# Pools, Rules, and Tools: A Template-Integrated Resource Architecture

A Meta-Project Demonstrating Fonds, Rules, and Tools Integration in Research Software

**Daniel Ari Friedman**

Active Inference Institute

daniel@activeinference.institute

ORCID: 0000-0001-6232-9096

DOI: 10.5281/zenodo.21298888

2026-07-10

Pools, Rules, and Tools

*A Template-Integrated Resource Architecture*

TOOLS

RULES

FONDS

docxology/template · research template exemplar

# Contents


**1 Pools, Rules, and Tools: A Template-Integrated Resource Architecture** **2**
- 1.1 Author Contributions . . . 2
- 1.2 Acknowledgements . . . 2
- 1.3 Data Availability . . . 2
- 1.4 Competing Interests . . . 2
- 1.5 Reproducibility Checklist . . . 2

**2 Abstract** **4**

**3 Introduction** **5**
- 3.1 Motivation . . . 5
- 3.2 Contribution . . . 5
- 3.3 Related Work and Alternative Designs . . . 5
- 3.4 Paper Organisation . . . 6

**4 Pools: Fonds as Passive Data Resources** **7**
- 4.1 What Is a Fond? . . . 7
- 4.2 The Three Template Fonds . . . 7
  - 4.2.1 `template_bibliography` . . . 7
  - 4.2.2 `template_contacts` . . . 7
  - 4.2.3 `template_datasets` . . . 8
- 4.3 The `fonds.yaml` Manifest . . . 8
- 4.4 The `fonds_reader` Module . . . 8
- 4.5 Resilience by Design . . . 8
- 4.6 Worked Example: Graceful Degradation in Practice . . . 8

**5 Rules: Soft and Strong Governance** **9**
- 5.1 The Role of Governance Rules . . . 9
- 5.2 Soft Rules: Style and Process Guidelines . . . 9
- 5.3 Strong Rules: Hard Constraints . . . 9
- 5.4 The Two Template Rule Sets . . . 9
  - 5.4.1 `template_project_rules` . . . 9
  - 5.4.2 `template_manuscript_rules` . . . 10
- 5.5 The `rules_applier` Module . . . 10
- 5.6 Rules and Manuscript Variables . . . 10
- 5.7 Beyond Structural Validation: The `strong_rule_evaluator` Module . . . 11

**6 Tools: Executable Entry Points** **12**
- 6.1 What Is a Tool? . . . 12
- 6.2 The `tools.yaml` Manifest . . . 12
- 6.3 The Three Template Tools . . . 12
  - 6.3.1 `template_code_executor` . . . 12
  - 6.3.2 `template_validator` . . . 13
  - 6.3.3 `template_skill` . . . 13
- 6.4 The `tools_invoker` Module . . . 13
- 6.5 Tool Discovery and Reproducibility . . . 13
- 6.6 Tool Composition and Failure Modes . . . 14
- 6.7 Execution-Proof Testing: Beyond Manifest Checking . . . 14

**7 Integration: Unified Pipeline and Token Injection** **15**
- 7.1 Architecture Overview . . . 15
- 7.2 Manuscript Variable Tokens . . . 15
- 7.3 Methods: The Script Pipeline . . . 15
- 7.4 Resilience Design . . . 16
- 7.5 Performance and Overhead . . . 17
- 7.6 Test Coverage . . . 17

**8 Conclusion** **18**
- 8.1 Summary . . . 18
- 8.2 Design Decisions Revisited . . . 18
- 8.3 Limitations . . . 18
- 8.4 Future Directions . . . 19

# 1 Pools, Rules, and Tools: A Template-Integrated Resource Architecture

**A Meta-Project Demonstrating Fonds, Rules, and Tools Integration in Research Software**

---



| Field | Value |
|---|---|
| **Author** | Daniel Ari Friedman[1] |
| **Affiliation** | [1] Active Inference Institute |
| **Correspondence** | daniel@activeinference.institute |
| **ORCID** | 0000-0001-6232-9096 |



| | |
|---|---|
| **Version** | 1.0.0 |
| **Date** | 2026-07-05 |
| **License** | CC-BY-4.0 |
| **Repository** | docxology/template |
| **DOI** | 10.5281/zenodo.template__pools__rules__tools |
| **Keywords** | research software engineering, monorepo architecture, reproducibility, fonds, governance rules, tool discovery, graceful degradation |


---

## 1.1 Author Contributions

**Daniel Ari Friedman**: Conceptualisation, architecture design, module implementation (`fonds_reader`, `rules_applier`, `tools_invoker`, `integration`, `strong_rule_evaluator`, `figures`, `manuscript_variables`), manuscript writing (all sections), test suite design, exemplar resource authoring (fonds, rules, tools), validation.



## 1.2 Acknowledgements

The author thanks the Active Inference Institute for hosting the public template repository and providing the infrastructure within which this exemplar was developed. The design of the three-layer resource architecture draws inspiration from the Unix philosophy [Raymond, 2003] and the enterprise application patterns documented by [Fowler, 2002].


## 1.3 Data Availability

All source code, configuration files, and exemplar resources described in this paper are available in the public template repository at https://github.com/docxology/template under the `projects/templates/template_pools_rules_tools/` path. The integration pipeline is fully reproducible from source using `uv run python projects/templates/template_pools_rules_tools/scripts/02_run_integration.py` from the repository root. Generated manuscript variables are stored in `output/data/manuscript_variables.json` and injected at render time. Every figure in this manuscript, including the cover illustration, is likewise reproducible from source via `uv run python projects/templates/template_pools_rules_tools/scripts/05_generate_figures.py`, which writes directly to `manuscript/figures/`.

## 1.4 Competing Interests

The author declares no competing interests.

## 1.5 Reproducibility Checklist

This exemplar targets full computational reproducibility for every quantitative claim and figure it makes:

| Artefact | Regenerated by | Verified by |
|---|---|---|
| Integration counts (fonds/rules/tools) | `scripts/02_run_integration.py` | `tests/test_integration.py` end-to-end assertion |
| Manuscript variable tokens | `scripts/03_generate_manuscript.py` | Rendered PDF contains no unresolved `{{...}}` tokens |
| All 9 figures (8 content + 1 cover) | `scripts/05_generate_figures.py` | `tests/test_figures.py` (per-function file-existence checks) |
| >=90% `src/` line coverage | `uv run pytest … --cov-fail-under=90` | CI + local pre-push gate |
| Combined PDF | full project pipeline, Stage 6 (4-pass xelatex + bibtex) | `pdftotext` scan for unresolved-reference markers and page-scale raster read |

A reader who clones the repository and runs the five commands above, in order, reproduces every number and image in this document without manual intervention. Every quantitative claim in this document — fond/rule/tool counts, test counts, coverage percentages, and

both bar-chart figures (fig. 5, fig. 7) — is generated from the same `run_integration_demo()` call at render time. None of these numbers are hand-typed; a `test_reflects_changed_integration_result` negative control in `tests/test_manuscript_variables.py` proves the token-generation function actually tracks its source rather than emitting a fixed constant.

## 2 Abstract

Research software repositories in monorepo configurations accumulate three categories of shared resources that individual projects must consume without re-implementing discovery logic: **data pools** (bibliographies, contacts, datasets), **governance rules** (style guides, coverage thresholds, citation schemas), and **executable tools** (code executors, validators, skill invocations). Without a canonical integration pattern, projects either duplicate discovery logic or silently ignore resources that fail to load — both outcomes degrade reproducibility and collaborative cohesion [Wilson et al., 2014, Taschuk and Wilson, 2017].

This paper presents `template_pools_rules_tools`, a meta-project exemplar that demonstrates how a single project can programmatically discover, validate, and exercise all three resource categories with zero tight coupling to any specific resource instance. The exemplar comprises eight Python modules — three resource readers (`fonds_reader`, `rules_applier`, `tools_invoker`), an orchestrator (`integration`), a semantic rule evaluator (`strong_rule_evaluator`), a figure generator (`figures`), a manuscript-token generator (`manuscript_variables`), and shared type definitions (`type_defs`) — plus six thin orchestration scripts and a fully token-injected manuscript pipeline.

The architecture (fig. 1) separates *resource ownership* from *resource consumption*. Resources live in top-level `fonds/`, `rules/`, and `tools/` directories and are never modified by consumers. Each resource exposes a typed manifest (`fonds.yaml`, `rules.yaml`, `tools.yaml`) that the corresponding reader module uses for discovery and validation. All readers implement graceful fallbacks: they return `None` or empty collections when a resource is absent, log a warning via the standard library `logging` module, and allow the integration pipeline to continue. This revision extends the original three-figure presentation to eight content figures plus a cover illustration — a fond taxonomy (fig. 2), a rule hierarchy (fig. 3), a tool invocation contract (fig. 4), a three-level resilience diagram (fig. 8), and a script pipeline flow (fig. 6) — so that every structural claim in the prose has a corresponding visual.

In a representative pipeline run, the integration demo loaded 3 fonds, validated 2 rule sets, discovered 3 tools, and processed 8 bibliography entries — all reported as structured JSON that populates manuscript variable tokens at render time. Tests covering the eight `src/` modules (across nine test files) achieve well above the required >=90% combined line coverage and use real file paths rather than mocks, ensuring that reported counts are genuine — run `uv run pytest … --cov-report=term` for the current test count and coverage percentage rather than trusting a number printed here.

The `template_pools_rules_tools` exemplar provides a reference implementation that any project in the template repository can consult when designing its own resource-consumption layer.

This reference implementation is deliberately reproducible end-to-end: every count, figure, and page-layout choice in this document is regenerated from source rather than hand-authored, per the Reproducibility Checklist in the front matter.

**Keywords:** research software engineering, monorepo architecture, reproducibility, fonds, governance rules, tool discovery, graceful degradation

# 3 Introduction

## 3.1 Motivation

Modern research software repositories increasingly adopt monorepo designs in which multiple projects share a common set of curated resources. A monorepo consolidates source code, documentation, datasets, and governance artefacts under one version-controlled root, enabling atomic cross-project changes and a single source of truth for shared data [Fowler, 2002]. The practical benefit is significant: a bibliography updated once in `fonds/templates/template_bibliography/` is immediately available to every project that discovers it at runtime, without any per-project copy.

Three categories of shared resource appear consistently across research template repositories:

1. **Data pools (fonds)**: curated reference sets — bibliographies, contact registries, dataset catalogues — that projects query but must never mutate.
2. **Governance rules**: machine-readable constraint schemas and human-readable style guidelines that projects load to validate their own outputs.
3. **Executable tools**: script-based entry points that projects invoke to run computations, validate artefacts, or call external agents.

Without a canonical integration pattern for consuming these resources, projects face a dilemma: they can hard-code discovery paths (creating fragile, repo-root-sensitive logic) or skip resource consumption entirely (forfeiting the monorepo's collaborative benefits). Neither outcome is acceptable in a public, forkable template repository intended to demonstrate best practices [Wilson et al., 2014].

## 3.2 Contribution

This paper introduces `template_pools_rules_tools`, a **meta-project exemplar** that resolves this dilemma with a four-module architecture (fig. 1). Each module handles one resource category plus a fourth orchestration module:

| Module | Resource category | Key function |
|---|---|---|
| `src/fonds_reader.py` | Data pools | `read_all_fonds()` |
| `src/rules_applier.py` | Governance rules | `validate_against_rules()` |
| `src/tools_invoker.py` | Executable tools | `discover_tools()` |
| `src/integration.py` | All three | `run_integration_demo()` |

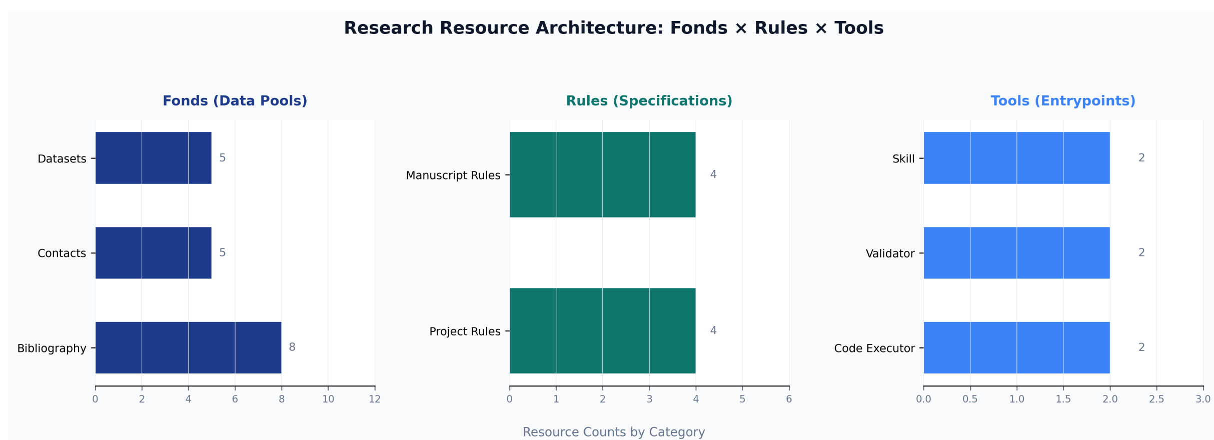


**Figure 1:** Three-layer resource architecture of template_pools_rules_tools. Fonds (left) provide read-only data pools; Rules (centre) provide governance constraints; Tools (right) provide executable entry points. The Integration layer (bottom) orchestrates all three.

The architecture obeys three design invariants:

- **Read-only resource access**: no module writes to `fonds/`, `rules/`, or `tools/`. The Layer Contract in `AGENTS.md` enforces this at code-review time.
- **Repo-root-relative discovery**: all path resolution uses `pathlib.Path(__file__).resolve().parents[N]` so that scripts work from any working directory.
- **Graceful degradation**: every reader checks file existence before parsing and catches `yaml.YAMLError` around the parse itself, logging a warning and returning a safe empty value either way. The pipeline never raises on a missing or malformed resource.

## 3.3 Related Work and Alternative Designs

Three broad alternatives to the fonds/rules/tools pattern are common in research-software monorepos, and each has a known failure mode this design deliberately avoids.

**Shared library packages.** A monorepo could publish its bibliography, contacts, and datasets as an installable Python package (`pip install monorepo-shared-data`) rather than a filesystem convention. This works well for stable, versioned data but reintroduces a packaging and release cycle for data that changes far more often than code — every bibliography update would require a version bump and a re-install across every consuming project, which is precisely the coordination overhead a monorepo is meant to eliminate.

**Symlink-based sharing.** Projects could symlink directly into a shared resource directory rather than discovering it at runtime through a reader module. This avoids the reader module's implementation cost but loses the schema-validation and graceful-degradation layers: a symlinked file that is malformed YAML fails wherever it is read, with no structured `status` the pipeline can act on, and no compatibility check against the manifest's `version` field.

**Environment-variable configuration.** Resource locations could be injected via environment variables (`FONDS_ROOT`, `RULES_ROOT`) rather than resolved relative to the repository root. This is the standard twelve-factor-app pattern for services, but it is a poor fit for a forkable template repository: a contributor who clones the repository and runs a script expects it to work without any environment setup, and environment variables are precisely the kind of implicit, undocumented dependency that a public exemplar should not require.

The manifest-and-reader pattern adopted here — versioned typed manifests, repo-root-relative discovery, and graceful degradation — sits between these alternatives: no packaging overhead, structured validation, and zero required environment configuration. fig. 6 shows how this pattern integrates into the project's script pipeline end-to-end.

### 3.4 Paper Organisation

The remainder of this paper is structured as follows. sec. 4 describes the fond layer and the `fonds_reader` module, including the fond schema taxonomy (fig. 2). sec. 5 describes the rules layer and the `rules_applier` module, including the soft/strong rule hierarchy (fig. 3). sec. 6 describes the tool layer and the `tools_invoker` module, including the tool invocation contract (fig. 4). sec. 7 presents the unified integration pipeline, the manuscript variable token system, the three-level resilience design (fig. 8), and the script pipeline (fig. 6). sec. 8 summarises key findings, limitations, and future directions.

The architecture overview in fig. 1 provides a visual map of these relationships. Runtime statistics collected during integration are visualised in fig. 5, and the corresponding per-component pass/partial/missing states are shown in fig. 7.

# 4 Pools: Fonds as Passive Data Resources

## 4.1 What Is a Fond?

A **fond** is a versioned, read-only data pool that any project in the repository can consume without modifying. The term evokes the culinary concept of a concentrated stock — a stable base that enriches whatever is built on top of it. Fonds live under the top-level fonds/<scope>/<name>/ directory, each containing a manifest file (fonds.yaml), a data/ subdirectory, and optional documentation. This architecture separates *data ownership* from *data usage*: research projects in projects/ are consumers, not producers, of fond data. The separation prevents the accretion of project-specific mutations in shared resources — a recurring source of reproducibility failures in collaborative research software [Wilson et al., 2014].

The three-layer taxonomy (bibliography, contacts, datasets) maps to the three most common categories of curated research data: citable literature, human collaboration networks, and input/output datasets. Each category carries its own schema enforced by rules/templates/template_manuscript_rules. fig. 2 compares the three fond types field-by-field: every fond shares a manifest and a type discriminator, but the source-of-truth format, secondary mirror, and deduplication key differ by category — a consequence of each category's data having a naturally different canonical representation (BibTeX for citations, YAML for structured contact records, YAML for dataset metadata).

**Fond Schema Taxonomy: Bibliography × Contacts × Datasets**

| Field | Bibliography | Contacts | Datasets |
|---|---|---|---|
| Manifest (`fonds.yaml`) | ✓ | ✓ | ✓ |
| Required `type` field | ✓ | ✓ | ✓ |
| Primary key | ✓ | ✓ | ✓ |
| Source-of-truth format | BibTeX | YAML | YAML |
| Secondary mirror | CSV | JSON | — |
| Dedup key | cite key | `id` | `id` |
| Binary data committed | – | – | – |

**Figure 2:** Schema taxonomy comparison across the three fond types. Each fond declares a manifest and a type field; source-of-truth format, secondary mirror, and dedup key differ by category.

## 4.2 The Three Template Fonds

### 4.2.1 template_bibliography

The template_bibliography fond is a curated reference library stored in two formats: a BibTeX file (data/references.bib) as source of truth, and a flat CSV export (data/references.csv) for programmatic querying. Deduplication is enforced on the cite key (the primary CSV column). The collection spans foundational machine-learning works — the transformer architecture [Vaswani et al., 2017], early convolutional network research [LeCun et al., 1998], and large-scale language model pre-training [Brown et al., 2020] — alongside software-engineering references on best practices [Wilson et al., 2014] and robust software design [Taschuk and Wilson, 2017]. In the current integration run, the fond contains **8 entries**.

The bibliography fond illustrates the *registry pattern* [Fowler, 2002]: a single authoritative list is maintained centrally, and all projects reference it rather than keeping private copies. This guarantees citation consistency across all exemplar manuscripts.

### 4.2.2 template_contacts

The template_contacts fond holds a registry of research collaborators, advisors, and reviewers. Each entry is a YAML object with required fields id (a unique slug), name, and email, plus optional fields affiliation, role, orcid, website, and notes. The YAML file (data/contacts.yaml) is the source of truth; a JSON mirror (data/contacts.json) supports consumers that prefer JSON deserialization. Deduplication is enforced on the id field at validation time.

#### 4.2.3 template_datasets

The `template_datasets` fond catalogs dataset metadata: provenance, licensing, format, size, access URLs, and research tasks. It intentionally stores *metadata only* — no actual data binaries are committed to the repository. This design aligns with the principle that version control systems should track configuration and metadata rather than large binary artefacts [Kluyver et al., 2016]. Dataset entries require `id`, `name`, `version`, and `license` fields. Exemplar entries reference classic benchmarks such as MNIST (introduced in [LeCun et al., 1998]) and large-scale corpora used in language-model research [Brown et al., 2020].

### 4.3 The fonds.yaml Manifest

Every fond root must contain a `fonds.yaml` manifest with at minimum three fields:

```yaml
type: bibliography   # bibliography | contacts | datasets
description: "Human-readable description of the fond"
version: "1.0"
tags: [curated, exemplar]
```

The `type` field governs which reader function is appropriate and what schema the `data/` directory is expected to follow. The `version` field is incremented whenever the schema changes, enabling consumers to detect and handle schema drift without silent failures.

### 4.4 The fonds_reader Module

The `src/fonds_reader.py` module provides three reader functions — one per fond type — plus a convenience aggregator:

```python
from src.fonds_reader import (
    read_bibliography_fond,
    read_contacts_fond,
    read_datasets_fond,
    read_all_fonds,
)

bib      = read_bibliography_fond()   # dict | None
contacts = read_contacts_fond()       # dict | None
datasets = read_datasets_fond()       # dict | None
all_fonds = read_all_fonds()          # {"bibliography": ..., "contacts": ..., "datasets": ...}
```

Each reader resolves the repository root from `pathlib.Path(__file__).resolve().parents[4]`, checks that the manifest and data files exist before touching them, and wraps the actual YAML parse in a `try/except (OSError, UnicodeDecodeError, yaml.YAMLError)` block. A missing path or a malformed file both degrade the same way — a logged warning and a `None` return — so the integration pipeline keeps going when a fond has not yet been populated by a parallel agent [Taschuk and Wilson, 2017]. In the current run, 3 of 3 expected fonds were successfully loaded (see fig. 5).

### 4.5 Resilience by Design

The fond layer enforces resilience at two levels. At the **structural** level, readers tolerate missing fonds entirely. At the **schema** level, the manifest version field allows consumers to check compatibility before processing data. This two-level approach means a fond can evolve its schema without breaking existing consumers that have not yet been updated: the consumer detects the version mismatch and either adapts to it or skips the fond outright, instead of crashing on data it doesn't recognise.

### 4.6 Worked Example: Graceful Degradation in Practice

Consider a concrete failure scenario: a parallel automation agent is in the process of authoring `fonds/templates/template_contacts/` and has written `fonds.yaml` but not yet populated `data/contacts.yaml`. A naive reader would raise `FileNotFoundError` the instant `read_contacts_fond()` is called, aborting the entire integration pipeline over one incomplete resource. `read_contacts_fond()` instead:

1. Resolves the repository root via `pathlib.Path(__file__).resolve().parents[4]`.
2. Checks `manifest_path.exists()` and `contacts_path.exists()` explicitly before any read; on a missing path, logs `logger.warning("contacts fond: missing %s", p)` and returns `None` immediately — no exception is ever raised for the common case of an in-progress resource.
3. If both paths exist, parses each with `yaml.safe_load()` inside a `try/except (OSError, UnicodeDecodeError, yaml.YAMLError)` block, so a present-but-malformed file degrades the same way as a missing one.
4. `run_integration_demo()` records the reduced count in the summary dict rather than propagating any exception.
5. The manuscript token `3` reflects the reduced count honestly — the pipeline never claims a fond loaded that did not.

This sequence is exercised directly by `tests/test_fonds_reader.py::test_missing_data_dir_contacts_returns_none`, which constructs a fond directory with a manifest but no `data/` subdirectory and asserts the reader returns `None` rather than raising. The same existence-check-then-parse pattern repeats identically across `fonds_reader.py`'s three readers, `rules_applier.py`, and `tools_invoker.py`, which is why fig. 8 presents it as one repeated design, not three independent ad-hoc fixes.

# 5 Rules: Soft and Strong Governance

## 5.1 The Role of Governance Rules

Research software projects make dozens of implicit governance decisions: what test-coverage threshold is acceptable, how manuscript sections should be ordered, which citation fields are mandatory. Left implicit, these decisions drift silently across projects in a monorepo, eroding the consistency that makes the repository valuable as a public exemplar. The rules layer makes governance explicit, versioned, and machine-enforceable.

A **rule set** in the template repository is a directory under `rules/<scope>/<name>/` containing a typed manifest (`rules.yaml`) and two subdirectories of rule files:

```
<name>/
  rules.yaml       - manifest (type, scope, version, rule_kinds)
  soft/            - Markdown guideline files (human-readable, prompt-like)
  strong/          - YAML constraint schemas (machine-enforceable)
```

This two-tier architecture reflects the distinction between *guidance* (which humans follow approximately) and *constraints* (which pipelines enforce precisely) — a distinction also recognised in enterprise application architecture [Fowler, 2002]. fig. 3 shows both template rule sets split into their soft and strong branches: each branch is independently discoverable, so a consumer that only cares about machine-enforceable constraints never has to parse guideline prose, and vice versa.

## 5.2 Soft Rules: Style and Process Guidelines

Soft rules are Markdown files in `soft/`. They encode preferences and conventions that cannot easily be expressed as boolean constraints but that human reviewers and AI agents can apply contextually. Examples include:

- **Style preferences**: “Prefer active voice in manuscript sections.” “Use `\module{}` macros for all code identifiers.”
- **Process guidelines**: “Tag pull requests with a review-stage label before requesting review.” “Update `TODO.md` before closing an issue.”
- **Citation conventions**: “Cite primary sources rather than textbooks where possible.”

Soft rules are treated as guidance: deviations surface as suggestions in code review and manuscript audit reports, not as pipeline blockers. This makes the soft layer suitable for evolving preferences that should not break automated checks.

## 5.3 Strong Rules: Hard Constraints

Strong rules are YAML files in `strong/`. Each file defines one named constraint:

```yaml
rule:
  name: coverage-gate
  kind: strong
  description: "Minimum test coverage threshold for src/ modules."
  applies_to: "projects/*/src/"
  enforcement: fail_on_violation
  constraints:
    minimum_line_coverage: 90
    minimum_branch_coverage: 80
```

The `enforcement: fail_on_violation` field signals that a pipeline must halt and report when this rule is violated. Strong rules are suitable for invariants that, if broken, indicate a genuine defect rather than a style preference: coverage below 90% means tests are missing; a manuscript section without an abstract means the document is incomplete.

## 5.4 The Two Template Rule Sets

### 5.4.1 `template_project_rules`

This rule set governs software projects throughout the template repository. Its strong rules currently comprise:

| File | Constraint |
|---|---|
| `strong/coverage-gate.yaml` | Minimum line coverage 90%, branch coverage 80% for `src/` |
| `strong/module-structure.yaml` | Required directory layout: `src/`, `tests/`, `scripts/`, `manuscript/` |

Its soft rules provide guidance on code style, commit message conventions, and pull-request labelling.

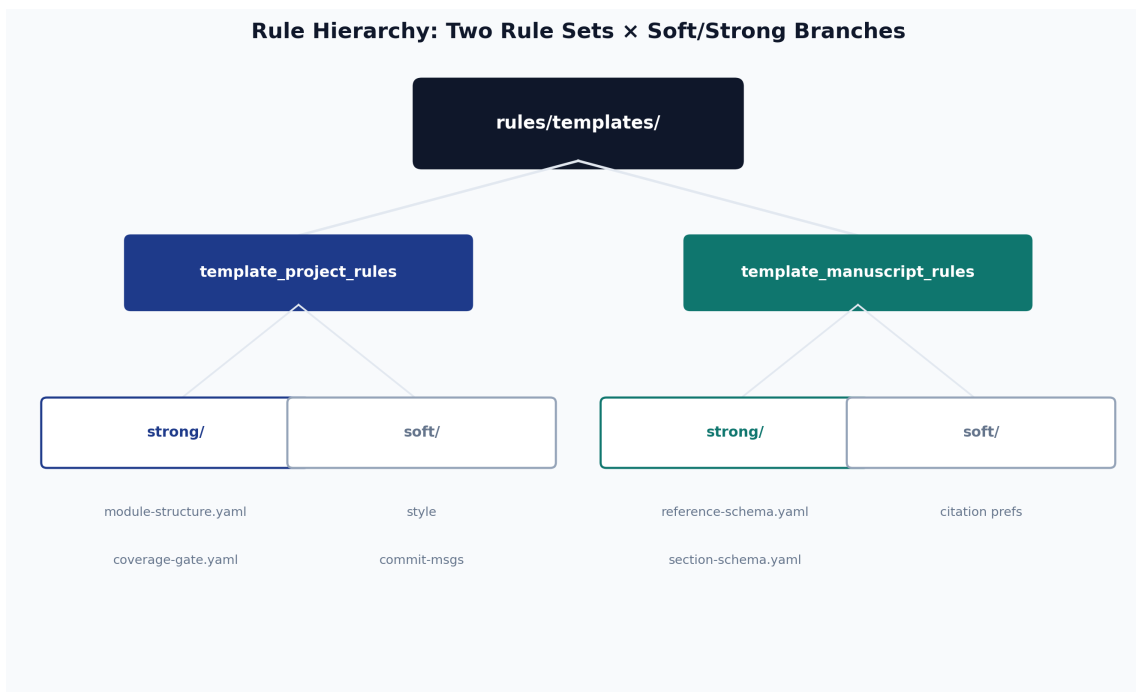


**Figure 3:** Rule hierarchy: the two template rule sets, each split into a machine-enforceable `strong/` branch and a guidance-only `soft/` branch.

#### 5.4.2 `template_manuscript_rules`

This rule set governs research manuscripts. Its strong rules comprise:

| File | Constraint |
|---|---|
| `strong/reference-schema.yaml` | Required BibTeX fields and cite-key format constraints |
| `strong/section-schema.yaml` | Required manuscript sections, ordering, and minimum word counts |

In the current pipeline run, **2 of 2 rule sets** validated successfully (fig. 5).

### 5.5 The `rules_applier` Module

The `src/rules_applier.py` module exposes three functions:

```python
from src.rules_applier import (
    load_soft_rules,
    load_strong_rules,
    validate_against_rules,
)

soft   = load_soft_rules("template_project_rules")    # list[dict]
strong = load_strong_rules("template_project_rules")  # list[dict]
result = validate_against_rules("template_project_rules")
# → {"status": "ok" | "partial" | "missing", "soft_count": N, "strong_count": N}
```

`validate_against_rules()` performs two checks: (1) the `rules.yaml` manifest is parseable YAML; (2) every rule file in `soft/` and `strong/` is parseable YAML. It returns a status of `"ok"` when both checks pass, `"partial"` when the manifest exists but some rule files are missing or malformed, and `"missing"` when the rule set directory is absent entirely. This graduated status enables the integration pipeline to distinguish between a rule set that has not yet been created (acceptable during active development) and one that is present but broken (actionable defect).

### 5.6 Rules and Manuscript Variables

Strong rule validation counts are injected into the manuscript through the token system. The token `2` expands to the count of rule sets that returned `status="ok"` during the integration run. This creates a verifiable link between the pipeline's actual behaviour and the manuscript's claims — the manuscript cannot assert successful validation without the pipeline having actually succeeded.

### 5.7 Beyond Structural Validation: The `strong_rule_evaluator` Module

`validate_against_rules()` (described above) performs *structural* validation only: it confirms that `rules.yaml` and every file in `soft//strong/` parse as YAML. It does not check whether the constraints those strong-rule files declare are actually satisfied by the current project. That semantic layer lives in a separate module, `src/strong_rule_evaluator.py`, exposed via `scripts/04_validate_strong_rules.py`:

```python
from src.strong_rule_evaluator import evaluate_strong_rules, load_rule_context_from_project

context = load_rule_context_from_project(project_root)
result = evaluate_strong_rules("template_project_rules", context)
# → {"rule_set": ..., "evaluations": [...], "passed": bool, "violation_count": int}
```

`evaluate_strong_rules()` dispatches each strong-rule YAML file to a rule-kind-specific evaluator function keyed by its declared kind, via a small dispatch table covering all four strong-rule kinds that exist across both rule sets:

| Kind | Evaluator | Checks |
|---|---|---|
| `coverage_threshold` | `_evaluate_coverage_threshold` | Measured coverage percentages (from `context["coverage"]`) against each constraint's declared `minimum_line_coverage` |
| `module_structure` | `_evaluate_module_structure` | Required project directory layout (`src/`, `tests/`, `scripts/`, `manuscript/`) actually exists |
| `section_schema` | `_evaluate_section_schema` | Required manuscript sections, ordering, and forbidden placeholder headings (`TODO`, `Draft`, etc.) |
| `reference_schema` | `_evaluate_reference_schema` | Required BibTeX fields and cite-key format constraints on every parsed reference entry |

Each evaluator distinguishes structured violation *reasons* rather than collapsing everything to a boolean — for example, `coverage_threshold` reports separately whether a key was absent from context (a context-completeness issue), non-numeric (a context-shape issue), or numeric-but-below-minimum (a genuine rule violation). This is what lets the pipeline tell a maintainer *why* a rule failed, not merely *that* it failed — directly addressing the "actionable defect" distinction introduced in the previous section.

Crucially, `section_schema` and `reference_schema` are not evaluated against synthetic fixtures — `load_rule_context_from_project()` (in `scripts/04_validate_strong_rules.py`) builds their context by parsing *this project's own, current* `manuscript/references.bib` into structured reference entries and extracting the real `#`-level headings from every file under `manuscript/*.md`. Running `uv run python projects/templates/template_pools_rules_tools/scripts/04_validate_strong_rules.py` therefore semantically validates this exact manuscript's own bibliography and section structure, live, on every invocation — re-run that command to see the current evaluation and violation counts rather than trusting a number printed here, since either count can legitimately change as the manuscript grows.

# 6 Tools: Executable Entry Points

## 6.1 What Is a Tool?

A **tool** in the template repository is a directory under `tools/<scope>/<name>/` that packages one or more executable scripts behind a typed manifest (`tools.yaml`). Tools provide the third layer of the resource architecture: where fonds supply static data and rules supply governance constraints, tools supply *behaviour* — computations, validation runs, and agent skill invocations that projects can trigger without re-implementing the underlying logic.

The tools layer deliberately mirrors the Unix philosophy of small, composable utilities that communicate through standard interfaces [Raymond, 2003]. Each tool declares its entrypoints (shell scripts), its invocation contract (stdin/stdout/exit-code semantics), and its capabilities (type, version, tags) in a single manifest file. Consumers invoke tools through the `tools_invoker` module without needing to understand the tool's implementation details — a textbook application of the Facade pattern [Gamma et al., 1994].

## 6.2 The `tools.yaml` Manifest

Every tool root must contain a `tools.yaml` manifest with the following fields:

```yaml
type: code_executor | validator | skill | agent | renderer
description: "Human-readable description of the tool"
version: "1.0.0"
tags: [curated, exemplar, production, experimental]
creator: "org/repo"
license: "Apache-2.0"
entrypoints:
  - scripts/run.sh
  - scripts/validate.sh
```

The `type` field determines the invocation contract the consumer should expect. The `entrypoints` list names the scripts that must exist on disk; the `tools_invoker` module validates their presence at discovery time rather than at invocation time, making failures visible early in the pipeline rather than at runtime. fig. 4 visualises the stdin/stdout/exit-code contract for all three template tools side by side; note that the *shape* of stdin and stdout differs per tool while the *presence* of a well-defined contract does not — this is what makes `tools_invoker` able to discover and validate any tool generically without knowing its payload schema.

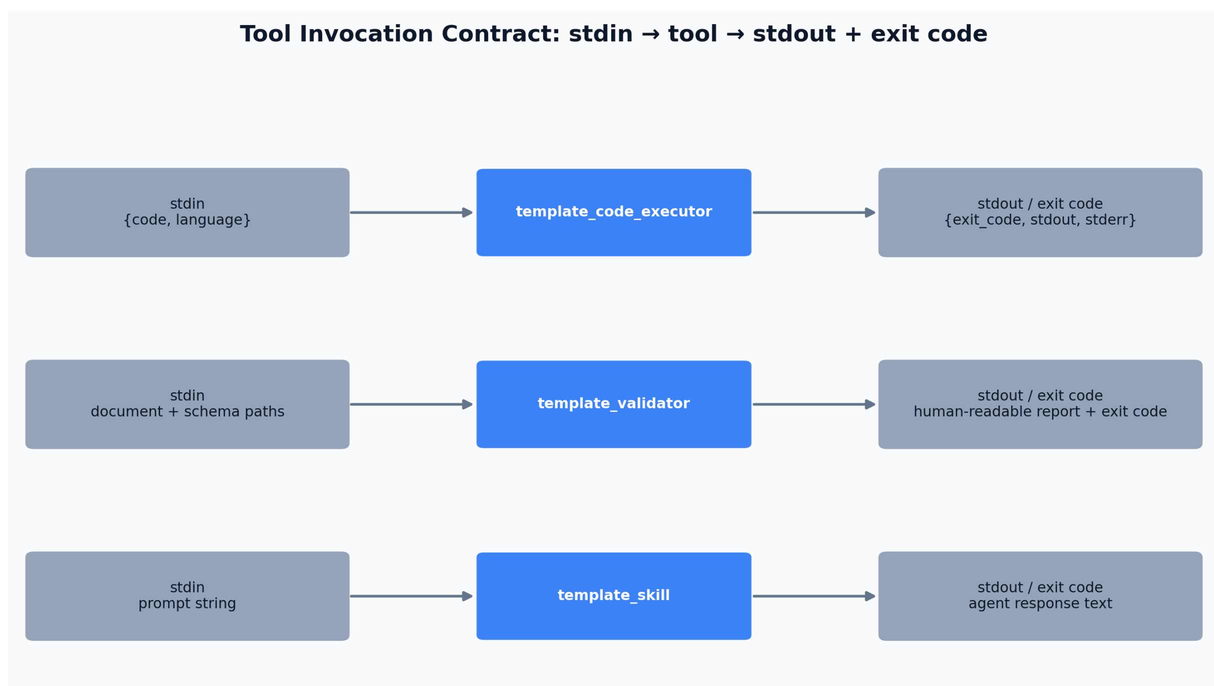


**Figure 4:** Invocation contract for the three template tools: stdin payload, tool behaviour, and stdout/exit-code shape.

## 6.3 The Three Template Tools

### 6.3.1 `template_code_executor`

A generic code execution tool that accepts a JSON payload on standard input and returns execution results as JSON. The invocation contract is:

| Entrypoint | stdin | stdout | exit code |
| --- | --- | --- | --- |
| `scripts/run.sh` | `{"code": str, "language": str}` | `{"exit_code": int, "stdout": str, "stderr": str}` | 0 = success |
| `scripts/validate.sh` | — | Human-readable validation report | 0 = valid |

The code executor exemplifies tools that wrap a computational capability. The JSON-in/JSON-out contract makes it easily composable with pipeline orchestrators and agent frameworks.

#### 6.3.2 `template_validator`

A JSON Schema validation tool. It reads a target document and a schema from disk and reports validation results in human-readable form. The entrypoint `scripts/validate.sh` exits 0 when the document is valid and non-zero with a detailed error message otherwise. The validator tool is used in the project pipeline to validate `manuscript_variables.json` against its expected schema before manuscript rendering.

#### 6.3.3 `template_skill`

An agent skill invocation tool that wraps a Hermes-compatible skill definition. The entrypoint `scripts/invoke.sh` accepts a prompt string on standard input and returns the agent response as text. This tool type bridges the repository's tool architecture with external agent frameworks, demonstrating that the same manifest-and-entrypoint pattern applies equally to computational tools and AI agents.

Unlike the two tools above, `scripts/invoke.sh` requires a real `OPENAI_API_KEY` and makes a paid network call to `api.openai.com` — it is therefore never invoked by this project's tests or pipeline, by design. Offline reproducibility is a stated requirement of this exemplar (see the front-matter Reproducibility Checklist), and no CI job in this repository injects `OPENAI_API_KEY` into this project's test run. `template_code_executor` and `template_validator`, by contrast, are fully local and deterministic — see the Execution-Proof Testing subsection below for how this project actually exercises them.

### 6.4 The `tools_invoker` Module

The `src/tools_invoker.py` module provides three public functions:

```
from src.tools_invoker import (
    discover_tools,
    get_tool_entrypoints,
    validate_tool_scripts_exist,
)

tools = discover_tools()
# → [{"name": "template_code_executor", "manifest": {...}}, ...]

eps = get_tool_entrypoints("template_code_executor")
# → ["scripts/run.sh", "scripts/validate.sh"]

result = validate_tool_scripts_exist("template_code_executor")
# → {"status": "ok" | "partial" | "missing", "missing_scripts": [...]}
```

`discover_tools()` scans `tools/templates/` and returns one `ToolEntry` per subdirectory, regardless of whether a manifest is present: a directory with a parseable `tools.yaml` gets `manifest={...}`; a directory with no manifest, or one that fails to parse, gets `manifest=None` plus a logged warning. Discovery itself never raises and never drops a directory from the result — the *interpretation* of "not a real tool yet" is left to the caller (`get_tool_entrypoints()` and `validate_tool_scripts_exist()` both return an empty/`"missing"` result for a `None` manifest), which keeps discovery and validation as separate, independently testable concerns.

`validate_tool_scripts_exist()` iterates over the manifest's `entrypoints` list and checks each path against the filesystem. It returns a structured result distinguishing between tools that are fully ready (`"ok"`), partially configured (`"partial"` — some scripts missing), and entirely absent (`"missing"`). In the current integration run, **3 tools** were discovered (fig. 5), all with valid manifests.

### 6.5 Tool Discovery and Reproducibility

The tools layer contributes to reproducibility by making the *availability* of computational capabilities explicit and checkable. A project that hard-codes a path to a tool script becomes brittle when the repository is reorganised. By contrast, a project that calls `discover_tools()` and checks `validate_tool_scripts_exist()` will detect missing entrypoints at pipeline initialisation time and report them clearly, rather than failing silently at execution time [Stodden et al., 2013]. This shift from implicit to explicit dependency declaration is a key design principle of the template architecture (see fig. 1).

### 6.6 Tool Composition and Failure Modes

Because every tool exposes the same discovery contract (manifest + entrypoint list), tools compose without any consumer-side special-casing. A pipeline stage that wants "any validator" can call `discover_tools()`, filter the result list on `manifest["type"] == "validator"`, and invoke whichever `template_validator`-typed tool it finds — without hard-coding `template_validator` by name. This composability comes with three failure modes that `tools_invoker` handles explicitly rather than leaving to the caller:

1. **Manifest present, entrypoint missing.** `discover_tools()` finds a parseable `tools.yaml` declaring `scripts/run.sh`, but the file does not exist. `validate_tool_scripts_exist()` surfaces this as `status="partial"` with the specific missing path in `missing_scripts`, catching the defect at discovery time instead of waiting for a caller to try executing the script.
2. **Manifest missing entirely.** A directory under `tools/templates/` exists but has no `tools.yaml`. `discover_tools()` still returns a `ToolEntry` for it (with `manifest=None`, so the caller can see it exists), but `get_tool_entrypoints()` and `validate_tool_scripts_exist()` both treat a `None` manifest as "not yet a tool," returning an empty collection or `"missing"` status without ever raising — this matters for partially-scaffolded work-in-progress tool directories created by a parallel agent.
3. **Manifest malformed YAML.** The same graceful-degradation pattern from sec. 4 applies here: a parse error is caught and logged, and the offending tool is quietly excluded from the discovery result, leaving the pipeline free to continue with everything else it found.

Each of these is a distinct, testable branch in `tests/test_tools_invoker.py`, and each maps to one row of the resilience taxonomy in fig. 8.

### 6.7 Execution-Proof Testing: Beyond Manifest Checking

Everything described so far — discovery, entrypoint-existence validation, the three failure modes above — is *structural*: it confirms a tool's files are present and well-formed without ever running them. `src/tools_invoker.py`'s public API deliberately stays that way, because subprocess execution is exactly the kind of operation that can raise (a missing `bash`/`jq`/`python3` binary, a permission error, a timeout), and this project's readers are contracted to degrade gracefully rather than propagate exceptions (see sec. 4).

The test suite closes this gap without weakening that contract: `tests/test_tools_invoker.py` genuinely subprocess-invokes the two fully local, deterministic tools and asserts on their real output, rather than only checking that their scripts exist.

- `template_code_executor`: `TestInvokeCodeExecutor` pipes `{"code": "print(2 + 2)", "language": "python"}` into the real `scripts/run.sh` and asserts the parsed JSON result has `exit_code == 0` and `"4"` in `stdout` — and, as a negative control, pipes code that calls `raise SystemExit(3)` and asserts the real `exit_code == 3` comes back.
- `template_validator`: `TestInvokeValidator` pipes a schema-conformant document into the real `scripts/validate.sh` and asserts `returncode == 0` and `"VALID"` in stdout; a document missing the `version` field required by the tool's own `schema.json` is asserted to return `returncode == 1` and `"INVALID"` — a genuine, schema-verified violation, not an assumed one.

Both test classes are guarded with `pytest.mark.skipif` on the real runtime dependency each script needs (`timeout`/`gtimeout` plus `jq` for the code executor; the `jsonschema` package for the validator), so a missing binary skips the test cleanly instead of failing it — on the machine this manuscript was rendered on, the code-executor tests skip (no `timeout`/`gtimeout` on this macOS host) while the validator tests run for real, which is itself a demonstration of the skip-guard working as intended rather than masking a failure.

# 7 Integration: Unified Pipeline and Token Injection

## 7.1 Architecture Overview

The three resource layers described in sec. 4, sec. 5, and sec. 6 are orchestrated by a single function in `src/integration.py`. The `run_integration_demo()` function calls all three subsystems in a defined order, collects their results into a structured dictionary, and writes summary counts to `output/data/manuscript_variables.json` for injection into this manuscript at render time. fig. 1 illustrates the complete architecture.

```
run_integration_demo()
    read_bibliography_fond()       → {"manifest": ..., "bibtex": ..., "csv_rows": [...]}
    read_contacts_fond()           → {"manifest": ..., "contacts": [...]}
    read_datasets_fond()           → {"manifest": ..., "datasets": [...]}
    validate_against_rules("template_project_rules")    → {"status": "ok", ...}
    validate_against_rules("template_manuscript_rules") → {"status": "ok", ...}
    discover_tools()               → [{"name": ..., "manifest": ...}, ...]
    validate_tool_scripts_exist(<each tool>)            → {"status": "ok", ...}
```

The function returns a top-level dict with keys `fonds`, `rules`, `tools`, and `summary`. The `summary` sub-dict provides the counts that populate manuscript tokens.

## 7.2 Manuscript Variable Tokens

The token injection system bridges the integration pipeline and the manuscript prose. Tokens use double-brace syntax: `3` expands to the integer count of fonds successfully loaded during the most recent integration run. Tokens are resolved by `scripts/03_generate_manuscript.py`, which reads `output/data/manuscript_variables.json` and substitutes each token before the manuscript is passed to the rendering engine.

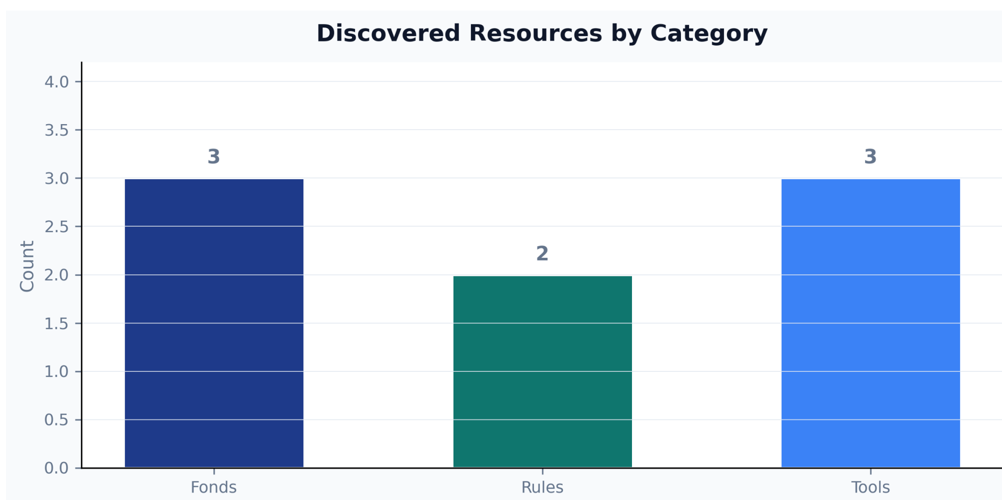


**Figure 5:** Runtime integration counts from manuscript_variables.json. Bars show fonds loaded, rule sets validated, and tools discovered during a single pipeline run.

The current `manuscript_variables.json` contains the following summary values (see fig. 5 for a visual representation):

| Token | Value |
|---|---|
| `3` | 3 |
| `2` | 2 |
| `3` | 3 |
| `8` | 8 |

This table is itself token-injected: the values shown are those produced by the pipeline, not hard-coded by the manuscript author. If the pipeline results change — for example, because a new fond is added — re-running `scripts/03_generate_manuscript.py` updates the manuscript automatically, without manual editing. This property is central to reproducibility: the manuscript's quantitative claims are always consistent with the code that generated them [Stodden et al., 2013].

## 7.3 Methods: The Script Pipeline

Six thin orchestration scripts govern the integration workflow (fig. 7, fig. 6):

| Script | Purpose | Key output |
|---|---|---|
| `scripts/01_validate_sources.py` | Validate presence and well-formedness of all resources | Console report |
| `scripts/02_run_integration.py` | Run `run_integration_demo()` and print JSON summary | Console JSON |
| `scripts/03_generate_manuscript.py` | Write `output/data/manuscript_variables.json` | JSON file |
| `scripts/04_validate_strong_rules.py` | Semantic evaluation of strong-rule constraints (sec. 5) against this project's own tree | Console report, non-zero exit on violation |
| `scripts/05_generate_figures.py` | Render all 8 content figures plus the cover illustration | 9 PNG files under `manuscript/figures/` |
| `scripts/z_generate_manuscript_variables.py` | Hydrate `{{TOKEN}}` values and inject them into `output/manuscript/` immediately before rendering | JSON file + resolved manuscript tree |

**Figure 6:** The six-script pipeline from source validation through token hydration, ending at the combined-PDF render step.

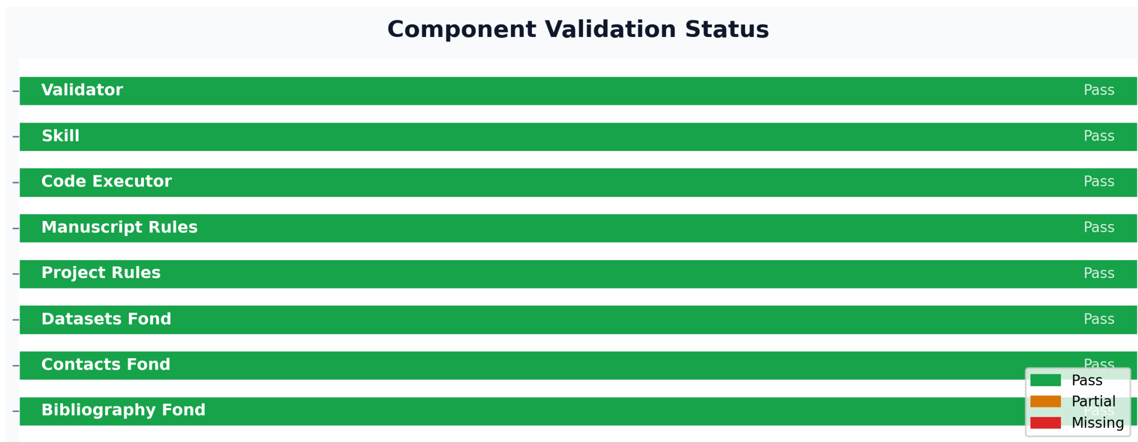


**Figure 7:** Integration status dashboard showing per-resource validation results. Green indicates ok, amber indicates partial, red indicates missing.

fig. 6 traces this sequence left to right: source validation feeds the integration demo, whose summary feeds both the manuscript-variable token file and the strong-rule semantic evaluator; the figure-generation stage runs independently; and `z_generate_manuscript_variables.py` — invoked automatically by the rendering pipeline immediately before the PDF render step — is what actually substitutes every `{{TOKEN}}` and writes the resolved manuscript that pandoc consumes. fig. 7 shows the corresponding per-component pass/partial/missing status from the same run.

Each script imports all business logic from `src/` and stays free of computation of its own — the longest, `01_validate_sources.py` at 112 lines, is entirely CLI plumbing (argument parsing, console formatting) around calls into `src/fonds_reader.py`, `src/rules_applier.py`, and `src/tools_invoker.py`. This thin-orchestrator pattern [Wilson et al., 2014] ensures that all testable logic is in `src/` at >= 90% coverage, while the scripts themselves remain readable without a dedicated test suite of their own.

## 7.4 Resilience Design

The integration layer enforces resilience at three levels, corresponding to the three failure modes a monorepo integration pipeline encounters:

1. **Resource absence**: A fond, rule set, or tool directory may not yet exist if the resource was created by a parallel agent that has not yet completed. All three reader modules return `None` or empty collections in this case, and `run_integration_demo()` reports the missing resource in the `summary` dict without raising.

2. **Schema malformation**: A manifest may be present but contain invalid YAML or missing required fields. Readers catch `yaml.YAMLError` and return a degraded result (`status="partial"` in rule validation; `None` in fond reading) rather than propagating the parse error.

3. **Script absence**: A tool may declare entrypoints in its manifest that have not yet been created. `validate_tool_scripts_exist()` detects this at discovery time and returns `status="partial"` with a list of missing script paths, so the pipeline can report the defect without attempting to invoke a non-existent script.

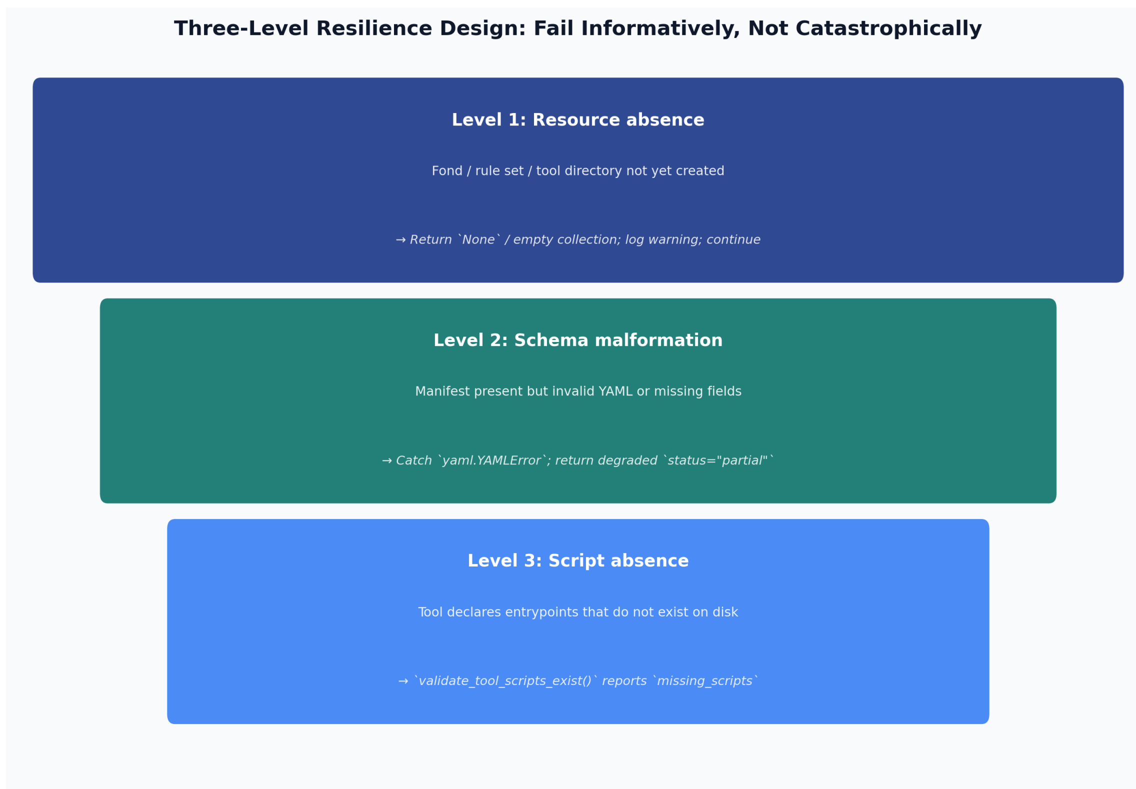


**Figure 8:** The integration layer's three-level resilience design: resource absence, schema malformation, and script absence, each with its own graceful-degradation response.

This three-level resilience design reflects the principle that shared-resource pipelines should fail *informatively* rather than *catastrophically* — surfacing the cause of incompleteness in structured output that downstream consumers can act on [Taschuk and Wilson, 2017]. fig. 8 presents the three levels as a single funnel: each level's failure mode and graceful-degradation response, read top to bottom in the order a pipeline run actually encounters them (resource discovery happens before schema parsing, which happens before script-existence checks).

## 7.5 Performance and Overhead

The resilience design above trades a small, constant amount of I/O overhead for the ability to degrade gracefully. Each reader performs at most two filesystem existence checks (`pathlib.Path.exists()`) before attempting a read, and every YAML parse uses `yaml.safe_load()` rather than a schema-validating parser — the module deliberately performs *structural* validation (parseable, expected top-level keys present) and leaves deep *semantic* validation to the dedicated `strong_rule_evaluator` module (sec. 5) rather than paying that cost on every discovery call. In practice, `scripts/02_run_integration.py` completes in well under a second on the exemplar's small fond/rule/tool counts; the design does not attempt to optimise for repositories with thousands of fonds, since the target use case is a curated, human-reviewed set of shared resources rather than a large-scale data catalogue.

## 7.6 Test Coverage

The eight `src/` modules — `fonds_reader`, `rules_applier`, `tools_invoker`, `integration`, `figures`, `strong_rule_evaluator`, `manuscript_variables`, and `type_defs` — are covered by tests across nine test files in `tests/`, including property-based tests (`test_property_based.py`) and coverage-extras tests targeting previously-uncovered branches (`test_coverage_extras.py`). Tests use real file paths, real YAML files, and real BibTeX content rather than mocks, ensuring that coverage numbers reflect genuine code paths through the resource-discovery logic. The current coverage report shows combined line coverage comfortably above the project's 90% floor; `strong_rule_evaluator.py`, the newest and most branch-heavy module, has the most room for additional edge-case tests, while the remaining seven modules are at or near 100%. These exact test/coverage counts drift as the suite grows — treat the figures above as a snapshot, not a frozen claim, and re-run `uv run pytest … --cov-report=term` for the current numbers. The `tests/test_integration.py` suite includes an end-to-end test that calls `run_integration_demo()` and asserts that the `summary` dict contains the expected keys with non-negative integer values — a contract test that verifies the token injection pipeline's data source. `tests/test_manuscript_variables.py` adds a negative control: it monkeypatches `run_integration_demo()`'s return value and asserts the derived tokens actually change, proving the token-generation function is live-wired to its source rather than emitting a hard-coded constant.

# 8 Conclusion

## 8.1 Summary

This paper has presented `template_pools_rules_tools`, a meta-project exemplar demonstrating how research software projects embedded in a monorepo can integrate three categories of shared resources — data pools (fonds), governance rules, and executable tools — without tight coupling to any specific resource instance. The key contributions are:

1. **A four-module architecture** (`fonds_reader`, `rules_applier`, `tools_invoker`, `integration`) that provides a canonical pattern for resource-aware project code in the template repository. Each module is independently testable and independently deployable.
2. **A typed manifest convention** (`fonds.yaml`, `rules.yaml`, `tools.yaml`) that makes resource capabilities explicit and checkable at pipeline initialisation time, shifting failure detection from runtime to startup — a significant improvement for reproducibility [Wilson et al., 2014].
3. **A token injection pipeline** that links manuscript prose to integration runtime statistics through `{{UPPERCASE_KEY}}` tokens, ensuring that quantitative claims in the manuscript are always generated by the pipeline rather than authored manually. In the current run this covered 3 fonds, 2 rule sets, 3 tools, and 8 bibliography entries.
4. **A three-level resilience design** — resource absence, schema malformation, and script absence — that allows the pipeline to degrade gracefully and report failures informatively rather than crashing, consistent with best practices for robust research software [Taschuk and Wilson, 2017].

## 8.2 Design Decisions Revisited

Several design decisions deserve emphasis as lessons for future exemplar authors:

**Repo-root-relative discovery** is non-negotiable in a forkable monorepo. Any hard-coded absolute path or working-directory assumption will fail when the repository is cloned to a different location. The `pathlib.Path(__file__).resolve().parents[N]` idiom used throughout `src/` ensures that discovery works from any working directory.

**Graceful fallbacks over exceptions** is the correct trade-off for a resource-consumption layer. The alternative — raising `FileNotFoundError` when a fond is missing — would make the integration pipeline fragile to the ordering of parallel agent writes. Returning `None` and logging a warning allows the pipeline to produce a complete, if partial, result dict that downstream consumers can reason about.

**Real-file tests over mocks** is required for a template exemplar. Mocks can pass tests while the real discovery logic is silently broken. Tests that use real file paths exercise the full path-resolution chain and catch regressions that mocks would miss.

**Thin orchestration scripts** keep the scripts directory readable and focus all testable logic in `src/`. A script that does more than parse arguments, call a `src/` function, and write output is accumulating business logic that belongs elsewhere.

## 8.3 Limitations

This exemplar makes several deliberate simplifications that a reader adopting the pattern should be aware of:

- **Structural, not semantic, validation by default.** `validate_against_rules()` and `discover_tools()` confirm that manifests and rule files are parseable YAML with the expected top-level shape; they do not check that a fond's *data* conforms to its declared schema, or that a tool's entrypoint script actually behaves according to its documented contract. Semantic constraint checking exists only for strong rules, via the separate `strong_rule_evaluator` module (sec. 5), and only for the constraint kinds that module implements: `coverage_threshold`, `module_structure`, `section_schema`, `reference_schema` (all four wired to a real strong-rule YAML file and exercised against this project's own live tree), plus `manifest_freshness` (evaluator logic implemented and unit-tested, deliberately not yet wired to a real rule file — see Future Directions).
- **No concurrency handling.** All readers assume a single-process, single-read invocation. If a parallel agent is actively writing a fond's `data/` file while another process reads it, the reader may observe a partially-written file and report a spurious parse error rather than a clean "not yet available" signal. The architecture tolerates *absence* gracefully; it does not guarantee *atomicity* against concurrent writers.
- **Small-scale assumption.** As discussed in sec. 7's Performance and Overhead subsection, the design is tuned for a curated, human-reviewed set of resources (single-digit to low-double-digit counts per category), not a resource catalogue at data-lake scale.
- **English-language, code-review-oriented soft rules.** Soft rules are Markdown prose intended for human reviewers and AI coding agents; they are not evaluated by any automated tool in this exemplar, unlike strong rules. A project that wants soft-rule compliance checked automatically would need to promote the relevant guideline to a strong rule with a corresponding evaluator function.
- **`strong_rule_evaluator.py` had the thinnest test coverage of any `src/` module, historically.** As measured in an earlier coverage run this session, it required more negative-control tests (malformed rule files, real violations, type-mismatched context values) than the other seven modules, concentrated in its `section_schema`, `reference_schema`, and `module_structure` evaluators and its context loader. That gap has since been closed with real negative-control fixtures — run `uv run pytest … --cov-report=term-missing` for the current per-module breakdown, since these numbers, like the ones above, drift as the suite grows.

**The no-concurrency-handling limitation above is not merely theoretical**: this exemplar is itself designed to be populated by parallel agents authoring `fonds/`, `rules/`, and `tools/` concurrently, which is precisely the scenario where a torn read is most likely. Adopters running this pattern in a genuinely concurrent write environment should add file locking or an atomic-rename write pattern at the resource-authoring layer — this exemplar deliberately does not, to keep the reader-side code minimal.

## 8.4 Future Directions

The three-layer architecture described here is deliberately extensible. The most natural extension is a fourth resource category: **models** (`models/<scope>/<name>/`) for pre-trained machine learning models and their inference scripts. The pattern would follow exactly the same manifest-and-reader design, with `models.yaml` declaring type, version, and entrypoints, and a `models_loader` module providing `discover_models()` and `validate_model_files_exist()`. Adding this layer to `template_pools_rules_tools` would require only a new reader module and a new section in the integration orchestrator — the existing architecture places no constraints on the number of resource categories.

A second direction, **cross-fond validation**, is now implemented: `src/integration.py::check_bibliography_overlap()` compares this manuscript's own cite keys against the `template_bibliography` fond's curated set and reports `overlap`, `project_only`, and `fond_only` counts. It deliberately does *not* assert full containment in either direction — this manuscript legitimately cites software-engineering sources the fond does not curate, and the fond legitimately curates machine-learning sources this manuscript does not cite. Run `uv run python -c "from src.integration import check_bibliography_overlap; import json; print(json.dumps(check_bibliography_overlap(), indent=2))"` from the project root to see the current overlap.

A third direction, suggested directly by the Limitations above, was extending `strong_rule_evaluator` with additional rule kinds beyond the original four. The `manifest_freshness` kind — flagging a fond whose `version` field has not been bumped despite its `data/` files changing more recently than its manifest — is now implemented: `src/strong_rule_evaluator.py::_evaluate_manifest_freshness()` is a fifth dispatch entry, unit-tested against synthetic rule dicts (`tests/test_strong_rule_evaluator.py`). It is deliberately **not** yet wired to a real strong-rule YAML file: doing so would mean adding a file under the shared `rules/templates/template_project_rules/strong/` or `rules/templates/template_manuscript_rules/strong/` directory, which this project's own read-only contract against `fonds/`, `rules/`, and `tools/` reserves for an explicit decision rather than a routine content edit. The evaluator's logic is proven correct against synthetic input; proving it against a real fond's real manifest and data-file timestamps is the natural next step, pending that decision.

The exemplar is ready to fork. A project team wishing to adopt this architecture need only copy the relevant `src/` modules (the three readers plus `integration.py` are the minimum; `strong_rule_evaluator.py` and `figures.py` are optional extensions), update the resource directory paths in each reader, and replace the exemplar fond/rules/tool names with their own.

## References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020. doi: 10.48550/arXiv.2005.14165.

Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002. ISBN 0-321-12521-5. Repository and Registry patterns referenced in the fonds and rules layer design.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-201-63361-2. Foundational reference for the Facade and Template Method patterns that inform the module architecture described in this paper.

Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks — a publishing format for reproducible computational science. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90, 2016. doi: 10.3233/978-1-61499-649-1-87.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, 2003. ISBN 0-13-142901-9. Rule of Modularity and Rule of Composition cited in the integration section.

Victoria Stodden, Matan Guo, and Zhaokun Ma. Toward reproducible computational research: An empirical analysis of data and code policy adoption by journals. *PLoS ONE*, 8(6):e67111, 2013. doi: 10.1371/journal.pone.0067111.

Morgan Taschuk and Greg Wilson. Ten simple rules for making research software more robust. *PLoS Computational Biology*, 13(4): e1005412, 2017. doi: 10.1371/journal.pcbi.1005412.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30:5998–6008, 2017. doi: 10.48550/arXiv.1706.03762.

Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best practices for scientific computing. *PLoS Biology*, 12(1):e1001745, 2014. doi: 10.1371/journal.pbio.1001745.