quartobot: Quarto-native automation for citation resolution and reproducible publishing

Sean Davis

TODO — Sean's section. ~250–300 words. Four moves: problem (manubot pattern, Quarto's broader publishing surface, pandoc-manubot-cite covers citation resolution but not the broader pattern); approach (pre-render hook architecture, committed `references.json` as a durable artifact, CLI + MCP + on-disk surfaces); results (wraps `citekey_to_csl_item`, installable via `uv tool install`, validated on two production manuscripts); conclusion (extends manuscript-as- software along five dimensions while preserving manubot's resolver expertise). Full bullets in outline.md.

Introduction

Implementation

quartobot is structured as a single Python package exposing a small command-line interface. The integration with Quarto runs through a `project.pre-render:` hook in `_quarto.yml`; the package is otherwise independent of Quarto and can be invoked directly. Citation resolution itself is delegated to manubot's `citekey_to_csl_item` function, so the resolver chain (Crossref for DOI, NCBI E-utilities for PubMed and PMC, the arXiv API, Open Library for ISBN, the Wikidata API for Wikidata items) is the one already validated by eight years of manubot use.

Architecture and the choice of pre-render over filter

Quarto's documentation describes two principal customization surfaces: pre-render scripts, which run before pandoc parses the document, and Lua filters, which run on the parsed abstract syntax tree after the document engine executes. Citation **formatting** — how a resolved reference appears once pandoc-citeproc has it in hand — is a natural Lua-filter concern, and indeed pandoc-citeproc is itself implemented as a filter. Citation **resolution**, in contrast, must

happen before the AST exists, because the bibliography is consumed at parse time. A Lua filter is therefore unable to add entries to the bibliography; by the time it runs, that file has already been read.

quartobot lives on the pre-render side for this reason. The `resolve` subcommand scans Quarto sources (`.qmd`, `.md`, and `.ipynb`) for citation keys of the form `@<prefix>:<identifier>`, resolves each through `citekey_to_csl_item`, and writes the resulting CSL-JSON to a file named `references.json`. Pandoc-citeproc then consumes that file as part of the normal render. The integration touches three things in a project: a single line under `project:` in `_quarto.yml`, the generated `references.json`, and optionally a hand-curated `references.bib` for entries the resolver does not cover (textbooks, gray literature, software releases with no DOI).

This choice differs from the existing path via the `pandoc-manubot-cite` filter, which provides the same resolver behavior as a pandoc filter and therefore runs during each pandoc invocation. The two approaches are not equivalent. A filter resolves at render time and produces no durable output. A pre-render hook resolves once, writes a file, commits it to the repository, and lets every subsequent render consume the cached bibliography without further network activity. For manuscripts rendered repeatedly — by continuous integration, by per-pull-request preview, by per-commit permalink — this difference becomes material. The bibliography becomes a versioned artifact in its own right.

The trade-off favors pre-render when several of the following hold: the same manuscript is rendered repeatedly, authors want offline-capable renders, the bibliography is treated as a versioned artifact reviewed alongside the prose, and determinism matters more than always-fresh metadata. These are precisely the conditions under which the manuscript-as-software pattern operates. Pre-render is therefore the structurally correct architectural choice for the pattern; per-invocation filters remain a valid choice for one-off pandoc renders where the manuscript is not under continuous integration.

Supported identifiers

Citation keys take the form `@<prefix>:<identifier>` in prose. The supported prefixes and their upstream resolvers are listed in Table 1.

Prefix	Identifier type	Upstream resolver
`@doi:`	Digital Object Identifier	Crossref
`@pmid:`	PubMed identifier	NCBI E-utilities
`@pmc:`	PubMed Central identifier	NCBI E-utilities
`@arxiv:`	arXiv identifier	arXiv API
`@isbn:`	International Standard Book Number	Open Library
`@wikidata:`	Wikidata QID	Wikidata API
`@url:`	Web resource URL	URL metadata extraction

Hand-curated entries continue to be supported through a project-level `references.bib`, which pandoc reads alongside the resolver output. The choice of which mechanism to use for a given citation is the author's; the two coexist in the same document. Reconciliation between the two sources is described below.

Input format coverage

The pre-render scan is format-agnostic on the input side. Jupyter notebooks (`.ipynb`), Quarto Markdown (`.qmd`), and plain Markdown (`.md`) are all scanned in the same pass, because Quarto presents these formats through a uniform interface. This is the dimension manubot's bespoke toolchain did not reach: notebooks were never first-class manuscript inputs. The same `@doi:` syntax that resolves in prose resolves equally in a notebook narrative cell, allowing a literate-programming manuscript (analysis code, narrative, and citations in the same file) to use the manuscript-as-software pattern without modification.

Cross-platform deployment

quartobot inherits Quarto's platform reach. The package runs on macOS, Linux, and Windows; the GitHub Actions workflows that ship with `quartobot use github-ci` (described below) use standard cross-platform actions. Manubot's bespoke toolchain assumed a Linux+Docker rendering environment; quartobot makes no such assumption.

CI scaffolding

The `use github-ci` subcommand lays down the continuous-integration scaffolding for the manuscript-as-software pattern. The default configuration produces three workflows. The first builds the manuscript on every push to the default branch, rendering to HTML, PDF, DOCX, and JATS XML, and publishes the result to GitHub Pages. The second builds a per-pull-request preview at a stable URL under `…/pr/<n>/`, posting a sticky comment with the preview link for reviewers who do not have a local Quarto installation. The third generates a versions index on `gh-pages` listing tagged releases and open pull-request previews.

With the optional `--with-versioned-snapshots` flag, each commit on the default branch also produces an immutable snapshot at `…/v/<sha>/`. This is the manubot per-commit permalink pattern, preserved exactly: a rendered manuscript on disk always knows which commit produced it, and a downloaded copy carries that provenance in the HTML.

Three surfaces for human and AI use

quartobot exposes its resolution functionality through three surfaces, each appropriate for a different consumer.

The **CLI** (`quartobot resolve`) is the primary interface and the one Quarto invokes through the pre-render hook. The same CLI is the surface AI authoring agents shell out to when drafting. An agent that has just written `@doi:10.1371/journal.pcbi.1007128` into prose can pass that identifier, or a batch of several dozen, to a single `quartobot resolve` call and receive the resolved CSL-JSON back. Two checks become cheap. First, the agent can confirm that every identifier resolved at all; an identifier that does not resolve is one the agent likely fabricated, and the citation can be removed or corrected before it reaches the manuscript. Second, the agent can compare the resolved metadata to what it intended (the work's title, the lead author's surname, the year of publication) and catch the case where the identifier resolves but does not name the paper the agent meant to cite. The pattern matches the broader recommendation in recent work on agent-tool efficiency to expose tools as code on a filesystem rather than as in-context schemas ("Code Execution with MCP: Building More Efficient AI Agents \ Anthropic," n.d.).

The **MCP server** (`quartobot mcp`) is an optional stdio-based Model Context Protocol surface for the case where an IDE or agent framework already operates inside an MCP-aware environment and shelling out adds ergonomic friction. The server exposes the same resolver function as the CLI; results are identical.

The **on-disk artifact** (`references.json`) is the surface that distinguishes quartobot most sharply from a filter-based approach. Once `resolve` has written the file and an author has committed it, the resolved citation graph is available to any downstream consumer without further network activity and without invoking pandoc. Citation linters can flag unused references, duplicate identifiers, or missing required fields. Bibliometric pipelines can ingest a manuscript's reference set as structured data. Other rendering toolchains can read the JSON directly. AI authoring agents drafting later prose can ground new claims against the manuscript's existing reference set, reading the committed file rather than re-querying the upstream APIs. This reuse is what `pandoc-manubot-cite` cannot offer: filter outputs are produced and consumed within a single pandoc invocation and are not available to anything else.

A growing body of work documents the per-turn token cost of MCP servers in agent workflows, with multi-server deployments routinely consuming tens of thousands of tokens before useful work begins (Sadani and Kumar 2026; "MCP Servers Use 35x More Tokens Than CLI Tools — And Reliability Drops to 72% on Hard Tasks" 2026). Anthropic's analysis recommends presenting tools as code on a filesystem and loading definitions on demand, reporting up to a 98.7% reduction in context overhead under that pattern ("Code Execution with MCP: Building More Efficient AI Agents \ Anthropic," n.d.). quartobot's design predates this literature but

lands on the same architecture: the CLI is the default surface, the MCP server is optional, and the on-disk artifact is the structurally efficient locus for downstream reuse.

Reconciliation of resolved and hand-curated entries

Real manuscripts mix resolver-generated and hand-curated citations. A `references.bib` may carry entries for textbooks, software releases, gray literature, or any work without a persistent identifier that the resolver covers. The `references.json` produced by `quartobot resolve` is regenerated on each pre-render pass. Without explicit handling, the two sources can collide on citation keys, especially as resolver-generated keys are derived from the identifier itself (e.g., `doi:10.1371/journal.pcbi.1007128`) and may overlap conceptually with hand-curated keys for the same work.

The `reconcile` subcommand handles this case explicitly. Three modes are supported: `prefer-bib` keeps the hand-curated entry and discards the resolver's version; `prefer-resolved` keeps the resolver's entry and removes the hand-curated duplicate; and `fail-on-collision` halts with a diagnostic listing the conflicting keys for the author to resolve. All three modes operate with backup-then-mutate semantics: the original files are saved before any change, so a `reconcile` step is reversible.

Operation

Installation

quartobot is distributed on PyPI and is installed as a standalone command-line tool using `uv`:

```
uv tool install quartobot
```

The `uv tool install` form is recommended over `pip install` because it isolates quartobot in its own environment and places the resulting executable on the user's `PATH` without polluting any project's Python environment. quartobot requires Python 3.10 or later; no other system dependencies are required beyond an installed Quarto runtime.

Canonical workflow

The shortest path from "I want to write a manuscript" to a working manuscript-as-software project is three commands:

```
quarto create project manuscript my-paper
cd my-paper
quartobot init
```

`quarto create` scaffolds a default Quarto manuscript with an `index.qmd` source file, a placeholder `references.bib`, and a `_quarto.yml` project configuration. `quartobot init` then layers the citation pipeline on top: it adds the `pre-render:` line to `_quarto.yml`, ensures `references.json` is listed in the `bibliography:` field, and appends the relevant entries to `.gitignore` for any transient files. The result is a Quarto project that resolves `@doi:`, `@pmid:`, and other persistent-identifier citation keys on every render.

Authors then write prose as they would in any Quarto project, with citation keys pasted directly into the text:

```
We follow @doi:10.1371/journal.pcbi.1007128, with the dataset described
in @pmid:31479462 and methods inspired by @arxiv:2104.10729.
```

A `quarto render` runs the pre-render hook automatically, produces an updated `references.json`, and proceeds with the normal multi-format render.

To add the continuous-integration scaffolding described above, a single additional command is run from inside the project directory:

```
quartobot use github-ci
```

The author commits the generated workflow files and pushes; the first push to GitHub initiates the render, publishes the result to GitHub Pages, and registers the pull-request preview pattern for subsequent contributions.

Alternative on-ramps

The canonical workflow assumes a new project. Two alternative paths cover other starting points.

For an **existing Quarto project** that already has prose and possibly a `references.bib`, `quartobot init` is run from inside the project root. It performs the same scaffolding steps and does not modify existing prose or bibliography entries. Authors can then begin adding `@doi:`-style keys incrementally without rewriting any existing citations.

For the **resolver-only case** — an author who wants citation resolution but has their own CI configuration and does not want quartobot to scaffold workflows — `quartobot init` provides everything needed without `quartobot use github-ci`. The author wires the resolver into any existing pipeline by ensuring `quartobot resolve` runs before pandoc.

Command-line reference

The full CLI is summarized in Table 2. Each subcommand has detailed help available via `quartobot <subcommand> --help`.

Subcommand	Purpose
`resolve`	Scan project sources for citation keys, resolve each via `citekey_to_csl_item`, write CSL-JSON to `references.json`. Invoked by Quarto's pre-render hook; also directly callable.
`init`	Scaffold the citation pipeline into a Quarto project. Idempotent.
`use github-ci`	Layer GitHub Actions workflows for render, pull-request preview, and (optionally) per-commit permalinks.
`scan`	Inventory citation keys across a project without resolving them. CI-lint surface.
`validate`	Static checks on `_quarto.yml` and the project layout. CI-lint surface.
`reconcile`	Resolve collisions between `references.bib` and `references.json` with explicit modes (`prefer-bib`, `prefer-resolved`, `fail-on-collision`).
`versions`	Generate the `/versions/` index page on `gh-pages` listing tagged releases and open pull-request previews.
`mcp`	Run a stdio Model Context Protocol server exposing the resolver for in-band use by AI authoring agents.

Use cases

Discussion

Software availability

- **Source repository:** github.com/seandavi/quartobot
- **Package index:** `quartobot` on PyPI

- **Documentation:** seandavi.github.io/quartobot
- **License:** MIT
- **Archived release:** Zenodo DOI to be minted at submission.
- **Latest version at submission:**

Data availability

This article describes a software tool; no primary research data are generated. The two production case studies referenced in the Use Cases section have their own data availability statements in their respective publications. The reproducibility artifacts for those case studies — the source `.qmd` files, the committed `references.json` bibliographies, the rendered outputs, and the GitHub Actions workflow history — are publicly available in the repositories cited under each case study. The present manuscript is itself a quartobot project; its source, including the committed `references.json` produced by `quartobot resolve`, is available at .

Author contributions

- **Sean Davis** — Conceptualization, Methodology, Software, Investigation, Writing — Original Draft, Writing — Review & Editing, Project Administration.

AI usage statement

Portions of this manuscript were drafted with the assistance of large language model coding agents (Anthropic Claude, operating through Claude Code) used as a literate-programming and drafting tool. AI assistance was used during outlining, prior-work surveying, structural drafting of the Implementation and Operation sections, and the development of the Discussion. All scientific claims, design decisions, citation choices, and final wording were reviewed and edited by the human authors. quartobot itself was developed by the human authors with AI assistance in implementation and documentation.

Funding

This work was supported by the National Cancer Institute under the NCI Information Technology for Cancer Research award U24CA289073 (NCI Bioconductor) and the University of Colorado Cancer Center Support Grant P30CA046934.

Acknowledgments

The authors thank the manubot project and its maintainers for eight years of resolver expertise embodied in `citekey_to_csl_item`, on which quartobot depends; the Quarto team at Posit for the Quarto Manuscripts project type that made this integration tractable; and early users of quartobot in the Bioconductor and CFDE communities whose feedback shaped the tool.

References

"Code Execution with MCP: Building More Efficient AI Agents \ Anthropic." n.d. Accessed May 29, 2026. https://www.anthropic.com/engineering/code-execution-with-mcp.

"MCP Servers Use 35x More Tokens Than CLI Tools — And Reliability Drops to 72% on Hard Tasks." 2026. May 9. https://www.mindstudio.ai/blog/mcp-servers-35x-more-tokens-cli-tools-reliability-benchmark/.

Sadani, Anuj, and Deepak Kumar. 2026. *Tool Attention Is All You Need: Dynamic Tool Gating and Lazy Schema Loading for Eliminating the MCP/Tools Tax in Scalable Agentic Workflows*. 2604.21816. arXiv. https://arxiv.org/abs/2604.21816.