Rhiza: A Living-Template System for Sustainable Python Project Infrastructure

Thomas Schmelzer Harry Campion Mark Richardson
thomas@jqr.ae

March 2026

Abstract

Software projects accumulate infrastructure debt: continuous-integration pipelines age out of date, linting rules diverge from community standards, and hard-won configuration improvements never propagate from the project that discovered them to its siblings. Ad-hoc fixes compound this drift without a systematic remedy. We present **Rhiza**—from the Greek *rhiza*, "root"—a living-template system for Python projects that decouples the *content* of best-practice infrastructure from the *mechanism* that applies it. Unlike one-time project generators such as Cookiecutter or Copier, Rhiza provides templates that continue to evolve and that downstream projects can pull in selectively at any time. The system is structured around three key abstractions: (i) a versioned template repository that acts as the authoritative source of curated infrastructure files; (ii) a lightweight CLI that fetches and merges those files; and (iii) a modular bundle system that lets teams opt into exactly the features they need. We describe the design rationale, the architectural decisions that guide the implementation, and the operational experience accumulated across more than twenty downstream projects.

Contents

1 Introduction

Every serious Python project carries with it a hidden second project: the infrastructure that makes the first one possible. Continuous-integration pipelines, linting and formatting rules, reproducible environment specifications, pre-commit hooks, security scanners, documentation generators, and Makefile-based development workflows are not incidental—they are the scaffolding on which reliable, maintainable software is built. Yet this scaffolding is almost universally treated as an afterthought.

The typical lifecycle of project infrastructure proceeds as follows. At project creation, a developer copies configuration files from a previous project, consults a blog post, or runs a scaffold tool. The resulting configuration is reasonable for that moment in time. Over months and years, however, the ecosystem moves on: the linter gains new rules, the CI service deprecates old syntax, the recommended Python version changes, a new security vulnerability scanner emerges. The infrastructure falls behind. Keeping it current requires manual effort that no one formally owns, so it rarely happens. The result is configuration drift: each project in an organization diverges from the others and from current best practice, with the gap widening over time.

Rhiza is a response to this problem. Its core thesis is that project infrastructure should be treated as a versioned, updatable artifact—a dependency to be upgraded, not a one-off scaffold to be generated and forgotten. This is not a novel observation; the DevOps and infrastructure-as-code communities have long advocated for treating infrastructure with the same discipline applied to application code [6,9]. What Rhiza contributes is a concrete, opinionated, Python-centric implementation of this philosophy that is lightweight enough for individual developers and teams to adopt without changing their existing workflows.

The name reflects the ambition: just as a root system sustains a plant by continuously supplying water and nutrients, Rhiza aims to continuously supply modern, well-maintained infrastructure to the projects built on top of it.

2 Background and Motivation

2.1 The Configuration-Drift Problem

Configuration drift in software projects is an instance of the broader problem of *technical debt* [3]. Unlike functional technical debt—code that works but is hard to change—infrastructure debt is often invisible: the tests still pass, the CI pipeline still runs, and no immediate failure signals a problem. The harm materialises gradually: security vulnerabilities go undetected, formatting inconsistencies slow code review, and onboarding new contributors takes longer because each project has idiosyncratic conventions.

Organisations that maintain many Python projects—research groups, consultancies, open-source umbrella projects—feel this cost acutely. A configuration improvement discovered in one project must be manually propagated to each other project. If there are ten projects, the propagation cost is multiplied by ten; if there are fifty, the improvement often simply does not propagate at all.

2.2 Existing Approaches and Their Limitations

Several tools exist to help with project scaffolding:

Cookiecutter [12]. Cookiecutter generates a project directory from a Jinja2 template. It solves the *initialisation* problem well but provides no mechanism for synchronising improvements after the initial generation. Once a project is created, it is on its own.

Copier [8]. Copier addresses the synchronisation gap with an `update` command that re-applies a template to an existing project. This is a significant advance. However, Copier's update model applies the entire template wholesale, relying on three-way merges to preserve local customisations. This works well for simple files but can produce conflicts for complex, structured files such as CI workflow YAML or modular Makefiles. Additionally, Copier requires projects to opt into its update mechanism from the start, and the learning curve for resolving update conflicts can be steep.

cruft [7]. cruft adds a Cookiecutter-update workflow via a stored diff approach. It shares Cookiecutter's all-or-nothing update semantics and inherits its limitations.

Template repositories on GitHub. Many organisations maintain "template" repositories that new projects are forked from. This solves initialisation but creates a permanently diverging fork: there is no built-in mechanism to pull upstream changes into child repositories.

Internal tooling. Large engineering organisations sometimes build bespoke internal tools that enforce a standard project layout. These tools solve the problem well within one organisation but are expensive to build, not shared across the community, and often not maintained long-term.

2.3 The Need for Living Templates

The common limitation of the above approaches is that they treat project infrastructure as a *point-in-time* artefact rather than a *continuous* one. Rhiza takes the opposite position: infrastructure files should be versioned, tagged, and upgraded in much the same way that library dependencies are. A project's `.rhiza/template.yml` file specifies which version of the Rhiza template to use, analogously to how `pyproject.toml` specifies which version of a library to install. Upgrading is a deliberate act, but it is easy, reversible, and incremental.

3 System Design

3.1 Architectural Overview

Rhiza is deliberately split into two independent components (ADR-0005):

1. **The template repository** (`jebel-quant/rhiza` on GitHub) is the authoritative source of curated infrastructure files. It is versioned with Git tags and contains the actual content that downstream projects consume.

2. **The CLI** (`rhiza` on PyPI) is the engine that fetches files from the template repository and applies them to a downstream project. It is a separate Python package with its own independent release cadence.

Separating these concerns means that the CLI can be improved—bug fixes, new commands, better conflict resolution—without forcing downstream projects to adopt new template content, and vice versa. It also means that organisations can fork the template repository and maintain their own curated variant while continuing to use the standard CLI.

3.2 Template Bundles

A naive template system maps every file in the template repository to a file in the downstream project. This all-or-nothing approach quickly breaks down: a project that does not use Docker should not receive Docker configuration; a project that is not hosted on GitHub should not receive GitHub Actions workflows.

Rhiza addresses this with *template bundles* (ADR-0006): named, coarse-grained groups of related files. The bundles defined in `.rhiza/template-bundles.yml` include:

Bundle	Contents
`core`	Makefile infrastructure, `pyproject.toml` skeleton, `.python-version`, `uv.lock` patterns
`tests`	pytest configuration, coverage settings, property-based testing scaffold
`github`	GitHub Actions workflows (CI, lint, security, release, sync, docs)
`gitlab`	GitLab CI equivalent with feature parity
`docker`	Dockerfile templates, `.dockerignore`, container Makefile module
`book`	MkDocs configuration, documentation structure, coverage badges
`marimo`	Marimo notebook scaffold, interactive documentation setup
`pre-commit`	`.pre-commit-config.yaml` wired to ruff, black, and isort

Table 1: Selected Rhiza template bundles.

Dependencies between bundles are declared explicitly. For example, the `book` bundle requires `tests` because the documentation site includes auto-generated coverage reports. The CLI enforces these constraints at initialisation and sync time, preventing invalid combinations.

A downstream project declares the bundles it uses in `.rhiza/template.yml`:

```
repository: jebel-quant/rhiza
ref: v0.8.16
templates:
  - core
  - tests
  - github
  - docker
```

Listing 1: Example `.rhiza/template.yml`.

3.3 Modular Makefile Architecture

A single monolithic Makefile is a maintenance burden: it is hard to read, harder to extend without conflicts, and impossible to selectively update. Rhiza adopts a three-layer architecture (ADR-0004):

1. **Root `Makefile`** (nine lines): owned by the downstream project, this file simply includes the Rhiza core.

2. **`.rhiza/rhiza.mk`**: the template-managed core. It auto-includes all files matching `.rhiza/make.d/*.mk` using a glob pattern.

3. **`.rhiza/make.d/*.mk`**: one file per feature area, named with numeric prefixes to control load order. Prefixes `00`–`19` define configuration variables; `20`–`79` define task targets; `80`–`99` define hooks.

Feature modules include `bootstrap.mk` (environment setup), `quality.mk` (linting, formatting), `test.mk` (pytest), `docs.mk` (MkDocs), `github.mk` (GitHub CLI helpers), `docker.mk` (container builds), and `marimo.mk` (notebook server). Targets use GNU Make's double-colon syntax (`::`) where appropriate, which allows pre-hook and post-hook targets in separate files to extend a task without modifying the file that defines it. Developers who need project-local shortcuts that should not be committed can place them in `local.mk`, which is listed in `.gitignore` by default.

This architecture provides more than forty documented `make` targets, all discoverable via `make help`.

3.4 Reproducible Environments with `uv`

A template system that mandates outdated or slow tooling will not be adopted. Rhiza standardises on `uv` [2]—Astral's unified Python version and package manager—for all environment management (ADR-0002). The decision was driven by three considerations:

1. **Single binary**: `uv` is distributed as a self-contained binary. Installing it does not require Python to be present, which simplifies bootstrapping on CI runners and developer machines alike.

2. **Speed**: `uv` resolves and installs packages 10–100× faster than `pip`, which meaningfully reduces CI wait times on projects with large dependency trees.

3. **Lock file**: `uv.lock` pins the entire transitive dependency graph, ensuring that every environment—local, CI, production—is identical.

The `.python-version` file, automatically read by `uv`, pins the Python interpreter version. Together these two files are the minimal, complete specification of a reproducible Python environment.

3.5 Continuous Quality Enforcement

Rhiza integrates a layered quality stack that operates at multiple stages of the development cycle:

Pre-commit hooks. The `pre-commit` bundle installs hooks that run `ruff` for linting and formatting, catching issues before they enter the repository.

CI pipelines. The `github` and `gitlab` bundles provide workflows that run on every pull request: unit tests across Python 3.11–3.14, type checking with `ty`, dependency validation with `deptry`, security scanning with `bandit`, and static analysis with `semgrep` and CodeQL.

Automated dependency updates. A Renovate configuration file, included in the `core` bundle, opens pull requests whenever a pinned dependency or the template `ref` is updated upstream. This closes the loop: Rhiza itself is treated as a dependency that can be automatically proposed for upgrade.

4 Operational Experience

4.1 Downstream Adoption

Rhiza has been applied to more than twenty Python projects spanning quantitative finance libraries, data-pipeline tools, and developer-tooling packages. Across these projects, the mean time to adopt a new infrastructure improvement—for example, upgrading to a new Python version or enabling a new `ruff` lint rule—has fallen from several weeks to typically one or two days: the time needed for Renovate to open a PR against the Rhiza template repository and for each downstream project to rebase against the new tag.

4.2 Conflict Resolution in Practice

The most common source of friction in living-template systems is merge conflicts when a project has customised a file that the template also manages. Rhiza mitigates this in several ways:

1. **Bundle granularity.** By grouping files into coarse bundles, Rhiza reduces the frequency with which any given downstream file is touched by an upstream update.
2. **Makefile hooks.** Downstream projects that need to extend a Makefile target can do so by adding a `::` target in `local.mk` rather than modifying the managed file. The managed file can therefore be updated freely.
3. **Configuration delegation.** Where possible, tool configuration resides in `pyproject.toml` (owned by the downstream project) rather than in separate managed files. The template provides a commented scaffold section; the downstream project fills it in.

4.3 Architecture Decision Records

All non-trivial design decisions in Rhiza are recorded as Architecture Decision Records [10] in `docs/adr/`. This practice serves two purposes. First, it provides newcomers with the *reasoning* behind choices, not just the choices themselves—critical for understanding why, say, `uv` was chosen over `poetry` or why the CLI and template repository are separate packages. Second, it creates a structured audit trail that makes revisiting decisions easier when the context changes.

Nine ADRs have been recorded to date, covering: the use of ADRs themselves (ADR-0001), adoption of `uv` (ADR-0002), CI platform choices (ADR-0003), the modular Makefile architecture (ADR-0004), separation of template from CLI (ADR-0005), the bundle system (ADR-0006), dual GitHub/GitLab support (ADR-0007), Marimo integration (ADR-0008), and pre-commit hook strategy (ADR-0009).

5 Related Work

Rhiza sits at the intersection of several research and engineering topics:

Infrastructure as Code. The IaC movement [6] established that infrastructure should be version-controlled, peer-reviewed, and automatically applied. Rhiza applies this principle to the layer *below* application infrastructure: the configuration files and tooling that enable developers to write and test application code.

Convention over configuration. Ruby on Rails [5] popularised the principle that opinionated defaults reduce cognitive overhead. Rhiza's template bundles encode opinionated defaults for Python projects, with the expectation that most projects will use them as-is and only override where genuinely necessary.

Package managers and dependency pinning. The reproducibility guarantees that Nix [4], Cargo, and modern Python tools like `uv` provide for application dependencies are extended by Rhiza to infrastructure configuration: the `ref` field in `template.yml` pins the exact version of infrastructure used, making environments reproducible not just in terms of library versions but also in terms of development tooling.

Polyglot monorepos. Organisations using monorepos (e.g., [11]) face a similar challenge: keeping build and CI configuration consistent across many packages. Rhiza offers a lighter-weight alternative for organisations that prefer independent repositories.

6 Future Directions

The Rhiza roadmap identifies several directions for future work:

Finer-grained synchronisation. The current sync model operates at the bundle level. Future work will explore file-level and section-level synchronisation, enabling, for example, the update of a single GitHub Actions job within a larger workflow file without touching the rest of the file.

Validation tooling. A `rhiza validate` command currently checks that a project's `template.yml` is consistent with the files on disk. Expanding this to a full compliance report—identifying files that have drifted from the template—would help teams understand the cost of deferred upgrades.

Multi-language support. The core abstractions (versioned templates, named bundles, modular task files) are language-agnostic. Extending Rhiza to support TypeScript and Rust projects would broaden its applicability without requiring fundamental architectural changes.

Community bundle registry. Currently, all bundles live in the canonical Rhiza repository. A community registry would allow third parties to publish bundles—for example, a bundle for a particular cloud provider's deployment tooling—that can be mixed in alongside the core set.

7 Conclusion

Software infrastructure is not a one-time concern. It requires the same ongoing investment as the application code it supports, yet most organisations lack the tooling to make that investment systematically rather than heroically. Rhiza addresses this gap with a living-template system that treats infrastructure configuration as a versioned, upgradeable dependency. Its key contributions are:

1. A two-component architecture that separates template content from the sync mechanism, enabling independent evolution of each.
2. A named bundle system that provides coarse-grained, dependency-aware feature selection.
3. A modular Makefile architecture with hook points that accommodate local customisation without preventing template updates.
4. First-class integration with modern Python tooling (`uv`, `ruff`, `pre-commit`, Marimo) and AI-assisted development workflows.
5. A documented, ADR-driven design process that makes the system's rationale transparent and revisable.

Rhiza is open source under the MIT licence. The template repository is available at https://github.com/jebel-quant/rhiza and the CLI is installable via `uvx rhiza`.

References

[1] Anthropic, Claude Code: Agentic coding in the terminal. https://claude.ai/code, 2025–.

[2] Astral, uv: An extremely fast Python package and project manager. https://docs.astral.sh/uv, 2024–.

[3] Cunningham, W., The WyCash portfolio management system. In *Proc. ACM OOPSLA*, 1992.

[4] Dolstra, E., de Jonge, M., and Visser, E., Nix: A safe and policy-free system for software deployment. In *Proc. LISA*, 2004.

[5] Hansson, D. H., Ruby on Rails. https://rubyonrails.org, 2004–.

[6] Humble, J. and Farley, D., *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[7] Kothari, S. et al., cruft. https://cruft.github.io/cruft, 2019–.

[8] Macías, J. M. et al., Copier. https://copier.readthedocs.io, 2019–.

[9] Morris, K., *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Media, 2016.

[10] Nygard, M. T., Documenting architecture decisions. https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions, 2011.

[11] Potvin, R. and Levenberg, J., Why Google stores billions of lines of code in a single repository. *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[12] Roy, A., Greenfeld, D. et al., Cookiecutter. https://cookiecutter.readthedocs.io, 2013–.