SPECMINE: A Large-Scale Corpus of Spec-Driven Development Artifacts

Shyam Agarwal, Bogdan Vasilescu
Carnegie Mellon University
shyamaga@andrew.cmu.edu, vasilescu@cmu.edu

***Abstract*—Spec-Driven Development (SDD) is a fast-emerging practice in which a structured natural-language specification, written by a developer, or (more often) drafted by an AI tool and then curated by the developer, drives an AI coding agent's implementation. A wave of tooling (GitHub Spec Kit [3], OpenSpec [4], AWS Kiro [5], and dozens of others) has appeared since 2025, yet the artifacts these tools produce have never been studied at scale. We present SPECMINE, a corpus that captures SDD in public GitHub repositories through two censuses: a broad census of `spec.md`/`specs.md` files covering most tools (470,795 files across 73,030 repositories, attributed to 17 named tools), and a Kiro census of its distinct `requirements/design/tasks` layout (98,574 files across 12,910 repositories). Each spec is enriched with full repository metadata, complete commit history, and parsed document structure. How a spec becomes code is itself an open question, so for 11 tools we sweep every pull request that touches a spec in their repositories with at least ten stars, capturing 5,992 such PRs across 581 repositories with their changesets. That makes the simplest workflow, spec and implementation changing together in one PR, directly observable, and a census-wide index of 2,421,323 typed references (1.28M to code files, 863k to sibling documents, 152k to PRs, 62k refs, 43k branches, 22k issues) gives a second, independent link from spec to code. SPECMINE lets the community study, for the first time, *how software is specified in the age of AI agents*.**

I. HIGH-LEVEL OVERVIEW

A. Why spec-driven development, and why now

Spec-Driven Development (SDD) tools help a developer capture what to build as one or more Markdown documents that an AI agent then uses to generate and modify code. Some tools keep this in a single specification file; others, such as Spec Kit and Kiro, split it into separate requirements, design, and task-list documents. The spec itself can be human-written, but is often augmented or generated by AI from a short prompt, brief, or product requirements document, then modified by a human. The practice barely existed before 2025; it now spans thousands of repositories and a rapidly growing set of competing tools. Because the spec, rather than the code, is becoming the primary artifact developers write, review, and maintain, it is an increasingly important unit of study, but there is no dataset that captures these artifacts, tells which tool produced them, or shows how they relate to the code that ships.

Recent MSR Challenges have looked at neighbouring facets of AI-assisted development: DevGPT (MSR 2024) [1] at developer–ChatGPT *conversations* and AIDev (MSR 2026) [2] at *agent-authored* pull requests. But both study the model's output. SPECMINE is the missing *intent* layer: the specification of *what* to build (whether a developer writes it or drafts it with an AI tool and refines it), captured across 18 SDD tools (17 named plus Kiro), and, on a subsample of mature repositories, followed into the pull requests that touch it.

B. What the data set contains

We capture SDD artifacts across public GitHub repositories in four parts, obtained through the code-search and REST APIs:

1) **The broad census.** Nearly every SDD kit stores its specification as a `spec.md`/`specs.md` file, so a census of that filename recovers the bulk of the ecosystem. Each kept file carries the full GitHub repository object (stars, license, language, topics, timestamps), file/commit provenance, and an attribution to the tool that produced it (17 named tools such as OpenSpec, Spec Kit, and Conductor), assigned by a path fingerprint, the directory layout each kit generates. A manual check of 30 specs spanning ten named tools found the attribution correct in every case: each spec sits in its tool's directory and matches that tool's published template.
2) **The Kiro census.** AWS Kiro is among the most adopted SDD tools but is invisible to a `spec.md` search: it stores `requirements.md`, `design.md`, and `tasks.md` under `.kiro/specs/`. We census that layout separately, recovering 98,574 artifacts across 12,910 repositories.
3) **The pull requests that change specs.** Across the eleven named tools with the clearest and most widely adopted spec-driven workflows, we sweep every repository with at least ten stars (949 repositories, 963 repository–tool targets, since a repository can adopt more than one tool) and capture every pull request that touches a spec file with its changeset, each file flagged spec or code, so a single PR shows whatever code changed alongside the spec. This yields 5,992 spec-touching pull requests across 581 of those repositories. We exclude the caffeine.ai app-generator (auto-generated single apps) and unattributed ad-hoc specs, neither a deliberate SDD process. Because every row carries live GitHub identifiers, the same procedure extends to more repositories, which we intend to do in later snapshots. How specs actually become code is an open question (Appendix B).
4) **The traceability index.** Around each spec sit pointers to pull requests, issues, and the files it should change. We assemble these from the spec text, its commit messages, its OpenSpec `tasks.md`, and its folder's git tree, and ship them

as a census-wide index of 2,421,323 references, each typed by relation and provenance and rolled up onto the spec it came from. For OpenSpec, whose `tasks.md` names code paths directly, we go further and resolve every task-to-code reference against the repository's git tree at the anchoring commit (435,401 references, each recording whether its task was checked off and whether the named file existed at that commit). This is a second channel from spec to code, independent of co-change: when a spec's commit references PR #123 it points at an implementation even if that PR touches no spec file, and a task naming a file that never appeared is evidence of a spec never implemented.

Every spec is further enriched with its full commit history and 39 parsed structural features (heading tree, code/table/diagram counts, requirement-template markers such as EARS [6] and Gherkin, and quality signals such as unfilled placeholders and `TODO`s). Attribution and enrichment are reversible and idempotent.

C. Summary statistics

This paper describes SPECMINE v1.0, the July 2026 snapshot. Nearly all specs (99.7%) were first committed in 2025 or later, matching when these tools emerged, and 92% in 2026, so the corpus captures the practice from its birth. At a glance it offers **scale** (470,795 specifications), **breadth** (73,030 repositories across 17 named tools, plus 12,910 Kiro repositories), and **depth** (every spec carrying full commit history and 39 structural features, 266,230 OpenSpec change artifacts, and, for 581 repositories, a curated layer of 5,992 spec-touching pull requests with their changesets). caffeine.ai leads by repository count but is an app *generator* whose repositories are largely auto-generated single apps; among developer-adopted tools the leaders are Kiro, Spec Kit, and OpenSpec. Table I gives summary counts; Table III (Appendix E) gives the full per-tool breakdown of all 17 named tools, the 4 path buckets, and the Kiro census.

TABLE I
SUMMARY STATISTICS.

Quantity	Count
Spec files (broad census)	470,795
distinct repositories	73,030
distinct owners	44,521
Named SDD tools (broad census)	17
Kiro artifacts (separate census)	98,574
distinct repositories	12,910
Spec-file commits	780,335
Files with content + structural features	468,307
Spec-touching pull requests (11 tools swept, subsample)	5,992
repositories with $\geq$1 such PR	581
repositories swept ($\geq$10 stars, 11 tools)	949
that also modify code in the same PR	81.2%
per-file diff rows captured	348,141
OpenSpec change artifacts (proposal/design/tasks)	266,230
Typed references (traceability index)	2,421,323
OpenSpec task$\rightarrow$code refs resolved on the tree	435,401
Licensed repositories	28,698
Repositories with $\geq$100 stars	923
Curated dataset size (uncompressed)	14.7 GB

II. INTERNAL STRUCTURE

The data are stored in a relational (MySQL) schema and are also released as flat CSV/Parquet exports plus a JSONL of spec contents. The broad census is the spine (`spec_files`, keyed on a 16-hex digest of the file's GitHub blob URL, `file_url_sha16`); the Kiro census and the PR layer attach to it by `repo_name`.

Around this spine sit its satellites: `spec_file_commits` (per-file commit history) and `spec_content_features` (the 39 features) keyed on `file_url_sha16`; `kiro_files` (the Kiro census); the per-tool ⟨`tool`⟩`_prs` and ⟨`tool`⟩`_pr_files` (the PR layer, keyed on `repo_name`, flagging each changed file `is_spec`/`is_code`); `openspec_artifact_files` and `openspec_code_refs` (OpenSpec change artifacts and their resolved task-to-code references); and `spec_links` with `repo_trees` (the traceability index). Fig. 1 is the entity-relationship diagram and Table II a condensed data dictionary; Appendix A describes every released table, and the complete dictionary (all 57 tables, 815 columns, with types and provenance notes) ships as `DATA_DICTIONARY.md`.

How both censuses are built (adaptive size partitioning around GitHub's 1,000-result cap, with idempotence and determinism checks), the documented *not-a-spec* filter and its funnel (822,901 rows $\rightarrow$ 575,633 distinct $\rightarrow$ 470,795 kept), what the filename census is made of and how to draw a maturity line through it, the scope and sampling of the PR layer (a subsample, not a census), and the co-change heuristic that treats a spec-and-code PR as implementing that spec (together with its limits) are all detailed in Appendix B.

III. HOW TO ACCESS

- **Obtain.** Each snapshot (from v1.0) ships on Zenodo with a citable DOI as a MySQL dump, CSV/Parquet exports, and a JSONL of spec contents; a GitHub mirror carries the schema, loader scripts, an example Jupyter/Colab notebook, and a 500-repository sample.
- **Recommended use.** Load the dump into MySQL/DuckDB, or read the Parquet directly with `pandas`/`polars`: for instance *"which specs change in the same PR as code?"* is a single join between ⟨`tool`⟩`_prs` (`touches_code=1`) and its `pr_files`. Because every identifier is a live GitHub URL, participants can *bring their own data*, re-fetching any row or joining external sources (issues, CI logs, registries).
- **Skills.** SQL and basic `pandas` suffice, with no special hardware for the Parquet layer; a free GitHub token is needed only to re-fetch or grow the corpus, and the full MySQL image needs about 14.7 GB (within Zenodo's per-dataset limit).

A. Licensing, ethics, and privacy

All artifacts come from public GitHub repositories, collected within the API terms of service. Every row keeps its

repository's license (28,698 carry a recognized SPDX license), so participants can filter to license-compatible subsets; redistribution follows each repository's terms.

The release ships public handles but not private contact details. Author and committer logins and numeric IDs are kept: they are public, a repository's slug already carries its owner's, and they are what makes authorship analysis possible (bot detection, newcomer-versus-maintainer, the developer–agent division of labour). Commit emails, avatar URLs, and the remaining profile fields are dropped before release; a participant who needs them can fetch them from GitHub directly, under GitHub's terms and their institution's approval rather than ours. Commit messages and spec text are authored prose that may name people, and we do not rewrite them. We ask participants to avoid deanonymization and to keep personal data out of published results beyond aggregate analysis.

IV. Research Questions

Making the specification a first-class unit of study, alongside the tool that produced it and the code it drives, opens agendas that a code-side dataset cannot reach. We group representative questions below, each tied to the layer that enables it. Each is an empirical study, and most cut across several tool families.

1) **Adoption and diffusion of spec-driven development.** *Enabled by the cross-tool census (18 SDD tools) over the 2024–2026 emergence window.*
   - Who adopts SDD (newcomers or experienced maintainers, which languages, domains, and team sizes), and how does adoption diffuse across ecosystems?
   - How do the competing tool families grow, coexist, or displace one another, and do their document templates converge toward a shared structure?
2) **Anatomy and quality of specifications.** *Enabled by raw content plus 39 structural features across 468,307 specs.*
   - What forms do specs take (EARS, Gherkin, user stories, free prose), and can a measurable notion of spec quality (completeness, testability, ambiguity, unfilled placeholders) be defined and validated?
   - How much of a spec is reused template boilerplate versus genuinely project-specific content, and does that ratio differ by tool or author?
3) **The spec–code relationship.** *Enabled by two independent channels: the PR layer (5,992 spec-touching PRs across 581 repositories, with complete, spec/code-flagged changesets) and the traceability index (2,421,323 references from spec text, commit messages,* `tasks.md`*, and folder trees, with 435,401 task-to-code references resolved against the repository tree).*
   - *Where does the implementation of a spec live?* Co-change in a single PR is the obvious first guess and is common in our sample, but other workflows are equally plausible: a spec merged on its own and implemented in one or several later PRs, an implementation that arrives before the spec is written down, or a spec that is never implemented at all. Can these patterns be recognized and told apart at scale, how prevalent is each, and does the mix differ by tool, team size, or repository maturity? This is the question behind the heuristic of Appendix B, and we regard it as open.
   - How often is a spec kept in sync as the code evolves, and when the two drift apart, which side moves first?
   - A spec names the files it expects to change. How often does that code actually exist? Resolving 435,401 OpenSpec task-to-code references against the tree at the anchoring commit makes the gap between what a spec declares and what the repository contains measurable: what predicts it, and does it widen as specs get longer?
   - Does the presence or quality of a spec correlate with PR size, review effort, or merge outcome?
4) **Human–AI collaboration around specs.** *Enabled by authorship, commit cadence, and template signals in the age of coding agents.*
   - Can human-authored and agent-generated specs be distinguished, and how is authorship of a single spec shared between developer and agent?
   - Do higher-quality specs predict smoother downstream implementation (fewer follow-up fixes), and how does review differ when the artifact under discussion is a spec rather than code?
5) **Lifecycle, evolution, and abandonment.** *Enabled by 780,335 commits of full per-spec history and lifecycle roles (living / proposed / archived).*
   - What are the churn and half-life of a spec, and how often are specs abandoned mid-flight (open tasks that never close, placeholders never filled), and what predicts abandonment?
   - Because every prior state is reconstructable from commit history, can spec-driven workflows be studied longitudinally without the data-leakage pitfalls of snapshot datasets?

V. Data Availability

SpecMine is the dataset for the MSR 2027 Mining Challenge. Version 1.0 (the July 2026 snapshot) is released in three tiers, so a user can start at whatever scale suits the study:

- **Full dataset (Zenodo).** The complete corpus is archived on Zenodo with a citable DOI (`10.5281/zenodo.22102779`) as a MySQL dump plus flat CSV/Parquet exports and a JSONL of spec contents.
- **Parquet mirror (Hugging Face).** A per-table Parquet mirror of every released table is published at https://huggingface.co/datasets/ShyAgarwal/specmine for direct use with `pandas`/`polars`/DuckDB, with no need to load the dump.
- **GitHub mirror and sample.** https://github.com/shyamagarwal13/specmine-official carries the schema (`schema.sql`), loader scripts, the complete data dictionary (`DATA_DICTIONARY.md`), an example Jupyter/Colab notebook, and a curated 500-repository sample. The sample is a faithful, reproducibly selected slice of every layer (28,583 specs, 18,585 Kiro artifacts, 5,992 spec-touching pull requests with their 288,267 per-file

changesets, and 261,032 traceability references) that clones and queries in minutes.

Every identifier in the release is a live GitHub value (repository slug, blob URL, pull-request number), so any row can be re-fetched or joined to external sources. All artifacts are drawn from public repositories, each carrying its own license (Sec. III-A); the SPECMINE compilation is released under CC BY 4.0. Later snapshots will be published as new versioned Zenodo records, each with its own DOI, so results stay reproducible against a fixed version.

REFERENCES

[1] MSR 2024 Mining Challenge, "DevGPT: a dataset of developer–ChatGPT conversations," 2024. [Online]. Available: https://github.com/NAIST-SE/DevGPT

[2] MSR 2026 Mining Challenge, "AIDev: a dataset of agent-authored pull requests," 2026. Preprint: https://arxiv.org/abs/2507.15003

[3] GitHub, "Spec Kit," 2025. [Online]. Available: https://github.com/github/spec-kit

[4] "OpenSpec," 2025. [Online]. Available: https://github.com/Fission-AI/OpenSpec

[5] Amazon Web Services, "Kiro," 2025. [Online]. Available: https://kiro.dev

[6] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (EARS)," in *Proc. IEEE Int. Requirements Engineering Conf. (RE)*, 2009, pp. 317–322.

APPENDIX A: SCHEMA AND ENTITY–RELATIONSHIP DIAGRAM

Figure 1 gives the entity–relationship diagram of the released schema; Table II is a condensed data dictionary of its core tables. The complete dictionary (all 57 tables and 815 columns) ships as `DATA_DICTIONARY.md`. Appendix B documents how the corpus is constructed and quality-controlled; Appendices C–E give the data-specific detail a participant needs to query the release directly.

The released tables, in full:

- `spec_files`: one row per spec file: the full repository object, file/commit provenance, tool attribution (`spec_tool`/`spec_role`), and raw content.
- `spec_file_commits`: the complete per-file commit history (author, committer, timestamps, message).
- `spec_content_features`: the 39 structural and requirement-template features per file.
- `kiro_files`: the Kiro `requirements`/`design`/`tasks` artifacts, with their kind and feature.
- ⟨`tool`⟩`_prs` and ⟨`tool`⟩`_pr_files`: for each of 11 tools, the spec-touching pull requests (state, merge, branch, line counts, and a `touches_code` flag) and their per-file changesets (each file flagged `is_spec`/`is_code`).
- `openspec_artifact_files` and `openspec_code_refs`: for the OpenSpec ecosystem, the full change proposals, designs, and task lists with parsed task counts, plus the task-to-code references they contain, resolved against the repository tree (`task_done`, `exists_at_anchor`).
- `spec_links`: the census-wide index of typed references, assembled from spec text, commit messages, OpenSpec `tasks.md`, and folder git trees (code files, sibling docs, PRs, issues, branches), by relation and provenance; `repo_trees` holds the git tree that references resolve against.

APPENDIX B: CONSTRUCTION AND QUALITY CONTROL

Because GitHub code search caps any query at 1,000 results and reports an unreliable `total_count`, both censuses are built by *adaptive size partitioning*: file-size ranges are recursively split until each leaf is safely below the cap, and each leaf is paged until a genuinely empty page (short pages, which GitHub returns silently under rate-limiting, are retried rather than treated as the end). We verified the procedure is idempotent (re-running adds $\approx$0 files) and deterministic (independent fetches converge on the same set), and we transparently document the residual limitation: the reachable set may be a strict subset of all matching files.

The sweep stored 822,901 rows across its size partitions, which reduce to 575,633 distinct files once deduplicated on the blob URL; of these, 470,795 survive a documented *not-a-spec* filter that removes files inside package caches (97,757) or vendored dependency trees (376) and specs first committed before 2024 (6,705). Filtered rows are retained with the flag and the reason that fired, so the decision can be audited and reversed. The size-partition plans for both censuses ship as audit trails.

a) What the corpus is made of: A filename census of a practice this young inevitably mixes serious projects with tutorials, template clones, and one-off experiments: only 923 of the 73,030 repositories have 100 stars or more. We do not pre-filter this for participants, because where the line falls depends on the study, but we ship what is needed to draw it: fork, template, and archived flags on every repository object, a content hash on every file so that copies of the same starter template can be collapsed, and per-spec `is_tiny`, `has_lorem`, and unfilled-placeholder flags that identify scaffolding nobody ever filled in. Reporting which filter was applied should be part of any study built on SPECMINE. The two censuses are keyed independently and may overlap: a repository can run Kiro and also carry a `spec.md`.

b) Scope of the PR layer: The PR layer is a subsample, not a census. Within each of the eleven tools with the clearest spec-driven workflows we ran discovery on every repository with at least ten stars, the selection rule for the sample: 963 repository–tool targets over 949 distinct repositories (a repository can adopt more than one tool), dominated by OpenSpec (524) and Spec Kit (305). Of these, 581 contain at least one spec-touching pull request and enter the released layer; the rest committed their specs directly to a branch or had no qualifying PR. Results from this layer describe those repositories in detail and should not be extrapolated to the full 73,030-repository census without care. Within a sampled repository the sweep is exhaustive for the PRs it targets: every PR that touches a spec file is captured with its changeset, complete up to GitHub's 3,000-file-per-PR API cap, at which a small number of very large PRs are truncated. Discovery is commit-based (a PR enters when a spec file's commit is associated with it through GitHub's `/commits/{sha}/pulls`), while the changeset is the

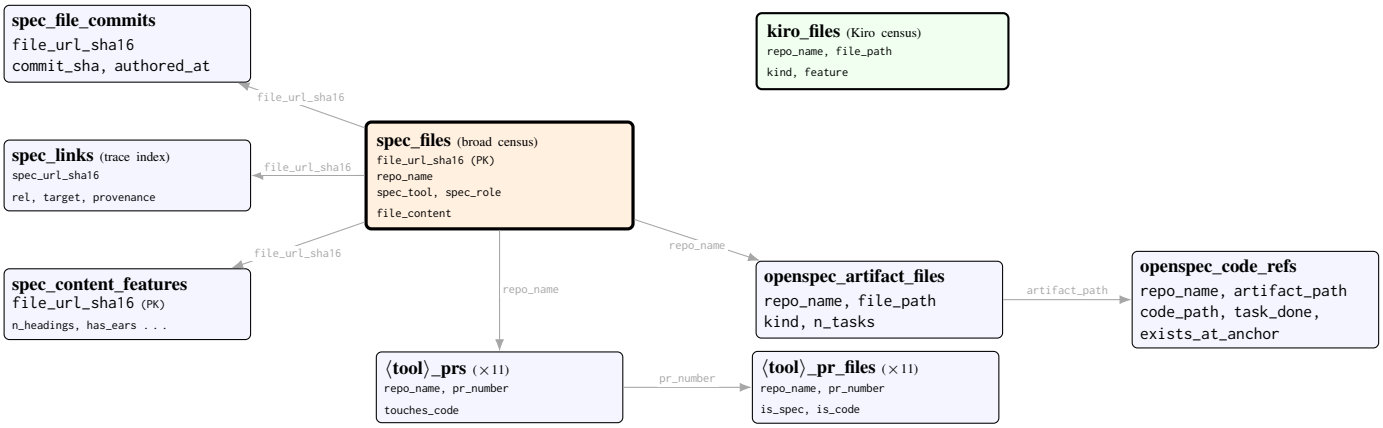

Fig. 1. Entity-relationship diagram of the released SPECMINE schema. The `spec_files` spine (orange) is the broad census; its file-keyed satellites join on `file_url_sha16`. The Kiro census (green) is captured independently and keyed on `repo_name`. The per-tool PR tables (11 tools, shown once) record the pull requests that change a spec and their full changesets, flagging each file as spec or code. `spec_links` and `openspec_code_refs` form the traceability index: each spec's links to code, PRs, issues, and sibling docs, the latter resolved against the repository tree. Plumbing tables (discovery queues, partition plans, raw staging) are omitted.

TABLE II
CONDENSED DATA DICTIONARY OF THE CORE RELEASED TABLES. THE FULL DICTIONARY (ALL TABLES AND COLUMNS, WITH TYPES AND PROVENANCE NOTES) SHIPS AS `DATA_DICTIONARY.md`.

Table	Rows	Purpose / join key
`spec_files`	470,795	broad census spine: full repo object, file/commit provenance, tool attribution, raw content. *PK* `file_url_sha16`.
`spec_file_commits`	780,335	Complete per-file commit history (author, committer, timestamps, message). *FK* `file_url_sha16`.
`spec_content_features`	468,307	39 structural and requirement-template features per spec. *PK* `file_url_sha16`.
`kiro_files`	98,574	Kiro census under `.kiro/specs`; `kind` = requirements / design / tasks. Key `repo_name, file_path`.
⟨`tool`⟩`_prs` (×11 tools)	5,992	Pull requests that modify a spec: state, merge, branch, line counts, `touches_code`. Key `repo_name, pr_number`.
⟨`tool`⟩`_pr_files` (×11)	348,141	Per-file changeset of each spec-touching PR, each file flagged `is_spec` / `is_code`. *FK* `repo_name, pr_number`.
`openspec_artifact_files`	266,230	OpenSpec change proposals, designs, and task lists, with parsed task counts. Key `repo_name, file_path`.
`openspec_code_refs`	435,401	Task-to-code references parsed from OpenSpec `tasks.md` and resolved against the tree at the anchoring commit (`task_done`, `exists_at_anchor`). *FK* `repo_name, artifact_path`.
`spec_links`	2,421,323	Census-wide index of typed references (from spec text, commit messages, `tasks.md`, and folder trees): code, sibling docs, PRs, issues, branches, by relation and provenance. *FK* `spec_url_sha16`.

PR's net diff, so the two occasionally disagree: five of the 5,992 PRs carry no spec file in their diff, the truncated ones plus two whose spec commit sits in shared or ancestor history. By construction, a PR that touches *no* spec file never enters the sweep, even when it implements one; recovering those means enumerating a repository's full PR history from the API and filtering it, which the released repository list and PR numbers make straightforward but which we have not done ourselves.

c) The co-change heuristic and its limits: How a spec actually becomes code is not directly observable, and we do not settle it here. The PR layer instead applies a deliberate heuristic that participants can inspect: *a pull request that edits a spec file and also changes source files is treated as implementing that spec*. Co-change is common in the sample (81.2% of spec-touching PRs also modify code, in the same reviewed changeset), so it is a reasonable place to start, but it is an assumption, not ground truth. A PR may touch a spec for unrelated reasons or implement it only partially, and a workflow that merges a spec on its own and implements it in later PRs is invisible to co-change altogether. The complete changeset of every spec-touching PR, with branch names, authors, and creation and merge timestamps, lets participants tighten or replace the heuristic (by file-path proximity, commit and merge timing, task-list checkbox transitions, or branch and title references to a named spec), and the traceability index covers cases co-change misses: the PRs and issues a spec names are recorded whether or not those PRs touch a spec file. Reconciling the two channels, and establishing how far

each one reaches, is itself one of the research questions this dataset is meant to open up (Sec. IV, item 3).

Appendix C: Controlled Vocabularies

The values each categorical column takes, with row counts in SpecMine v1.0.

`spec_files.spec_role` (lifecycle role): archived (156,936), na (137,779), living (73,253), feature (49,370), change_proposal (41,806), config (11,651). *living*/*change_proposal*/*archived* are the OpenSpec lifecycle; *feature* is a Spec Kit numbered feature; *config* is a tool scaffold file; *na* is everything else.

`spec_links.rel` (what a reference points to): code (1,278,828), sibling (863,445), pr (151,563), ref (62,267), branch (42,970), issue (22,139), anchor (111).

`spec_links.provenance` (how the reference was found): tasks (1,194,660, from an OpenSpec `tasks.md`), tree (937,815, from listing the repository git tree), commit_msg (231,284, from the spec's commit messages), branch_header (42,970, the `Feature Branch:` line), content (14,594, parsed from the spec body).

`kiro_files.kind`: requirements (32,972), design (32,123), tasks (31,592), bugfix (1,539), other (348).

`openspec_artifact_files.kind`: proposal.md (93,307), tasks.md (93,061), design.md (79,459), plan.md (245), research.md (78), data-model.md (41), quickstart.md (39).

PR-file flags (⟨`tool`⟩`_pr_files`, per changed file): `is_spec`, `is_code`, `is_specdir` (booleans); `change_status` (added / modified / removed). The PR-level `touches_code` is 1 when a PR changes at least one code file outside the spec directory.

Appendix D: Content and Structural Features

Each spec's Markdown is parsed into 39 features (`spec_content_features`, one row per spec, keyed by `file_url_sha16`), grouped below. Count columns are integers; marker columns are 0/1.

- **Size:** `n_bytes`, `n_lines`, `n_words`.
- **Structure:** `n_headings`, `max_heading_depth`, `n_code_fences`, `n_mermaid`, `n_tables`, `n_links`, `n_images`, `n_list_items`, `n_checkboxes`, `n_checked`.
- **Requirement syntax (counts):** `n_shall`, `n_gherkin`, `n_scenario_blocks`, `n_delta_headers`.
- **Section / requirement markers (0/1):** `has_ears`, `has_gherkin`, `has_user_story`, `has_requirements`, `has_acceptance_criteria`, `has_user_scenarios`, `has_non_goals`, `has_out_of_scope`, `has_overview`, `has_success_criteria`, `has_edge_cases`, `has_open_questions`, `has_dependencies`, `has_purpose`, `has_mandatory_marker`.
- **Quality / completeness:** `n_needs_clarification`, `n_todo`, `is_tiny`, `has_unfilled_placeholder`, `has_lorem`.
- **Language:** `lang` (detected natural language), `content_family` (template family, e.g. openspec / speckit).

Appendix E: Full Tool Attribution

Table III lists every attribution category in the broad census: all 17 named tools, the 4 interpretable path buckets (groupings of unattributed files, not tools), and the separate Kiro census, with file and repository counts. caffeine.ai leads by repository count but is an app *generator* whose repositories are largely auto-generated single apps; among developer-adopted tools the leaders are Kiro, Spec Kit, and OpenSpec.

TABLE III
Every attribution category in SpecMine v1.0 (kept spec set).

Tool / bucket	Files	Repos
OpenSpec	274,955	8,926
Spec Kit	54,640	10,619
caffeine.ai	14,262	13,749
Conductor	7,752	936
Trae	2,004	434
opencode	1,963	66
dot_specs	1,619	250
Claude Code	1,351	833
Agent OS	1,278	237
MoAI-ADK	1,156	63
specweave	963	4
kitty_specs	840	67
codex	550	69
zenflow	234	64
gsd	196	181
cursor	80	65
kilocode	3	3
Path buckets (not tools):		
adhoc_other	38,575	10,239
specs_generic	31,725	3,301
docs_embedded	19,437	8,655
root_bare	17,212	17,104
Kiro (separate census)	98,574	12,910