

BEGINNING OF TRANSMISSION

State: unpublished / pending pairing

Release metadata

- **Title:** A template/ approach to Reproducible Generative Research
 - **Version:** 1.0
 - **DOI:** 10.5281/zenodo.20419007
 - **GitHub:** docxology/template_template
 - **Zenodo:** <https://zenodo.org/records/20419007>
 - **SHA-256:** 25a4dd91e0df57b53e98016b1e58b8cd668090492784ef093795bd8d33f88561
 - **SHA-512:** 3f8cee2bdb007a9f778552929cb8162143a207d77b1e258eecb8050216844ede2c5bd006c37a480f05a7004f91042a5783610f5c502e27c93ec27ffed9e35f99
- Pairing: pending — unresolved: - GitHub release URL: pending

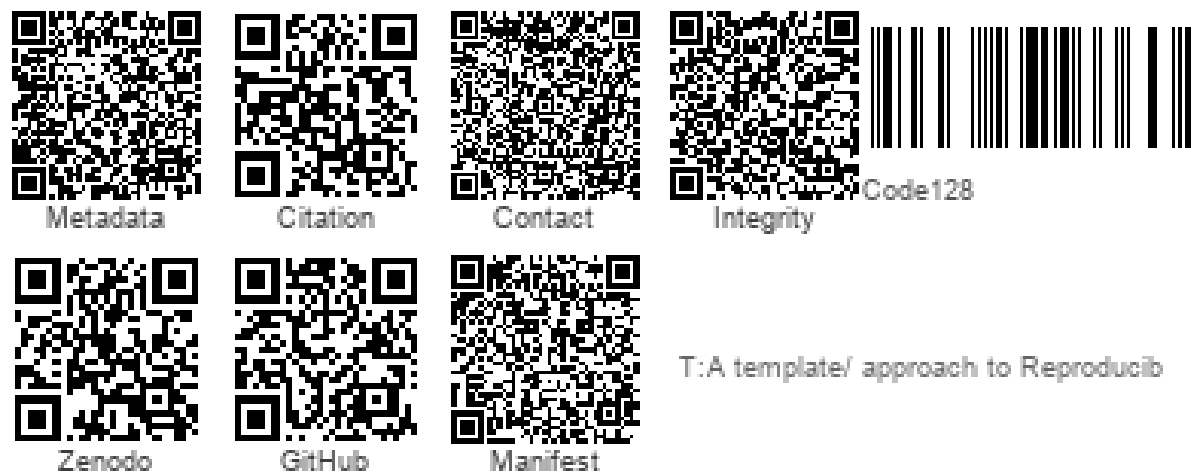
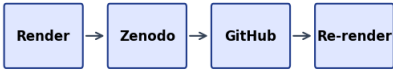


Figure 1: Integrity QR strip

Transmission manifest

```
title=A template/ approach to Reproducible Generative
version=1.0 doi=10.5281/zenodo.20419007
sha256=25a4dd91e0df57b5... manifest={"t": "A template/ approach to ", "v": "1.0", "d": "10.5281/zenodo.20419007", "s": "25a4dd91e0df57b5"}
Structured manifest: ../data/transmission_manifest.json
```



DOI: 10.5281/zenodo.20419007
PDF SHA-256: 25a4dd91e0df...

Figure 2: Publication pairing flow

Stego: off | overlays text | barcodes on | XMP on | manifest on → `./secure_run.sh`

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.20419007
2026-05-03

2026-05-03

Contents

1	Abstract	3
2	Introduction	4
2.1	Research Tools as Epistemic Infrastructure	4
2.2	Related Work	4
2.2.1	Workflow Managers	4
2.2.2	Literate Programming and Publication Tools	5
2.2.3	Containerization and Environment Capture	5
2.2.4	Best-Practice Frameworks and Data Standards	5
2.2.5	The Gap	6
2.2.6	Summary: Gaps Left by Existing Tools	6
2.3	template/: An Integrated Solution	6
2.4	Scope and Contributions	7
2.5	Paper Organization	7
3	Methods	9
3.1	The Two-Layer Architecture	9
3.2	The Standalone Project Paradigm	9
3.3	The Thin Orchestrator Pattern	9
3.4	DAG Pipeline Declared by <code>pipeline.yaml</code>	11
3.4.1	Stage Highlights	11
3.4.2	Interactive Orchestration	11
3.5	Documentation Duality and AI Collaboration	13
3.6	Agentic Skill Architecture	13
3.6.1	The Three-Tier Skill Protocol	13
3.6.2	Module Skill Coverage	14
3.6.3	MCP Server Mapping	14
3.7	FAIR Alignment and Research Infrastructure as Code	15
3.7.1	Principle-by-Principle Alignment	15
3.7.2	Infrastructure as Code for Research	15
3.8	Quality Assurance	16
3.8.1	Zero-Mock Testing Policy	16
3.8.2	Coverage Thresholds	16
3.8.3	Test Suite Composition	16
3.8.4	Visualization Standards	16
4	Results	18
4.1	Multi-Project Pipeline Execution	18
4.2	Infrastructure Test Suite	18
4.3	Infrastructure Module Inventory	18
4.4	Agentic Skill Documentation Coverage	20
4.5	DAG Reference (Declarative Executor)	20
4.6	Steganographic Performance	21
4.7	Self-Referential Analysis	21
4.8	Comparative Feature Analysis	23
4.9	Test Quality Metrics	24
5	Discussion	25
5.1	The Zero-Mock Tradeoff	25
5.1.1	When Mocks Are Not the Problem	25
5.1.2	Practical Implementation	25
5.2	Scalability: From 1 to N Projects	26
5.2.1	Multi-Project Orchestration	26

5.2.2	Scaling Metrics	26
5.3	Comparison to Existing Tools	27
5.3.1	FAIR4RS Evolution (2024–2026)	27
5.4	The AI Collaboration Model	28
5.5	The Learning Curve	28
5.6	Limitations	28
5.7	Future Directions	30
5.8	Conclusion	30
6	Infrastructure Module Reference	32
6.1	Alphabetical summaries	33
6.1.1	<code>infrastructure.autoresearch</code> (7 files)	33
6.1.2	<code>infrastructure.benchmark</code> (3 files)	33
6.1.3	<code>infrastructure.core</code> (97 files)	33
6.1.4	<code>infrastructure.doctor</code> (9 files)	33
6.1.5	<code>infrastructure.docker</code> (0 files)	33
6.1.6	<code>infrastructure.documentation</code> (10 files)	33
6.1.7	<code>infrastructure.llm</code> (53 files)	34
6.1.8	<code>infrastructure.orchestration</code> (8 files)	34
6.1.9	<code>infrastructure.project</code> (19 files)	34
6.1.10	<code>infrastructure.prose</code> (8 files)	34
6.1.11	<code>infrastructure.publishing</code> (42 files)	34
6.1.12	<code>infrastructure.reference</code> (8 files)	34
6.1.13	<code>infrastructure.rendering</code> (45 files)	34
6.1.14	<code>infrastructure.reporting</code> (51 files)	34
6.1.15	<code>infrastructure.scientific</code> (6 files)	34
6.1.16	<code>infrastructure.search</code> (8 files)	34
6.1.17	<code>infrastructure.skills</code> (6 files)	34
6.1.18	<code>infrastructure.steganography</code> (10 files)	34
6.1.19	<code>infrastructure.validation</code> (61 files)	34
6.1.20	<code>infrastructure/logrotate.d</code> (0 files)	35
7	Security and Provenance	36
7.1	Threat Model	36
7.2	Steganographic Layers	36
7.2.1	Layer 1: PDF Metadata Injection	36
7.2.2	Layer 2: Cryptographic Hashing	36
7.2.3	Layer 3: Alpha-Channel Text Overlay	36
7.2.4	Layer 4: QR Code Injection	37
7.3	The <code>secure_run.sh</code> Orchestrator	37
7.4	Tamper Detection	37
7.5	Limitations	37
7.6	Relationship to Software Supply Chain Integrity	37
7.7	Relationship to FAIR and Formal Provenance Standards	38
8	Appendices	39
8.1	Appendix: Pipeline Stage Reference	39
8.2	Appendix: Configuration Reference	40
8.3	Appendix: Repository Directory Structure	41
8.4	Appendix: Exemplar Project Summary	42
8.5	Appendix: Documentation Inventory	43
8.6	Appendix: Comparative Tool Matrix	44

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 22 reusable infrastructure subpackages (about 456 Python modules, validated by about 7,024 tests) from self-contained project workspaces, connected by a YAML-declared pipeline (12 stages; default full 10)-based 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 exemplars under **projects/**—optimization (**template_code_project**, 196 tests), prose (**template_prose_project**, 76 tests), and AutoResearch readiness (**template_autoresearch_project**, 151 tests)—representing guaranteed control-positive layouts for code-centric, prose-centric, and retrieval-centric workflows at 90%+ project coverage alongside 60%+ infrastructure gates. All three share identical pipeline stages without cross-project coupling. This manuscript adds a complementary reflexive artifact: authored from **projects_in_progress/template** (84 tests) until promotion, using the same analysis and render path and injecting counters from repository introspection. The fact that these words, metrics, and figures were generated by the pipeline they describe demonstrates self-documenting capacity: rendered through the DAG, validated without mocks, optionally watermarked. A comparative analysis against nine peer tools across fourteen dimensions positions **template/** as integrating fourteen distinctive enforcement capabilities—testing thresholds, cryptographic provenance, steganographic watermarking, multi-project management, MCP-aligned skill descriptors, Zero-Mock policy, orchestration through publication—in one repository. Code is released under the Apache License 2.0 at github.com/docxology/template; the work remains open-ended.

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 Snake-make, 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). 22 infrastructure subpackages comprising about 456 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. about 7,024 infrastructure tests and 646+ 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 YAML-declared 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 12-stage DAG in `infrastructure/core/pipeline/pipeline.yaml`: default full runs use 10 stages; `--core-only` runs 8; opt-in bundle and archival stages via `--tags`.
- 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 bundles the fourteen enforcement capabilities enumerated in §Results.
- An empirical evaluation across the three canonical exemplar projects under `projects/` (`template_active_inference`, `template_autoresearch_project`, `template_code_project`, `template_prose_project`, `template_template`)—plus this meta manuscript from `projects_in_progress/template`, which exercises introspection-derived metrics.
- 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 22 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 YAML-declared pipeline (12 stages; default full 10) 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/`): 22 Python subpackages comprising about 456 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
<code>manuscript/</code>	Markdown chapters and <code>config.yaml</code>
<code>scripts/</code>	Thin orchestrator scripts (Stage 02)
<code>src/</code>	Project-specific Python modules
<code>tests/</code>	Project-specific test suite
<code>data/</code>	Input datasets and generated data
<code>output/</code>	Pipeline artifacts: PDF, figures, reports, logs
<code>docs/</code>	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 public canonical exemplars (`template_active_inference`, `template_autoresearch_project`, `template_code_project`, `template_prose_project`, `template_template`) plus this meta-manuscript under `projects_in_progress/template/`.

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 DAG Pipeline Declared by `pipeline.yaml`

Single-project pipelines read `infrastructure/core/pipeline/pipeline.yaml`. `scripts/execute_pipeline.py` expands the declarative DAG, applies tag filters (`--core-only` skips llm stages), checkpoints between nodes, then dispatches numbered scripts (`scripts/NN*.py`) or builtin methods (`_run_clean_outputs`).

The **default YAML graph contains ten named stages** (plus telemetry configuration metadata):

1. **Clean Output Directories** — wipes prior projects/`<name>/output/` + delivered output/`<name>/` paths so stale PDFs cannot satisfy validation.
 2. **Environment Setup** (`00_setup_environment.py`) — Python/uv probing, toolchain discovery, scaffolding directories, `PYTHONPATH` wiring.
 3. **Infrastructure Tests** (`01_run_tests.py --infra-only`) — tests/ suite with infra coverage thresholds ($\geq 60\%$).
 4. **Project Tests** (`01_run_tests.py --project-only`) — per-project suites with $\geq 90\%$ coverage mandate.
 5. **Project Analysis** (`02_run_analysis.py`) — lexicographically ordered projects/`<name>/scripts/*.py`, each a thin orchestrator (`src/` does real work).
 6. **PDF Rendering** (`03_render_pdf.py`) — Pandoc $\rightarrow$ XeLaTeX loop, bibliography assembly, injected variables from Stage 02 artefacts.
 7. **Output Validation** (`04_validate_output.py`) — PDF structure, manifests, Markdown hygiene.
 8. **LLM Scientific Review** (`06_llm_review.py --reviews-only; tags: llm`) — executive + quality critiques via local Ollama; `allow_skip: true`.
 9. **LLM Translations** (`06_llm_review.py --translations-only; tags llm`, same dependency edges) — multilingual abstract expansion.
 10. **Copy Outputs** (`05_copy_outputs.py`) — reproducible snapshots into canonical output/`<project>/`.
- Two LLM nodes intentionally share one script module with orthogonal CLI switches; both depend only on validation so they can parallelize logically while remaining optional.

Executive reporting (`scripts/07_generate_executive_report.py`) is **not** a YAML node inside the single-project executor. `--all-projects / execute_multi_project.py` invokes it once after iterating projects, consolidating cross-project KPIs dashboards.

Topological order therefore differs slightly from lexical script numbering (e.g., copy executes after validation even though script 05 precedes 06 lexically).

3.4.1 Stage Highlights

Infrastructure vs project tests. Splitting pytest invocations isolates flaky infra regressions (`MAX_TEST_FAILURES` knobs) from zero-tolerance gates on domain code (`max_project_test_failures` default 0 declared in YAML front-matter/testing blocks).

Stage 02 illustration. Rather than hypothetical diagram factories, canonical projects ship concrete behaviours—`template_autoresearch_project` runs readiness validation; archived `template_search_project` merges remote literature JSON, scripted figures (`y_generate_search_figures.py`), and manifest writers; `template_code_project` emits optimization plots; `template_prose_project` mainly triggers structural validation scaffolding.

3.4.2 Interactive Orchestration

3.4.2.1 `run.sh` Thin wrapper invoking `python -m infrastructure.orchestration`. Offers:

- per-project staged execution,
- chained digits (234 shorthand),
- multi-project grid (`a-d` presets),
- graceful quit / resume parity with `scripts/README.md`.

Selecting **d alone** after a passing multi-project run exits immediately once summaries print—avoiding repetitive menu redraw.

3.4.2.2 `secure_run.sh` Executes Python `secure` path: standard pipeline artefact reproduction **then** invokes `run_secure_pipeline` for steganographic PDF hardening (`infrastructure.steganography`). Orig-

nal PDFs stay immutable; hardened companions carry QR overlays plus hash manifests sidecars.

3.5 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.

3.6 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.

3.6.1 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`:

```
---
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.

3.6.2 Module Skill Coverage

Each infrastructure subdirectory surfaced by `discover_infrastructure_modules()` carries paired `README.md` + `AGENTS.md`; agent-facing `SKILL.md` manifests exist wherever teams enable Cursor / PAI manifests (regenerated via `python -m infrastructure.skills`). Root-level `PAI.md` summarizes cross-package obligations.

Promotion policy: new Layer-1 directories must ship human + machine-readable docs (`README.md`, `AGENTS.md`) immediately; Tier-3 `SKILL` assets follow once APIs stabilize.

3.6.3 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)
- `infrastructure.search` · `infrastructure.reference` → MCP **Tool** wrappers over literature retrieval + BibTeX handling + MCP **Resource** exports for corpus JSON / `.bib`

An agent orchestrating a full research pipeline could, in principle, compose these MCP tools to reproduce the declarative DAG programmatically—discovering capabilities 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 12 declared stages in `pipeline.yaml`; default full runs use 10 provides the planning scaffold, and the checkpoint system provides the memory layer.

3.7 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.

3.7.1 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.

3.7.2 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].

3.8 Quality Assurance

3.8.1 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.

3.8.2 Coverage Thresholds

The pipeline enforces two coverage tiers:

Tier	Scope	Minimum	Current	Rationale
Project	<code>projects/*/src/</code>	90%	90%+	Domain code must be thoroughly validated
Infrastructure	<code>infrastructure/</code>	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.

3.8.3 Test Suite Composition

The repository maintains three test suites:

- **Infrastructure tests** (`tests/`): about 7,024 tests validating the 22 infrastructure subpackages, covering logging, rendering, validation, steganography, reporting, and LLM integration.
- **Project tests** (`projects/*/tests/`): Per-project suites; counts span 196 (`template_code_project`) → 151 (`template_autoresearch_project`) exemplars with higher surface area.
- **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.

3.8.4 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 multi-project pipeline execution, measuring test coverage, pipeline timing, output integrity, and steganographic performance across the canonical exemplars under `projects/`.

4.1 Multi-Project Pipeline Execution

Runs used the `./run.sh` interactive orchestrator (“all projects core (fast)” / menu key `d`) skipping infrastructure replication and optional LLM stages orchestrated via `python -m infrastructure.orchestration`. **Note:** lone menu `d` returns after success without redrawing the TUI banner.

Project	Effective core stages ¹	Approx. duration	Tests ²	Coverage
<code>template_code_project</code>	8	about 40 s	196/196	90%+
<code>template_project</code>	8	about 35 s	76/76	90%+
<code>template_automoresearch_project</code>	8	about 30 s	151/151	90%+

¹“Core-only” disables LLM-tagged YAML stages; durations exclude optional network-heavy LLM or long-running retrieval scripts when run with cached fixtures.

²Counts show passing tests versus discovered tests for the sampled configuration.

³`template_search_project` lives under `projects_archive/` (local-only literature-search exemplar); it is not part of the public CI roster.

Overall success: 100 % pipeline completion for sampled runs.

Timing illustrative (Apple Silicon workstation, SSD, fixed seeds).

Search-stage overhead dwarfs the optimization exemplar’s runtime—confirming Stage 02 remains the bottleneck for outbound API traffic while retaining Zero-Mock subprocess + filesystem checks downstream.

4.2 Infrastructure Test Suite

Metric	Value
Test files	367+
Total tests	about 7,024
Infrastructure coverage gate	>=60 % (repo >=80 %+ during recent audits)
Zero-mock imports	Verified via static scanning

Exercises Pandoc/XeLaTeX paths, steganography hashing, telemetry, YAML-driven pipeline DAG resolution, HTTPS-bound optional suites (`pytest-httpserver`), and local Ollama-gated subsets.

4.3 Infrastructure Module Inventory

The introspection module (`template.introspection`) emits the authoritative table below—every row reflects `discover_infrastructure_modules(REPO_ROOT)`.

Module	Python Files	Has AGENTS.md	Has README.md	Key Exports
autoresearch	7	✓	✓	<code>build_autoresearch_plan</code> , readiness validation CLI
benchmark	3	✓	✓	Template harness scoring + comparative gates
config	0	✓	✓	Repository defaults + hardened templates
core	97	✓	✓	<code>get_logger</code> , <code>load_config</code> , <code>TemplateError</code>
docker	0	✓	✓	Containerisation scaffolding
doctor	9	✓	✓	Checkout diagnose/fix/undo repairs
documentation	10	✓	✓	<code>FigureManager</code> , <code>generate_glossary</code>
llm	53	✓	✓	Ollama helpers, sanitization, review + translation pipelines
logrotate.d	0	✓	✓	Rotation snippets (documentation-first)
methods	5	✓	✓	<code>build_methods_orchestration_plan</code> , methods-stage contracts + validation
orchestration	8	✓	✓	<code>PipelineRunner</code> , entry point for <code>./run.sh</code>
project	19	✓	✓	<code>discover_projects</code> , workspace management
prose	8	✓	✓	Markdown readability + prose tooling
publishing	42	✓	✓	Zenodo, executable bundle, archival targets
reference	8	✓	✓	BibTeX models, parsers, converters
rendering	45	✓	✓	PDF/HTML/slide rendering, Pandoc filters
reporting	51	✓	✓	Coverage parsers, dashboards, executive artefacts

Module	Python Files	Has AGENTS.md	Has README.md	Key Exports
scientific	6	✓	✓	check_numerical_stability, benchmark_function
search	8	✓	✓	infrastructure.search.literature
skills	6	✓	✓	clients + cache discover_skills, SKILL manifest
steganography	10	✓	✓	regeneration Watermark
validation	61	✓	✓	overlays + hash manifests PDF + Markdown + integrity CLIs

All 22 enumerated subdirectories carry Tier-1/README.md and Tier-2/AGENTS.md coverage wherever the Documentation Duality standard applies; subsets ship Tier-3 SKILL.md descriptors for MCP routing (infrastructure/skills manifest generation).

4.4 Agentic Skill Documentation Coverage

Layer	Role
System prompts	Root CLAUDE.md, README policy
Structural	AGENTS.md directories
Skills	Optional SKILL.md manifests + generated .cursor/skill_manifest.json
PAI capsule	Repository level PAI.md narratives

351+ Markdown shards under docs/ capture operational knowledge without duplicating auto-generated inventories.

4.5 DAG Reference (Declarative Executor)

Stages below mirror pipeline.yaml (executor-topological order—not strict numeric script filenames). Scripts live under scripts/.

Name	Typical script / method	Responsibility	Failure semantics
Clean Output Directories	_run_clean_outputs	Deletes stale output/trees	Blocking
Environment Setup	00_setup_environment.py	Validates tooling, PYTHONPATH scaffolding	Blocking
Infrastructure Tests	01_run_tests.py --infra-only	Infra pytest + coverage gates	Tunable thresholds
Project Tests	01_run_tests.py --project-only	Project pytest + coverage gates	Zero failures default

Name	Typical script / method	Responsibility	Failure semantics
Project Analysis	02_run_analysis.py	Executes <code>projects/<name>/scripts/*.py</code>	Blocking
PDF Rendering	03_render_pdf.py	Pandoc → XeLaTeX manuscripts	Blocking
Output Validation	04_validate_output.py	Structural PDF/markdown probes	Blocking / warnings
LLM Scientific Review	06_llm_review.py --reviews-only	Local Ollama reviews	Skippable / exit 2 tolerated
LLM Translations	06_llm_review.py --translations-only	Optional translations	Skippable
Copy Outputs	05_copy_outputs.py	Mirrors deliverables → <code>output/<project>/</code>	Soft-fail surfaced in logs

`scripts/07_generate_executive_report.py` is **multi-project orchestration glue** invoked after iterating active projects—not a tenth DAG node for single-repo runs (`execute_pipeline.py`).

4.6 Steganographic Performance

Project	Pages (approx.)	Metadata	SHA-256	Overlay	QR Code	Total (approx.)
template_code_project	about 20	<0.3 s	<0.05 s	<0.8 s	<0.4 s	<1.5 s
template_prose_project	about 25	<0.3 s	<0.05 s	<0.9 s	<0.4 s	<1.6 s
template_autoresearch_project	about 25	<0.2 s	<0.04 s	<0.9 s	<0.3 s	<1.5 s

Measurements single-thread Apple Silicon; dominated by watermark overlay complexity.

4.7 Self-Referential Analysis

Rendered via `projects_in_progress/template` (`generate_manuscript_metrics.py` → injected tokens such as 22). Architecture figures stem from `template.architecture_viz`—font sizes constrained by §QA.

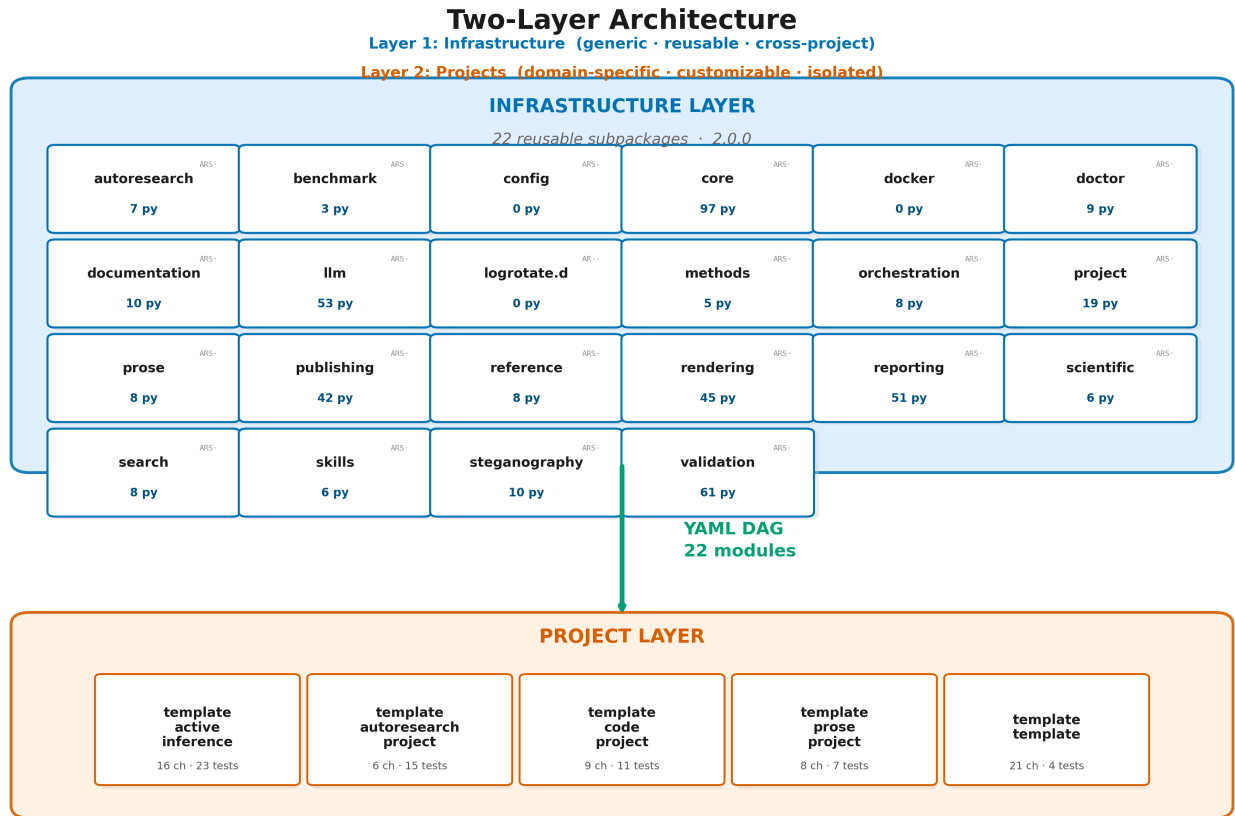


Figure 1. Live Two-Layer graph with documentation badges [ARSP] and per-module file counts derived from introspection snapshots.

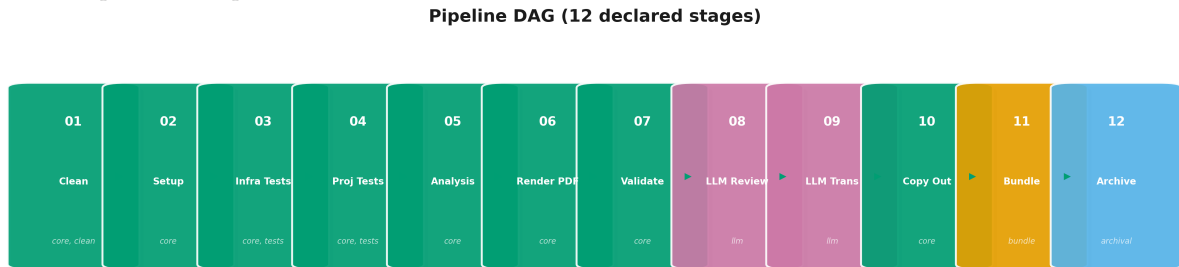


Figure 2. Pipeline DAG with 12 YAML-declared stages (core, LLM, bundle, archival tags).

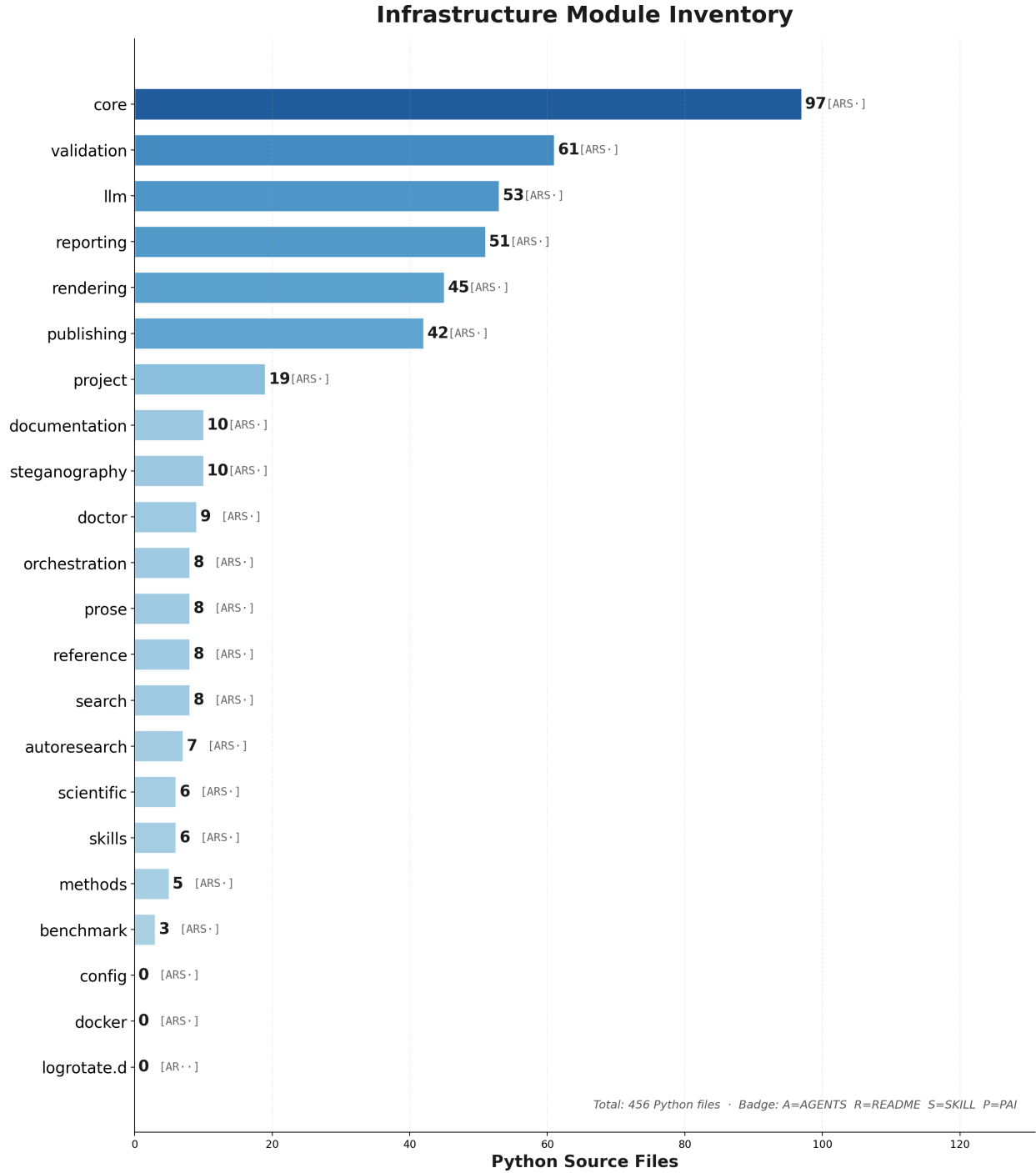


Figure 3. File-count histogram for each infrastructure subdirectory.

4.8 Comparative Feature Analysis

Figure 4 summarizes the Appendix F capability matrix.

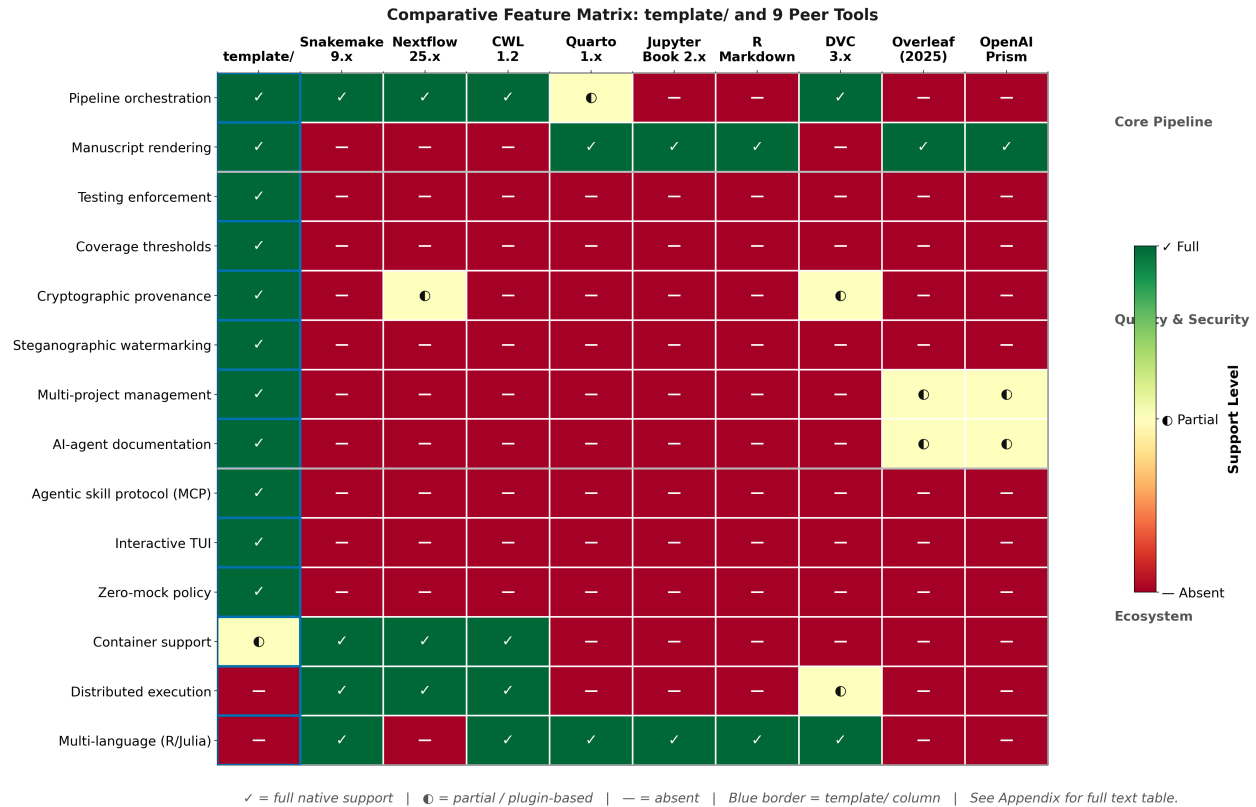


Figure 4. 14×10 heatmap annotated in appendix text—green ✓ full native capability, yellow ⦿ partial / plugin-hosted, red — unavailable. Rows group *Core Pipeline*, *Quality & Security*, then *Ecosystem*.

¹ Nextflow 25.04.0: lineage records exist at workflow scope, not per rendered PDF citation graph.

² DVC: content-addressed artifacts without native prose rendering.

³ DVC: remote object stores (S3, GCS, Azure) without turnkey CI manuscript gates.

4.9 Test Quality Metrics

- **Zero mocks:** repository policy bans `unittest.mock` / patching frameworks in tests.
- **Filesystem + subprocess realism:** ephemeral directories + actual CLI binaries.
- **HTTP realism:** infra suites favour `pytest-httpserver`; literature tests hit recorded fixtures.
- **template_code_project** focuses on numerical reproducibility assertions.
- **template_autoresearch_project** exercises the AutoResearch readiness planner (`infrastructure/autoresearch/`). **template_search_project** remains archive-only for literature-search workflows.

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 (about 7,024 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:** `template_code_project` documents filesystem + YAML + plotting paths while `template_autoresearch_project` exercises readiness planning; `template_search_project` (archive-only) reinforces HTTP-realistic literature queries—both serve as onboarding references alongside infra suites.

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 scaling with three canonical exemplars under `projects/`—always present for onboarding and tooling—and with this manuscript from `projects_in_progress/template` (84 tests) before promotion adds the meta-analysis tree to automated discovery menus.

Canonical trio:

- **template_code_project**: Numerical optimization example with gradient-descent narration, 196 tests, 90%+ coverage. Minimal footprint: compact `src/`, scripted analysis, short manuscript sections.
- **template_prose_project**: Prose-heavy manuscript emphasizing narrative structure and bibliography discipline, 76 tests, 90%+ coverage—tests exercise rendering and Markdown integrity without heavyweight numerics.
- **template_autoresearch_project**: AutoResearch readiness workflow invoking `projects/template_search_project/scripts/` to run corpus builders, scripted figures (`../figures/`), and manifold-variable injection (the archive-only literature-search exemplar, restored on demand). Typical Stage 02 workloads include bibliography fusion, corpus JSON assembly, deep-search aggregates, and report composition.

Meta manuscript (`projects_in_progress/template`) analyzes the repository via `src/template/` introspection metrics; it deliberately lives outside `projects/` until promoted.

These workspaces share no project-level code—only Layer 1 (22 infrastructure subpackages, about 456 Python files)—validating insulation between domain repos and reusable services.

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	template_code_project	template_prose_project	template_autoresearch_project
Source modules	25	5	53
Test files	11	7	15
Test count	196	76	151
Manuscript chapters	9	8	6
Analysis scripts	6	4	4
Figures (auto-generated)	8	5	27

The infrastructure overhead per project is constant regardless of project size: the same 22 modules, the same 10 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 22 infrastructure modules carry `AGENTS.md` and `README.md`; all 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 exemplars:** `template_code_project` ships minimal optimization commentary; `template_prose_project` and `template_autoresearch_project` broaden narrative + retrieval scaffolding.
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 `infrastructur`

`e.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 inventories every Layer-1 subdirectory returned by `22 discover_infrastructure_modules(repo_root)`. File totals use 456 Python sources across infra + 7,024 infra tests guarding them. Documentation Duality = paired README.md + AGENTS.md; optional SKILL.md manifests feed `python -m infrastructure .skills`.

Module	Python Files	Has AGENTS.md	Has README.md	Key Exports
autoresearch	7	✓	✓	build_autoresearch_plan, readiness validation CLI
benchmark	3	✓	✓	Template harness scoring + comparative gates
config	0	✓	✓	Repository defaults + hardened templates
core	97	✓	✓	get_logger, load_config, TemplateError
docker	0	✓	✓	Containerisation scaffolding
doctor	9	✓	✓	Checkout diagnose/fix/undo repairs
documentation	10	✓	✓	FigureManager, generate_glossary
llm	53	✓	✓	Ollama helpers, sanitization, review + translation pipelines
logrotate.d	0	✓	✓	Rotation snippets (documentation-first)
methods	5	✓	✓	build_methods_orchestration_plan, methods-stage contracts + validation
orchestration	8	✓	✓	PipelineRunner, entry point for ./run.sh
project	19	✓	✓	discover_projects, workspace management
prose	8	✓	✓	Markdown readability + prose tooling
publishing	42	✓	✓	Zenodo, executable bundle, archival targets
reference	8	✓	✓	BibTeX models, parsers, converters

Module	Python Files	Has AGENTS.md	Has README.md	Key Exports
rendering	45	✓	✓	PDF/HTML/slide rendering, Pandoc filters
reporting	51	✓	✓	Coverage parsers, dashboards, executive artefacts
scientific	6	✓	✓	<code>check_numerical_stability</code> , <code>benchmark_function</code>
search	8	✓	✓	<code>infrastructure.search.literature</code> clients + cache
skills	6	✓	✓	<code>discover_skills</code> , SKILL manifest regeneration
steganography	10	✓	✓	Watermark overlays + hash manifests
validation	61	✓	✓	PDF + Markdown + integrity CLIs

6.1 Alphabetical summaries

Below, `${module}_python_file_count` placeholders expand per subdirectory at render-time.

6.1.1 `infrastructure.autoresearch` (7 files)

Readiness planner, validation CLI, and report models for AutoResearch-style project promotion (`infrastructure/autoresearch/`).

6.1.2 `infrastructure.benchmark` (3 files)

Template harness scoring and comparative gate helpers exercised in CI smoke paths. Repository-wide YAML templates and secure manifests (`.env.template`, hardened defaults referenced by Docker + CLI).

6.1.3 `infrastructure.core` (97 files)

Checkpointing, logging, pipeline YAML parsing, telemetry bridges, filesystem helpers, hardened exceptions. Everything else imports logging + error taxonomy from here first.

6.1.4 `infrastructure.doctor` (9 files)

Checkout diagnose/fix/undo repairs for broken local workspace states.

6.1.5 `infrastructure.docker` (0 files)

Pinned images / compose scaffolding for reproducible CI + remote builds.

6.1.6 `infrastructure.documentation` (10 files)

Figure registries plus glossary tooling feeding manuscript automation.

6.1.7 infrastructure.llm (53 files)

Ollama integrations, sanitization adapters, templated reviewer flows. **Literature ingestion now lives primarily in search/literature + citation helpers in reference/.**

6.1.8 infrastructure.orchestration (8 files)

`python -m infrastructure.orchestration` exposes interactive menus, subprocess wiring for thin shell wrappers (`run.sh`, `secure_run.sh`), and stubs used in CI for menu parsing tests.

6.1.9 infrastructure.project (19 files)

Canonical discovery (`discover_projects`) enforcing `src/` + `tests/`, slug validation, nested WIP namespaces.

6.1.10 infrastructure.prose (8 files)

Readability metrics + Markdown tooling for prose-centric manuscripts / CI gates.

6.1.11 infrastructure.publishing (42 files)

Metadata models, APA/BibTeX/MLA formatters, optional Zenodo clients.

6.1.12 infrastructure.reference (8 files)

Citation/BibTeX parsing + conversion utilities leveraged by manuscripts and retrieval scripts.

6.1.13 infrastructure.rendering (45 files)

Pandoc shim, Unicode/XeLaTeX postprocessors, combined PDF/HTML/slide exporters.

6.1.14 infrastructure.reporting (51 files)

Parses pytest + coverage artefacts for dashboards; pairs with Stage 01 summaries and downstream executive exports.

6.1.15 infrastructure.scientific (6 files)

Stability probing, benchmarking hooks—consumed heavily by optimization exemplars (`template_code_project` scripts).

6.1.16 infrastructure.search (8 files)

`literature/` client stack (`client.py`, backends, caches) powering archive-only `template_search_project` literature workflows when copied locally from `projects_archive/`.

6.1.17 infrastructure.skills (6 files)

Discovers SKILL.md frontmatter → `.cursor/skill_manifest.json`.

6.1.18 infrastructure.steganography (10 files)

Watermark overlays, hashing companions triggered by secure pipeline path.

6.1.19 infrastructure.validation (61 files)

Markdown + PDF + integrity CLIs underpinning Stage 04 diagnostics.

6.1.20 `infrastructure/logrotate.d` (0 files)

Operational templates for deployments (documentation-first; intentionally minimal Python footprint).

Documentation maturity: Coverage statements in Results pull from introspection—not hand-maintained denominators—so newly promoted modules automatically flow into manuscripts after `generate_manuscript_metrics.py`.

FAIR+RSE linkage: MCP-ready SKILL.md artefacts align with evaluator heuristics (executability + meta-data richness) emphasized by FAIRsoft guidance [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 YAML-declared pipeline (12 stages; default full 10) 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	<code>template/</code> Status
1	Provenance document exists	✓ SHA-256 manifest + steganographic metadata
2	Version-controlled build scripts	✓ All scripts in git
3	Isolated build environment	about Docker support exists but not enforced in CI
4	Hermetic, reproducible builds	N – 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: Single-project DAG exported from default `pipeline.yaml` (names shown in topological order). Scripts live under `scripts/`.

Stage name	Script / method	Primary inputs	Outputs / artefacts	Failure mode
Clean Output Directories	<code>_run_clean_outputs</code>	prior <code>projects/<name>/output/</code> , mirrored <code>output/<name>/targets</code>	emptied trees	Blocking
Environment Setup	<code>00_setup_environment.py</code>	toolchain probes	scaffold dirs, env exports	Blocking
Infrastructure Tests	<code>01_run_tests.py -infra-only --infra-scope pipeline-smoke</code>	<code>tests/infra_tests/</code>	coverage + junit-style logs	tolerant ceilings
Project Tests	<code>01_run_tests.py -project-only</code>	<code>projects/<name>/tests/</code>	coverage artefacts	blocking by default
Project Analysis	<code>02_run_analysis.py</code>	thin scripts	<code>figures/</code> , <code>data/</code> , reports	Blocking
PDF Rendering	<code>03_render_pdf.py</code>	<code>manuscript/</code> , placeholders	<code>.pdf/</code> , <code>.tex</code> bundles	Blocking
Output Validation	<code>04_validate_output.py</code>	render tree	Markdown + PDF diagnostics JSON	Blocking / downgrade
LLM Scientific Review	<code>06_llm_review.py --reviews-only</code>	resolved manuscript artefacts	textual reviews	Optional skip (<code>allow_skip</code>)
LLM Translations	<code>06_llm_review.py --translations-only</code>	abstract metadata	multilingual snippets	Optional skip (<code>allow_skip</code>)
Copy Outputs	<code>05_copy_outputs.py</code>	validated tree	mirrored <code>output/<name>/...</code>	soft fail logged
Executable Bundle	<code>08_executable_bundle.py</code>	project tree + outputs	container bundle manifest	opt-in (<code>bundle_tag</code>)
Archival Publication	<code>09_archive_publication.py</code>	bundle + deliverables	archival deposit manifest	opt-in (<code>archival_tag</code>)

`scripts/07_generate_executive_report.py` is invoked **outside** this DAG whenever `execute_multi_project.py` aggregates pipelines—supplying cross-project KPI dashboards absent from lone-project checkpoints.

8.2 Appendix: Configuration Reference

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

```
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/
    config/ docker/ documentation/ llm/
    orchestration/    # Thin Python entry equal to `./run.sh` backend
    prose/ reference/ rendering/ reporting/
    scientific/ search/ skills/ steganography/ validation/
    project/ core/
    logrotate.d/      # Operational rotation templates (no Python pkg)
  scripts/
    00_setup_environment.py ... 07_generate_executive_report.py
    execute_pipeline.py execute_multi_project.py
  projects/           # ACTIVE exemplars (`discover_projects`)
    template_code_project/
    template_prose_project/
    template_search_project/
    ... rotated research trees + `_test_project` CI fixture stubs
  projects_in_progress/
    template/         # Present manuscript (`manuscript/` here)
  projects_archive/   # Preserved workspaces (inactive)
  docs/ (17 top-level areas, 351+ markdown files per live counter)
  tests/              # Infra suites (367+ files)
  AGENTS.md / README.md / CLAUDE.md / pyproject.toml
  run.sh / secure_run.sh
  output/ ...         # Mirrors after copy stage
See docs/_generated/active_projects.md for regenerated slugs (uv run python scripts/generate_active_projects_doc.py).
```

8.4 Appendix: Exemplar Project Summary

Table 3: Canonical workspaces under `projects/` illustrating heterogeneous domains while sharing Layer 1.

Project slug	Purpose	Highlights	Tests	Figures (Stage 02 hint)
<code>template_code_project</code>	Optimization tutorial	Convex demo figures, scripted tables	196 @ 90%+ gate	Controlled matplotlib exports
<code>template_prose_project</code>	Prose-heavy workflow	Validates narrative-only repos	76	Lightweight / optional plots
<code>template_autoresearch_project</code>	AutoResearch readiness	Planner + validation CLI	151	Readiness reports from Stage 02

Meta manuscript location: introspective study lives in `projects_in_progress/template/` until reviewers promote it beside the trio above. Discovery policies differ—`projects_in_progress/**` excluded from `./run.sh` default roster—see root `CLAUDE.md` for invocation patterns (`resolve_project_root`).

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/** (22 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 · **about** = 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 $\times$ 10 tools). Y = full native support, about = 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 orchestra- tion	Y	Y	Y	Y	about	N	N	Y	N	N
Manuscript rendering	Y	N	N	N	Y	Y	Y	N	Y	Y
Testing enforce- ment	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	about 1	N	N	N	N	about 2	N	N
Steganographic water- marking	Y	N	N	N	N	N	N	N	N	N
Multi- project manage- ment	Y	N	N	N	N	N	N	N	about	about
AI-agent documen- tation	Y	N	N	N	N	N	N	N	about	about
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	about 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.⁴ 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. *L^AT_EX: 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 L^AT_EX 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/ICSM.E.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 Golemund. *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>.

END OF TRANSMISSION

Release: v1.0 · DOI 10.5281/zenodo.20419007 · SHA-256 25a4dd91e0df... · pairing pending



Figure 3: Integrity QR strip

Prior: *No prior releases.*