Capturing Cell-Level Metadata in Jupyter Notebooks for Reproducibility and Reuse

Guillermo Camacho Hortelano
guillermo.camacho.hortelano@student.uva.nl

October 13, 2025, 32 pages

Academic supervisor: dr. Zhiming Zhao, z.zhao@uva.nl
Daily supervisor: Koen Greuell, koen.greuell@lifewatch.eu
Research group: MultiScale Networked Systems,
https://ivi.uva.nl/research/multiscale-networked-systems-mns.html

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Here goes your abstract. Be concise, introduce context, problem, known approaches, your solution, your findings.

Contents

Chapter 1

Introduction

For virtual researchers, most of the work happens in notebooks. They load a dataset, try an idea, tweak a parameter, rerun a few code cells, and move on. It feels productive. The problem shows up later: *where did this table come from? which cell created* `df_clean`*? why does reordering cells change the result?* Studies confirm this is common; many public notebooks are not directly runnable and need manual fixes or institutional knowledge [1–4]. Reproducibility and reuse get harder as the notebook grows.

1.1 Motivation

The day-to-day issues are simple and persistent:

- **Context disappears.** Figures and tables lose their origin. It is hard to trace a result back to the code, inputs, and parameters that produced it [5–7].
- **Dependencies are implicit.** Execution order diverges from cell order; variables live in hidden state; the same concept gets different names over time [1].
- **Sharing is brittle.** Copying a few cells rarely carries the *explanations* others need to run and adapt them.
- **FAIR is "somewhere else".** Guidance exists (e.g., Ten Simple Rules; FAIR principles), but tools are often outside Jupyter and metadata is handwritten [8, 9].

We want something small but concrete: make the relationships inside a notebook *visible* and *portable* without asking authors to annotate every step.

1.2 Problem Statement

Even small notebooks produce many digital objects: cells, variables, derived files, figures. Their relationships are only implicit in code and execution state. Without a lightweight way to capture those relationships:

- reproducibility is fragile (out-of-order execution; undeclared dependencies),
- reuse is slow (what depends on what?),
- and results management is manual (no machine-actionable context).

At a high level, NaaVRE is the surrounding environment for this work: it extends Jupyter with services aimed at turning notebook steps into reusable components and composing pipelines [4, 10]. This is valuable for making work run elsewhere. However, it does not maintain a notebook-wide record of how results are produced, nor does it bundle that context into a portable, standards-based package that others can inspect and reuse [5, 11, 12]. CellScope addresses this gap by capturing the relationships inside a notebook and packaging them for human inspection and machine action, while remaining compatible with NaaVRE as a downstream environment.

1.3 Research Questions

We scope the work to **Jupyter**. NaaVRE is treated as a use case for integration, not as the scope.

> **Main RQ:** How can we automatically capture, package, and expose *cell-level* relationships in Jupyter notebooks so that results are easier to understand, reproduce, and reuse?

We explore three sub-questions:

- **RQ1.** How can we and store, per cell, function definitions, variable definitions, and variable uses reliably (without modifying user code) [13, 14]?
- **RQ2.** How can we represent those relationships in a portable, web-native format that tools can index and humans can inspect [5, 11, 12]?
- **RQ3.** What interface helps users *see* and *navigate* these relationships directly from the notebook (interactive graph, lightweight metadata search) [15–18]?

Why not just reuse the NaaVRE analyzer? We *align* with it (static analysis via AST; no runtime instrumentation) but pursue a different deliverable: *portable, FAIR metadata and a human-facing graph.* Our Python extractor is purposely small and self-contained to (a) work outside NaaVRE, (b) emit RO-Crate + PROV for any downstream, and (c) interoperate with NaaVRE by exchanging JSON (I/O sets, edges) when integrated.

1.4 Contributions

This thesis delivers:

1. **Cell-level capture** for Python notebooks via static analysis (AST): functions, variable definitions/uses, and inter-cell edges. *(RQ1)*
2. **Storage and indexing** where authors review/confirm metadata before commit; the RO-Crate remains the source of truth and a lightweight index is maintained for findability across projects. *(RQ1)*
3. **Portable packaging** of the notebook and its explanations using RO-Crate JSON-LD with PROV relations (inputs, outputs, generation) [5, 11]. *(RQ2)*
4. **Interactive graph** where clicking a node shows a code snippet and the relevant metadata; edges explain "this depends on that" (e.g., a shared variable). *(RQ3)*

The prototype runs stand-alone and can be integrated into NaaVRE's notebook flow later. It *complements* the existing NaaVRE containerizer/analyzers by adding portable provenance packaging and graph-first exploration rather than replacing their containerization pipeline [10, 19].

1.5 Outline

Chapter 2 reviews the current state of metadata capture and storage, as well as existing solutions and their identified gaps. Chapter 3 explains the approach, requirements analysis and a conceptual overview of the platform. Chapter 4 shows the technical details of the implementation. Chapter 5 evaluates the results against relevant metrics.

Chapter 2

State of the Art

2.1 Background

We keep two threads in view. First, evidence: notebooks are popular but fragile in practice. Large-scale studies report common quality issues; hidden state, non-linear execution, missing dependencies, and frequent need for manual fixes [1–3]. Second, guidance: reproducibility frameworks emphasise environment capture, provenance, automation, and sharing [8, 9]. The gap is not a lack of principles; it is the distance between those principles and everyday notebook work.

Packaging and provenance. RO-Crate is a community specification for packaging data, code, and metadata as JSON-LD next to the actual files [11, 12]. It is web-native, aligns with schema.org, and can incorporate PROV for provenance [5] and DCAT for cataloging [20, 21]. Profiles for workflows and workflow runs allow us to describe inputs, outputs, parameters, and execution traces [22, 23].

Where notebooks meet VREs. WorkflowHub and Galaxy show how to register and run workflows; they are strong at tool/workflow granularity [17, 24]. FAIRDOM-SEEK focuses on data/model assets and sharing [18]. NaaVRE demonstrates a notebook-centric cloud VRE [10]. What is generally missing is *cell-level* metadata capture and an in-place way to inspect those relationships inside Jupyter.

NaaVRE (technical summary). NaaVRE is a notebook-centric VRE that turns notebook steps into reusable components via a *component containerizer*. Two analyzers matter for this thesis. First, a **Code Analyzer / I/O Detector** that parses cells to infer inputs/outputs and code structure; it is used at *containerization time* to make a cell deployable as a component (e.g., detect file and variable I/O, derive simple types) [10, 19]. Second, a lightweight **Type Detector** that can query the running kernel for types when needed (e.g., issuing `typeof(...)` or equivalent calls through the Jupyter messaging protocol) [4, 19]. These services are effective for pipeline composition, but they do not persist a *notebook-wide* record of cell→data relationships nor package that record as a portable research object. Our work complements NaaVRE by (i) reusing its Code Analyzer for AST/I/O detection where appropriate, and (ii) *packaging* the resulting relationships as RO-Crate with PROV so they travel with the notebook. **(TBD: Missing other components from where metadata is obtained (e.g., experiment manager)**

Digital objects in a Jupyter-based VRE. For clarity we explicitly enumerate the digital-object types used throughout this thesis (see Table 2.1). We keep a pragmatic list that reflects everyday notebook work and VRE practice [4, 10–12]. We distinguish *as-authored* objects (inside the notebook) from *packaged* objects (in the RO-Crate).

Metadata profiles per object type. We reuse RO-Crate and PROV where possible and keep profiles small and practical [5, 11, 21]. Table 2.2 maps each object to core fields and vocabularies; details of how we *obtain* each field appear later in Chapter 4. **(TBD: Missing metadata obtained from NaaVRE components, update table when done. Also maybe should go in chapter 3)**

Notebook analysis and helpers. Static and dynamic tools flag issues or recover some provenance. noWorkflow captures provenance for Python scripts without modifying code; YesWorkflow recovers work-

Type	Short definition & examples
Notebook	The `.ipynb` document as authored (cells, metadata).
Cell (code/markdown)	Executable or narrative unit; may define functions, variables, figures.
Variable / symbol	Logical data produced/used across cells (e.g., `df_clean`, `model`).
Parameter / config	Tunables that influence results (e.g., file paths, hyperparameters).
File / artifact	Concrete outputs/inputs on disk (e.g., `.csv`, `.parquet`, figures).
Dataset (curated)	A collection of files with stable identity and description (externally cited).
Figure / table	Rendered visual/tabular result derived from data and code.
Environment / kernel	Runtime image/interpreter that executes the notebook.
Component (containerized step)	A cell or group of cells packaged to run as a reusable step in a VRE [10].
Metadata record / Crate	The RO-Crate JSON-LD + sidecar files that carry human- and machine-readable context [11, 12].

Table 2.1: Digital objects considered in this thesis (10 types).

flow structure from annotations [13, 14]. Linters such as Julynter/PyNBlinT catch anti-patterns [2, 25]. These are useful, but they typically do not produce a *portable research object* with cell-level relationships and a human-navigable graph.

What a user can *do* with an RO-Crate in a Jupyter context. In practical terms, a crate lets a notebook author (or a reviewer) *carry* the story of a result alongside the code:

1. Package the notebook, per-cell scripts (`cells/cell_*.py`), and a machine-actionable description of cell → variable relations (`ro-crate-metadata.json` + GraphML).
2. Open the crate locally: double-click a self-contained HTML (`cell_graph.html`) to browse the dependency graph; hover a node to see the code snippet and metadata; jump to the exact cell file.
3. Exchange the crate: upload to a registry (e.g., SEEK item or WorkflowHub attachment) or archive (e.g., Zenodo); downstream tools can index JSON-LD, and humans can read the graph without services.
4. Reuse: search for "cells that produce `df_clean`", or "cells that consume `rainfall_data`" (locally or, later, in a triple store), and copy only the needed, well-explained parts.

This is complementary to execution platforms: it focuses on *explainability and portability* of relationships, not just re-running.

Object lifecycles (conceptual). We refer to lightweight lifecycles that reflect notebook practice; Chapter 4 explains how CellScope captures and packages the associated metadata.

- **Cell** — *author → (optional) execute → analyze (derive defs/uses, I/O) → package as Activity → reuse (copy/refactor or containerize).*
- **Variable/symbol** — *defined → used/derived → packaged as logical Dataset with provenance → referenced downstream.*
- **File/artifact / Figure/Table** — *produced → consumed/published → versioned → archived*; captured via `prov:wasGeneratedBy`/`prov:used`.
- **Notebook** — *draft → analyzed → packaged as RO-Crate → shared/archived* (registry/DOI) [11, 12].
- **Environment/kernel** — *selected → recorded* (image, packages) → *replayed* (optional).
- **Component** — *derived from cells* (containerization) → *published to VRE → invoked/composed* [10].

These lifecycles frame the evaluation criteria used later (coverage of defs/uses/edges, portability/-FAIRness of packaged objects, and usefulness of the graph/list for discovery).

Object	Entity type(s)	Key fields (examples / vocab)
Notebook	`CreativeWork`	*name*, *description*, *version*, *author*, *license*; *encodingFormat*=`application/x-ipynb+json`.
Cell	`prov:Activity`	*name* ("Cell 7"); *programmingLanguage*; *text* (snippet); links via `prov:used`/`prov:wasGeneratedBy`.
Variable / symbol	`Dataset` (logical)	*name* (symbol), optional *description*; provenance via `prov:wasGeneratedBy`/`prov:used`.
Parameter / config	`PropertyValue`	*name*, *value*; link to the cell (`prov:used`) that consumes it.
File / artifact	`File`	*name*, *contentSize*, *checksum*, *encodingFormat*; provenance as above.
Dataset (curated)	`Dataset`	*name*, *description*, *distribution* (DCAT); *license*, *identifier/DOI*.
Figure / table	`File` or `ImageObject`	*name*, *encodingFormat* (e.g., PNG/SVG, CSV), *about*/*keywords*.
Environment / kernel	`SoftwareApplication`	*name*, *version*, *softwareRequirements* (e.g., image tag, packages).
Component	`SoftwareApplication` + `prov:Activity`	*name*, *identifier/URI*, *inputs*/*outputs*; link back to notebook cells.
Metadata record / Crate	`CreativeWork`	*name*, *license*, *creator*; crate graph (`ro-crate-metadata.json`).

Table 2.2: Compact per-type profiles using RO-Crate, PROV, and DCAT.

2.2 Related Work

Method. We followed a lightweight literature study. Sources: Google Scholar, ACM Digital Library, IEEE Xplore, and project/documentation sites referenced by primary works. Core search strings included: *"RO-Crate specification 1.2"*, *"workflow registry WorkflowHub"*, *"Galaxy update 2024"*, *"Jupyter provenance"*, *"noWorkflow"*, *"YesWorkflow"*, *"ProvBook"*, and *"CF Conventions"*. We screened titles/abstracts and prioritised: (i) peer-reviewed or official specs; (ii) practical systems used by communities; (iii) materials that explicitly address packaging, provenance, or notebook/script analysis. We included recent updates where available (e.g., Galaxy 2024 and WorkflowHub 2024/2025).

Packaging/specifications. *RO-Crate* provides a web-native, JSON-LD based packaging format to co-locate files and machine-actionable metadata; version 1.2 is the current stable specification. It emphasises schema.org alignment and extensibility, and offers workflow/run profiles to describe inputs/outputs/parameters and execution traces. This is the backbone we target for portable "notebook + context" packages. Limitation with respect to our gap: RO-Crate *defines how to describe and exchange*, but does not *extract* cell-level relations from notebooks, that is our capture step [11, 12, 22, 23].

PROV-O (W3C) and *DCAT v2* (W3C) are key vocabularies we reuse: PROV for "wasGeneratedBy/used/wasDerivedFrom", DCAT for dataset/catalog semantics. Limitation: they are general; mapping cell-level semantics still requires our analysis layer. [5, 20, 21]

VRE registries and execution platforms. *WorkflowHub* serves as a registry for FAIR computational workflows (with updated lifecycle publications through 2025). It is valuable for discovery and

citation across engines, but operates above notebook cells; reproducing a notebook typically means *refactoring it into a workflow language* (e.g., CWL/Nextflow), which loses exploratory cell-level nuance [17, 26]. *Galaxy* is a mature web platform for accessible, reproducible analysis. To "reproduce a notebook" in Galaxy, one would either (i) rewrite cells as Galaxy tools/workflows, or (ii) use interactive environments; neither path captures *cell-to-variable* lineage as a portable object [24, 27]. *FAIRDOM-SEEK* excels at managing datasets, models, and SOPs with rich metadata and DOIs, but it does not execute notebooks nor infer intra-notebook dataflow; a notebook + files can be *hosted*, yet cell relations remain implicit [18]. *Describo* (RO-Crate editor) helps curate crates for arbitrary research objects, but metadata entry is manual; it does not *derive* cell relations from code [28]. *nf-core/Nextflow* communities provide high-quality, versioned pipelines; porting a Jupyter exploration into a pipeline is valuable for production, yet it requires *rewriting*, not extracting relationships from the original notebook.

Notebook/script provenance and analysis. *noWorkflow* collects provenance for Python scripts without modifying code; strong on execution-time lineage and queries. Valuable conceptually, but not notebook-cell aware out of the box. [13]

YesWorkflow exposes prospective provenance (workflow structure/dataflow) from scripts via lightweight, language-agnostic annotations in comments. It supports graph visualisation; however, it requires user annotations and targets scripts rather than live notebook cells. [29]

ProvBook captures and visualises Jupyter execution provenance (e.g., cell runs) and can export RDF. Useful for run-time histories; less focused on static data-dependence between cells as inferred from code. [30, 31]

Domain conventions and semantics. For climate notebooks, the *CF Conventions* and *CF Standard Names* underpin interoperable data/metadata practice. They matter when we lift our crates into domain search or validation. [32, 33]

Illustrative reproduction paths (and pitfalls). *WorkflowHub:* publish a workflow; to replay a notebook, you must refactor cells into a workflow description. Pitfall: loss of interactive, exploratory context; no cell-level lineage unless separately encoded. *FAIRDOM-SEEK:* upload notebook, datasets, and descriptive metadata for sharing/DOI. Pitfall: no automatic extraction of cell relations; consumers still guess "which cell produced what". *Galaxy:* wrap each step as a tool or use an interactive environment. Pitfall: tool-level provenance but not notebook cell graph; manual effort to reach runnable parity. *nf-core/Nextflow:* re-implement as a pipeline. Pitfall: great for production, but requires rewriting; loses the "as-authored" notebook structure. *Describo:* curate an RO-Crate by hand. Pitfall: manual metadata entry; no static analysis to derive the dependency graph.

What these works provide for us. In short: RO-Crate/PROV/DCAT give us the *interchange language*; WorkflowHub/Galaxy show FAIR packaging and registry integration at the workflow level; noWorkflow/YesWorkflow/ProvBook show how provenance can be captured or exposed for scripts/notebooks. The missing piece, and our contribution, is *automatic cell-level relationship capture inside Jupyter*, packaged as RO-Crate and exposed via an interactive, developer-friendly graph.

Classification (summary)

A summary of the researched work and their respective categories can be found in table 2.3.

2.3 Existing solutions

We group the most related systems and tools and compare them along axes that matter for this thesis: *granularity* (cell/notebook/workflow), *capture mode* (static, runtime, annotations), *packaging/portability* (e.g., RO-Crate), *Jupyter UX* (in-place aid vs. external), and notable *limitations*. This complements the narrative review and the classification table above.

Overview.

A head-to-head summary appears in Table 2.4.

Takeaways. Tools closest in spirit either (i) emphasise *runtime* histories (ProvBook, noWorkflow) without producing a *portable research object*, (ii) require *annotations* (YesWorkflow), or (iii) operate at *workflow/platform* scope (WorkflowHub, Galaxy, SEEK) rather than inside notebooks. None combine:

Category	Representative work
Packaging/specs	RO-Crate core/spec [11, 12], PROV-O [5], DCAT v2 [21], Describo [28]
VRE registries/execution	WorkflowHub [17, 26], Galaxy [24, 27], FAIRDOM-SEEK [18], **NaaVRE** [10], nf-core/Nextflow
Notebook/script provenance	noWorkflow [13], YesWorkflow [29], ProvBook [30, 31]
Notebook quality/linters	Empirical studies and tooling [1, 2, 25]
Domain conventions	CF Conventions [32], CF Standard Names [33]
Visualization building blocks	NetworkX [15], vis.js (Network) [16]

Table 2.3: Related work classification used in this thesis.

automatic *cell-level* relationship capture, *confirm-first* curation, and *RO-Crate + PROV* packaging with in-place Jupyter views. This gap motivates CellScope's design.

2.4 Gap Analysis

Summarising: we need *automatic*, low-friction capture of cell-level relationships inside Jupyter, packaged in a web-native way (so others and services can use it), and an interactive view that makes these relationships obvious. Existing VREs and registries operate at a higher level; existing notebook tools do not emit portable research objects with cell-level provenance. This is the specific gap we address. While NaaVRE's analyzers support containerization, they stop short of producing a persistent, notebook-wide provenance model or a portable RO-Crate package, which is precisely the gap CellScope fills [10].

Tool / System	Granularity	Capture mode	Packaging / Portability	Jupyter UX	Notes / Limitations
ProvBook [31, 34]	Notebook / cell (exec)	Runtime provenance (execution history)	RDF export possible; not RO-Crate-native	In-notebook visualisation	Focuses on execution traces; limited static data-dependence; no portable research object by default.
noWorkflow [13]	Script / cell (exec)	Runtime provenance without code mods	Provenance store; not RO-Crate-native	External UI; not Jupyter-first	Strong execution lineage; less cell-aware for live notebooks; export requires mapping.
YesWorkflow [29]	Script-level	Annotations (prospective provenance)	Graphs / docs; no RO-Crate	External visualisation	Requires user annotations; targets scripts rather than live notebook cells.
WorkflowHub [17, 26]	Workflow	Registry / metadata	FAIR workflow records; attachments	External web UI	Discovery / citation over workflows; replaying notebooks requires refactoring to a workflow language.
Galaxy [24, 27]	Tool / workflow	Execution + histories	Platform internal; not RO-Crate	Web platform	Rich pipeline provenance; notebooks reproduced via tools / IEs; no cell-level lineage as a portable object.
FAIRDOM-SEEK [18]	Dataset / model / asset	Curation (manual)	Rich metadata + DOIs	Web portal	Hosts "notebook + files", but intra-notebook relations remain implicit unless encoded separately.
Describo [28]	Any research object	Manual curation	RO-Crate editor	Desktop / web app	Helpful for curation; does not derive cell relations automatically.
NaaVRE Component Containerizer [10, 19]	Cell / component	Static I/O detection (+type hints)	Container specs; not RO-Crate-wide	Jupyter extensions environment	Effective for containerisation; currently no notebook-wide, portable cell→data provenance package.
CellScope (this work)	Cell + file + variable	Static first; optional runtime hints	RO-Crate 1.2 + PROV (portable)	Jupyter-first (graph + list)	Automatic cell-level relationships; confirm-first metadata capture; generate→then visualise; NaaVRE-friendly.

Table 2.4: Comparison of alternatives closest to our goals (cell-level understanding, portability, and Jupyter integration). CellScope row shown for context.

Chapter 3

CellScope: A Human-Centered Approach

To address the gap identified in Chapter 2, CellScope is designed around a simple idea: *make notebook internals visible and portable for people and machines.* We outline methodology, requirements, and a technology-agnostic architecture; implementation details are reserved for Chapter 4.

3.1 Methodology

We adopt a design science research (DSR) approach [35] to build and evaluate an artifact that answers our research questions. The cycle and its concrete instantiations in this thesis are:

1. **Problem framing and context.** Ground the work in observed notebook pain points (lost context, hidden state, brittle sharing) and evidence from empirical studies [1–3]. Clarify the NaaVRE setting and intended users at a conceptual level (Ch. 1).
2. **Requirement elicitation and prioritisation.** Derive candidate requirements from the gap analysis (Ch. 2), supervisor feedback, and a small user-story survey. Prioritise with a lightweight scheme that emphasises *relevance* and *value* (MoSCoW variant), producing R1-R4 (capture, packaging, views, polyglot readiness) plus non-functional goals (portability, findability, near-real-time feedback).
3. **Design principles and conceptual architecture.** Formulate principles that guide the solution: *storage-first*, *confirm-first* metadata (draft → user review → store), *generate then visualise*,... Consolidate these into the conceptual architecture and lifecycle in §3.3.
4. **Build minimal instantiations.** Realise the principles in a small, Jupyter-first artifact that (i) captures a draft of intra-notebook relationships, (ii) enables author confirmation, (iii) packages the result as a portable research object, and (iv) renders views that read from what is stored (Ch. 4). We adopt a *static-first* capture stance for safety and speed, with the option to complement it later with lightweight runtime hints where strictly necessary.
5. **Evaluation and iteration.** Specify objectives, baselines, tasks, KPIs, and analyses in advance (Ch. 5). We evaluate along *coverage* (does capture reflect the intended relations?), *comprehension/efficiency* (do people answer provenance questions faster and as accurately?), and *portability/FAIRness* (is the package machine-actionable and reusable) [5, 11, 12, 36, 37]. Results inform iterative refinements.
6. **Reflection and communication.** Document the design trade-offs (e.g., profiles vs. single schema; confirm-first vs. auto-capture), report limitations, and release artifacts to support reuse and scrutiny.

3.2 Requirement Analysis

Elicitation & prioritisation. Requirements were collected from the literature and gap analysis (Ch. 2), and validated and refined through a small user-story survey (4 respondents: 3 NaaVRE developers, 1 non-developer user; all experienced Jupyter users). Participants rated each story's *clarity*, *relevance* and *value* from 1 to 5 and provided free-text feedback.

To prioritise, we compute a *weighted score* per story:

$$\text{Score} = 0.45 \cdot \text{Relevance} + 0.45 \cdot \text{Value} + 0.10 \cdot \text{Clarity}$$

and map the score to MoSCoW categories with fixed thresholds: *MUST* (≥ 4.5), *SHOULD* (≥ 4.0 and < 4.5), *COULD* (≥ 3.0 and < 4.0), *WON'T (for now)* (< 3.0). This gives more weight to *relevance* and *value*—which better reflect user impact—while still considering *clarity*.

Must-have functionalities (weighted average ≥ 4.5)

- **Filter by type/tags/relations** (precise discovery)

Should-have functionalities ($4.0 \leq$ weighted average < 4.5)

- **Link digital objects across projects**
- **FAIR feedback** on completeness when publishing
- **Publish selected objects** to a marketplace
- **Jump** from a marketplace card to the code+graph

Could-have functionalities ($3.0 \leq$ weighted average < 4.0)

- Auto-capture metadata (no burden)
- See cell dependencies
- Explore graph
- Search for cells/data snippets to reuse

Won't-have functionalities (weighted average < 3.0)

- Advanced ontology-level reasoning beyond lightweight typing

Additional feedback to highlight:

- the need for a *non-graph view* (table/filter) alongside the graph,
- *compatibility cues* (IDs, versions) to avoid breakage,
- and *credit/citation* for reuse.

These results directly inform the requirements and prioritisation below.

Capture (R1). Per cell: functions, variable definitions, variable uses; derive edges from last definition to subsequent uses. Prefer static analysis to avoid executing user code [13].

Package (R2). Represent cells as *activities* and variables/files as *data*; add *hasInput*/*hasOutput*/*wasGeneratedBy* links; keep the crate human-readable and machine-actionable. Include basic identity/version fields to support cross-project linking and compatibility cues requested by users. [5, 11, 12].

Explore (R3). Graph view that opens from the crate; hover to see code snippet and metadata; edges carry a short rationale ("uses variable `x`"). Add a compact, *non-graph list view with filters* (type/tags/relations) to address discoverability preferences raised in the survey. [16].

Scope (R4). Python first. Multi-language support can be added by detecting kernels and common hand-offs (e.g., SoS syntax, file materialisation). Marketplace publishing and FAIR feedback are tracked as "Should-have" features contingent on the core capture/packaging path. [38].

3.3 Architecture

Purpose. This section explains *what* CellScope is conceptually and *how it is expected to behave* from a user and system point of view, without committing to specific technologies (details are in Chapter 4).

Concept at a glance

CellScope makes the internal structure of a notebook *visible* and *portable*. It does so by:

1. **Capturing and storing** relationships between code cells and the data they define/use (R1).
2. **Packaging** the notebook together with those relationships into a shareable research object that carries provenance and context (R2).
3. **Exposing** the result through two complementary views: a structural graph and a non-graph list/filter view (R3).
4. **Accommodating multiple languages** (e.g., Python and R) by recognising common hand-off patterns between cells and kernels (R4).

Actors and interactions

- **Author** works in a notebook; asks CellScope to *analyse* the current state.
- **Reviewer/Collaborator** opens the overview to understand "what depends on what," or opens a packaged artifact received from someone else.
- **Registry/Marketplace** (optional for MVP) can index packaged artifacts to support discovery and reuse.

Storage-first principle. We treat storage as first-class. Every digital object is handled in three parts: its *content* (the bytes), its *metadata* (a concise description that people and machines can understand), and its *control & links* (identity, versioning, provenance relations, and access cues). The author keeps a portable package of metadata co-located with the content; the platform maintains a separate index optimised for discovery and cross-object linking. This separation preserves portability for authors while making objects *easy to find* across projects.

Common core with per-type profiles. To balance consistency and expressiveness, we use a small *common core* (identifier, type, name, license, version, checksum, sensitivity flag) and *per-type* profiles (Notebook, Cell/Activity, File/Dataset, Figure/Table, Parameter, Workflow). The common core enables uniform handling and validation; the profiles capture what is specific to each object without overloading a single schema.

Confirm-first metadata lifecycle. CellScope follows a confirm-first flow: *(1) capture* a draft description from the notebook and its context; *(2) review/confirm* through a short form where authors can add missing information (e.g., units, licenses) and redact sensitive items; *(3) store* by writing the author-side package and updating the platform index; *(4) visualise or export from what is stored.* This prioritises correctness of stored information over presentation.

Conceptual components

Capture. Interprets each cell to identify the *things* it defines (e.g., variables/files) and *things* it uses, then infers relationships between cells. The capture is designed to be *fast, safe, and minimally intrusive* (no changes to user code), acknowledging known fragilities of notebooks (R1).

Packaging. Turns the notebook plus its relationships into a *portable bundle* with explicit provenance so others—and tools—can inspect it outside the original environment (R2).

Views. Two complementary lenses over the same relationships: (i) a graph for "structure at a glance" and (ii) a list+filter view for precise discovery by type, tags, or relation (R3).

Typical flow (happy path)

1. The author runs *Analyse*. CellScope builds a conceptual map of cells and their inputs/outputs.
2. The author explores the result: follows upstream/downstream, filters by type/tags/relations, and checks short rationales for edges.
3. The author exports a *portable package* to share with a collaborator or attach to a record/registry.
4. A collaborator opens the package locally and reaches the same views without needing the original environment.

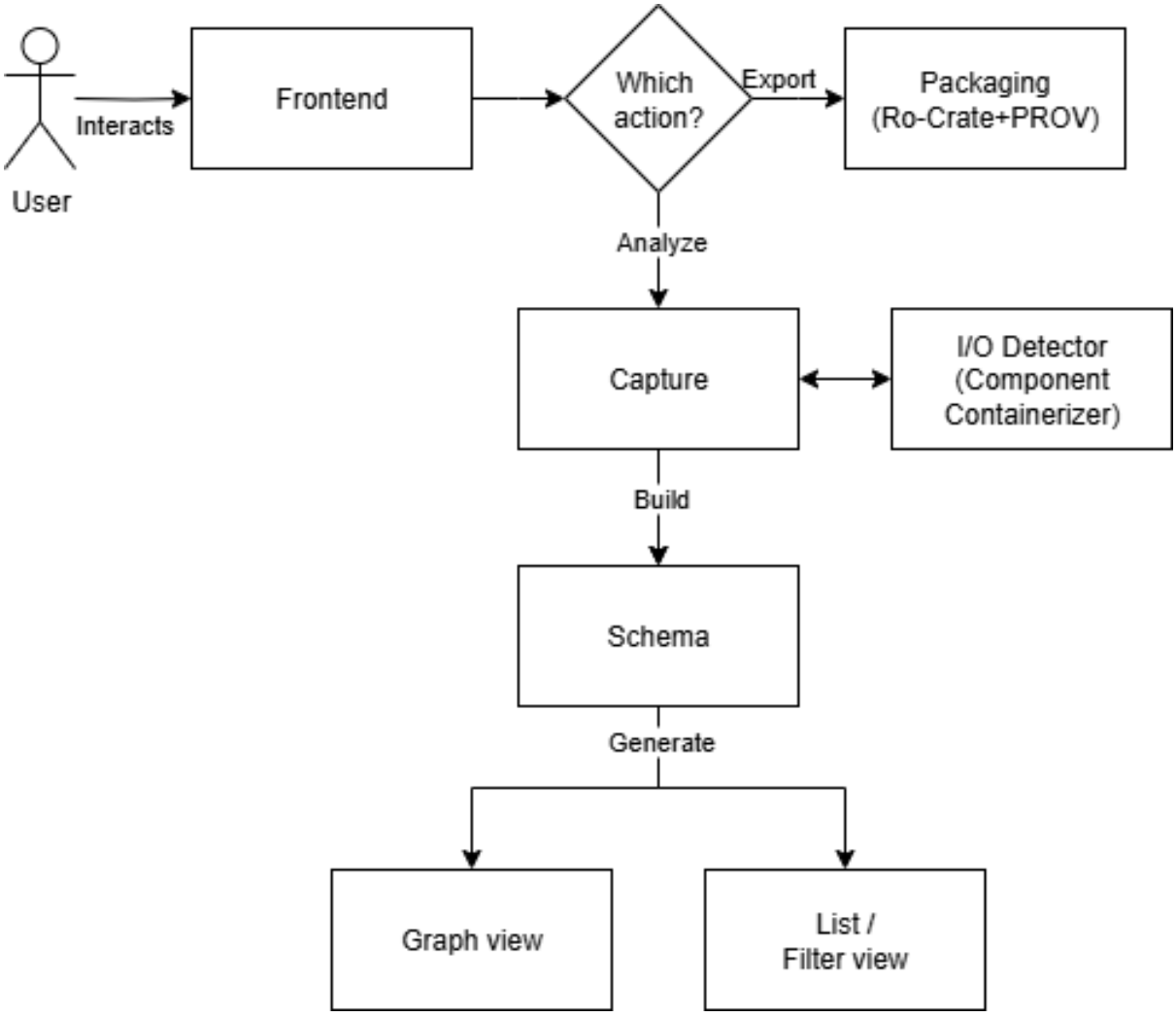

Figure 3.1: Conceptual architecture of CellScope: the author triggers *Analyze* or *Export*; capture derives relationships; packaging produces a portable package with provenance; views (graph and list/filter) render the same underlying model.

Near-real-time updates. The platform reacts to authoring events (e.g., saving or executing work) by refreshing the draft description, proposing updates for confirmation, and then applying small increments to storage and index. This keeps feedback timely without requiring the author to manage operational details.

Generate, then visualise. Human-facing views are read-only lenses over the stored relationships. We first *generate* the information and record it, then *visualise* it from that source.

Chapter 4

Implementation

We implemented CellScope as a *Jupyter-first* solution with three parts: a JupyterLab extension (user interface), a thin Jupyter Server extension (REST API), and a reusable Python library (`cellscope`) that performs analysis and packaging. A small CLI mirrors the server endpoints for development and batch use. The goal is modest: generate useful metadata with zero manual steps and present it in views people can act on. The design aligns with NaaVRE's extension-oriented environment [10, 17] and *operationalises* the conceptual principles introduced in Chapter 3 (§3.3: storage-first, confirm-first, generate then visualise).

Figure 4.1 shows the implementation-level architecture and the integration points with the Component Containerizer and kernels.

4.1 Implementation lifecycle (link to Chapter 3)

We follow the *confirm-first* flow from §3.3: **capture** a draft → **review/confirm** → **package** → **store+index** → **visualise or export**. Concretely:

1. **Draft capture** (`/cellscope/analyze`) produces a graph JSON and draft fields.
2. **Review/confirm** (UI forms) lets users add units, licenses, and redact sensitive fields.
3. **Package** (§4.6) writes an author-side bundle consistent with the draft.
4. **Store & index** (§4.7) persist the bundle and post an index delta for discovery.
5. **Visualise/export** (§4.9) render from stored metadata; exports reuse the same bundle.

This realises the "generate, then visualise" principle from §3.3.

4.2 Review and confirmation

Before committing to storage/indexing, CellScope shows per-object forms: auto-captured fields, missing/low-confidence items (e.g., license, units, sensitivity), and editable values. On *Commit*, we (a) write the crate, (b) increment `version` if needed, and (c) push an index delta.
Flow (pseudocode).

```
draft = capture(notebook)            # /cellscope/analyze
review = ui.review(draft)            # user confirms/edits (confirm-first)
crate  = write_crate(review)         # package (author-side bundle)
delta  = to_rdf_delta(crate)         # store+index (discoverability)
post("/cellscope/index", delta)
```

4.3 Capture: Static first, hybrid when needed

Design choice. We favour *static, per-cell* analysis for speed, safety, and transparency (no execution needed), acknowledging common notebook pitfalls [1, 2]. We enrich static results with *optional runtime hints* when available [13]. This balances practicality with fidelity. This implements the *Capture* concept in §3.3.

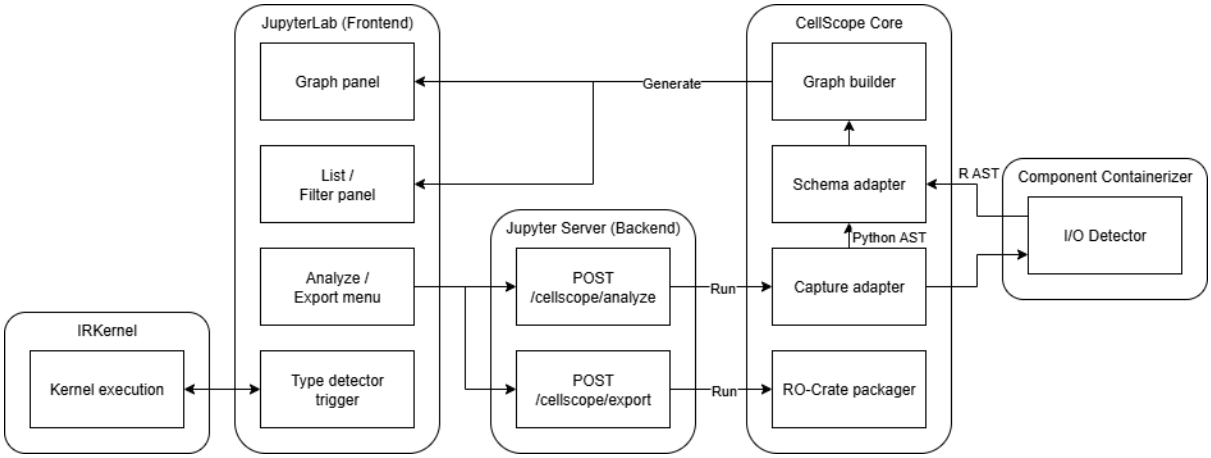

Figure 4.1: Implementation architecture: JupyterLab frontend (Graph/List panels), Jupyter Server endpoints (`/cellscope/analyze`, `/cellscope/export`, `/cellscope/index`), CellScope core (capture, schema/graph builder, RO-Crate packager), and external analyzers (Component Containerizer I/O Detector; kernels for type queries).

This capture path corresponds to the *Analyze* flow in Figure 4.1 (JupyterLab → /cellscope/analyze → capture adapters → graph builder).

Python (static)

We parse each code cell into an AST and extract:

1. **Definitions** (left-hand side of assignments, function names),
2. **Uses** (identifier reads not defined in the same cell).

Edges are drawn from the last definition of a symbol to subsequent uses (producer → consumer).

R (static, conservative)

For R cells we parse source with `parse()` and `getParseData()` to identify simple assignments and symbol uses. We deliberately avoid deep flow analysis; the goal is to capture common notebook patterns with low effort.

Runtime hints (optional)

When enabled, pre/post-execution hooks record lightweight snapshots (e.g., newly created names, simple file writes). Hints *add* edges, increase confidence, or flag mismatches; the static graph remains authoritative [13].

Cross-kernel edges

We detect *file materialisation* patterns (e.g., `write.csv`, `saveRDS`, `pandas.DataFrame.to_csv`, `open(...,'wb')`) and corresponding reads (`read.csv`, `readRDS`, `pandas.read_csv`, etc.). When a path written in one cell/language is read in another, we add `produces/consumes` edges. This enables Python↔R hand-offs without prescribing a polyglot framework; SoS-style multi-kernel workflows are also recognised conceptually [38].

Algorithm (sketch)

```
def analyze_notebook(cells, lang_of, hints=None):
    G = DiGraph(); last_def = {}
    for i, cell in enumerate(cells):
        lang = lang_of(i)              # "py" or "r"
        defs, uses, io = static_pass(cell.source, lang)
```

```
        if hints: defs, uses, io = merge_hints(defs, uses, io, hints.get(i))
        G.add_node(i, lang=lang, defs=defs, uses=uses, io=io)
        for v in defs: last_def[v] = i
    for j, data in G.nodes(data=True):
        for v in data['uses']:
            if v in last_def: G.add_edge(last_def[v], j, type='uses', via='
                symbol')
    for i, data_i in G.nodes(data=True):              # file-based hand-offs
        for path in data_i['io'].get('writes', []):
            for j, data_j in G.nodes(data=True):
                if path in data_j['io'].get('reads', []) and i != j:
                    G.add_edge(i, j, type='consumes', via='file', path=path)
    return G
```

Listing 4.1: Per-notebook capture with optional hints

Complexity and limits. Parsing and def/use extraction are linear in cell size. The approach under-approximates dynamic flows (reflection, aliasing), which we surface in §4.12 [1, 2].

4.4 Per-type profiles, units, and sensitive data

Per-type profiles. We use a common core (`@id`, `@type`, `name`, `license`, `version`, `sha256`, `sensitive`) and small extensions per object type (Notebook, Cell/Activity, File/Dataset, Figure/Table, Parameter, Workflow). Profiles are expressed directly in RO-Crate JSON-LD with RO-Crate, PROV-O, and DCAT terms.

Config files (`conf_`). Configuration files (e.g., YAML/JSON/TOML/INI) are included as digital objects. Non-sensitive keys may be promoted into `schema:PropertyValue` entries after user confirmation. Keys matching sensitive patterns (e.g., `token`, `password`, `secret`) are redacted by default.

Secrets (`secret_`). We do not embed secrets in crates. Instead, we record placeholders (`value="[REDACTED]"`, `sensitive=true`) and a short rights note at crate or file level. Secrets are resolved via platform mechanisms (environment variables or vaults); crates are designed to be shared.

Workflows (`.naavrewf`). NaaVRE workflow files are included as *Files* with proper `encodingFormat` (JSON or YAML) and linked to a lightweight *Workflow* (or *CreativeWork* with `additionalType="NaaVREWorkflow"`). We connect them via PROV to the notebook/crate they are exported from.

Quantities and units (manual). Numeric fields with units (parameters, measured variables) are captured through `schema:QuantitativeValue` (`value`, `unitText`, optional `unitCode`). Units are entered by the user during confirmation; optional domain annotations (e.g., CF Standard Names) are supported.

4.5 Field acquisition by object type

Table 4.1 summarises, for each digital object, the fields we record, *how* we obtain them in CellScope, and *where* they are stored in the RO-Crate graph (using RO-Crate, PROV, and DCAT terms) [5, 11, 12, 21]. Static analysis is authoritative; runtime hints are opt-in and only augment edges (Section 4.3). *Notes.* (i) For R, CellScope calls the existing I/O Detector and normalises its JSON output into the same symbol/file model as Python before creating PROV links. (ii) When a field cannot be derived reliably (e.g., license), we include an editable placeholder to keep the crate valid without blocking export.

4.6 Packaging: RO-Crate with PROV relations

Crate contents. We emit a self-contained RO-Crate with:

- the notebook and per-cell code snippets,
- a graph export (`graph/graph.json` and GraphML for external tools [15]),
- JSON-LD metadata describing entities and relations (RO-Crate + PROV) [5, 11, 12].

Entity mapping. We represent:

- **Cells** as activities (`prov:Activity`) with human-readable names,
- **Files/Datasets** as data entities,
- **Logical variables** as internal identifiers when helpful for comprehension.

Edges use standard PROV predicates (`prov:used`, `prov:wasGeneratedBy`, `prov:wasDerivedFrom`) [5]. We avoid custom ontologies; any additional typing is included as lightweight annotations for readability.

Portability. The crate opens locally; the bundled HTML viewer renders the same overview outside the author's environment [16]. This fulfils the portability requirement derived in Chapter 3 (§3.3) and specified here with concrete structures [11, 12].

Sharing and protection

Sharing. The RO-Crate is the canonical, portable bundle: notebook, files, and JSON-LD provenance live together and can be opened offline or attached to registries (e.g., SEEK/WorkflowHub/Zenodo) [11, 12]. The crate includes a graph export so reviewers can browse dependencies without services.
Protection. We record `license` and `rights` at the crate root and for individual files where relevant; sensitive assets can be represented by metadata with restricted distributions (e.g., DCAT `accessURL` only) rather than embedding data [21]. CellScope avoids storing secrets (tokens, passwords) and supports redacting paths or excluding files at export. Access control and embargo live in the destination platform; the crate carries the *policy metadata*, not enforcement.
Good practice. Prefer SPDX licenses; include creators/contributors; checksum files for integrity; for third-party datasets, cite identifiers (DOIs/URLs) instead of copying data; include environment info to support replay.

4.7 Storage and indexing service

Canonical package. The RO-Crate written next to content (FS or object storage) is the source of truth.

Index (findability). An *Indexer* service serialises a projection of the crate to RDF and posts SPARQL Update deltas to a triple store. We index ids/types, names, creators, versions, checksums, sensitivity flags, and core PROV edges (`prov:used`, `prov:wasGeneratedBy`, `prov:wasDerivedFrom`).

API (internal). `POST /cellscope/index` accepts a crate path or JSON-LD payload, computes the RDF delta, and posts it to the triple store. Indexing runs on *save* and on explicit *Export*; updates are debounced to keep latency low.

Rationale. Crates remain portable and human-readable; the index delivers fast cross-object queries and "easy to find" requirements without imposing a DB on authors. This implements the storage-first separation discussed conceptually in §3.3.

4.8 Versioning and persistent identifiers

Versioning policy. Each object has `version` and `sha256`. Any content or metadata change creates a new version with `prov:wasRevisionOf` to the prior `@id`. Files live under `content/{id}/{version}/...` (or object storage with bucket versioning). The crate root references the latest versions.
JSON-LD example.

```
{ "@id": "#cell-7-v3", "@type": "prov:Activity", "name": "Cell 7",
  "version": "3", "prov:wasRevisionOf": { "@id": "#cell-7-v2" } }
```

PID minting (publish). A PID is minted only on *Publish*. We ensure a stable local `@id`, then the platform PID service (e.g., Handle/DataCite) returns a global identifier recorded as `identifier` and `sameAs`. Until publish, internal links use local URIs.
PID example.

```
{ "@id": "#dataset-precip-v1", "@type": "Dataset", "name": "Daily precipitation",
  "identifier": "doi:10.1234/abcd.efgh", "sameAs": "https://doi.org/10.1234/abcd.efgh" }
```

4.9 Views: Graph and non-graph list/filter

Graph. A self-contained HTML (`cell_graph.html`) renders cells and edges. Hover shows a code snippet and key metadata; upstream/downstream highlighting supports impact analysis [16]. The render pipeline follows the architecture in Figure 4.1 and implements the storage-first, confirm-first principles introduced in section 3.3. Views are read-only lenses over what is stored (section 4.1), reflecting the "generate, then visualise" principle from section 3.3.

List/filter view. To match user-story priorities, we provide a complementary panel with:

- facets by *type* (cell, file, dataset, variable),
- *tags* and free-text search,
- quick actions (show producers/consumers, open in viewer, copy path).

Both views are fed by the same graph JSON (see Figure 4.1).

4.10 Jupyter integration

The endpoints and panels align with Figure 4.1: the frontend calls `/cellscope/analyze` and `/cellscope/export`, which in turn invoke the core capture and packaging components, and `/cellscope/index` to push deltas to the index.

Server API. The Jupyter Server extension exposes:

- `POST /cellscope/analyze`: returns graph JSON for the active notebook,
- `POST /cellscope/export`: writes an RO-Crate and returns its path [11, 12].

See Figure 4.1 for their placement in the stack.

JupyterLab extension. The UI adds a side panel (graph + list/filter) and an *Export* button. Re-analysis is debounced on save/execute to keep feedback responsive [16].

CLI (developer mode). A small CLI mirrors the endpoints: `cellscope build` (analyze+export) and `cellscope vis` (open viewer). This aids reproducible tests and batch runs.

NaaVRE. The crate can be indexed by NaaVRE services; because exchange is RO-Crate JSON-LD, CellScope remains decoupled from specific registries [10, 12, 17].

Real-time update path

On *save* and/or *cell executed*, the server emits an event that triggers: (i) `/cellscope/analyze` (fast graph update), and (ii) `/cellscope/index` (crate delta → SPARQL Update). We debounce events to keep UI responsiveness. Service objectives: P95 analyze+index < 400 ms for ≤ 50 cells; P95 < 2 s at ≈ 100 cells (see O4 in Chapter 5). This implements the near-real-time behaviour sketched in §3.3.

APIs (summary).

- `POST /cellscope/analyze` → graph JSON (fast).
- `POST /cellscope/export` → write RO-Crate; return path.
- `POST /cellscope/index` → post RDF/SPARQL Update delta to triple store.
- `GET /cellscope/objects?id=...` → fetch one object for the review forms.

4.11 Alternatives considered

Kernel-agent first. Agents inside IPython/IRkernel emit runtime events for high-fidelity provenance; we did not adopt this due to overhead and tighter coupling to kernel internals [13].

SoS-centric. Building on SoS multi-kernel DAGs gives explicit polyglot structure, but restricts adoption to SoS users. Our approach is notebook-agnostic and infers hand-offs via files [38].

4.12 Limitations

Static analysis misses some dynamic flows (reflection, late binding) and may under-approximate cross-language aliasing. Variable-level logical entities improve comprehension but are not a formal data model; files/datasets are the stable bridge for polyglot flows. Large notebooks can create dense graphs; we cap snippet length and rely on filtering and focus modes. Runtime hints remain opt-in to avoid overhead and security concerns [1, 2, 13].

Object	Fields (examples)	How obtained	Stored as
Notebook	name, description, author, license; encodingFormat=`application/x-ipynb+json`	From notebook metadata (if present) or user entry; file sniffing for format	`CreativeWork` (`name`, `description`, `creator`, `license`); file entity for the `.ipynb`
Cell (code/markdown)	name ("Cell 7"), programmingLanguage, code snippet	From notebook cell contents and metadata; language from kernel/-cell tags	`prov:Activity` with `name`/`programmingLanguage`; snippet as annotation
Variable / symbol (logical)	name (e.g., `df_clean`); generatedBy / usedBy	Python: AST; R: I/O Detector (existing) → normalised model; optional hints add confidence	Logical `Dataset`; links via `prov:wasGeneratedBy` (producer cell) and `prov:used` (consumer cell)
Parameter / config	`name=value` pairs influencing results	Static heuristics (simple assignments, CLI args, constants); else user-supplied	`PropertyValue` attached to activities; optionally `prov:used` from consumer cell
File / artifact	path, encodingFormat (MIME), size, checksum	Filesystem stat; MIME from extension/sniff; provenance from analysis	`File` with `contentSize`, `encodingFormat`, `sha256`; linked via `prov:used`/`prov:wasGeneratedBy`
Dataset (curated)	name, description, identifier/-DOI, license, distribution	User-supplied reference (URL/DOI) or inferred from citations/paths	`Dataset` + DCAT `distribution`; `identifier`, `license`
Figure / table	filename (PNG/SVG/CSV) about/keywords	Detection of `savefig`/exports in code; file stat	`File` or `ImageObject`; provenance edges as for files
Environment / kernel	name, version, requirements (image/packages)	Kernel spec + optional environment export (pip/conda); user-provided image tag if available	`SoftwareApplication` (`name`, `version`, `softwareRequirements`)
Component (containerised step)	name, identifier/URI, declared I/O	From existing containerizer pipeline (if used)	`SoftwareApplication` + links back to producing cells; store component URI in crate
Metadata record / Crate	crate metadata (title, creator, license)	`ro-crate-py` writer fills standard fields; user may add license/creators	`CreativeWork` root; payload in `ro-crate-metadata.json`

Table 4.1: Acquisition and storage of fields per object. Static analysis is primary; runtime hints are optional. RO-Crate/PROV/DCAT provide the storage vocabulary [5, 11, 12, 21].

Chapter 5

Validation

We evaluate CellScope on its main promise: make it easier to understand, reproduce, and manage results in Jupyter notebooks, while keeping artifacts portable. We follow two complementary tracks aligned with the checklist: (i) *design of experiments* that map directly to requirements and contributions, and (ii) clear *KPIs*, procedures, and analyses per experiment.

5.1 Objectives

- **O1 (Coverage).** How completely do we capture cells, variable definitions/uses, and producer→consumer edges? *(RQ1)*
- **O2 (Comprehension & efficiency).** Do users find producers, inputs, and impacts faster and with fewer errors? *(RQ3)*
- **O3 (Portability/FAIRness).** Does packaging as RO-Crate increase machine-actionable quality? *(RQ2)*
- **O4 (Real-time & scale).** Can CellScope store and index objects fast enough for interactive use, and does the index scale for discovery? *(RQ2/RQ3)*

Objective	Primary KPI(s)	Maps to
O1 Coverage	F1 (defs/uses/edges), Cohen's κ	RQ1
O2 Comprehension	Time-to-answer, success, SUS/TLX	RQ3
O3 Portability	JSON-LD validation, F-UJI, FAIRshake	RQ2
O4 Real-time & scale	P50/P95 analyze+index; SPARQL latency	RQ2/RQ3

Table 5.1: Validation objectives mapped to RQs and KPIs.

Baselines and conditions. We compare against widely used, conceptually similar tools from the state of the art:

- **ProvBook** (notebook execution provenance/visualisation; Python notebooks) [34].
- **noWorkflow** (execution provenance for Python without modifying code; script- and cell-level provenance) [13].

We use *three* conditions for the user study: **C0** Notebook-only (baseline), **C1** ProvBook, **C2** CellScope.[1]

Operationalisation and hypotheses. O1 (Coverage). Build a small, hand-labelled gold standard for two notebooks: (i) a curated Python "stress-test", and (ii) a cross-kernel Python↔R notebook (file hand-offs). Label per cell: (a) function names, (b) variable definitions, (c) variable uses, (d)

[1] noWorkflow is used in O1/O3 (coverage/FAIRness) rather than O2 because its UI is not notebook-centric for end-user comprehension tasks.

producer→consumer edges. Compute precision/recall/F1 at the symbol level for defs/uses and at the (source cell, symbol, target cell) level for edges. Inter-annotator agreement (Cohen's κ) on a 20% subset benchmarks labelling reliability.

H1a: CellScope achieves $\mathrm{F1}_{\mathrm{defs}} \geq 0.85$, $\mathrm{F1}_{\mathrm{uses}} \geq 0.80$, $\mathrm{F1}_{\mathrm{edges}} \geq 0.80$ in the curated Python notebook.
H1b (improvement). Enabling runtime *hints* increases $\mathrm{F1}_{\mathrm{edges}}$ by ≥ 0.08 on dynamic patterns (vs. static only) [13].
H1c (polyglot). For Python↔R hand-offs, edge recall ≥ 0.75 when edges are materialised via files.

O2 (Comprehension & efficiency). Within-subjects task battery (T1-T3) with *three* interface conditions (C0, C1, C2), counterbalanced by a 3×3 Williams design. Measure time-to-answer and correctness; collect SUS and NASA-TLX [36, 37].

H2a: CellScope reduces median time-to-answer by $\geq 30\%$ vs. Notebook-only (C2 < C0).
H2b: CellScope reduces median time-to-answer by $\geq 20\%$ vs. ProvBook (C2 < C1).
H2c: Error rate with CellScope is non-inferior to both baselines ($\Delta \leq 5$ percentage points).
H2d: SUS ≥ 70; *H2e:* TLX ≤ 45.

O3 (Portability/FAIRness). Assess RO-Crate with (i) F-UJI (selected indicators) and (ii) a FAIRshake rubric tailored to *notebook + context*; validate JSON-LD against RO-Crate 1.2 and open the crate offline (HTML report) and in a curator (e.g., Describo) or as a WorkflowHub/SEEK attachment [11, 12, 39, 40].

H3: F-UJI overall ≥ 0.80 (A1/I1/R1 indicators); FAIRshake median $\geq 3.5/5$; JSON-LD validates; HTML graph viewable without services.

O4 (Real-time & scale). Measure end-to-end latency for *analyze+index* on notebooks of 10/50/100 cells under single-user and scripted multi-user loads. Record P50/P95 latencies and update throughput; time common SPARQL templates (producers, consumers, datasets per notebook). *H4a:* For $\leq$50 cells, P95 analyze+index < 400 ms; for ≈ 100 cells, P95 < 2 s. *H4b:* Median SPARQL query time < 200 ms on the indexed store.

KPI-requirement mapping. O1→R1 (capture), O2→R3 (views), O3→R2 (packaging), cross-kernel edges in O1→R4 (polyglot).

5.2 Datasets, baselines, and tasks

Notebooks.

1. **N1: Python stress-test** (functions, reuse, plotting; single-kernel).
2. **N2: Rainforest climate** (Py→R or R→Py via CSV/Parquet/RDS hand-offs).

Baselines & eligibility. For O2, we use N1 (Python) so that ProvBook (C1) and CellScope (C2) are comparable in a notebook UI [34]. For O1/O3, we run all tools on both N1 and N2; for noWorkflow we run on the Python parts and map its provenance to comparable edges [13].

Tasks (performed on each condition).

- **T1** Find where `df_clean` is produced and list its inputs.
- **T2** If cell X changes, identify downstream results that must be re-run.
- **T3** Locate the code that produced Figure 1 and list its inputs.

5.3 Participants, design, and procedure

Participants. Target n = 12-16 (mixed Jupyter experience). Recruitment via supervisor channels; consent procedure and anonymisation in place.

Design. Within-subjects with three interface conditions (C0, C1, C2), counterbalanced by a Williams design to control order/carryover. Each participant performs T1-T3 on N1 under each condition with a short washout task between blocks.

Procedure. (1) Briefing and consent. (2) 5-minute familiarisation with the notebook UI (no tools). (3) Block A (condition 1): T1-T3; SUS/TLX. (4) Washout. (5) Block B (condition 2): T1-T3; SUS/TLX. (6) Block C (condition 3): T1-T3; SUS/TLX. (7) Short semi-structured interview (5-8 min).

Instrumentation. We log start/stop timestamps per task, success/partial/fail outcomes, and interaction events in each UI (e.g., graph hovers, edge clicks). For O1, two annotators build the gold standard and resolve disagreements by discussion; κ is computed on a 20% overlap.

5.4 Metrics and analysis

Coverage (O1). Symbol- and edge-level confusion matrices against the gold set. Report precision/recall/F1 for (a) defs, (b) uses, (c) edges. Bootstrap 95% CIs over cells. Report Cohen's κ for inter-annotator agreement. For noWorkflow/ProvBook, map runtime provenance to edges via `used`/`wasGeneratedBy` relations [13, 34]. Analyse error modes (dynamic name creation, aliasing, cross-cell shadowing).

Efficiency and comprehension (O2). For time-to-answer across three conditions use a *Friedman* test with *post-hoc* Wilcoxon signed-rank tests (Holm-corrected); report effect sizes (r) and 95% CIs. For success rate, use Cochran's Q with McNemar post-hoc (Holm). Report per-task deltas (T1-T3) to identify where the graph helps most (producer lookup vs. impact analysis). SUS and TLX analysed with Friedman + post-hoc Wilcoxon [36, 37].

Portability and FAIRness (O3). Validate JSON-LD against RO-Crate 1.2; verify crate opens offline (HTML) and in a curator. Run F-UJI and record per-indicator scores (A1, I1, R1, etc.). Run a FAIRshake assessment with a small panel (3-5 reviewers) using a rubric adapted to "notebook + context". Success criteria: FUJI ≥ 0.80, FAIRshake median $\geq 3.5/5$, core PROV links present [11, 12, 39, 40].

Latency and scalability (O4). Report P50/P95 latencies for *analyze+index* by notebook size; update throughput (updates/min); and SPARQL response times for a fixed set of templates. Use bootstrapped CIs and include resource footprints (CPU/memory) for indexer and store. Summarise failures/regressions with remediation (e.g., batching deltas).

Power note. With $n = 14$ and a within-subjects design, we expect $\approx 80\%$ power to detect a medium effect ($r \approx 0.5$) in pairwise Wilcoxon tests at $\alpha = 0.05$; omnibus Friedman has higher sensitivity.

5.5 Threats to validity

Small samples and domain specificity limit generalisation; AST under-approximates dynamic flows; tool familiarity varies. We mitigate by (i) using a within-subjects design and counterbalancing; (ii) reporting per-notebook and per-task results (avoiding over-aggregation); (iii) preregistering acceptance thresholds (H1-H3); and (iv) releasing all materials (notebooks, gold labels, crates, HTML graphs) to enable reanalysis [3, 6].

5.6 Materials and availability

We release anonymised datasets (N1-N2), gold labels, analysis scripts, exported RO-Crates, and the HTML graph viewers to a public repository, enabling independent reproduction of results [11, 12].

Chapter 6

Discussion

In this chapter, we discuss the results of our experiment(s) on ...

Finding 1: Highlight like this an important finding of your analysis of the results.

Refer to Finding 1.

6.1 Lessons learned

Chapter 7

Conclusions

7.1 Future work

Acknowledgements

If so inclined, thank people.

Bibliography

[1] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about jupyter notebooks," *Proceedings of the 16th International Conference on Mining Software Repositories (MSR)*, pp. 507–517, 2019.

[2] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "Understanding and improving the quality and reproducibility of Jupyter notebooks," *Empirical Software Engineering*, vol. 26, no. 4, p. 65, 2021. DOI: 10.1007/s10664-021-09961-9.

[3] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *CHI*, ACM, 2018, 32:1–32:12.

[4] T. Kluyver, B. Ragan-Kelley, F. Pérez, and et al., "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, IOS Press, 2016, pp. 87–90.

[5] T. Lebo *et al.*, "Prov-o: The prov ontology," W3C, W3C Recommendation, Apr. 2013. [Online]. Available: https://www.w3.org/TR/prov-o/.

[6] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature*, vol. 533, no. 7604, pp. 452–454, 2016.

[7] J. P. A. Ioannidis, "Why most published research findings are false," *PLoS Medicine*, vol. 2, no. 8, e124, 2005.

[8] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, "Ten simple rules for reproducible computational research," *PLOS Computational Biology*, vol. 9, no. 10, e1003285, 2013. DOI: 10.1371/journal.pcbi.1003285.

[9] M. D. Wilkinson and et al., "The fair guiding principles for scientific data management and stewardship," *Scientific Data*, vol. 3, p. 160 018, 2016.

[10] Z. Zhao *et al.*, "Cloud-native data analytics workflows for science: From devops to DataOps," *Software: Practice and Experience*, vol. 52, no. 8, pp. 1784–1808, 2022. DOI: 10.1002/spe.3091.

[11] S. Soiland-Reyes *et al.*, "Packaging research artefacts with ro-crate," *Data Science*, vol. 5, no. 2, pp. 97–138, 2022. DOI: 10.3233/DS-210053.

[12] R.-C. Community, *Ro-crate specification 1.2*, https://www.researchobject.org/ro-crate/1.2/, Accessed 2025-09-05, 2025.

[13] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "Noworkflow: A tool for collecting, analyzing, and managing provenance from python scripts," in *PVLDB*, vol. 10, 2017, pp. 1841–1844. DOI: 10.14778/3137765.3137810.

[14] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher, "Yesworkflow: A user-oriented, language-independent tool for recovering workflow information from scripts," in *IPAW*, Springer, 2015, pp. 230–244.

[15] A. Hagberg, D. Schult, and P. Swart, "Exploring network structure, dynamics, and function using networkx," in *SciPy*, 2008.

[16] *Vis.js network*, https://visjs.org/, 2024.

[17] M. R. Crusoe, M. Abueg, K. Alam, *et al.*, "Workflowhub: A registry for fair computational workflows," in *2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, IEEE, 2022, pp. 2685–2692. DOI: 10.1109/BIBM55620.2022.9995761.

[18] K. Wolstencroft, S. Owen, A. R. Williams, C. Goble, S. Soiland-Reyes, *et al.*, "Fairdom-seek: A systems biology data and model management platform," *Bioinformatics*, vol. 33, no. 11, pp. 1748–1750, 2017. DOI: 10.1093/bioinformatics/btx265.

[19] J. Boon, "Enhancing r code containerization through automated input/output detection and type inference," Master Software Engineering thesis, M.S. thesis, University of Amsterdam, Aug. 2024.

[20] F. Maali, J. Erickson, and P. Archer, "Data catalog vocabulary (DCAT)," W3C, W3C Recommendation, Jan. 2014. [Online]. Available: https://www.w3.org/TR/vocab-dcat/.

[21] S. J. D. Cox *et al.*, "Data catalog vocabulary (dcat)—version 2," W3C, W3C Recommendation, Feb. 2020. [Online]. Available: https://www.w3.org/TR/vocab-dcat-2/.

[22] S. Leo *et al.*, "Recording provenance of workflow runs with RO-Crate," *PLOS ONE*, vol. 19, no. 9, e0309210, 2024. DOI: 10.1371/journal.pone.0309210.

[23] S. Leo *et al.*, "Recording provenance of workflow runs with RO-Crate," *PLOS ONE*, vol. 19, no. 9, e0309210, 2024. DOI: 10.1371/journal.pone.0309210.

[24] E. Afgan *et al.*, "The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update," *Nucleic Acids Research*, vol. 46, no. W1, W537–W544, 2018. DOI: 10.1093/nar/gky379.

[25] L. Quaranta, D. Di Castro, S. Scardapane, and F. A. Lisi, *Eliciting best practices for collaboration with computational notebooks*, arXiv preprint arXiv:2202.07233, 2022. [Online]. Available: https://arxiv.org/abs/2202.07233.

[26] D. Garijo, C. Goble, S. Soiland-Reyes, S. Owen, M. R. Crusoe, *et al.*, "Workflowhub: A registry for fair computational workflows," *Scientific Data*, 2025, in press.

[27] E. Afgan, J. Taylor, A. Nekrutenko, *et al.*, "The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2024 update," *Nucleic Acids Research*, vol. 52, no. W1, W83–W90, 2024. DOI: 10.1093/nar/gkae410.

[28] ARDC, *Describo: Ro-crate creation tool*, https://arkisto-platform.github.io/describo/, Accessed 2025-09-05, 2025.

[29] T. McPhillips *et al.*, "Yesworkflow: A user-oriented, language-independent tool for recovering workflow information from scripts," in *International Provenance and Annotation Workshop (IPAW)*, ser. Lecture Notes in Computer Science, vol. 8628, Springer, 2015, pp. 230–244. DOI: 10.1007/978-3-319-16462-5_19.

[30] S. Samuel and B. König-Ries, "Provbook: Provenance-based semantic enrichment of interactive notebooks for reproducibility," in *ISWC 2018 Posters & Demos (CEUR-WS Vol. 2180)*, Cited in this thesis as "Kau2020"; ProvBook line of work relevant to 2020–2022 follow-ups, 2018, 57:1–57:4. [Online]. Available: http://ceur-ws.org/Vol-2180/paper-57.pdf.

[31] S. Samuel, F. Löffler, and B. König-Ries, *Reproducemegit: A tool to support reproducibility of machine learning pipelines*, arXiv:2006.12110, 2020. [Online]. Available: https://arxiv.org/abs/2006.12110.

[32] C. Community, *Cf (climate and forecast) metadata conventions*, https://cfconventions.org/, Accessed 2025-09-05, 2025.

[33] C. Community, *Cf standard name table*, https://cfconventions.org/standard-names.html, Accessed 2025-09-05, 2025.

[34] S. Samuel and B. König-Ries, "Provbook: Provenance-based semantic enrichment of interactive notebooks for reproducibility," in *ISWC 2018 Posters & Demos (CEUR-WS Vol. 2180)*, Cited in this thesis as "Kau2020"; ProvBook line of work relevant to 2020–2022 follow-ups, 2018, 57:1–57:4. [Online]. Available: http://ceur-ws.org/Vol-2180/paper-57.pdf.

[35] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[36] J. Brooke, "Sus: A "quick and dirty" usability scale," *Usability Evaluation in Industry*, pp. 189–194, 1996.

[37] S. G. Hart and L. E. Staveland, "Development of nasa-tlx (task load index): Results of empirical and theoretical research," in *Advances in Psychology*, vol. 52, Elsevier, 1988, pp. 139–183.

[38] B. Peng, K. Tang, *et al.*, "Sos notebook: An environment for multi-language data analysis and reproducible research," in *The Journal of Open Source Software*, JOSS article; SoS Notebook, 2017.

[39] A. Devaraju and R. Huber, "An automated solution for measuring the progress toward fair research data," *Patterns*, vol. 2, no. 11, p. 100 370, 2021. DOI: 10.1016/j.patter.2021.100370.

[40] D. J. B. Clarke *et al.*, "Fairshake: Toolkit to evaluate the fairness of research digital resources," *Cell Systems*, vol. 9, no. 5, pp. 417–421, 2019. DOI: 10.1016/j.cels.2019.09.011.

Appendix A

Non-crucial information