that, plus the ability for a consumer to change a setting without editing a synced file, requires an explicit resolution order (Table 1).

Layer 4 is what replaces target shadowing: a consumer that disagrees with a threshold writes it in a file it already owns. Validation happens once, when the configuration is built, so a misspelled `typechecker` fails before any tool is provisioned — where the shell `case` that previously checked it ran after.

One detail is worth recording because it was a shipped bug rather than a hypothetical. A value arriving from a `.env` file or the environment is a string, and the field it lands in may be a tuple; assigning the string leaves it to be splatted one *character* per argument, turning `UV_SYNC_ARGS="--group test"` into `uv sync - - g r o u p`. The field's type is known only where the record is constructed, so that is where the coercion belongs. This and

	source	role
1	dataclass defaults	the shipped answer
2	`.rhiza/.env`	developer-local, untracked
3	`rhiza.toml`	committed, language-neutral
4	the manifest's `[tool.rhiza-task]`	committed, next to the project's own metadata
5	the environment	CI overrides
6	command-line flags	one invocation

Table 1: Resolution order, lowest precedence first. Layer 3 exists because a Go module has no TOML manifest to namespace a table into.

one sibling bug are now covered by property-based tests over arbitrary strings, because both are statements about a class of input rather than about any value a fixture list would have enumerated.

3.5 One name per gate, three implementations

Each of `test`, `coverage`, `typecheck`, `security`, `deps`, `license` and `docs-coverage` exists in the Python, Rust and Go layers, keyed `layer:name` in the registry. Which one answers is decided per repository by detecting the manifests present, so a crate carrying a Python binding package resolves to one gate rather than an ambiguity, and `rust:test` addresses a specific layer explicitly. This replaces `make`'s double-colon rules and the no-op stub declarations a task needed in order to depend on a target that might not have been synced.

The parity is a contract about names, not about mechanism. The Go coverage gate must read a `total:` line out of `go tool cover` because `go test` has no `--fail-under`; the Rust one delegates to `cargo-llvm-cov`; the Python one passes a flag. What consumers depend on is that `coverage` means the same thing and writes `_tests/coverage.xml` in all three.

3.6 A layering invariant held by discipline

The modules form a strict order — configuration and the task model below the tool wrappers, below the task modules, below the runner, below the CLI — with no cycles and no function-local imports. Nothing enforces it. The choice to say so plainly, in the contributor documentation, alongside the two `grep` invocations that check it, is itself part of the design: an invariant that a reader can verify in one command is more durable than one asserted in prose. A function-local import introduced to break a cycle would survive for years, where a module-level cycle crashes immediately and gets fixed.

4 The argument vector as contract

The recipes stated their contract as shell: which tool, which flags, which working directory. Testing that contract previously meant running it, which meant provisioning a toolchain.

The observation this package rests on is that the contract is not the execution — it is the *argument vector*. All tool invocations funnel through the public functions of a single module, which differ in how the tool is provisioned: an isolated one-shot tool, a tool that must import the project's own code, a toolchain binary expected on `PATH`, one whose *stdout* is interpolated into a later argument, and `uv` itself. A test fixture patches every one of them and records what would have run:

```
def test_passes_coverage_flags_when_source_exists(cfg, recorder):
```

```
python.test(cfg)
call = recorder.find("pytest")
assert "--cov=src" in call.flags
assert f"--cov-fail-under={cfg.coverage_fail_under}" in call.flags
```

No test in the suite runs `uv`, or any other tool. The consequences are practical: the suite finishes in under a second, needs no network, and runs identically on all three operating systems — so Windows, which the shell recipes could not support without a forty-line probe, becomes an ordinary matrix cell.

The approach has a precise boundary, and naming it is more useful than defending it. A vector assertion proves that the command is the one intended. It cannot prove that the tool accepts it. A flag that a future `cargo` rejects passes every such test. Section 6 returns to this.

Two categories of behaviour are deliberately tested differently. Configuration parsing is property-based, for the reason given above. And five docstrings carry 39 executable examples covering configuration resolution, exit-status aggregation, registry lookup and guard evaluation; a test collects and runs them, rather than enabling a global doctest flag — which in the shipped gate's argument list would export the policy to every consumer, where whether to gate doctests is their decision.

5 Implementation

The package is 20 modules and roughly 4,200 lines, against 4,900 lines of tests. It declares three runtime dependencies, held down deliberately: the package is provisioned by `uvx` on every gate invocation in CI, so its resolve time is paid once per job. Everything a *task* needs — `pytest`, `ty`, `bandit`, `interrogate` — is provisioned by that task on demand, exactly as the recipes did.

measure	value
registered tasks	62
language-layered / neutral	34 / 28
sections	12
tests	301
statement coverage of `src/`	100 %
suite wall-clock	< 1 s
runtime dependencies	3
CI matrix (OS × Python)	3×4

Table 2: Implementation at the version described. The coverage floor is configured at 100 and enforced wherever the test gate runs.

Two aspects of the validation are worth separating from the numbers.

The floor justifies a structural exemption. The test suite is organised by behaviour rather than mirroring `src/` one-to-one: the twelve task modules share one shape — a guard, a provisioning call, an argument vector — and per-module test files would be twelve copies of one test. A parity checker would report that as disorder, so the repository declares the exemption together with a machine-readable reason, and that reason appeals to the 100 % floor: no module a missing test file would have covered can go unexercised, because an unexercised module would put the total below the floor. The floor and the exemption move together, or neither moves.

The package runs its own gates. A CI job invokes `rhiza-task all`, the same aggregate consumers invoke, so a regression in prerequisite resolution or outcome reporting fails there first. This is one place where being the tool has a cost: a repository consuming the template would run its gates through a published copy of this code rather than the working tree, which is circular, so this repository is deliberately not template-managed and owns its own workflow files.

6 Limitations

Vectors are not executions. As Section 4 states, a vector assertion cannot detect a flag the tool rejects. The mitigation is end-to-end coverage in CI, and it is uneven: the aggregate gate exercises the Python layer against real tools, while the Rust and Go layers — despite full statement coverage — are validated only as vectors unless a job provisions those toolchains. Since the cross-language naming is the package's headline claim, this is the gap most worth closing, and coverage figures actively conceal it.

Statement coverage is not assertion quality. A 100 % floor establishes that no statement is unvisited. It does not establish that any is meaningfully asserted, and once the suite reaches the floor the number can no longer ratchet. No gate here measures assertion quality, so the property is held by review; the executed docstring and fenced examples narrow the gap only where the documentation makes a checkable claim.

Complexity limits stated in prose decay. Four blocks exceed a conventional cyclomatic-complexity threshold, each carrying a comment arguing why the flat form is preferred; three are bounded by closed sets, and the fourth grows by one branch per validated setting and therefore names an explicit ceiling. A ceiling that only a reader checks is the same class of artefact as an unexecuted docstring example: correct when written, and unmonitored precisely where growth is expected.

The single-name assumption. One task name per gate per repository is a simplification. A monorepo with several Python packages under one root has one `source_folder`, and the resolution order offers no per-directory scope.

7 Related work

Modern task runners — `just`, `task`, `invoke`, `poethepoet`, `mise` — improve substantially on `make`'s ergonomics for defining commands, and `nox` and `tox` add environment management. All of them, however, define tasks in a file that lives in the consumer's repository. Adopting one changes the language the recipes are written in without changing how they are distributed, which is the problem addressed here.

Template tools — `cookiecutter`, `copier`, `cruft` — attack distribution directly, and `copier` in particular can re-apply an updated template with a three-way merge. That is a genuine improvement over re-copying, but it remains a merge into files the consumer owns: the update can conflict, and what the consumer runs is still a copy rather than a version. Packaging the behaviour and leaving only configuration in the repository sidesteps the merge entirely, at the cost of requiring that the shared part be expressible as a program with a stable interface.

The closest relative is the sibling package that applied the same reasoning to the template's synced test folder, distributing those checks as a pytest plugin. This work generalises that result from one folder to the gate layer.

8 Conclusion

The gate layer's difficulties were not caused by `make` being an awkward language. They were caused by `include` being unable to name a remote file, which left copying as the only distribution mechanism available and made every subsequent problem a consequence. Replacing the layer with a versioned package removes the cause: there is nothing to copy, nothing to exclude, and nothing to drift.

Two secondary results seem more general than the specific migration. First, expressing a gate's contract as an argument vector rather than as an execution makes an entire infrastructure layer testable without provisioning anything, which is what allowed the Windows support the shell recipes never had — while also drawing a boundary that must then be closed deliberately, per language, in CI. Second, recovering `skipped` as an outcome distinct from `passed` turns out to matter more than its size suggests: a gate that quietly measures nothing is worse than one that fails, because it is indistinguishable from one that works.

References

[1] Jebel Quant Research. *rhiza-task*. `https://github.com/jebel-quant/rhiza-task`.

[2] Jebel Quant Research. *pytest-rhiza*. `https://github.com/jebel-quant/pytest-rhiza`.

[3] Jebel Quant Research. *rhiza*: the project template. `https://github.com/jebel-quant/rhiza`.

[4] Astral. *uv: an extremely fast Python package and project manager*. `https://github.com/astral-sh/uv`.

[5] S. I. Feldman. *Make — A Program for Maintaining Computer Programs*. Software: Practice and Experience, 9(4):255–265, 1979.

[6] *Copier*: a library and CLI app for rendering project templates. `https://github.com/copier-org/copier`.