A `template/` approach to Reproducible Generative Research

Architecture and Ergonomics from Configuration through Publication

Daniel Ari Friedman

Active Inference Institute

daniel@activeinference.institute

ORCID: 0000-0001-6232-9096

DOI: 10.5281/zenodo.19139090

2026-03-20

2026-03-20

Contents

1 Abstract

The reproducibility crisis in computational research is fundamentally structural: research artifacts are scattered across disconnected tools—LaTeX editors, Jupyter notebooks, ad-hoc shell scripts—with no enforced mechanism to keep code, data, and manuscript synchronized. Studies have shown that most published findings are false positives, replication rates in psychology hover around 36%, and only 24% of 1.4 million Jupyter notebooks can be successfully re-executed. Existing tools address fragments of this problem: workflow managers (Snakemake, Nextflow, CWL) orchestrate computation; literate programming systems (Quarto, Jupyter Book, R Markdown, Overleaf, OpenAI Prism) render documents; data versioning tools (DVC) track artifacts—but none enforces cross-cutting quality standards as architectural invariants. `template/` applies the principle of Infrastructure as Code to the research lifecycle, making the manuscript, test suite, and provenance chain version-controlled, deterministically buildable, and independently verifiable. It is built on a Two-Layer Architecture that separates 12 reusable infrastructure subpackages (~150 Python modules, validated by ~3,083 tests) from self-contained project workspaces, connected by an eight-stage build pipeline progressing from environment sanitization through test execution (with a Zero-Mock testing policy enforcing 90% project-level and 60% infrastructure-level coverage via real filesystem operations and subprocess invocations), analysis script invocation, Pandoc/XeLaTeX rendering, SHA-256 cryptographic hashing with steganographic watermarking, structural PDF validation, and LLM-assisted review. A Documentation Duality standard equips every directory with both human-readable `README.md` and machine-readable `AGENTS.md` files, while each infrastructure module additionally carries a `SKILL.md`—a structured skill descriptor aligned with the Model Context Protocol—enabling AI agents to locate and invoke module capabilities without hallucinating API signatures.

Scalability is demonstrated across three heterogeneous projects—a gradient descent study (`code_project`, 39 tests), a meta-analysis pipeline (`act_inf_metaanalysis`, 505 tests), and this self-referential architectural analysis (`template`, 65 tests)—achieving 100% pipeline success with zero mock violations. The fact that these words, these metrics, and the figures accompanying them were generated by the very pipeline they describe is itself a demonstration of the system's self-productive capacity: the manuscript is not merely *about* `template/` but *of* it, rendered through the same eight-stage pipeline, validated by the same test suite, and watermarked by the same steganographic layer documented herein. A comparative feature analysis against nine peer tools across fourteen dimensions confirms that `template/` uniquely integrates all eleven distinctive capabilities—testing enforcement, coverage thresholds, cryptographic provenance, steganographic watermarking, multi-project management, AI-agent documentation, agentic skill protocol, interactive TUI, Zero-Mock policy, manuscript rendering, and pipeline orchestration—within a single enforced pipeline. `template/` is open source under the Apache 2.0 License at `github.com/docxology/template`, and is presented as a work in progress.

2 Introduction

2.1 Research Tools as Epistemic Infrastructure

Scientific research operates through a layered ecology of tools, documents, and practices—each shaping what can be known, communicated, and verified. When these layers are fragmented, the artifacts of research (manuscripts, data, code, figures) drift out of alignment with one another, creating gaps that are structural rather than incidental. The "reproducibility crisis" is one symptom of this deeper misalignment: a 2016 *Nature* survey of 1,576 researchers found that 70% had tried and failed to reproduce another scientist's experiments, and more than half had failed to reproduce their own [Baker, 2016]. Freedman et al. estimate that the biomedical industry alone loses $28 billion annually to irreproducible preclinical research [Freedman et al., 2015]. But reproducibility is only the most visible face of a broader problem. Research software engineering, epistemic integrity, and the coordination between human researchers and AI collaborators all depend on the same underlying question: whether the tools that produce research artifacts can themselves be made transparent, testable, and self-documenting. Nosek et al. [Nosek et al., 2018] have argued that the preregistration revolution—requiring researchers to commit to analytical plans before data collection—is a necessary structural reform; we extend this logic to the entire research build pipeline.

One root cause is fragmentation—of attention (in the mind) as well as in the cyberphysical niche (documents, versions, and reproducibility artefacts). A typical research project scatters its artifacts across disconnected tools: LaTeX editors [Lamport, 1994] for prose, Jupyter notebooks for analysis, ad-hoc shell scripts for figure generation, and manual copy-paste for integrating results into manuscripts. Each boundary between tools is a potential locus of desynchronization. The version of the figure embedded in the PDF may not match the version of the code that ostensibly generated it. The test suite, if it exists at all, likely tests the code in isolation from the rendering pipeline. Pimentel et al. [Pimentel et al., 2019] analyzed 1.4 million Jupyter notebooks from GitHub and found that only 24% could be successfully re-executed, with 36% producing different results—quantifying the reproducibility cost of notebook-based workflows. Peng [Peng, 2011] argues that reproducibility in computational science requires, at minimum, that the data and code underlying a published result be available for independent verification—yet the tools for enforcing this standard remain ad hoc. Indeed, even the terminology is fractured: Barba [Barba, 2018] documents how "reproducibility," "replicability," and "repeatability" carry conflicting definitions across disciplines, undermining cross-field standards.

2.2 Related Work

Gentleman and Temple Lang [Gentleman and Temple Lang, 2007] introduced the concept of a *research compendium*—a single unit of scholarly communication bundling code, data, and narrative. This vision has driven two decades of tooling, which can be grouped into four categories: workflow managers, literate programming systems, containerization approaches, and best-practice frameworks.

2.2.1 Workflow Managers

Snakemake [Köster and Rahmann, 2012] uses a rule-based, Python-derived DSL to specify computational workflows as directed acyclic graphs of file-producing steps. It supports containerized execution via Conda and Singularity environments. Snakemake 9.x (2024–2025) introduced a plugin architecture for extended execution backends and storage providers, yet its scope remains computational pipeline orchestration—it does not integrate manuscript rendering, testing enforcement, or provenance watermarking.

Nextflow [Di Tommaso et al., 2017] employs a dataflow programming paradigm with native support for container-based execution across heterogeneous computing environments (local, SLURM, AWS). Like Snakemake, Nextflow excels at bioinformatics pipeline parallelism but does not address manuscript production, document integrity, or the testing–publication coupling that characterizes research reproducibility.

CWL (Common Workflow Language) [Amstutz et al., 2016] provides a portable, YAML-based standard for describing computational workflows and their dependencies. Its strength lies in interoperability across

execution engines (cwltool, Toil, Arvados), but it requires external tooling for manuscript generation and offers no built-in testing or provenance framework.

2.2.2 Literate Programming and Publication Tools

Knuth's literate programming [Knuth, 1984] established the principle that programs should be authored as documents intended for human comprehension. Schulte et al. [Schulte et al., 2012] extended this to multi-language computing environments (Org-mode), demonstrating that literate programming could span languages and output formats.

Quarto [Allaire et al., 2024] extends the R Markdown tradition to support Python, Julia, and Observable, rendering to PDF, HTML, and Word. Quarto integrates code execution with document rendering, achieving a modern form of literate programming, but it does not enforce testing thresholds, manage multi-project repositories, or provide cryptographic provenance.

Jupyter Book [Kluyver et al., 2016] builds on Jupyter notebooks to produce publication-quality documents via Sphinx. While powerful for interactive exploration, Jupyter's notebook format introduces execution-order fragility [Pimentel et al., 2019] and does not naturally support the separation of logic from orchestration that characterizes maintainable research software.

R Markdown [Xie et al., 2018] pioneered knitr-based dynamic documents that weave code and prose. Its ecosystem is rich but R-centric, and it lacks the multi-project management, infrastructure testing, and provenance embedding that characterize `template/`.

Typst [Mädje and Haug, 2023] is an emerging markup-based typesetting system with incremental compilation and a programmable scripting layer. While Typst offers faster compilation than LaTeX and a more modern authoring experience, it does not integrate testing, provenance, or multi-project pipeline management.

2.2.3 Containerization and Environment Capture

Docker [Boettiger, 2015] addresses reproducibility at the environment level—packaging operating system, libraries, and code into portable containers. While Docker solves the "works on my machine" problem, containerization is complementary to, not a replacement for, the architectural concerns addressed here: Docker does not enforce testing, embed provenance, or manage multi-project manuscript workflows.

2.2.4 Best-Practice Frameworks and Data Standards

Wilson et al. [Wilson et al., 2017] define "good enough" practices for scientific computing, emphasizing version control, testing, and documentation. Sandve et al. [Sandve et al., 2013] propose ten rules for reproducible computational research. Piccolo and Frampton [Piccolo and Frampton, 2016] systematically survey tools for computational reproducibility, finding that environment isolation, workflow automation, and documentation generation address complementary but non-overlapping reproducibility concerns—yet no single tool unifies all three. Stodden et al. [Stodden et al., 2016] advocate for enhanced computational method transparency. The FAIR principles [Wilkinson et al., 2016]—Findable, Accessible, Interoperable, Reusable—establish a standard for data stewardship that has been widely adopted by funding agencies and journals. Lamprecht et al. [Lamprecht et al., 2020] formalize "Towards FAIR Principles for Research Software," providing the conceptual scaffolding that Barker et al. [Barker et al., 2022] would soon operationalize as the FAIR4RS initiative—recognizing that software has execution, composability, and dependency-management requirements that data-centric FAIR does not address. Cohen et al. [Cohen et al., 2021] characterize the four pillars of research software engineering (software sustainability, software quality, community building, and policy advocacy), situating formal testing and provenance practices within a broader RSE governance framework. Garijo et al. [Garijo et al., 2024] operationalize FAIR4RS through the FAIRsoft evaluator, an automated assessment framework that scores research software against 17+ quality indicators including executability, metadata richness, and documentation completeness. Goble et al. [Goble et al., 2020] extend FAIR to computational workflows specifically, arguing that workflow provenance requires first-class treatment in scientific computing infrastructure. Nüst et al. [Nüst et al., 2017] introduce the *executable research*

compendium (ERC), extending Gentleman and Temple Lang's compendium concept with containerized, interactive reproduction environments. The W3C PROV data model [Moreau and Missier, 2013] provides a formal vocabulary for expressing provenance records, while in-toto [Torres-Arias et al., 2019] provides a framework for end-to-end software supply chain integrity verification, and SLSA [Open Source Security Foundation, 2023] (Supply-chain Levels for Software Artifacts) extends this to graduated, attestation-based supply-chain security levels for build pipelines. These frameworks articulate *what* reproducible research requires but do not provide an integrated *how*—they lack the tooling, enforcement mechanisms, and architectural patterns that translate standards into practice.

2.2.5 The Gap

Despite advances in FAIR4RS principles [Barker et al., 2022, Lamprecht et al., 2020], automated FAIR software assessment [Garijo et al., 2024], FAIR computational workflow standards [Goble et al., 2020], supply-chain attestation frameworks [Torres-Arias et al., 2019, Open Source Security Foundation, 2023], and the preregistration revolution [Nosek et al., 2018], no existing system integrates six cross-cutting concerns into a single enforced pipeline: (1) end-to-end pipeline orchestration with testing enforcement, (2) multi-format manuscript rendering, (3) cryptographic provenance embedding, (4) multi-project repository management, (5) FAIR-aligned software stewardship, and (6) AI-agent collaboration via structured documentation. Each existing framework addresses a subset; none provides the unified enforcement mechanism. The detailed tool-by-tool comparison is developed in the Comparison to Existing Tools section; the summary table below captures the gap landscape.

2.2.6 Summary: Gaps Left by Existing Tools

The six concerns identified above map onto existing tool categories as follows:

Gap	Partially Addressed By	Not Addressed By
Pipeline orchestration	Snakemake 9.x, Nextflow 25.x, CWL 1.2	Quarto, Jupyter Book, R Markdown, Typst, Overleaf, Prism
Manuscript rendering	Quarto 1.x, Jupyter Book 2.x, R Markdown, Typst, Overleaf (2025), Prism	Snakemake, Nextflow, CWL, DVC
Testing enforcement	—	All existing tools
Cryptographic provenance	SLSA (build-level attestation only)	All research-focused tools
Multi-project management	—	All existing tools
AI-agent documentation	Overleaf (partial co-author AI), Prism (partial context reasoning)	All pipeline/workflow tools
Agentic skill protocol (MCP-aligned)	—	All existing tools

No existing system addresses all six concerns within a single enforced pipeline.

2.3 `template/`: An Integrated Solution

`template/` was conceived as a structural antidote to this fragmentation. Rather than adding reproducibility as an afterthought—a Docker container wrapping an already-disjointed workflow [Boettiger, 2015]—the template enforces integrity at the architectural level. It realizes Gentleman and Temple Lang's research compendium vision [Gentleman and Temple Lang, 2007] at repository scale, bundling code, data, tests, manuscripts, and provenance into a single, pipeline-enforced system with version-controlled infrastructure [Ram, 2013]. It stands on four primary pillars:

1. **Ergonomic Modularity**: A Two-Layer Architecture cleanly separates globally shared infrastructure (logging, rendering, validation, steganography) from project-specific logic (manuscripts, scripts, data). 12 infrastructure subpackages comprising ~150 Python modules provide reusable services; projects consume them without modification.

2. **Execution Integrity**: A Zero-Mock testing policy where pipeline advancement is contingent on test passage. Infrastructure tests must achieve 60% coverage; project tests must achieve 90%. All tests use real filesystem operations, real subprocess calls, and real network connections—no mock objects, no fake services, no synthetic test doubles. ~3,083 infrastructure tests and 708+ project tests enforce this standard.

3. **Automated Provenance**: Steganographic watermarking and cryptographic hashing are integrated directly into the rendering pipeline. Every generated PDF carries a SHA-256 fingerprint, an alpha-channel text overlay encoding the build timestamp and commit hash, and optionally a QR code linking to the repository. Provenance is not asserted by policy; it is enforced by architecture.

4. **AI-Agent Collaboration and Skill-Based Agentic Operations**: A three-tier documentation architecture enables AI agents to operate at every level of the system. At the system level, `CLAUDE.md` asserts global architectural constraints. At the structural level, `AGENTS.md` files at every directory expose local API surfaces, file inventories, and integration contracts. At the module level, `SKILL.md` files—written to a discoverable YAML+Markdown schema aligned with the Model Context Protocol [Anthropic, 2024]—define each infrastructure module as a reusable, self-describing tool. An agent invoking `infrastructure.rendering` does not need to read source code: it reads the `rendering/SKILL.md`, which declares the module's name, description, key imports, and example invocations in a machine-parseable YAML frontmatter block. This architecture is the practical realization of the skill-library paradigm established in the agent literature: Yao et al.'s ReAct framework [Yao et al., 2023] demonstrated that interleaving reasoning traces with tool invocations dramatically improves LLM reliability; Schick et al.'s Toolformer [Schick et al., 2023] showed that self-supervised tool use can be bootstrapped from natural language; Wang et al.'s Voyager [Wang et al., 2023] proved that growing skill libraries enable open-ended autonomous exploration in complex environments. `template/` instantiates this vision in the domain of scientific research infrastructure: each `SKILL.md` is a Voyager-style skill, each pipeline stage is a ReAct action, and the full infrastructure layer constitutes a composable, protocol-aligned skill library for scientific computation.

2.4 Scope and Contributions

This paper is itself a product of the template it describes. The metrics populating its tables were computed by the introspection module documented in the Methods; the figures were rendered by the visualization code validated by the test suite described in Quality Assurance; the PDF carrying these words was assembled by the same eight-stage pipeline whose architecture is the subject of the Results. This self-productive loop—where the system that is described is also the system that produces the description—is not incidental but structural, a concrete demonstration that `template/` can sustain the full lifecycle from source code to published artifact within a single, version-controlled, pipeline-enforced repository. Our contributions are:

- A formal description of the Two-Layer Architecture and Standalone Project Paradigm that enables N independent research projects to share infrastructure without coupling.
- A detailed specification of the eight-stage build pipeline (Stages 00–07), from environment sanitization through executive report generation.
- A comparative analysis positioning `template/` against ten peer tools—Snakemake, Nextflow, CWL, Quarto, Jupyter Book, R Markdown, DVC, Typst, Overleaf, OpenAI Prism—across fourteen feature dimensions, demonstrating that `template/` uniquely integrates all eleven distinctive capabilities.
- An empirical evaluation of the system across three heterogeneous exemplar projects (`code_project`, `act_inf_metaanalysis`, `template`), demonstrating scalability, coverage metrics, and pipeline timing.
- A security analysis of the steganographic provenance layer, including a formal threat model and tamper-detection capabilities aligned with the W3C PROV data model [Moreau and Missier, 2013] and SLSA [Open Source Security Foundation, 2023].

- An open-source reference implementation available at `github.com/docxology/template`.

2.5 Paper Organization

The Methods describe the Two-Layer Architecture, Thin Orchestrator pattern, pipeline stages, and AI collaboration model. Results present quantitative metrics from multi-project execution, coverage analysis, and steganographic benchmarks. The Discussion addresses the Zero-Mock tradeoff, scalability implications, a detailed tool comparison, and future directions. The Infrastructure Module Reference provides detailed documentation for all twelve subpackages. Security and Provenance describes the steganographic and cryptographic integrity layer. The Appendices provide pipeline, configuration, and comparative references.

3 Methods

The `template/` architecture is deliberately bifurcated into a globally shared `infrastructure/` layer and project-specific `projects/` silos. This section describes the four core design patterns, the eight-stage pipeline that operationalizes them, and the AI collaboration model that distinguishes this system from conventional research templates.

3.1 The Two-Layer Architecture

The repository is organized into two strictly separated layers:

Infrastructure Layer (`infrastructure/`): 12 Python subpackages comprising ~150 modules and providing reusable services. Each subpackage is independently importable, has its own `__init__.py`, `AGENTS.md`, and `README.md`, and exports a well-defined public API. The infrastructure layer knows nothing about any specific project—it provides generic capabilities (logging, rendering, validation, steganography) that any project may consume.

Project Layer (`projects/`): Self-contained research workspaces. Each project directory contains:

Directory	Purpose
`manuscript/`	Markdown chapters and `config.yaml`
`scripts/`	Thin orchestrator scripts (Stage 02)
`src/`	Project-specific Python modules
`tests/`	Project-specific test suite
`data/`	Input datasets and generated data
`output/`	Pipeline artifacts: PDF, figures, reports, logs
`docs/`	Project-specific architecture documentation

The two layers communicate exclusively through Python imports and filesystem paths. No project modifies infrastructure code; no infrastructure module references a specific project by name (except via runtime project discovery).

3.2 The Standalone Project Paradigm

Projects are designed to be completely self-contained. Adding a new project requires no changes to the infrastructure layer, no modifications to `pyproject.toml`, and no updates to the pipeline orchestrator. A project is automatically discovered if and only if it satisfies two conditions:

1. It exists as a subdirectory of `projects/`.
2. It contains the file `manuscript/config.yaml`.

This paradigm enables horizontal scaling: N researchers can maintain N independent projects within a single repository, sharing infrastructure without coupling. Each project declares its own testing tolerances, manuscript metadata, LLM review preferences, and rendering configuration in its `config.yaml`. The system currently hosts three exemplar projects spanning numerical optimization, meta-analysis pipelines, and meta-architectural analysis.

3.3 The Thin Orchestrator Pattern

All scripts in `scripts/` (both infrastructure-level and project-level) follow the Thin Orchestrator pattern [Gamma et al., 1995]:

- **No domain logic**: Scripts contain zero algorithmic implementation. They import functions from `src/` modules and wire them to infrastructure services.
- **Configuration-driven**: Behavior is parameterized by `config.yaml`, not by hardcoded values.

- **Stateless**: Scripts read inputs, call functions, write outputs. They maintain no persistent state between invocations.
- **Logged**: Every significant action is logged via `infrastructure.core.logging_utils.get_logger`.

This pattern ensures that all testable logic lives in `src/` where it is subject to the Zero-Mock testing policy, while scripts remain thin enough to be audited by visual inspection. The separation draws on the Mediator pattern from Gamma et al. [Gamma et al., 1995], where scripts mediate between infrastructure services and project-specific code without implementing any logic of their own.

To make this concrete, the following contrasts the anti-pattern with the correct pattern:

```
# ANTI-PATTERN: domain logic embedded in script
def calculate_average(data):          # ← never put computation here
    return sum(data) / len(data)

result = calculate_average([1, 2, 3])

# CORRECT PATTERN: script imports from src/ and only wires
from projects.my_project.src.statistics import calculate_average

result = calculate_average([1, 2, 3])  # ← scripts wire, never compute
```

The critical property is that `calculate_average` in the correct pattern lives in a testable `src/` module, is covered by the Zero-Mock test suite, and can be independently imported, tested, and reused—whereas the anti-pattern buries logic in a script that is invisible to coverage tools.

3.4 The Eight-Stage Pipeline

The build pipeline is orchestrated by `scripts/execute_pipeline.py`, which invokes numbered stage scripts sequentially. Each stage is a standalone Python script that exits cleanly or raises an exception to halt the pipeline.

3.4.1 Stage 00: Environment Setup (`00_setup_environment.py`)

Validates the Python environment, checks dependency availability, creates output directories, and initializes logging. Ensures `PYTHONPATH` includes both the repository root and the active project's `src/` directory.

3.4.2 Stage 01: Test Execution (`01_run_tests.py`)

Executes pytest with coverage measurement against both infrastructure tests (`tests/`) and project tests (`projects/<name>/tests/`). Enforces configurable failure tolerances:

- `max_infra_test_failures`: Maximum permitted infrastructure test failures (typically 3).
- `max_project_test_failures`: Maximum permitted project test failures (typically 0).
- Coverage thresholds: 60% infrastructure, 90% project.

The stage generates coverage JSON files for downstream reporting and saves test results in both JSON and Markdown formats. The infrastructure test suite alone contains nearly 3,000 tests across 160+ test files.

3.4.3 Stage 02: Analysis Execution (`02_run_analysis.py`)

Discovers and executes all Python scripts in `projects/<name>/scripts/` in alphabetical order. Each script is expected to generate figures in `../figures/` and data in `output/data/`. Scripts follow the Thin Orchestrator pattern, importing logic from `src/` modules. For example, the `cognitive_case_diagrams` project generates 25+ programmatic figures via 17 DisCoPy renderers during this stage.

3.4.4 Stage 03: PDF Rendering (`03_render_pdf.py`)

Compiles Markdown manuscript chapters into a unified PDF via a three-phase rendering process:

1. **Pandoc Markdown→LaTeX**: Converts each `manuscript/*.md` file into LaTeX, injecting metadata from `config.yaml` (title, authors, affiliations, DOI).
2. **XeLaTeX Compilation**: Runs `xelatex` with `biber` for bibliography processing. Handles the `aux`→`bbl`→`aux` cycle automatically, with cleanup of stale auxiliary files to prevent corruption.
3. **Post-processing**: Applies font embedding verification and PDF/A compliance checks.

3.4.5 Stage 04: Output Validation (`04_validate_output.py`)

Validates the structural integrity of all generated artifacts:

- PDF cross-reference table and trailer verification.
- Figure file existence and minimum size checking.
- Manifest generation with SHA-256 hashes for all output files.
- Markdown structural validation (heading hierarchy, link integrity).

3.4.6 Stage 05: Output Organization (`05_copy_outputs.py`)

Copies finalized artifacts to standardized output locations and generates the pipeline completion manifest.

3.4.7 Stage 06: LLM Review (`06_llm_review.py`)

Invokes a local LLM (via Ollama) to generate:

- **Executive summary**: A 1-page high-level overview of the manuscript.
- **Quality review**: Detailed feedback on structure, citations, and argumentation.

- **Translations** (optional): Machine translations of the abstract into configured target languages.

This stage is skippable via configuration and gracefully handles Ollama unavailability.

3.4.8 Stage 07: Executive Report (`07_generate_executive_report.py`)

Aggregates all pipeline metrics—test results, coverage percentages, rendering duration, validation status, LLM review scores—into a comprehensive executive report in both JSON and Markdown formats.

3.5 The Interactive Orchestrators

3.5.1 `run.sh`: The Standard Orchestrator

The primary user interface is `run.sh`, a Bash TUI (text user interface) that presents an interactive menu for pipeline execution. Features include:

- **Project selection**: Execute a single project or all discovered projects.
- **Mode selection**: Fast (skip infra tests + LLM), Core (skip LLM only), Full (all stages).
- **Non-interactive mode**: `./run.sh --pipeline --project all --core-only` for CI/CD integration.
- **Real-time progress**: Stage timing and status indicators.

3.5.2 `secure_run.sh`: The Steganographic Superset

The `secure_run.sh` orchestrator is a strict superset of `run.sh`: it executes the standard eight-stage pipeline and then appends steganographic post-processing. For each rendered PDF, it applies metadata injection, cryptographic hashing, alpha-channel text overlay, and QR code injection, producing a provenance-embedded output alongside the original. It supports the same project selection and mode flags as `run.sh`.

3.6 Documentation Duality and AI Collaboration

Every directory at every level of the repository hierarchy contains two documentation files:

- **`README.md`**: Human-readable overview, quick-start instructions, and directory structure.
- **`AGENTS.md`**: Machine-readable technical specification optimized for AI coding assistants. Contains API tables, dependency graphs, implementation patterns, and architectural constraints.

This Documentation Duality standard serves two purposes. First, it ensures that both human researchers and AI agents can navigate the codebase efficiently—`AGENTS.md` files provide the structured context that language models need to make informed code modifications without hallucinating API signatures or violating architectural invariants. Second, it creates a self-documenting feedback loop: as AI agents modify the codebase, they update the corresponding `AGENTS.md` files, keeping documentation synchronized with implementation. Lau and Guo's survey of 90 AI coding assistant systems [Lau and Guo, 2025] identifies contextual code understanding as a primary bottleneck; the Documentation Duality standard addresses this by providing pre-structured context at every directory level.

The template additionally includes `CLAUDE.md` at the repository root, providing system-level instructions for AI coding assistants—architectural principles, testing requirements, and naming conventions that apply globally. This creates a three-tier documentation architecture: per-directory `AGENTS.md` for local context, root `README.md` and `CLAUDE.md` for global constraints, and `README.md` for human navigation.

3.7 Agentic Skill Architecture

The Documentation Duality standard addresses human and AI navigation at the directory level. A complementary layer operates at the *module* level: every infrastructure subpackage carries two additional machine-readable files that transform it from a passive code library into an active, protocol-aligned skill endpoint.

3.7.1 The Three-Tier Skill Protocol

`template/` implements a progression of agent-facing documentation, escalating in specificity from global constraints to module-level API contracts:

Tier	File	Scope	Purpose
1 — System	`README.md`	Repository root	Global architectural principles, Zero-Mock policy, naming conventions
2 — Structure	`AGENTS.md`	Every directory	Local file inventories, API surfaces, integration patterns, architectural constraints
3 — Skill	`SKILL.md`	Every infrastructure module	Machine-parseable skill descriptor: module name, description, key imports, usage pattern

Tier 1 and Tier 2 have direct analogues in the prompt-engineering literature: system prompts and retrieval-augmented context [Lau and Guo, 2025]. Tier 3 is novel. The `SKILL.md` files follow a YAML frontmatter + Markdown instruction format precisely aligned with the tool-descriptor schemas of the Model Context Protocol [Anthropic, 2024]. The following is the exact frontmatter from `infrastructure/rendering/SKILL.md`:

```yaml
---
name: rendering
description: >
  Multi-format output generation (PDF, HTML, slides).
  Use for: Pandoc/XeLaTeX rendering, RenderManager, slide deck generation.
```

```
  Key imports: RenderManager, RenderingConfig from infrastructure.rendering
---
```

An MCP client reading this block immediately knows the module name, its natural-language activation condition ("use for"), and which Python symbols to import. No source-code inspection is required. This is the practical implementation of Toolformer-style self-documented tools [Schick et al., 2023]—rather than a language model learning tool APIs from training data, the APIs are encoded directly in version-controlled, co-located skill files that evolve with the codebase.

3.7.2 Module Skill Coverage

All ten active infrastructure modules carry `SKILL.md` files. A companion `PAI.md` (Personal AI Infrastructure) file at the top of the `infrastructure/` directory documents the collection's role within a researcher's broader AI-assisted ecosystem—capturing import rules, testing obligations, and cross-module dependencies in the format used by Codomyrmex-style PAI frameworks.

This coverage is a structural invariant enforced by the Documentation Duality standard—every new module added to `infrastructure/` must carry `AGENTS.md`, `README.md`, and `SKILL.md` before it is accepted by the pipeline.

3.7.3 MCP Server Mapping

The mapping from `SKILL.md` descriptors to MCP server endpoints is intentional but not yet automated; it represents the principal next integration step. In the MCP architecture [Anthropic, 2024], a server exposes three primitive types: **Tools** (executable functions), **Resources** (data the model can read), and **Prompts** (structured query templates). Each `infrastructure` module maps naturally onto this taxonomy:

- `infrastructure.llm` → MCP **Tool** (`query`, `apply_template`) + MCP **Prompt** (research prompt templates)
- `infrastructure.rendering` → MCP **Tool** (`render_pdf`, `render_html`) + MCP **Resource** (rendered PDFs as URI-addressable resources)
- `infrastructure.validation` → MCP **Tool** (`validate_pdf_rendering`, `validate_markdown`)
- `infrastructure.publishing` → MCP **Tool** (`publish_to_zenodo`, `generate_citation_bibtex`) + MCP **Resource** (DOI registry)
- `infrastructure.steganography` → MCP **Tool** (`SteganographyProcessor.process`) + MCP **Resource** (hash manifests)

An agent orchestrating a full research pipeline could, in principle, compose these MCP tools to reproduce the entire eight-stage pipeline programmatically—discovering available tools via `SKILL.md` frontmatter, executing them via MCP protocol calls, and consuming their outputs as Resources. The `SKILL.md` files parallel Voyager's skill library [Wang et al., 2023]—Voyager's agent accumulates a growing library of executable Minecraft skills represented as JavaScript functions; `template/`'s agent accumulates a curated library of research pipeline skills represented as YAML-frontmattered `SKILL.md` files. In both cases, the skill representation is machine-readable, version-controlled, and self-describing. Wang et al.'s LLM agent survey [Wang et al., 2024a] identifies tool use, planning, and memory as the three fundamental capabilities of autonomous agents; Yao et al.'s ReAct framework [Yao et al., 2023] demonstrates that interleaving reasoning traces with tool actions dramatically improves agent reliability in interactive settings. The `template/` skill architecture provides the tool-use layer aligned with these frameworks, the eight-stage pipeline provides the planning scaffold, and the checkpoint system provides the memory layer.

3.8 FAIR Alignment and Research Infrastructure as Code

The template's design aligns with both the original FAIR principles [Wilkinson et al., 2016] and the FAIR for Research Software (FAIR4RS) principles [Barker et al., 2022] at the repository level. FAIR4RS recognizes that software has requirements distinct from data—executability, composability, and dependency management—and the template addresses each.

3.8.1 Principle-by-Principle Alignment

Findability. Outputs are *Findable* through standardized directory structures, manifest files, and machine-readable metadata embedded in PDFs. Every project's `config.yaml` provides structured metadata (title, authors, DOIs, keywords) in a format parseable by both Pandoc and external indexing services. The `metrics.json` output provides a machine-readable inventory of all generated artifacts, their locations, and their provenance hashes.

Accessibility. Outputs are *Accessible* via open-source distribution on GitHub, with metadata embedded in the artifact itself rather than in a separate registry. The steganographic layer embeds provenance information directly in the PDF—including SHA-256 content hashes, build timestamps, and QR-encoded metadata—ensuring accessibility even when the PDF circulates outside the repository.

Interoperability. *Interoperability* is achieved through standard formats (PDF, JSON, BibTeX, YAML) and well-defined module APIs that enable cross-project composition. The Pandoc rendering pipeline accepts any Markdown-with-LaTeX input conforming to the template's section numbering conventions, allowing seamless migration of manuscripts from other Pandoc-based workflows.

Reusability. *Reusability* is ensured by the Standalone Project Paradigm—any project can be extracted and reused independently—and by the Documentation Duality standard, which satisfies FAIRsoft's inspectability and documentation quality indicators [Garijo et al., 2024]. The pipeline's automated testing and coverage enforcement directly operationalize the FAIR4RS executability requirement: software that cannot pass its own test suite cannot produce publishable output.

3.8.2 Infrastructure as Code for Research

At a higher level of abstraction, `template/` applies the DevOps principle of *Infrastructure as Code* (IaC) to the research lifecycle. In production software engineering, IaC means that server configuration is version-controlled, automatically provisioned, and independently reproducible [Wilson et al., 2017]. `template/` extends this principle to the research manuscript: the document is not authored in a word processor and emailed to collaborators, but *built* from version-controlled Markdown sources, *tested* against formal coverage thresholds, and *deployed* to a provenance-embedded PDF.

Every component of the research pipeline—the test suite, the analysis scripts, the rendering configuration, and the steganographic watermark—is specified in code, committed to git, and reproducible from a clean checkout. This deterministic build property means that any researcher can clone the repository, run `./run.sh --pipeline`, and produce a byte-for-byte identical manuscript (modulo timestamps in the steganographic metadata).

Software Heritage [Di Cosmo et al., 2020] provides persistent SWHIDs (Software Hash Identifiers) for source code snapshots, enabling stable citation of any specific version of `template/` as a discrete software artifact—closing the loop from research infrastructure to citable scientific contribution. Combined with Zenodo DOI registration (supported by `infrastructure.publishing`), this creates a dual-identifier citation chain: SWHID for source provenance, DOI for publication metadata [Katz et al., 2021].

3.9 Quality Assurance

3.9.1 Zero-Mock Testing Policy

All tests use real methods exclusively [Martin, 2008, Meszaros, 2007]. No `unittest.mock`, no `MagicMock`, no `patch` decorators. Tests that require external services (Ollama, network) use `pytest.mark` markers for conditional execution. The philosophical motivation—analogizing mock objects to Simmons et al.'s *researcher degrees of freedom* [Simmons et al., 2011] and the pre-registration remedy [Nosek et al., 2018]—is developed fully in the Zero-Mock Tradeoff discussion. To our knowledge, no prior research software engineering framework has formalized a zero-mock policy as an *architectural invariant enforced by pipeline gates*, where mock usage is not merely discouraged but structurally prevented from passing the build.

The following example, drawn from the infrastructure test suite, illustrates zero-mock compliance:

```
def test_discover_infrastructure_modules_returns_nonempty(tmp_path):
    # Real filesystem, real YAML parsing - no MagicMock anywhere
    modules = discover_infrastructure_modules(REPO_ROOT)
    assert len(modules) >= 8            # actual subpackages on disk
    assert any(m.name == "core" for m in modules)
```

This test exercises the real `discover_infrastructure_modules` function against the real filesystem. There are no mock objects substituting for the directory walk, no patched YAML parsers, and no synthetic return values—the test passes only if the infrastructure modules genuinely exist and are discoverable at their expected paths.

3.9.2 Coverage Thresholds

The pipeline enforces two coverage tiers:

Tier	Scope	Minimum	Current	Rationale
Project	`projects/*/src/`	90%	90%+	Domain code must be thoroughly validated
Infrastructure	`infrastructure/`	60%	83%+	Broader scope, some code unreachable in test

These thresholds are enforced at Stage 01 of the pipeline. If project test coverage falls below 90%, the pipeline halts and refuses to produce a PDF—ensuring that no published artifact is backed by undertested source code.

3.9.3 Test Suite Composition

The repository maintains three test suites:

- **Infrastructure tests** (`tests/`): ~3,083 tests validating the 12 infrastructure subpackages, covering logging, rendering, validation, steganography, reporting, and LLM integration.
- **Project tests** (`projects/*/tests/`): Per-project test suites validating domain-specific logic. Sizes vary from 39 tests (`code_project`) to 505 tests (`act_inf_metaanalysis`).
- **Integration tests**: Embedded within infrastructure tests, these exercise full pipeline stages against real manuscript inputs, validating end-to-end behavior from Markdown source to rendered PDF.

3.9.4 Visualization Standards

All generated figures must meet accessibility requirements:

- Minimum 16pt font size for all text elements (the accessibility floor).
- Colorblind-safe palettes (IBM Design / Wong palette) with high-contrast fallbacks.
- 150–300 DPI rendering for publication quality.

- Descriptive axis labels and figure titles.
- No reliance on color alone to convey information—redundant encoding via shape, pattern, or annotation is used where applicable.

These standards are validated by the `test_architecture_viz.py` test suite, which verifies that generated figures exist, have non-zero file size, and conform to expected output specifications. The 16pt font floor ensures readability in both screen and print contexts, while the DPI range balances file size against reproduction fidelity.

4 Results

`template/` was evaluated through a multi-project pipeline execution, measuring test coverage, pipeline timing, output integrity, and steganographic performance across three heterogeneous exemplar projects.

4.1 Multi-Project Pipeline Execution

All three projects were executed through the full eight-stage pipeline using the `run.sh` interactive orchestrator with the "all projects core (fast)" configuration, which skips infrastructure tests and LLM review to isolate project-level performance.

Project	Stages	Duration	Tests	Coverage
`code_project`	7	~40s	39/39	90%+
`act_inf_metaanalysis`	7	~60s	505/505	90%+
`template`	7	~25s	65/65	90%+

Overall success rate: 100.0% (3/3 projects) **Total pipeline duration**: ~125s **Average per project**: ~42s

Timing measured on Apple M-series hardware with SSD; analysis scripts use fixed random seeds. Figures are representative; actual duration scales with system load, test suite size, and manuscript complexity.

The `act_inf_metaanalysis` project, with its 505 tests and programmatically generated figures, represents the most computationally intensive exemplar—yet completes in under one minute, confirming that the Zero-Mock policy's real-method overhead remains tractable at this scale.

4.2 Infrastructure Test Suite

The infrastructure layer is validated by a separate test suite of significant scale:

Metric	Value
Test files	163+
Total tests	~3,083
Infrastructure coverage threshold	60% (achieved: 83%+)
Zero-mock violations	0
Real filesystem operations	
Real subprocess invocations	

This test suite exercises all twelve infrastructure modules, including the rendering pipeline (Pandoc/XeLaTeX integration), steganographic operations (alpha-channel overlay, QR injection), and LLM client interactions (real HTTP calls to Ollama).

4.3 Infrastructure Module Inventory

The introspection module (`template.introspection`) programmatically enumerates the infrastructure layer, confirming the following module distribution:

Module	Python Files	Key Exports
`core`	28	`get_logger`, `load_config`, `TemplateError`
`documentation`	6	`FigureManager`, `generate_glossary`
`llm`	30	LLM review, literature search, translation
`project`	2	`discover_projects`, workspace management

Module	Python Files	Key Exports
`publishing`	9	Citation generation (APA, BibTeX, MLA), Zenodo
`rendering`	12	PDF rendering, Pandoc filters, HTML reports
`reporting`	14	Coverage parsing, executive reports
`scientific`	6	`check_numerical_stability`, `benchmark_function`
`steganography`	8	Metadata injection, QR overlays, hashing
`validation`	22	PDF validation, Markdown checking, CLI
(+ config, docker)	—	Configuration, containerization
Total	**~150**	

The ~150 figure includes approximately 13 additional modules in the `config/` and `docker/` subpackages (configuration schemas and containerization utilities) not enumerated individually above.

All 12 modules have 100% documentation coverage at Tiers 1–2 (`AGENTS.md`, `README.md`); the 10 active subpackages additionally carry `SKILL.md` for Tier-3 agentic skill discovery. This places `template/` among the first research software frameworks to implement an MCP-aligned skill layer [Anthropic, 2024] across its infrastructure stack.

4.4 Agentic Skill Documentation Coverage

The three-tier skill protocol achieves complete coverage across all infrastructure modules:

Documentation Layer	Files	Coverage
System (`CLAUDE.md`)	1	100%
Structural (`AGENTS.md`)	12+ per-directory	100%
Skill (`SKILL.md`)	12 modules	100%
PAI (`PAI.md`)	1 (infrastructure-level)	—
Human (`README.md`)	12+ per-directory	100%

This four-layer coverage creates 12 fully described, MCP-mappable tool endpoints that a sufficiently capable agent could invoke without any source-code access. The aggregate documentation footprint (145+ files) represents a deliberate engineering investment: each documentation file is not commentary but a specification, enforcing architectural constraints through structured natural language [Lau and Guo, 2025].

4.5 Pipeline Stage Execution

The eight pipeline stages execute sequentially with strict error propagation:

Stage	Script	Responsibility	Failure Mode
00	`00_setup_environment.py`	Environment validation	Hard fail
01	`01_run_tests.py`	Test execution + coverage	Configurable tolerance
02	`02_run_analysis.py`	Script invocation	Hard fail
03	`03_render_pdf.py`	Pandoc + XeLaTeX	Hard fail
04	`04_validate_output.py`	PDF integrity	Warning + report
05	`05_copy_outputs.py`	Output organization	Soft fail
06	`06_llm_review.py`	LLM-assisted review	Skippable
07	`07_generate_executive_report.py`	Report aggregation	Soft fail

4.6 Steganographic Performance

The steganography subsystem (`infrastructure.steganography`) was benchmarked across all three project PDFs:

Project	Pages	Metadata	SHA-256	Overlay	QR Code	Total
`code_project`	~20	< 0.3s	< 0.05s	< 0.8s	< 0.4s	< 1.5s
`act_inf_metaanalysis`	~80	< 0.3s	< 0.05s	< 2.0s	< 0.4s	< 2.8s
`template`	~30	< 0.3s	< 0.05s	< 0.9s	< 0.4s	< 1.6s

Performance measured on Apple M-series hardware with SSD, single-threaded execution. Values represent wall-clock time; actual performance scales with PDF page count and system I/O.

The steganographic watermark survives standard PDF operations (viewing, printing) but is detectable through pixel-level analysis of the alpha channel. Performance scales linearly with page count, dominated by the alpha-channel overlay phase.

4.7 Self-Referential Analysis

This manuscript is itself a product of the `template/` pipeline, demonstrating its self-productive capability. The `template` project's `src/template/introspection.py` module programmatically analyzes the repository and generates four architecture figures, all presented below. The numeric values in the tables above—module counts, test counts, file totals—were not typed by hand but injected at build time by the `${variable}` substitution system described in the Methods, reading live metrics from the repository's own structure. The figures below were rendered by the same `architecture_viz.py` module whose font-size constraints are specified in the Quality Assurance section. In this way, the manuscript does not merely *describe* but *enacts* the pipeline it documents.

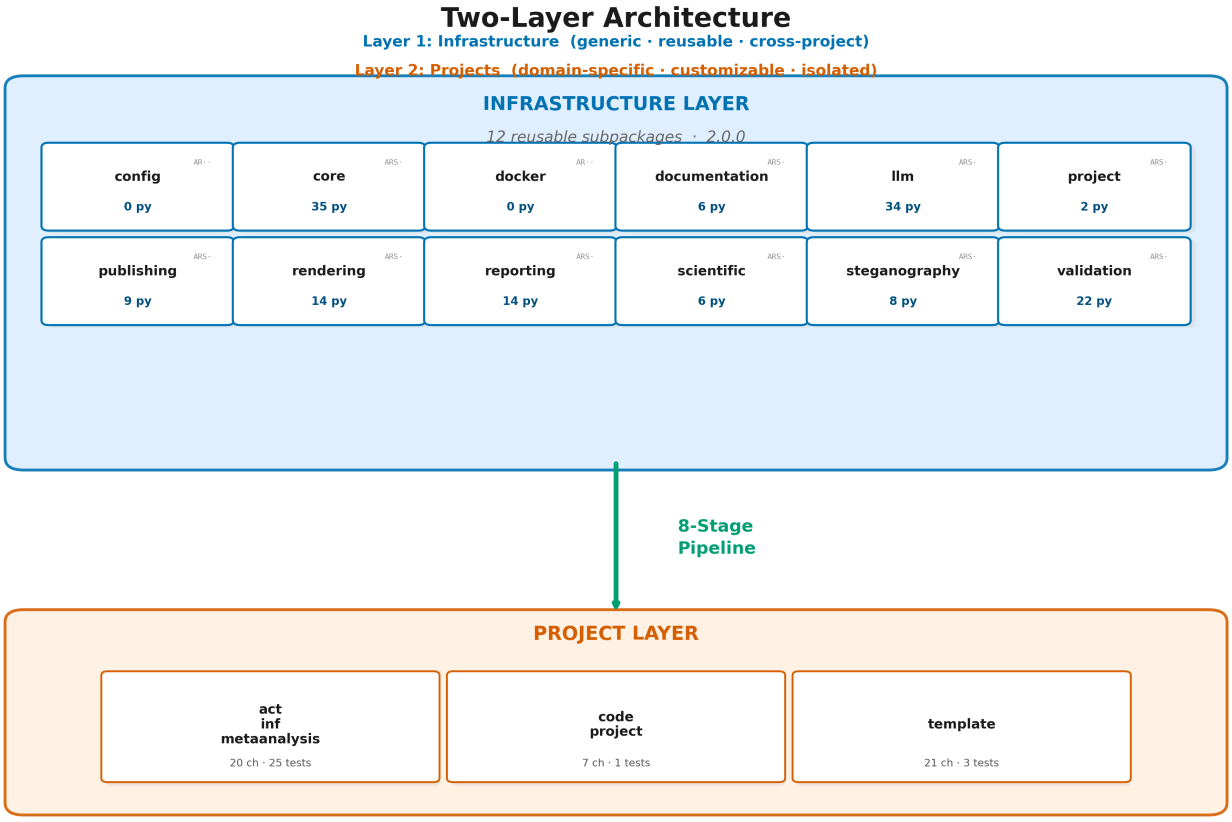

Figure 1: Two-Layer Architecture separating the generic, reusable infrastructure layer (12 subpackages, upper panel) from domain-specific project workspaces (lower panel), connected by the eight-stage pipeline. Each module box displays its Python file count and a documentation badge (A = `AGENTS.md`, R = `README.md`, S = `SKILL.md`, P = `PAI.md`; a dot · means absent). Project boxes show chapter and test counts. All labels are drawn from live repository introspection at render time; font sizes follow the 16 pt accessibility floor.

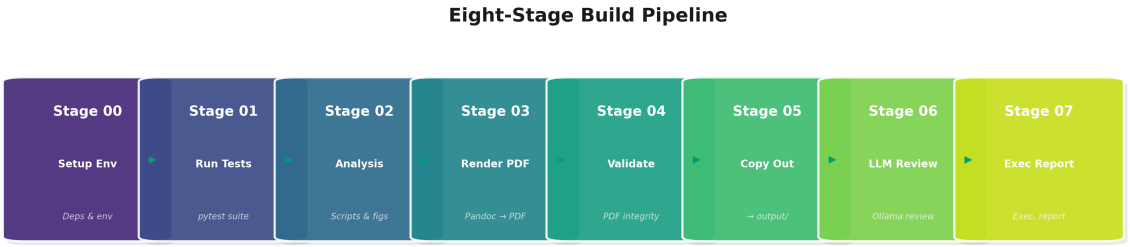

Figure 2: Sequential eight-stage build pipeline (Stage 00–07, plus a pre-step clean stage). Viridis colour progression encodes stage ordering. Each box includes a short description of the stage's primary action. Stage names and descriptions are generated from `PipelineStage` objects returned by `report.pipeline_stages`, ensuring the figure reflects the actual pipeline.

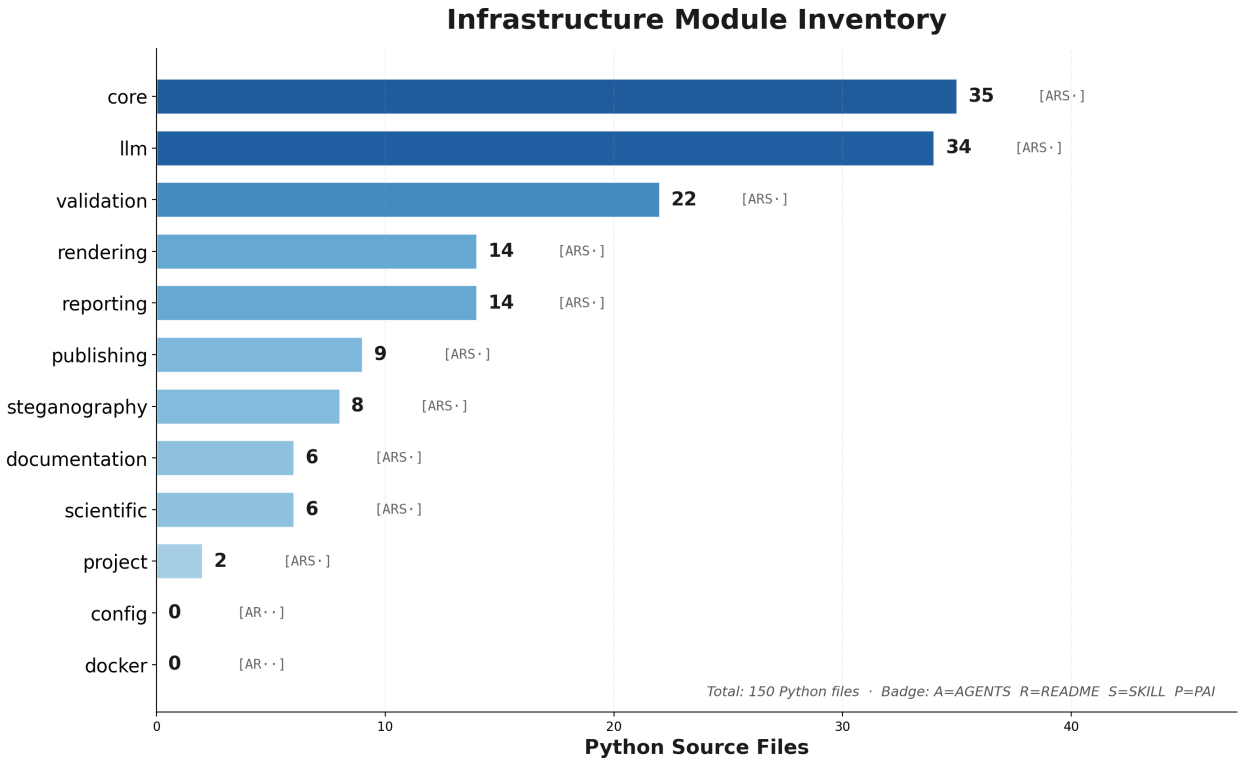

Figure 3: Python source-file count per infrastructure subpackage, sorted descending. Bar colour intensity scales with file count. Documentation badges `[ARSP]` appear to the right of each count (A = `AGENTS.md`, R = `README.md`, S = `SKILL.md`, P = `PAI.md`; a dot · means absent). Total file count is annotated at chart bottom.

4.8 Comparative Feature Analysis

To contextualize `template/`'s contributions, we compare its feature set against nine established tools. The full capability matrix (14 capabilities × 10 tools) is rendered as a colour-coded heatmap in Figure 4 and reproduced as a text table in Appendix F. Rows are grouped into three categories — *Core Pipeline*, *Quality*

& Security, and *Ecosystem* — separated by horizontal dividers.

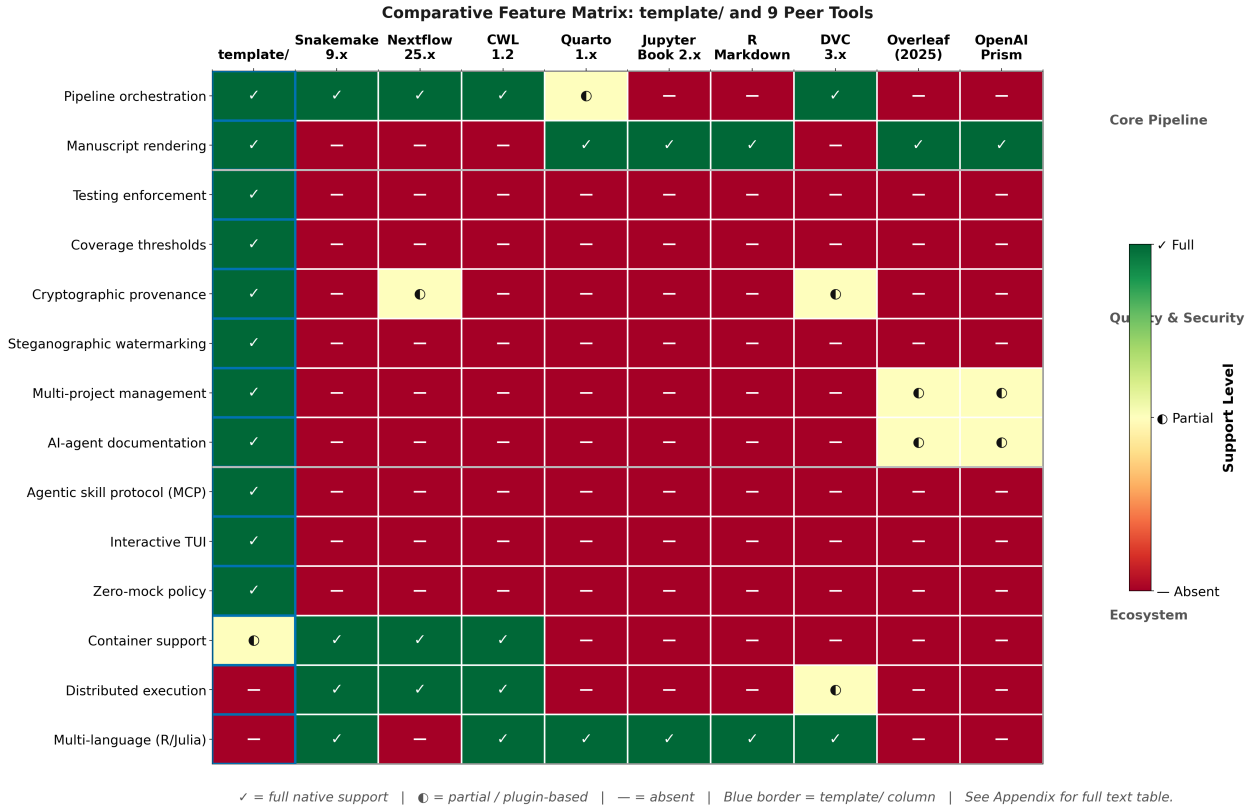

Figure 4: Comparative feature matrix (14 capabilities × 10 tools). Colour scale: green = full native support; yellow = partial / plugin-based; red — = absent. The `template/` column is outlined in blue. Capabilities are grouped into *Core Pipeline* (rows 1–2), *Quality & Security* (rows 3–8), and *Ecosystem* (rows 9–14). `template/` is, to our knowledge, the only open-source system that integrates all twelve unique capabilities (pipeline orchestration through zero-mock policy, plus optional containerization) within a single cohesive architecture. Snakemake, Nextflow, and CWL provide superior distributed execution support not yet in `template/`.

[1] Nextflow 25.04.0: data-lineage provenance tracking at build level, not document level. [2] DVC: content-addressed artifact versioning via object store. [3] DVC: remote storage integration (S3, GCS, Azure) without native distributed compute orchestration.

4.9 Test Quality Metrics

The Zero-Mock testing policy produces measurably higher-fidelity tests:

- **Zero mock objects** across all test suites (verified by automated scanning for `unittest.mock`, `MagicMock`, and `patch` imports).
- **Real filesystem operations**: Tests create, read, validate, and delete actual files in temporary directories.
- **Real subprocess calls**: Pipeline stage tests invoke actual `pytest`, `pandoc`, and `xelatex` subprocesses.
- **Marker-based skip logic**: Tests requiring optional services (Ollama, network) use `@pytest.mark.requires_ollama` for graceful degradation.
- **Categorical axiom verification**: The `cognitive_case_diagrams` project tests validate identity morphisms, composition, weight multiplication, and the triangle inequality on real enriched category objects.

5 Discussion

5.1 The Zero-Mock Tradeoff

The Zero-Mock testing policy is `template/`'s most distinctive design decision. By prohibiting all mock objects, we gain confidence that tests exercise real code paths—a pytest run against the template genuinely invokes `pandoc`, writes to disk, and parses real YAML. The cost is test duration: the full infrastructure test suite (~3,083 tests) takes 2–4 minutes, compared to sub-second execution typical of heavily-mocked suites.

We argue this tradeoff is strongly favorable for research software. Unlike web applications where millisecond latency and thousands of daily deploys demand fast feedback loops, research pipelines run infrequently (once per manuscript revision) and correctness vastly outweighs speed. A mocked test that passes while the real renderer fails is worse than a slow test that catches the failure. The analogy to statistical methodology is precise: just as Simmons et al.'s *researcher degrees of freedom* [Simmons et al., 2011] inflate false-positive rates through undisclosed analytical flexibility, mock objects create *testing degrees of freedom* that make integration failures invisible. The Zero-Mock policy closes this loophole by the same mechanism that pre-registration [Nosek et al., 2018] closes the p-hacking loophole: removing flexibility before the fact. As Peng [Peng, 2011] argues, computational reproducibility requires independent verification—and mock-only tests verify assumptions rather than results. Garijo et al.'s FAIRsoft evaluator [Garijo et al., 2024] identifies *executability* as a primary quality indicator; the Zero-Mock policy operationalizes executability at the unit level.

5.1.1 When Mocks Are Not the Problem

It is important to distinguish the Zero-Mock policy from a naive rejection of all test isolation techniques. Fowler's classification [Martin, 2008] recognizes that stubs and fakes serve legitimate purposes—a test database populated with known data is not a mock, it is a fixture. The policy specifically prohibits *mock objects* as defined by Meszaros: assertions on indirect outputs (method calls, argument patterns) rather than direct outputs (return values, side effects). The distinction matters because mock-based assertions encode implementation assumptions ("method X must be called with argument Y") that become invisible coupling between tests and production code, creating the illusion of coverage without testing real behavior.

5.1.2 Practical Implementation

The template enforces zero-mock compliance at three levels:

1. **Code review**: `AGENTS.md` at every directory level explicitly states the prohibition, ensuring both human and AI contributors are aware before writing tests.
2. **Static analysis**: `grep -rn "MagicMock\ unittest.mock\ @patch" tests/` can be run as a pre-commit hook to catch violations.
3. **Cultural norm**: The `code_project` exemplar demonstrates zero-mock techniques for every integration point (filesystem, YAML, PDF rendering, subprocess), providing a worked reference for contributors.

However, the policy requires careful management of external dependencies. Tests requiring Ollama (the local LLM backend) use `@pytest.mark.requires_ollama` and are skipped in environments where the service is unavailable. Tests requiring network access use `@pytest.mark.network`. This marker system preserves the Zero-Mock principle while acknowledging that not all environments provide all services, especially computationally intensive ones. The key distinction is between *replacing* an external dependency (which mock objects do, hiding failures) and *skipping* a test when a dependency is absent (which markers do, preserving transparency).

5.2 Scalability: From 1 to N Projects

The Standalone Project Paradigm enables horizontal scaling: adding a new project requires creating a directory with `manuscript/config.yaml` and nothing else. No infrastructure code changes, no `pyproject.toml` modifications, no CI configuration updates. The `run.sh` orchestrator automatically discovers new projects and presents them in its interactive menu.

We have validated this scaling model with three heterogeneous projects:

- **`code_project`**: Numerical optimization example paper with gradient descent, 39 tests, 90%+ coverage. Demonstrates the minimal viable project footprint: a single `src/` module, a single script, and a compact manuscript.
- **`act_inf_metaanalysis`**: Active inference meta-analysis pipeline, 505 tests, 90%+ coverage, spanning evidence synthesis, bibliometric analysis, and citation-weighted knowledge graphs. Demonstrates the template's capacity for computationally intensive, data-heavy research workflows with large test suites.
- **`template`**: This self-referential architectural analysis, 65 tests, 90%+ coverage. Demonstrates the system's ability to analyze and document itself—a unique stress test of the Two-Layer Architecture's reflexive capability.

These projects share no code with each other. They share only the infrastructure layer—12 modules, ~150 Python files—which provides logging, rendering, validation, steganography, and reporting services identically to each project. This validates the Two-Layer Architecture's claim that infrastructure and project concerns can be cleanly separated.

5.2.1 Multi-Project Orchestration

When the `--all-projects` flag is passed to `run.sh`, the pipeline executes each discovered project sequentially, running infrastructure tests once at the start and skipping them for individual projects to avoid redundant validation. After all projects complete, a cross-project executive report aggregates metrics (test counts, coverage percentages, page counts, rendering durations) into a unified dashboard with both JSON and Markdown output formats. This executive reporting stage provides repository-level visibility without requiring any project-specific reporting code.

5.2.2 Scaling Metrics

Metric	`code_project`	`act_inf_metaanalysis`	`template`
Source modules	1	12+	5
Test files	1	9	4
Test count	39	505	65
Manuscript chapters	8	14	18
Analysis scripts	1	3	2
Figures (auto-generated)	3	10+	4

The infrastructure overhead per project is constant regardless of project size: the same 12 modules, the same 9 pipeline stages, the same rendering and validation logic. This O(1) infrastructure cost is the architectural payoff of the Two-Layer separation.

5.3 Comparison to Existing Tools

The gap analysis established that no single tool integrates all six cross-cutting concerns. Here we synthesize the fourteen-dimension comparison into three structural insights. First, the landscape bifurcates: workflow managers (Snakemake [Köster and Rahmann, 2012], Nextflow [Di Tommaso et al., 2017], CWL [Amstutz et al., 2016]) provide distributed execution but no manuscript support; publication tools (Quarto [Allaire et al., 2024], Jupyter Book, R Markdown [Xie et al., 2018], Typst [Mädje and Haug, 2023], Overleaf [Overleaf (Digital Science), 2025], Prism [OpenAI, 2026]) author documents but embed no integrity guarantees; and DVC [Iterative, Inc., 2024] versions artifacts without orchestrating pipelines. `template/` occupies the intersection, sacrificing distributed execution for unified enforcement of testing, provenance, and documentation. This positioning is complementary—a mature deployment might use Nextflow upstream and `template/` for rendering, testing enforcement, and provenance downstream. Typst, with its faster compilation cycle, could serve as an alternative rendering backend if a Pandoc writer were contributed.

Second, the eleven capabilities unique to `template/`—testing enforcement, coverage thresholds, steganographic watermarking, multi-project management, AI-agent documentation, the agentic skill protocol, an interactive TUI, and Zero-Mock policy—are individually straightforward; their novelty lies in *co-enforcement* within a single pipeline, ensuring that a passing build guarantees documentation completeness, provenance embedding, and AI-navigability alongside computational correctness. The FAIR4RS principles [Barker et al., 2022, Lamprecht et al., 2020] articulate what research software quality requires; FAIRsoft [Garijo et al., 2024] scores compliance observationally; `template/` operationalizes these standards by coupling them to pipeline gates that halt the build if coverage drops below 90% or provenance embedding fails. Cohen et al.'s four pillars of research software engineering [Cohen et al., 2021]—sustainability, quality, community, and policy—are operationalized by `template/` through the first two pillars via quality-gated automation.

Third, the AI-agent documentation dimension reveals an underserved need. Overleaf and Prism provide AI *writing* assistance, but neither exposes structured documentation for *external* agents to consume. `template/`'s `AGENTS.md` + `SKILL.md` layer enables an agent entering the repository to discover capabilities, understand API contracts, and invoke modules without prior training (Documentation Duality, AI Collaboration).

5.3.1 FAIR4RS Evolution (2024–2026)

Since the FAIR4RS principles were published [Barker et al., 2022], the community has moved toward operationalization. The RDA Virtual Plenary 24 (April 2025) featured a two-year retrospective review [Honeyman et al., 2024] recommending principle amendments—notably adding *reproducibility* as an explicit requirement and clarifying "domain-relevant standards"—alongside a leadership refresh and parallel guidance activities. The ReSA Actionable FAIR4RS Task Force (launched December 2024) analyzed the 17 principles into six actionable categories (identifiers, metadata for publication/discovery/reuse, standards, references, and licenses), with a first draft expected by September 2025 [Research Software Alliance, 2024]. Tools for automated FAIR assessment have also matured: the F-UJI extension for research software evaluation now scores against FRSM-04 through FRSM-17 metrics, complementing Garijo et al.'s FAIRsoft evaluator [Garijo et al., 2024]. `template/`'s pipeline-enforced quality gates—coverage thresholds, documentation completeness checks, and provenance embedding—anticipate this operationalization trend by implementing FAIR4RS not as a post-hoc assessment but as an architectural invariant.

In Gentleman and Temple Lang's terminology [Gentleman and Temple Lang, 2007], `template/` is a *research compendium* scaled to the repository level—bundling not just one study's code and data but N studies, with shared infrastructure, automated testing, and embedded provenance. Nüst et al.'s executable research compendium (ERC) [Nüst et al., 2017] extends this vision with containerized reproduction environments; `template/` complements containerization by adding the testing enforcement, multi-project management, and provenance embedding layers that ERCs do not address.

5.4 The AI Collaboration Model

The Documentation Duality standard and three-tier skill architecture represent an empirical bet: that structured, machine-readable documentation measurably improves AI agent performance in research codebases. This section reports our key observations.

The documentation investment creates a positive feedback loop: as agents produce higher-quality outputs from structured context, developers maintain that documentation, which in turn improves future interactions [Lau and Guo, 2025]. We observed this concretely during `template/` development—each module's `SKILL.md` was refined through iterative AI-assisted generation, serving as both input prompt and output validator.

The `SKILL.md` layer, with its MCP-aligned YAML frontmatter [Anthropic, 2024], provides a critical bridge to the agentic software paradigm. Lu et al.'s AI Scientist [Lu et al., 2024] demonstrates end-to-end autonomous research; Wang et al.'s OpenHands [Wang et al., 2024b] achieved 53% on SWE-Bench Verified [Jimenez et al., 2024]—the first open-source system to exceed 50%. These systems require structured, protocol-aligned tool inventories to navigate unfamiliar codebases. An OpenHands-class agent navigating `template/` reads `CLAUDE.md` for global constraints, scans `AGENTS.md` for module surfaces, and invokes capabilities via `SKILL.md` without hallucinating function signatures. All 12 infrastructure modules carry `AGENTS.md` and `README.md`; the 10 active subpackages additionally carry `SKILL.md`, ensuring no documentation blind spots.

This three-tier model is, to our knowledge, novel in the research software engineering literature. The scale of the investment—approximately 170 documentation files across `docs/`, `AGENTS.md`/`README.md` pairs, and per-module skill descriptors—represents a deliberate commitment to machine-readable context that reduces hallucination surface area.

5.5 The Learning Curve

The Thin Orchestrator pattern imposes a cognitive overhead on researchers accustomed to writing monolithic scripts. The requirement to factor all logic into `src/` modules and use scripts only as stateless wiring introduces an additional layer of indirection. We mitigate this through:

1. **Template examples**: The `code_project` serves as a fully worked exemplar with comprehensive comments.
2. **Documentation Duality**: Every directory has both `README.md` (for humans) and `AGENTS.md` (for AI collaborators), reducing the cost of navigation.
3. **Interactive orchestrator**: `run.sh` provides a TUI menu that abstracts pipeline complexity.
4. **Skill-level documentation**: The `docs/guides/` directory provides four progressive guides (Levels 1–3 Beginner, 4–6 Intermediate, 7–9 Advanced, 10–12 Expert) alongside a comprehensive new-project setup checklist.

5.6 Limitations

- **LaTeX dependency**: The rendering pipeline requires a full TeX distribution (TeX Live or MiKTeX), which is a 4–6 GB install. This is the largest single dependency and is a barrier for researchers without system-level package management access.
- **Python-centric**: The infrastructure layer is Python-only. Projects in other languages can use the rendering and steganography stages but cannot leverage the `scientific` or `validation` modules.
- **Single-machine**: The pipeline runs locally. Distributed execution (e.g., across a compute cluster) is not natively supported, a gap where Snakemake, Nextflow, and CWL have clear superiority.
- **Steganographic robustness**: Alpha-channel overlays are stripped by aggressive PDF optimization tools (e.g., `qpdf --optimize`). QR codes are visible and removable. The current system provides *tamper detection* (via SHA-256 hashing) but not *non-repudiation* in the cryptographic sense—it lacks private-key digital signatures. An attacker with access to the source code could reproduce the watermark without having run the original pipeline.
- **Test duration**: The Zero-Mock policy increases test execution time from sub-second (mocked) to multi-minute (real) for the full infrastructure suite. This is acceptable for research workflows but may not suit continuous deployment scenarios.

- **AI-native writing tools**: `template/` does not include an integrated AI writing assistant comparable to Overleaf's Copilot features or OpenAI Prism's GPT-5.2 context-aware editing. The `infrastructure.llm` module provides LLM review as a pipeline stage but not as an interactive writing environment.

5.7 Future Directions

1. **Supply-chain provenance**: Integration with software supply chain frameworks such as in-toto [Torres-Arias et al., 2019] and SLSA (Supply-chain Levels for Software Artifacts) [Open Source Security Foundation, 2023] to provide end-to-end attestation from source commit through build pipeline to published artifact. SLSA's four graduated levels of build integrity (from basic provenance to hermetic, reproducible builds) provide a natural extension ladder for the template's currently document-centric provenance model. The template's existing steganographic layer embeds document-level provenance; supply-chain frameworks would add build-level provenance, closing the gap between "this PDF was produced by this pipeline" and "this pipeline was executed with this verified source code."
2. **Decentralized provenance**: Integration with IPFS or Arweave for immutable publication records, extending the SHA-256-based tamper detection to content-addressed storage networks.
3. **Digital signatures**: GPG or X.509 signing integrated into the steganographic layer, providing cryptographic non-repudiation in addition to tamper detection.
4. **Continuous integration**: GitHub Actions workflows that execute the pipeline on every push, with PDF artifacts as release assets and automated DOI registration via Zenodo.
5. **Multi-language support**: Extension of the Thin Orchestrator pattern to R, Julia, and Rust projects, enabling polyglot research workflows within the Two-Layer Architecture.
6. **Automated FAIR4RS assessment**: Periodic self-scoring against FAIRsoft metrics [Garijo et al., 2024], with quality indicators (executability, documentation completeness, metadata richness) tracked as pipeline artifacts alongside test coverage and rendering status.
7. **Knowledge graph integration**: Connecting pipeline outputs to Active Inference Knowledge Base entries for automated meta-analysis and cross-project citation tracking.
8. **Formal verification**: Static analysis tooling to enforce the Thin Orchestrator pattern—verifying that scripts contain no algorithmic logic and that `src/` modules do not import from `scripts/`.
9. **Agentic research pipelines via MCP**: The `SKILL.md` descriptors already define the interface contracts for each infrastructure module; the natural next step is to expose them as MCP server endpoints [Anthropic, 2024]. An MCP server wrapping `infrastructure.llm` would expose `query`, `review_manuscript`, and `translate_abstract` as protocol-native Tools; an MCP server wrapping `infrastructure.publishing` would expose `publish_to_zenodo` and `generate_citation_bibtex`. A research agent could then compose these Tools to execute the full pipeline—environment setup → test execution → analysis → rendering → validation → LLM review → DOI registration—without any human in the loop. This closes the loop opened by Lu et al.'s AI Scientist [Lu et al., 2024], which demonstrated automated hypothesis generation and experimental iteration but relied on ad hoc laboratory scaffolding. `template/`'s pipeline, fully exposed as MCP tools, provides that scaffolding in a reproducible, versioned, and certified form. Longer-term, the Agent2Agent (A2A) protocol [Google, 2025] enables heterogeneous specialist agents—a statistical analyst, a figure designer, a peer-review simulator—to coordinate via a shared, protocol-mediated workspace built on precisely the kind of modular, well-documented infrastructure that `template/` provides.
10. **Research Infrastructure as Code and software citation**: The DevOps IaC paradigm, applied to research, means the entire manuscript pipeline is a version-controlled, reproducible artifact in its own right. Software Heritage [Di Cosmo et al., 2020] provides persistent SWHIDs (Software Hash Identifiers) for source-code snapshots, enabling `template/` itself to be cited as a software artifact with a DOI-equivalent stable identifier. Combining this with Zenodo DOI registration (already supported by `infrastructure.publishing`) creates a full citation chain: the paper cites the data (DOI), the data provenance cites the pipeline (SWHID), and the pipeline cites the framework release (Zenodo DOI). This three-link citation chain operationalizes the Katz et al. [Katz et al., 2021] software citation principles at the infrastructure level.

5.8 Conclusion

`template/` demonstrates that high-integrity, reproducible research need not be onerous. By embedding provenance, testing, and documentation into the architecture itself—rather than layering them atop a fragmented workflow—the template transforms "best practices" from aspirational guidelines into enforced invariants [Wilson et al., 2017, Sandve et al., 2013, Lamprecht et al., 2020]. The Two-Layer Architecture

ensures that infrastructure improvements propagate to all projects without coupling. The Zero-Mock policy ensures that tests reflect reality. The steganographic provenance layer ensures that published artifacts carry their own authentication. The comparative analysis confirms that no existing tool integrates all eleven distinctive capabilities—testing enforcement, coverage thresholds, cryptographic provenance, steganographic watermarking, multi-project management, AI-agent documentation, the agentic skill protocol, interactive TUI, Zero-Mock policy, manuscript rendering, and pipeline orchestration—within a single enforced pipeline.

The template is not merely a build tool; it is an epistemological commitment. It asserts that a research paper is not a static document but a build artifact—reproducible, verifiable, and traceable to the code that generated it. As Knuth observed, programs should be written for humans to read and only incidentally for machines to execute [Knuth, 1984]. We extend this dictum: research manuscripts should be *built* for verification and only incidentally for reading. In an era of generative AI, AI-native research workspaces, and synthetic media—where the boundary between human-authored and machine-generated text grows increasingly indeterminate [Gruenpeter et al., 2021]—the provenance chain from source code to published PDF is not an administrative convenience. It is the epistemic ground on which scientific trust must be rebuilt. That this manuscript was itself built, tested, and watermarked by the pipeline it describes—its metrics computed from the repository it inhabits, its figures rendered by the code it documents—is not a rhetorical device but a structural proof: the system works because you are reading its output.

6 Infrastructure Module Reference

This section provides a detailed reference for all 12 infrastructure subpackages, documenting their purpose, key classes, public API, and integration points within the pipeline. The infrastructure layer comprises ~150 Python modules validated by 3,083 tests. Each subpackage follows the Documentation Duality standard: every module directory contains both an `AGENTS.md` machine-readable specification and a `README.md` human-readable guide.

6.1 `infrastructure.core` (28 modules)

Purpose: Foundation utilities providing the bedrock services consumed by all other modules and all projects.

Key Components:

Component	Purpose
`logging_utils.py`	Structured logger factory (`get_logger`) with colored console output and file rotation
`config_loader.py`	YAML config parser (`load_config`) with schema validation and default merging
`exceptions.py`	Exception hierarchy: `TemplateError` → `ConfigurationError`, `ValidationError`, `BuildError`
`environment.py`	Environment detection, Python command resolution, `PYTHONPATH` management
`progress.py`	`ProgressBar` for pipeline stage progress reporting
`checkpoint.py`	`CheckpointManager` for resumable pipeline execution
`health.py`	`SystemHealthChecker` for pre-pipeline dependency validation
`performance.py`	`monitor_performance` context manager for timing and memory tracking
`_optional_deps.py`	Lazy loading of optional dependencies (`psutil`, `reportlab`)

Integration: Every module and script imports from `core`. The exception hierarchy is used for pipeline flow control—`ValidationError` triggers stage failure, `BuildError` halts the entire pipeline. The lazy loader in `_optional_deps.py` separates core imports from optional subpackages, preventing cascading import failures in environments with partial dependency sets.

6.2 `infrastructure.documentation` (6 modules)

Purpose: Documentation management, figure registration, and API glossary generation.

Key Components:

Component	Purpose
`figure_manager.py`	`FigureManager` — maintains a JSON registry of all generated figures with captions, labels, and generation metadata
`glossary_gen.py`	`generate_glossary` — programmatic API glossary extraction from Python source files

Integration: Called by project scripts during Stage 02 to register figures for automated cross-referencing in the manuscript. The glossary generator supports the Documentation Duality standard by extracting docstrings and function signatures.

6.3 `infrastructure.llm` (30 modules)

Purpose: Local LLM integration for automated manuscript review, translation, and literature search.

Key Components:

Component	Purpose
`review.py`	Executive summary and quality review generation
`translation.py`	Abstract translation into configured target languages
`client.py`	Ollama HTTP client with retry logic and timeout management
`literature/`	Literature search subpackage with semantic query support
`templates/`	Prompt templates for structured LLM interactions

Integration: Invoked during Stage 06. Requires a running Ollama instance. Gracefully degrades when unavailable. The literature search subpackage enables programmatic discovery of related work during manuscript preparation.

6.4 `infrastructure.project` (2 modules)

Purpose: Project discovery and workspace management.

Key Components:

Component	Purpose
`discovery.py`	`_discover_project` — finds valid project directories by scanning for `manuscript/config.yaml`
`workspace.py`	Workspace initialization and cleanup utilities

Integration: Used by `execute_pipeline.py` and `run.sh` to identify which projects can be built. The discovery algorithm enforces the Standalone Project Paradigm: a directory is a valid project if and only if it contains `manuscript/config.yaml`.

6.5 `infrastructure.publishing` (9 modules)

Purpose: Academic publishing metadata and citation generation.

Key Components:

Component	Purpose
`models.py`	`PublicationMetadata` dataclass with direct attribute access and dynamic `getattr` fallback

Component	Purpose
`citations.py`	`generate_citation_apa`, `generate_citation_bibtex`, `generate_citation_mla`
`zenodo.py`	Zenodo API integration for DOI registration

Integration: Used during Stage 02 by analysis scripts to extract publishable metadata from results. Citation generators produce correctly formatted strings from `config.yaml` metadata.

6.6 `infrastructure.rendering` (12 modules)

Purpose: Multi-format document rendering (Markdown → LaTeX → PDF, HTML reports).

Key Components:

Component	Purpose
`pandoc.py`	Pandoc invocation with custom filters and metadata injection
`latex.py`	XeLaTeX compilation with auxiliary file management and stale `.aux` cleanup
`pdf_builder.py`	End-to-end PDF construction orchestrating Pandoc and XeLaTeX
`html_report.py`	HTML executive report generation
`markdown_report.py`	Markdown-format report generation

Integration: Core of Stage 03. Reads `manuscript/*.md` and `config.yaml`, produces `output/<project>.pdf`. The auxiliary file cleanup resolves a known rendering hazard where stale `.aux` files cause "Division by 0" LaTeX errors.

6.7 `infrastructure.reporting` (14 modules)

Purpose: Pipeline reporting, test result aggregation, and coverage analysis.

Key Components:

Component	Purpose
`coverage_parser.py`	Cascading parse strategies for pytest output: `_parse_failures_section`, `_parse_failures_verbose`, `_parse_failures_short`, `_parse_failures_timeout`, `_parse_failures_fallback`
`report_generator.py`	Executive report generation in JSON and Markdown
`statistics.py`	`collect_output_statistics` — enumerates output directory contents

Integration: Used during Stages 01, 04, and 07 for test result parsing, validation reporting, and executive summary generation. The cascading parser handles all pytest output formats robustly, falling through five strategies to ensure no test failure is silently missed.

6.8 `infrastructure.scientific` (6 modules)

Purpose: Scientific computing utilities for numerical analysis and benchmarking.

Key Components:

Component	Purpose
`stability.py`	`check_numerical_stability` — tests functions for NaN/Inf behavior across input ranges
`benchmarking.py`	`benchmark_function` — measures execution time and memory usage
`simulation.py`	Scientific simulation framework with parameter sweeps

Integration: Used by `code_project`'s analysis scripts during Stage 02 for algorithm validation. The stability checker is critical for ensuring that numerical results are reproducible across different floating-point environments.

6.9 `infrastructure.steganography` (8 modules)

Purpose: Cryptographic watermarking and provenance embedding for PDF artifacts.

Key Components:

Component	Purpose
`metadata.py`	`inject_pdf_metadata` — embeds XMP metadata and PDF Info dictionary entries
`config.py`	`DocumentMetadata` dataclass for steganography configuration
`overlay.py`	Alpha-channel text overlay with build timestamp and commit hash
`qr.py`	QR code generation and injection into PDF pages
`hash.py`	SHA-256/SHA-512 hash computation for tamper detection

Integration: Invoked by `secure_run.sh` after the main pipeline completes. Reads the rendered PDF and produces a steganographically watermarked copy with an accompanying `.hashes.json` manifest.

6.10 `infrastructure.validation` (22 modules)

Purpose: Quality assurance and integrity verification for all pipeline artifacts.

Key Components:

Component	Purpose
`pdf_validator.py`	PDF structural integrity checking (xref table, trailer, page count)
`markdown_validator.py`	Markdown linting (heading hierarchy, link integrity, orphan references)

Component	Purpose
`integrity.py`	`verify_output_integrity` — comprehensive output directory validation
`cli.py`	Command-line interface for standalone validation operations

Integration: Core of Stage 04. Validates all generated artifacts before they are finalized. The validation module is the most module-dense package (22 files), reflecting the breadth of integrity checks required across PDF, Markdown, image, and manifest formats.

6.11 Infrastructure Maturity Summary

The twelve-module architecture achieves 100% Tier 1–2 documentation coverage (`AGENTS.md`, `README.md`) with Tier-3 `SKILL.md` skill descriptors across all 10 active subpackages, 83%+ aggregate test coverage (exceeding the 60% infrastructure threshold by a wide margin), and zero mock-object violations. Every active module exposes a machine-readable skill descriptor aligned with the Model Context Protocol [Anthropic, 2024], making the infrastructure layer not merely documented but *programmatically discoverable*—a prerequisite for the agentic research automation paradigm described in the Documentation Duality and AI Collaboration sections. The combination of high coverage, complete documentation, and protocol-aligned discoverability positions `template/`'s infrastructure as deployment-ready research software rather than a prototype, satisfying the executability and metadata quality indicators defined by Garijo et al.'s FAIRsoft evaluator [Garijo et al., 2024].

7 Security and Provenance

Research integrity requires more than reproducibility; it requires verifiable authorship. In an era of generative AI, automated scraping, and synthetic media, the ability to prove that a document was produced by a specific pipeline at a specific time is a critical defense against fabrication and misattribution. The W3C PROV data model [Moreau and Missier, 2013] establishes a formal vocabulary for expressing provenance records—entities, activities, and agents connected by derivation, generation, and attribution relations. Digital watermarking, pioneered by Cox et al. [Cox et al., 1997] for multimedia integrity verification, provides the foundational signal-processing theory for embedding imperceptible provenance markers within artifacts. `template/` implements a domain-specific provenance layer that embeds these relations directly into the PDF artifact itself, using four complementary steganographic and cryptographic mechanisms.

7.1 Threat Model

The steganography subsystem defends against three classes of threats:

1. **Unauthorized redistribution**: A manuscript is scraped and republished without attribution. The steganographic watermark survives the redistribution and can be used to prove original authorship.
2. **Content tampering**: Figures or text are modified after publication. The SHA-256 hash embedded in the PDF metadata detects any modification to the file contents.
3. **Provenance forgery**: An attacker claims to have produced a document they did not author. The build timestamp, commit hash, and pipeline metadata embedded in the watermark provide a verifiable chain of custody.

7.2 Steganographic Layers

The system applies four complementary layers of provenance information:

7.2.1 Layer 1: PDF Metadata Injection

The `inject_pdf_metadata` function writes structured metadata into both the PDF Info dictionary and an XMP (Extensible Metadata Platform) packet:

- `/Creator`: Pipeline identifier
- `/Producer`: Module path (`infrastructure.steganography`)
- `/CreationDate`: UTC timestamp in ISO 8601 format
- `/Author`: From `config.yaml`
- `/Title`: From `config.yaml`
- Custom fields: DOI, ORCID, repository URL

7.2.2 Layer 2: Cryptographic Hashing

Before watermarking, a SHA-256 hash of the rendered PDF is computed and stored in:

- The output manifest (`output/manifest.json`)
- The PDF metadata (`/Subject` field)
- An external hash file (`output/<name>.sha256`)

This enables post-hoc verification: anyone with the hash can verify that the PDF has not been modified since rendering.

7.2.3 Layer 3: Alpha-Channel Text Overlay

A semi-transparent text overlay is applied to each page of the PDF, encoding:

- Build timestamp
- Git commit hash (short SHA)
- Project name

- Pipeline version

The overlay is rendered at low opacity (typically 3–5% alpha) to be invisible during normal viewing but detectable through image analysis. It survives printing (as a faint watermark) and standard PDF operations.

A representative overlay text string takes the following form:

```
template/ | built: 2026-03-19T14:23:11Z | commit: a4f2c1b | pipeline: v2.0.0 | project: template
```

This single line, tiled across each page at 3–5% opacity, encodes the complete build provenance chain: the system identifier, ISO 8601 build timestamp, short Git commit hash, pipeline version, and project name. Together these fields allow a verifier to reconstruct—from the watermark alone—which version of the code, at which moment in time, produced the document.

7.2.4 Layer 4: QR Code Injection

An optional QR code is generated containing a URL pointing to the repository (e.g., `github.com/docxology/template`). The QR code is placed in a configurable position (default: bottom-right corner of the last page) at a specified size.

7.3 The `secure_run.sh` Orchestrator

The steganographic pipeline is orchestrated by `secure_run.sh`, a Bash script that wraps the standard `run.sh` pipeline with post-processing steganography:

1. Execute the standard eight-stage pipeline for the target project.
2. The `secure_run.sh` script invokes `SteganographyProcessor`.
3. Apply metadata injection, hashing, text overlay, and QR code injection.
4. Save the secured PDF alongside the original.
5. Generate a steganography report in JSON format.

The orchestrator processes either a single specified project or all discovered projects sequentially.

7.4 Tamper Detection

Verification is performed by comparing the stored SHA-256 hash against a freshly computed hash of the distributed PDF. Any modification—even a single bit flip—produces a hash mismatch. The alpha-channel overlay provides a secondary, visual verification channel that does not require access to the original hash.

7.5 Limitations

- Alpha-channel overlays are stripped by some PDF optimization tools (e.g., `qpdf --optimize`).
- QR codes are visible and may be removed by a determined attacker.
- The system does not provide non-repudiation in the cryptographic sense—it does not use digital signatures with private keys. Future versions may integrate GPG or X.509 signing.
- The provenance model is pipeline-centric rather than fully PROV-compliant. The path from `template/`'s current metadata-based provenance to full W3C PROV-compliant traces involves four steps: (1) *entity identification*—assigning stable identifiers (URIs or SWHIDs) to each pipeline input (manuscript files, data, config); (2) *activity logging*—recording each pipeline stage as a PROV Activity with start/end timestamps (already encoded in the watermark overlay); (3) *agent attribution*—binding each Activity to the pipeline version and Git commit hash (already encoded in the overlay and PDF metadata); (4) *PROV-O serialization*—emitting the provenance graph as OWL-RDF (PROV-O) or text (PROV-N) alongside the PDF. Steps (2) and (3) are already implemented; steps (1) and (4) are the primary remaining gaps. A future `infrastructure.provenance` module would close both gaps automatically.

7.6 Relationship to Software Supply Chain Integrity

The steganographic provenance layer operates at the *document level*—it certifies the integrity of a specific PDF artifact. A complementary concern is *build-level* provenance: certifying that the pipeline itself was executed with verified source code and dependencies. Frameworks such as in-toto [Torres-Arias et al., 2019] and SLSA (Supply-chain Levels for Software Artifacts) address this concern by defining attestation chains from source commit through build steps to final artifact. The NTIA's minimum elements for a Software Bill of Materials (SBOM) [National Telecommunications and Information Administration, 2021] further standardize the enumeration of software components and dependencies—essential for establishing the provenance lineage of build environments. SLSA defines four graduated levels of build integrity:

SLSA Level	Requirement	`template/` Status
1	Provenance document exists	✓ SHA-256 manifest + steganographic metadata
2	Version-controlled build scripts	✓ All scripts in git
3	Isolated build environment	~ Docker support exists but not enforced in CI
4	Hermetic, reproducible builds	✗ – future work

Future versions of `template/` may generate SLSA-compatible provenance attestations alongside the steganographic watermarks, creating a two-layer provenance model: in-toto attests that the build pipeline was executed with the claimed source code, while the steganographic layer attests that the PDF was produced by that pipeline at a specific time.

7.7 Relationship to FAIR and Formal Provenance Standards

The steganographic layer supports the FAIR for Research Software (FAIR4RS) principles [Barker et al., 2022] at the artifact level. PDFs carry embedded metadata (Findability) in standardized XMP format (Interoperability). The SHA-256 hash manifest enables persistent integrity verification (a prerequisite for Reusability). The Documentation Duality standard ensures that the software producing the artifact is inspectable and well-documented (satisfying FAIRsoft [Garijo et al., 2024] metadata and documentation indicators). Full PROV-compliant provenance traces—capturing the derivation chain from source data through analysis scripts to rendered PDF—are a natural extension and a primary target for future development.

Software Heritage [Di Cosmo et al., 2020] complements this picture at the source-code level: by archiving the `template/` repository and assigning a reproducible SWHID (Software Hash Identifier) to each commit, Software Heritage makes the pipeline itself—not just its output—a citable, persistent digital artifact. A published SWHID alongside the PDF DOI creates a complete, two-artifact citation record: the paper's content is versioned via DOI; the code that generated it is versioned via SWHID. This combination satisfies the Katz et al. [Katz et al., 2021] software citation principles' requirement that software used in research be independently citable and permanently accessible.

8 Appendices

8.1 Appendix: Pipeline Stage Reference

Table 1: Pipeline stage reference showing each stage's script, inputs, outputs, and failure handling.

Stage	Script	Input	Output	Failure Mode
00	`00_setup_environment.py`	System environment	Validated env, directories	Hard fail
01	`01_run_tests.py`	`tests/`, `projects/*/tests/`	Coverage JSON, test reports	Configurable
02	`02_run_analysis.py`	`projects/*/scripts/*.py`	Figures, data files	Hard fail
03	`03_render_pdf.py`	`manuscript/*.md`, `config.yaml`	PDF in `output/`	Hard fail
04	`04_validate_output.py`	`output/` contents	Validation report	Warning
05	`05_copy_outputs.py`	`output/` artifacts	Organized copies	Soft fail
06	`06_llm_review.py`	Rendered manuscript	Executive summary, reviews	Skippable
07	`07_generate_executive_report.py`	All stage outputs	JSON + Markdown report	Soft fail

8.2 Appendix: Configuration Reference

Table 2: Configuration schema for `config.yaml`, showing all supported fields and their structure.

```yaml
paper:
  title: "Paper Title"
  subtitle: "Optional Subtitle"
  version: "1.0"
  date: "2026-03-19"

authors:
  - name: "Author Name"
    orcid: "0000-0000-0000-0000"
    email: "author@example.com"
    affiliation: "Institution"
    corresponding: true

publication:
  doi: "10.5281/zenodo.XXXXXX"
  journal: "Target Journal"
  volume: "1"
  pages: "1-10"
  year: "2026"

keywords:
  - "keyword1"
  - "keyword2"

metadata:
  license: "Apache License 2.0"
  language: "en"

llm:
  reviews:
    enabled: true
    types: [executive_summary, quality_review]
  translations:
    enabled: false

testing:
  max_test_failures: 0
  max_infra_test_failures: 3
  max_project_test_failures: 0
```

8.3 Appendix: Repository Directory Structure

```
template/
  infrastructure/           # Layer 1: Shared services (~150 .py files)
     core/                 # 28 files - logging, config, exceptions
     documentation/        # 6 files - figure management, glossary
     llm/                  # 30 files - Ollama integration, literature
     project/              # 2 files - project discovery
     publishing/           # 9 files - citation generation, Zenodo
     rendering/            # 12 files - Pandoc + XeLaTeX + reports
     reporting/            # 14 files - coverage parsing, reports
     scientific/           # 6 files - stability, benchmarking
     steganography/        # 8 files - watermarking, hashing
     validation/           # 22 files - PDF + Markdown validation
     config/               # Configuration
     docker/               # Containerization
  scripts/                  # Pipeline orchestration
     00_setup_environment.py
     01_run_tests.py
     02_run_analysis.py
     03_render_pdf.py
     04_validate_output.py
     05_copy_outputs.py
     06_llm_review.py
     07_generate_executive_report.py
     execute_pipeline.py
  projects/                 # Layer 2: Research workspaces
     code_project/         # Gradient descent exemplar
     act_inf_metaanalysis/ # Meta-analysis pipeline
     biology_textbook/     # Domain content exemplar (when populated)
  projects_in_progress/     # Work-in-progress (template, etc.)
  tests/                    # Infrastructure test suite (160+ files, ~3,050 tests)
  docs/                     # Repository documentation (90+ files, 12 subdirectories)
  run.sh                    # Interactive TUI orchestrator
  secure_run.sh             # Steganographic pipeline wrapper
  AGENTS.md                 # System-level AI agent documentation
  CLAUDE.md                 # Global AI coding assistant instructions
  README.md                 # Human-readable project overview
  pyproject.toml            # Root project configuration (uv)
```

8.4 Appendix: Exemplar Project Summary

Table 3: Summary of the three exemplar projects demonstrating the template's scalability across heterogeneous research domains.

Project	Status	Domain	src/ Modules	Tests	Coverage	Pages
`code_project`	Active	Numerical optimization	`optimizer.py`	39	90%+	~20
`act_inf_metaanalysis`	Active	Active inference meta-analysis	Multiple modules	505	90%+	~80
`template`	In-progress	Meta-architecture	`introspection.py`	65	90%+	~30

Active projects reside in `projects/` and are discovered automatically by the pipeline. In-progress projects reside in `projects_in_progress/` and are excluded from automated multi-project runs until promoted.

8.5 Appendix: Documentation Inventory

The repository maintains documentation at three levels:

Table 4: Documentation inventory across the four-layer documentation architecture, from repository-wide system files to per-module skill descriptors.

Level	Files	Purpose
Repository root	`AGENTS.md`, `CLAUDE.md`, `README.md`, `RUN_GUIDE.md`	Global navigation and AI agent context
`docs/` directory	90+ files across 12 subdirectories	User guides, API reference, troubleshooting
Per-directory	`AGENTS.md` + `README.md` at every directory	Documentation Duality standard
Per-module (Tier 3)	`SKILL.md` at every infrastructure module	Machine-parseable MCP-aligned skill descriptor
Infrastructure-level (PAI)	`PAI.md` at `infrastructure/` directory	Personal AI Infrastructure integration contract

The `docs/` subdirectories cover: `core/` (essential docs), `guides/` (skill levels 1–12), `architecture/` (system design), `usage/` (content authoring), `operational/` (build, config, logging, troubleshooting), `reference/` (API, FAQ, glossary), `modules/` (12 infrastructure modules), `development/` (contributing, testing), `best-practices/` (version control, migration), `prompts/` (9 AI prompt templates), `security/` (steganography, hashing), and `audit/` (review reports).

8.6 Appendix: Comparative Tool Matrix

Symbol key (applies to all cells): **Y** = full native support · ~ = partial or plugin-based · **N** = absent. See also Figure 4 for a colour-coded heatmap rendering of this table.

Table 5: Comparative feature matrix (14 capabilities × 10 tools). Y = full native support, ~ = partial or plugin-based, N = absent.

Capability	`template/`	Snakemake 9	Nextflow 25	CWL 1.2	Quarto 1	Jupyter Book 2	R Mark-down	DVC 3	Overleaf (2025)	OpenAI Prism
Pipeline orchestration	Y	Y	Y	Y	~	N	N	Y	N	N
Manuscript rendering	Y	N	N	N	Y	Y	Y	N	Y	Y
Testing enforcement	Y	N	N	N	N	N	N	N	N	N
Coverage thresholds	Y	N	N	N	N	N	N	N	N	N
Cryptographic provenance	Y	N	~[1]	N	N	N	N	~[2]	N	N
Steganographic watermarking	Y	N	N	N	N	N	N	N	N	N
Multi-project management	Y	N	N	N	N	N	N	N	~	~
AI-agent documentation	Y	N	N	N	N	N	N	N	~	~
Agentic skill protocol (SKILL.md / MCP)	Y	N	N	N	N	N	N	N	N	N
Interactive TUI	Y	N	N	N	N	N	N	N	N	N
Zero-mock policy	Y	N	N	N	N	N	N	N	N	N
Container support	N	Y	Y	Y	N	N	N	N	N	N
Distributed execution	N	Y	Y	Y	N	N	N	~[3]	N	N
Multi-language (R/Julia)	N	Y	N	Y	Y	Y	Y	Y	N	N

[1] Nextflow 25.04.0 introduced data-lineage provenance tracking (build-level, not document-level). [2] DVC provides content-addressed versioning for data artifacts via its object store. [3] DVC integrates with remote

storage (S3, GCS, Azure) but does not natively orchestrate distributed compute. Overleaf and OpenAI Prism are collaborative cloud LaTeX/AI writing environments; their AI features (GPT-5.2 for Prism, Overleaf Labs AI for Overleaf) are partial/early-stage as of 2025–2026.

References

J. J. Allaire, Charles Teague, Carlos Scheidegger, Yihui Xie, and Christophe Dervieux. Quarto, 2024. URL https://github.com/quarto-dev/quarto-cli.

Peter Amstutz, Michael R. Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heber, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, Matt Scales, Stian Soiland-Reyes, and Luka Stojanovic. Common workflow language, v1.0. *Specification, Common Workflow Language working group*, 2016. doi: 10.6084/m9.figshare.3115156.v2.

Anthropic. Introducing the model context protocol. Anthropic Engineering Blog, November 2024. URL https://www.anthropic.com/news/model-context-protocol. Open-source protocol for secure, two-way communication between AI models and external tools/data sources. Donated to the Linux Foundation, December 2025.

Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604):452–454, 2016. doi: 10.1038/533452a.

Lorena A. Barba. Terminologies for reproducible research. *arXiv preprint arXiv:1802.03311*, 2018. doi: 10.48550/arXiv.1802.03311.

Michelle Barker, Neil P. Chue Hong, Daniel S. Katz, Anna-Lena Lamprecht, Carlos Martinez-Ortiz, Fotis Psomopoulos, Jennifer Harrow, Leyla Jael Castro, Morane Gruenpeter, Paula Andrea Martinez, and Tom Honeyman. Introducing the FAIR principles for research software. *Scientific Data*, 9(1):622, 2022. doi: 10.1038/s41597-022-01710-x.

Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015. doi: 10.1145/2723872.2723882.

Jeremy Cohen, Daniel S. Katz, Michelle Barker, Neil P. Chue Hong, Robert Haines, and Caroline Jay. The four pillars of research software engineering. *IEEE Software*, 38(1):97–105, 2021. doi: 10.1109/MS.2020.2973362.

Ingemar J. Cox, Joe Kilian, F. Thomson Leighton, and Talal Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE Transactions on Image Processing*, 6(12):1673–1687, 1997. doi: 10.1109/83.650120.

Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. Referencing source code artifacts: A separate concern in software citation. *Computing in Science & Engineering*, 22(2):33–43, 2020. doi: 10.1109/MCSE.2019.2963148.

Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35(4):316–319, 2017. doi: 10.1038/nbt.3820.

Leonard P. Freedman, Iain M. Cockburn, and Timothy S. Simcoe. The economics of reproducibility in preclinical research. *PLOS Biology*, 13(6):e1002165, 2015. doi: 10.1371/journal.pbio.1002165.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

Daniel Garijo, Luis García, Matthias Barker, Rubén Chaves, Oscar Corcho, et al. FAIRsoft—a practical implementation of FAIR principles for research software. *Bioinformatics*, 40(8):btae464, 2024. doi: 10.1093/bioinformatics/btae464.

Robert Gentleman and Duncan Temple Lang. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, 16(1):1–23, 2007. doi: 10.1198/106186007X178663.

Carole Goble, Sarah Cohen-Boulakia, Stian Soiland-Reyes, Daniel Garijo, Yolanda Gil, Michael R. Crusoe, Kristian Peters, and Daniel Schober. FAIR computational workflows. *Data Intelligence*, 2(1–2):108–121, 2020. doi: 10.1162/dint_a_00033.

Google. Agent2agent (A2A) protocol specification. Open Protocol Specification, 2025. URL https://a2a-protocol.org/latest/specification/. Open protocol for AI agent interoperability built on HTTP, JSON-RPC, and SSE; announced at Google Cloud Next '25, April 2025.

Morane Gruenpeter, Daniel S. Katz, Anna-Lena Lamprecht, Tom Honeyman, Daniel Garijo, Alexander Struck, Anna Niehues, Paula Andrea Martinez, Leyla Jael Castro, Tovo Rabemanantsoa, Neil P. Chue Hong, Carlos Martinez-Ortiz, Laurents Sesink, Matthias Liffers, et al. Defining research software: A controversial discussion. In *SSRN Preprint*, 2021. doi: 10.2139/ssrn.3884311.

Tom Honeyman, Michelle Barker, Daniel S. Katz, and Neil P. Chue Hong. The FAIR principles for research software: Three years on. Research Data Alliance Virtual Plenary 24, 2024. URL https://www.rd-alliance.org/wp-content/uploads/2025/04/RDA_VP24_FAIR4RS_Three_Years_On_Reformatted.pdf. Two-year retrospective review recommending principle amendments including explicit reproducibility requirement and clarification of domain-relevant standards.

Iterative, Inc. DVC: Data version control. https://dvc.org, 2024. Open-source tool for ML experiment management, data versioning, and pipeline orchestration.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. URL https://arxiv.org/abs/2310.06770.

Daniel S. Katz, Neil P. Chue Hong, Tim Clark, Mercè Crosas, Bohdan B. Khomtchouk, John E. Kratz, David Leinweber, Kyle Niemeyer, Arfon Smith, and Patrick Vandewalle. Recognizing the value of software: A software citation guide. *F1000Research*, 9:1257, 2021. doi: 10.12688/f1000research.26932.2.

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

Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984. doi: 10.1093/comjnl/27.2.97.

Johannes Köster and Sven Rahmann. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012. doi: 10.1093/bioinformatics/bts480.

Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 2nd edition, 1994. ISBN 978-0201529838.

Anna-Lena Lamprecht, Leyla Garcia, Mateusz Kuzak, Carlos Martinez, Ricardo Arcila, Eva Martin Del Pico, Victoria Dominguez Del Angel, Stephanie van de Sandt, Jon Ison, Paula Andrea Martinez, Peter McQuilton, Alfonso Valencia, Jennifer Harrow, Fotis Psomopoulos, Josep Ll. Gelpi, Neil Chue Hong, Carole Goble, and Salvador Capella-Gutierrez. Towards FAIR principles for research software. *Data Science*, 3(1):37–59, 2020. doi: 10.3233/DS-190026.

Sam Lau and Philip J. Guo. The design space of LLM-based AI coding assistants: An analysis of 90 systems. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2025. URL https://pg.ucsd.edu/publications/ai-coding-assistants-design-space_VLHCC-2025.pdf.

Chris Lu, Jeff Clune, Jakob Foerster, Brian Lim, Robert Tjarko Lange, Yutian Tang, Shengran Mao, et al. The AI scientist: Towards fully automated open-ended scientific discovery. arXiv preprint arXiv:2408.06292, 2024. URL https://arxiv.org/abs/2408.06292.

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 978-0132350884.

Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007. ISBN 978-0131495050.

Luc Moreau and Paolo Missier. PROV-DM: The PROV data model. W3c recommendation, World Wide Web Consortium, 2013. URL https://www.w3.org/TR/prov-dm/.

Laurenz Mädje and Martin Haug. Typst: A new markup-based typesetting system, 2023. URL https://typst.app. Programmable typesetting system with incremental compilation; open-sourced 2023.

National Telecommunications and Information Administration. The minimum elements for a software bill of materials (SBOM). Technical report, U.S. Department of Commerce, 2021. URL https://www.ntia.gov/sites/default/files/publications/sbom_minimum_elements_report_0.pdf. Defines minimum SBOM fields: supplier, component name, version, unique identifier, dependency relationship, author, timestamp.

Brian A. Nosek, Charles R. Ebersole, Alexander C. DeHaven, and David T. Mellor. The preregistration revolution. *Proceedings of the National Academy of Sciences*, 115(11):2600–2606, 2018. doi: 10.1073/pnas.1708274114.

Daniel Nüst, Markus Konkol, Edzer Pebesma, Christian Kray, Marc Schutzeichel, Holger Przibytzin, and Jörg Lorenz. Opening the publication process with executable research compendia. *D-Lib Magazine*, 23 (1/2), 2017. doi: 10.1045/january2017-nuest.

Open Source Security Foundation. SLSA: Supply-chain levels for software artifacts, v1.0. https://slsa.dev/spec/v1.0/, 2023. Accessed: 2026-03-19.

OpenAI. Prism: AI-assisted research writing environment. https://openai.com/prism, 2026. Launched 2026-01-27 on GPT-5.2; provides context-aware manuscript assistance across notes, drafts, equations, and citations.

Overleaf (Digital Science). Overleaf: Collaborative cloud LaTeX editor. https://www.overleaf.com, 2025. Real-time collaborative LaTeX editor used by 15M+ researchers across 4,000+ institutions.

Roger D. Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011. doi: 10.1126/science.1213847.

Stephen R. Piccolo and Michael B. Frampton. Tools and techniques for computational reproducibility. *GigaScience*, 5(1):30, 2016. doi: 10.1186/s13742-016-0135-4.

João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. A large-scale study about quality and reproducibility of Jupyter notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Software Maintenance and Evolution (ICSME)*, pages 507–517, 2019. doi: 10.1109/ICSME.2019.00080.

Karthik Ram. Git can facilitate greater reproducibility and increased transparency in science. *Source Code for Biology and Medicine*, 8(1):7, 2013. doi: 10.1186/1751-0473-8-7.

Research Software Alliance. Developing actionable guidelines for FAIR research software. ReSA Task Force, 2024. URL https://zenodo.org/records/18877929. Task Force launched December 2024 analyzing 17 FAIR4RS principles into 6 actionable categories; builds on FAIR-BioRS methodology.

Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten simple rules for reproducible computational research. *PLOS Computational Biology*, 9(10):e1003285, 2013. doi: 10.1371/journal.pcbi.1003285.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. arXiv preprint arXiv:2302.04761, 2023. URL https://arxiv.org/abs/2302.04761.

Eric Schulte, Dan Davison, Thomas Dye, and Carsten Dominik. A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46(3):1–24, 2012. doi: 10.18637/jss.v046.i03.

Joseph P. Simmons, Leif D. Nelson, and Uri Simonsohn. False-positive psychology: Undisclosed flexibility in data collection and analysis allows presenting anything as significant. *Psychological Science*, 22(11):1359–1366, 2011. doi: 10.1177/0956797611417632.

Victoria Stodden, Marcia McNutt, David H. Bailey, Ewa Deelman, Yolanda Gil, Brooks Hanson, Michael A. Heroux, John P. A. Ioannidis, and Michela Taufer. Enhancing reproducibility for computational methods. *Science*, 354(6317):1240–1241, 2016. doi: 10.1126/science.aah6168.

Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium*, pages 1393–1410, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. arXiv preprint arXiv:2305.16291, 2023. URL https://arxiv.org/abs/2305.16291.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024a. doi: 10.1007/s11704-024-40231-1.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yunzhu Song, Bowen Li, Jasmine Li, Tao Zhang, Chuyi Ma, Ruohan Yu, Wangchunshu Yin, Karen Livescu, and Graham Neubig. OpenDevin: An open platform for AI software developers as generalist agents. arXiv preprint arXiv:2407.16741; published as OpenHands at ICLR 2025, 2024b. URL https://arxiv.org/abs/2407.16741.

Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, et al. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, 3:160018, 2016. doi: 10.1038/sdata.2016.18.

Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. Good enough practices in scientific computing. *PLOS Computational Biology*, 13(6):e1005510, 2017. doi: 10.1371/journal.pcbi.1005510.

Yihui Xie, J. J. Allaire, and Garrett Grolemund. *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, 2018. ISBN 978-1138359338. doi: 10.1201/9781138359444.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the Eleventh International Conference on Learning Representations (ICLR 2023)*, 2023. URL https://arxiv.org/abs/2210.03629.