

jss-lint: Automated Style Checking for *Journal of Statistical Software* Manuscripts

Manuel Koller

Abstract

Manuscripts submitted to the *Journal of Statistical Software* (JSS) must satisfy a comprehensive style guide covering preamble macros, semantic markup, citation conventions, capitalization, code formatting, bibliography completeness, and typography. Style violations that reach peer review cost editors, reviewers, and authors additional revision rounds for problems a machine can find. We present **jss-lint**, an open-source, deterministic style checker for JSS manuscripts. A single Rust core implements 62 rules in 16 categories for `.tex`, `.Rnw`, and `.Rmd` sources and ships through four channels: a command-line tool, a Python package, an R package, and a browser version compiled to WebAssembly that processes manuscripts entirely locally. The rules were written entirely by large language models under human direction and hardened by an evaluation-improvement loop against a pinned corpus of 254 JSS-format manuscripts. After 115 recorded iterations, precision estimated over 20,060 adjudicated violation instances is 97.2%; because most of those labels are model-issued, we treat this as a label-set estimate and bracket it with the human-only subset (93.8%) and a labeler-error-propagated band (85.4%–91.2%), and measured recall against a hand-annotated ground-truth corpus is 80.7% per instance. The methodology — specification-driven development, adversarially checked AI labeling, and per-rule precision gates with documented exemptions and rule retirement — is described in detail and offers a replicable template for similar tools. This manuscript was checked by **jss-lint** before submission; the replication script reproduces every number, table, and listing shown.

Keywords: Style checking, L^AT_EX, linting, reproducibility, Rust, Python, R, large language models.

1. Introduction

The *Journal of Statistical Software* reviews both a software artifact and the manuscript describing it.¹ Its style guide ([Journal of Statistical Software 2026b](#)) specifies requirements well beyond what general L^AT_EX experience suggests: dedicated preamble macros such as `\Plainauthor` and `\Address`, semantic markup macros (`\proglang`, `\pkg`, `\code`), **natbib** citation conventions ([Daly 2010](#)), capitalization rules that differ between titles, section headings, and captions, code-formatting and line-width conventions, and completeness requirements on the BibT_EX database. Each requirement is individually reasonable; collectively they are hard to hold in mind, and manuscripts that violate them generate editorial back-and-forth that neither improves the science nor teaches the author much beyond the specific correction.

¹The software presented here is an independent, third-party project; it is not affiliated with, endorsed by, or connected to the *Journal of Statistical Software*, its editors, or its publisher.

Our position is simple: the peer review process should not be burdened with problems that can be detected — and often repaired — mechanically. Statistical computing already follows this principle for code and results. The line of tooling from **Sweave** (Leisch 2002) through **knitr** (Xie 2014) and **SASweave** (Lenth and Højsgaard 2007) exists to guarantee mechanically that a manuscript’s code and output agree, precisely so that humans need not verify concordance by eye. **jss-lint** extends the same reasoning from a manuscript’s computational content to its editorial form.

This paper describes **jss-lint** version 1.1.0, a deterministic style checker for JSS manuscripts. Its contributions are threefold. First, the tool itself: 62 rules across 16 categories, each traceable to a clause of the style guide, applied by a single Rust engine that ships as a command-line tool, a Python package, an R package, and a fully client-side browser application — with an auto-fix mode that repairs the mechanical subset of violations. Second, a measured account of its accuracy: precision evaluated over 254 JSS-format manuscripts and 20,060 adjudicated violation instances, and recall evaluated against a hand-annotated ground-truth corpus — numbers of a kind that linters rarely report. Third, a development methodology: the tool was built in an intensive human–AI collaboration, and we document where specification-driven development helped, where it did not, and how an evaluation-improvement loop with per-rule precision gates turned a plausible-looking rule set into a trustworthy one.

Everything reported here is reproducible: a single replication script regenerates every statistic, table body, and terminal listing in this manuscript from the repository’s pinned evaluation artifacts, and — self-referentially — verifies that the manuscript itself passes **jss-lint** with zero violations.

Section 2 reviews the style requirements and gives a motivating example. Section 3 describes the architecture. Section 4 summarizes the rule catalogue. Section 5 presents the development methodology, including the role of AI assistance. Section 6 reports the empirical validation. Section 7 demonstrates the tool, Section 8 compares it with existing tools, and Section 9 concludes.

2. The style requirements and a motivating example

2.1. What the style guide asks for

The JSS style guide and the accompanying author instructions (Journal of Statistical Software 2026b,a) constrain a manuscript at several levels. At the document level, the manuscript uses the **jss** document class, provides an abstract, keywords, and at least one author address, and declares its bibliography as a BibTeX database. Plain-text companions (`\Plaintitle`, `\Plainauthor`, `\Plainkeywords`) are required whenever the corresponding rich field contains markup, because they feed the PDF metadata. At the markup level, programming languages, software packages, and code fragments are wrapped in `\proglang`, `\pkg`, and `\code` respectively; `\code` is preferred for inline code, though `\verb` is also allowed. Citations use **natbib** commands, and brackets-within-brackets citation constructs — a `\cite` command already wrapped in parentheses — are not permitted; the guide asks for `\citep` or, when parentheses are shared with other text, `\citealp`. Titles are set in title style, section headings and captions in sentence style, and keywords in sentence case. Code shown in the manuscript must fit the text width, use spaces around operators and after commas, and keep commentary in the prose

rather than inside verbatim blocks. House style adds smaller conventions: *e.g.*, and *i.e.*, carry a trailing comma, *p* value and *t* statistic are typeset with an italicized symbol and a tie, expectation and probability operators use the class-provided macros, and cross-references spell out “Section”, “Figure”, and “Equation”.

Some submission requirements are out of the linter’s scope by design: **jss-lint** checks editorial form, not that the manuscript compiles under pdfL^AT_EX or that its results reproduce. Book and software reviews, which have their own class options and provisions, are likewise supported only through the document-class rule.

2.2. A motivating example

The fragment below is a complete, intentionally non-compliant manuscript; it is shipped with the replication materials as `examples/demo.tex` and drives every demonstration in this paper.

```
\documentclass[article]{jss}
\author{Ann Author\and Ben Butcher}
\title{A Package for Robust Estimation in R}
\abstract{We present a package for robust estimation.}
\keywords{Robust Statistics, R}
\begin{document}
\section{Our New Method}
The package is implemented in R and builds on lme4
(\cite{bates2015}). The estimator attains a smaller p-value
than the classical approach; see the table below.
\end{document}
```

Running `jss-lint examples/demo.tex` reports seven violations, shown in Figure 1 exactly as the tool prints them. Four are mechanical: the authors are separated by `\and` instead of `\And` (**JSS-STRUCT-005**), the citation wraps `\cite` in parentheses (**JSS-CITE-003**), the language name in prose lacks `\proglang` (**JSS-MARKUP-001**), and the hyphenated *p* value spelling should use an italicized symbol and a tie (**JSS-OPER-001**). The remaining three require judgment: the missing `\Address` (**JSS-PRE-002**), the title-style section heading “Our New Method” (**JSS-CAP-002**), and the title-cased keyword entry “Robust Statistics” (**JSS-CAP-004**). Each report carries the source position, the style-guide clause it enforces, and a concrete suggestion; the mechanical four are repaired automatically by the auto-fix mode shown in Section 7. The fragment is not exhaustively caught: the bare package name **lme4** (Bates, Mächler, Bolker, and Walker 2015) in the prose should be wrapped in `\pkg`, but **lme4** is absent from the markup rule’s recognized-package set and so goes unflagged — a concrete instance of the recall boundary quantified in Section 6.3.

3. Design and architecture

3.1. Design principles

Four principles, fixed early and maintained throughout, shape the implementation.

Parse, do not pattern-match. Rules operate on an abstract syntax tree (AST) of the manuscript, not on raw lines. The original Python implementation parses L^AT_EX with **pylatexenc** (Faist

\$ jss-lint examples/demo.tex				examples/demo.tex	
Line:Col	Severity	Rule	Message	Suggestion	
1:1	error	JSS-PRE-002	Preamble defines \Address{} with author affiliation and contact (see §2.1 Preamble)	Add \Address{...} in the preamble with author affiliation and e-mail.	
2:19	warning	JSS-STRUCT-005	\author{} separates authors with \And or \AND (not lowercase \and) (see §2.2 Structure)	Separate authors with \And (inline) or \AND (line break), not lowercase \and.	
5:1	warning	JSS-CAP-004	\Keywords{} is comma-separated and in sentence case (see §6.2 Capitalisation)	Use sentence case for keywords: do not capitalise words after the first in an entry (proper names and abbreviations excepted). Use sentence style: capitalise only the first word (proper names remain capitalised).	
7:1	warning	JSS-CAP-002 (medium conf.)	Section titles are in sentence style (first word capitalised; others lowercase except proper names) (see §6.2 Capitalisation)	Wrap 'R' in \proglang{R}.	
8:31	warning	JSS-MARKUP-001 (medium conf.)	Programming-language names in prose are wrapped in \proglang{} (see §4.1 Markup)	Wrap 'R' in \proglang{R}.	
9:2	warning	JSS-CITE-003	Avoid bracket-in-bracket citation forms like \cite{...}; use \citep{...} instead (see §3.2 Citations)	Citation inside parens: replace \cite{...} with \citep{...}, or use \citealp{...} when additional text shares the parens.	
9:53	warning	JSS-OPER-001	Symbol-plus-noun constructs like p-value and t-statistic are typeset as \$p\$-value and \$t\$-statistic (tie, no hyphen) (see §4.4 Mathematical notation)	Replace 'p-value' with '\$p\$-value' (italicized symbol, tie).	

Figure 1: Author-mode report for the demonstration manuscript, headed by the command line that produced it (fixed-width terminal capture, regenerated by the replication script).

2025) and BibT_EX with **bibtexparser** (Noack *et al.* 2026); the Rust engine re-implements the same two-pass document model. AST-level analysis is what lets a rule distinguish a bare language name in prose from one inside a comment, a verbatim block, a citation key, or a macro definition — the distinctions that separate a useful checker from a noisy one. **.Rnw** (Sweave/knitr) and **.Rmd** (R Markdown) sources are supported by extracting their L^AT_EX and code components with source positions preserved, and multi-file manuscripts are handled by recursively resolving `\input` and `\include` directives.

Deterministic rules only. The same input always produces the same violations, byte for byte. AI plays a large role in the *development* of the rules (Section 5) but no role at all in their execution — an intentional contrast to AI-based writing assistants, and the property that makes the tool usable as a CI gate and its precision measurable.

Fail soft. A parse error in one file degrades that file’s analysis and is itself reported as a violation; it never aborts the run. Real manuscripts contain unbalanced braces in verbatim environments and other legacy constructs, and a checker that dies on them never gets to report anything useful.

Two audiences. Author mode reports each violation with its position and a fix suggestion. Reviewer mode aggregates the same findings into a per-category compliance summary (Section 7) so that editors and reviewers can see at a glance whether a submission is ready for content review.

3.2. One engine, four channels

jss-lint began as a pure Python package, distributed on the Python Package Index as **jss-style-checker** with the import module `texlint` and the `jss-lint` command-line entry point; it remains the reference implementation and parity oracle. Version 1.1.0 moves the engine — parsing, all 62 rules, auto-fixes, and every output renderer — into a single Rust (Rust Project Developers 2026) crate, **jsslint-core**, compiled into four distribution channels:

- **jsslint-cli**, a standalone native binary exposing the full command-line interface;
- **jsslint**, a Python wheel on the Python Package Index built as a **PyO3**-based extension module (PyO3 Project Developers 2026) that preserves the original Python API (distinct from the pure-Python **jss-style-checker** reference package above);
- **jsslintr**, an R package built with **extendr** (ExtendR Project Developers 2026), so that the community JSS serves most directly can run the checker from the language it already uses;
- **jsslint-wasm**, a WebAssembly build via **wasm-bindgen** (Rust and WebAssembly Working Group 2026) powering an in-browser checker. The WebAssembly module links no networking code at all, so an unpublished manuscript checked in the browser provably never leaves the author’s machine — a property commercial checking services cannot offer.

The port is validated by differential testing rather than by trust: every rule and every renderer is required to produce byte-identical output to the Python engine, enforced by per-rule parity tests and a differential sweep over the full evaluation corpus (Appendix C prints the regenerated

Surface	Description
<code>jss-lint</code>	Lint files; author or reviewer mode
<code>--fix, --dry-run</code>	Apply or preview deterministic repairs
<code>explain</code>	Per-rule documentation with examples
<code>init</code>	Interactive project bootstrap
<code>report</code>	One-page editor conformance report
<code>diff</code>	Compare violations between two runs
<code>lsp</code>	Language Server Protocol server
VS Code extension	Diagnostics and quick fixes while editing
GitHub Action	Reusable CI gate with SARIF upload
<code>--output</code>	Terminal, JSON, HTML, or SARIF
<code>--crossref</code>	Opt-in online DOI verification

Table 1: The **jss-lint** tool surface. All checking is local; only the explicitly opt-in `--crossref` mode performs network requests.

Measure	Value
Corpus size	255 manuscripts
Total wall time (parse + lint)	67.3 s
Throughput	3.8 manuscripts/s
Median per manuscript	231 ms
95th percentile	662 ms
Maximum	1.54 s

Table 2: Linting performance over all 255 scanned directories (the 254 pinned corpus members plus the auxiliary multi-version study directory), **Python** reference engine, single thread, parse and rule execution excluding interpreter startup, one timed pass per manuscript, measured on Linux 6.12.76-linuxkit, aarch64, 10 logical CPUs; the median and 95th percentile summarize the per-manuscript distribution.

evidence for the demonstration manuscript). This byte-parity contract is also what allows the empirical results of Section 6, produced with the **Python** engine, to stand for the **Rust** engine verbatim. The **Python** implementation remains in the repository as the executable specification and parity oracle.

Beyond the core linter, the ecosystem summarized in Table 1 integrates the checker into editing and review workflows.

3.3. Performance

Table 2 reports single-threaded timings of the **Python** reference engine over every scanned directory; checking a typical manuscript takes a fraction of a second, so the tool is comfortably interactive both at the command line and behind the language server.

Category	Rules	Reviewed	Precision
Preamble	8	120	100.0%
Structure	6	175	100.0%
Markup	4	3,656	91.1%
Citations	3	616	95.3%
References	6	5,883	97.4%
Bibtex	5	218	100.0%
Naming	2	205	100.0%
Capitalization	3	821	98.7%
Typography	4	354	100.0%
Abbreviations	1	40	100.0%
Code style	3	3,206	99.3%
Code width	1	520	100.0%
Operators	4	1,204	98.3%
Crossrefs	7	2,299	99.9%
House style	3	743	100.0%
Project	2	0	—

Table 3: Rule categories with the number of adjudicated violation instances and measured precision at the pinned evaluation iteration. Per-rule figures appear in Appendix B.

4. The rule catalogue

Version 1.1.0 ships 62 active rules in 16 categories. Every rule is defined in a single machine-readable catalogue that records its identifier, category, severity, the style-guide clause it enforces (with a citation anchor into the guide), whether it is auto-fixable, and its measured-precision confidence tier. The catalogue is the single source of truth: the engine’s rule table, the **explain** documentation, and Appendix A of this paper are all generated from it, so the paper cannot claim a rule that the tool does not ship. Table 3 summarizes the categories together with their measured precision from the evaluation of Section 6.

Three catalogue mechanisms deserve mention. First, rules whose firing is a purely syntactic fact — line length, presence of a required macro, a label with no matching reference — are marked *deterministic* (34 of 62); Section 5.5 explains why this flag matters for evaluation. Second, rules carry a confidence tier reflecting their measured precision, and users can set a floor with `--min-confidence`. Third, rules can be *retired*: the catalogue records 4 rules that were implemented, evaluated, and deliberately removed — for example a caption-case rule whose residual false positives required domain knowledge no syntactic gate could supply (Section 6.4). Retired identifiers are never reused, and retirement is documented rather than silent, because a rule that quietly disappears would erode trust in the ones that remain.

Conditional requirements are encoded as such. For example, the `\Plain...` companions are required only when the corresponding rich field actually contains markup ([JSS-PRE-003](#), [JSS-PRE-007](#), [JSS-PRE-008](#)); a plain `\title` needs no `\Plaintitle`.

5. Development methodology: Specifications, AI, and an evaluation loop

jss-lint was developed in an intensive collaboration between one human author and large language models of the Claude family (Anthropic 2026). The division of labor was stark, and we state it exactly because the accountability question deserves a precise answer: the models wrote every line of code and every sentence of this manuscript. Approximately 750 of the repository’s more than 900 commits carry an AI co-author trailer, and the author’s own direct commits are a handful of conveniences. The human contribution was directional and evaluative: setting requirements; issuing corrective instructions whenever the tool or the process drifted from the style guide or from intent (Section 5.3); hand-annotating the recall ground truth of Section 6.3; and working the interactive label-review queue of Section 5.5, retaining final authority over every disputed evaluation label. Notably, the human author never reviewed the code — only its measured behavior. The verification apparatus this paper describes — determinism, 1,915 tests, byte-parity between two independent engines, per-rule precision gates, measured recall, and a replication script — is not incidental to that choice; it is what makes the choice defensible, and we believe this substitution of measurement for line review is the load-bearing lesson of the project. We document the methodology in some depth because the combination of specification-driven development, test discipline, and a measurement loop is what made AI authorship trustworthy rather than merely fast, and because several of its lessons contradicted our expectations.

5.1. Specification-driven development for cross-cutting features

Each of the 17 major features — the core engine, the rule catalogue, .Rnw/.Rmd support, SARIF output, auto-fix, the language server, multi-file resolution, the evaluation harnesses, and so on — went through an explicit specify–clarify–plan–tasks–implement workflow on a dedicated branch: a written specification with acceptance criteria, recorded clarification questions, an implementation plan, and a task breakdown, all committed alongside the code. For work of this shape the overhead paid for itself. Specifications forced interface decisions to happen in prose review, where they are cheap, rather than in code review; the recorded clarifications repeatedly prevented re-litigating settled questions; and an AI collaborator given a written spec stays on task over a multi-day feature in a way one given a chat instruction does not.

Two forms of test discipline complemented the specifications. Rule logic is developed test-first and held to complete branch coverage, which for rules is not bureaucracy: a rule’s value lies almost entirely in its edge-case behavior, and the test suite — 1,915 tests at the pinned release — is where that behavior is pinned down. The Rust port added the second form, differential testing against the Python engine as described in Section 3.2.

5.2. Where specification-driven development did not work

Our project constitution now states the boundary explicitly: specifications are reserved for cross-cutting work, and mechanical changes bypass the workflow. We arrived at this rule empirically. The bulk of the project’s commits are small, evidence-driven rule adjustments produced by the evaluation loop of Section 5.4 — add a guard for citation keys, stop scanning program output for operator spacing, normalize whitespace in label matching. Early attempts to route such fixes through the specification workflow produced documents that restated a

one-line diff at ten times the length and went stale immediately. A candid methodological note: an earlier draft of this paper listed “specification overhead for small changes” as a lesson we had merely observed; in fact we initially misdescribed those fixes as spec-driven when they were not, and the correction — direct commits inside the measurement loop, specifications for everything cross-cutting — is itself the lesson.

The second boundary is knowledge, not size. Some rules bottomed out in judgments no specification could settle in advance — whether a capitalized word in a caption is a proper noun, whether `\texttt` wraps code or emphasis. For those, the productive cycle was not specify-then-implement but measure-then-decide, with the human supplying domain rulings the AI could not (Section 6.4).

5.3. AI assistance in implementation

The AI collaborators wrote all of the rule implementations, tests, and evaluation tooling, working from the specifications and from labeled false-positive evidence. Three patterns proved particularly effective. First, *investigation before code*: when a rule’s precision stalled, a subagent was dispatched to read every labeled false positive and return a written taxonomy of failure clusters before any fix was attempted; fixes then targeted named clusters, and their effect was measured per cluster. Second, *adversarial defaults*: AI-proposed fixes were required to state which true positives they might sacrifice, because our early experience matched the general finding that model-proposed guards are overconfident — several proposed false-positive guards would have silenced more true violations than false ones, a trade only visible because the loop measures both directions (Section 6.3). Third, *human course-correction at the requirements level*: the author intervened not in the code but in what the rules should mean, whenever measured behavior diverged from the style guide or from the journal’s published practice. Two adjacent examples: the keyword-case rule initially demanded a leading capital, but the journal’s template defines keyword “sentence case” by example as fully lowercase — its canonical list literally reads “comma-separated, not capitalized” — so the rule was relaxed to follow the template, the higher authority; conversely, the capital-after-colon requirement, on which both capitalization styles agree, was found to be enforced for reference titles but not for section titles, and — published practice not being an authority — was tightened for consistency. Both corrections entered as instructions and flowed through the standard loop, tests first.

This manuscript itself was drafted and revised by the AI collaborators under the same direct-and-verify workflow as the code; the human author wrote none of its text and reviewed its claims and rendered output rather than its source. The byline, however, follows the prevailing convention of scholarly publishing: an AI system cannot take responsibility for a work, so the listed author is the human who does, and the AI systems are credited in the acknowledgments.

5.4. The evaluation-improvement loop

The central methodological claim of this paper is that a linter’s rule set is an empirical object: whether a rule is *correct* is decided by running it against real manuscripts and adjudicating what it finds. The loop that operationalizes this has six steps: (1) extend or re-fetch the pinned corpus; (2) scan the corpus and persist every violation; (3) route new violations to labelers (Section 5.5); (4) record a snapshot of per-rule true/false-positive counts into an append-only history database; (5) when a rule underperforms, investigate its false-positive

clusters and implement a targeted fix; (6) re-scan and record the next iteration. The corpus grew in batches — from an initial 50 manuscripts to the final 254 — so that early rule fixes were re-tested against every later corpus expansion. At the pinned release the history database records 115 iterations.

The loop is governed by a per-rule precision gate: every rule must reach 90% measured precision on the full corpus. Aggregate precision alone is not an acceptable target, because a high-volume accurate rule can mask an inaccurate one. The gate has exactly one documented exemption mechanism: a rule whose sub-gate precision reflects an inherent ambiguity (not a tuning gap) may be declared exempt, with the rationale recorded in a reviewed configuration file; 1 rule is currently exempt (Section 6.4). Rules that can neither pass nor justify exemption are retired.

5.5. Label provenance

Every precision figure in this paper is a ratio of labeled violations, so the labels’ provenance matters as much as the rules’. Of the 20,073 labels behind the pinned evaluation, 17,715 (88.3%) are model-issued, 1,773 are human, and 585 are deterministic auto-labels; the headline precision is thus predominantly an estimate against machine labels, which is why we audit those labels and propagate their error into the interval reported in Section 6.2. AI labeling uses two locally run open-weight models — Bonsai-8B (an MLX build) and Qwen3-30B-A3B — routed per rule through a committed routing table, because the two models’ agreement profiles are strongly complementary. Both are benchmarked against a gold set of 1,410 independently human-labeled violations: Qwen3 agrees with the human verdict on 82.7% of the gold set and Bonsai on 82.9%, and on the true-positive verdicts specifically their error rates are 6.9% and 13.7% respectively — the two numbers that drive the adjusted-precision band. A skip list excludes rules on which AI labeling proved unreliable — those go directly to interactive human review — and uncertain or disputed labels are adjudicated by the human at the manuscript source.

Three audit findings shaped this design, and we report them because they generalize. First, at one recorded iteration the AI labelers were found to contradict their own earlier verdicts on re-presented violations, which prompted the gold-set benchmark and periodic re-adjudication sweeps; treating AI labels as fallible measurements to be audited — rather than as ground truth — is, in our view, a requirement for using them at all. Second, for the 34 mechanically decidable rules (line length, macro presence), routing firings to an AI labeler only *manufactured* disagreement: two such rules sat visibly below their true precision for weeks because a language model second-guessed objective facts. Such rules are now auto-labeled by construction, tagged distinctly, and kept out of the model gold set. Third, a gold-set decontamination sweep on 2026-07-06 found two hundred sixty-eight model-generated adjudications recorded under human-namespace reviewer tags — including one workflow in which a model had overridden prior human verdicts. These were re-namespaced to their true machine origin and excluded from the gold set, which is what forced us to treat “human-labeled” as itself an audited category rather than a trusted one. The general lesson: use an AI judge only where judgment is actually required, and audit the provenance tags too, not just the verdicts.

5.6. What transfers

For readers building a comparable tool, the retrospective compresses to five points. (1) Build

the measurement loop before writing many rules; every later decision was cheaper because it could be measured. (2) Write rules narrow and widen them under measurement. This policy optimizes precision first, and its cost is exactly the false negatives a recall corpus makes visible — which is why we built one (Section 6.3) rather than reporting precision alone. (3) Encode requirements as data, not prose: the machine-readable catalogue is what keeps the tool, its documentation, and this paper consistent. (4) Give AI collaborators specifications and measured evidence, not instructions and trust; audit AI labels against human gold data. (5) Retire rules publicly. A checker earns trust by what it refuses to claim as much as by what it catches.

6. Empirical validation

6.1. Corpus construction

The precision corpus is pinned by a manifest of 254 members: 232 vignettes of published JSS papers shipped in CRAN package archives and 16 standalone manuscripts collected from public repositories — the real manuscripts — together with 6 synthetic fixtures retained from early development. The synthetic fixtures are counted separately and never folded into a “real manuscript” total. One further directory, an auxiliary multi-version study of a single manuscript preserved in four revision stages, is scanned but left out of the pinned manifest, so 255 directories are scanned in total while every precision figure is reported over the 254 pinned members. By source format, 199 are `.Rnw`, 26 are `.Rmd`, and 29 are plain $\text{\LaTeX}$. The manifest records an immutable source URL and SHA-256 hash per member, so every evaluation run sees byte-identical inputs; it contains no duplicate manuscripts and no same-paper format doubles (an earlier, larger corpus used during development contained near-duplicates and was replaced by the pinned, deduplicated manifest).

Because CRAN vignettes describe R software exclusively, the repository-sourced portion was hand-curated to include 14 manuscripts describing software in other languages (Python, C++, Julia, Zig). These manuscripts exercise the rules on prose and code conventions from outside the R ecosystem — and immediately earned their place: their installation-guide sections exposed two over-firing patterns that R vignettes never trigger, both fixed and re-measured within the loop. The corpus remains R-weighted, and per-rule precision on non-R manuscripts is measured with correspondingly less data; we scope our claims accordingly.

One selection effect deserves emphasis, because it cuts against the headline. The 232 CRAN vignettes — the bulk of the corpus — are versions of papers that JSS has already published, and therefore already copy-edited: the style violations that reached review were, by construction, either caught by a human editor or judged acceptable. The tool’s actual use case is the opposite population, unedited submissions, where both the base rate and the difficulty of violations are almost certainly higher than this corpus shows. Our precision estimate is therefore measured on the easier distribution, and recall on submissions may be lower than Section 6.3 reports. The [JSS-CODE-003](#) episode of Section 6.4 is in-paper evidence: a false-positive class that no published R vignette in the corpus could surface appeared the moment the corpus was extended with a genuinely new manuscript genre. New submissions will move the boundary again.

6.2. Precision

At the pinned iteration (`new-additions-through-resolve-input`, recorded 2026-07-19), the corpus yields 20,060 adjudicated violation instances: 19,496 true positives and 564 false positives, an overall precision of 97.2% with no unlabeled instances outstanding. Of the 57 rules that fired at least once, 56 meet the 90% gate and 45 have no recorded false positive at all; per-rule figures are listed in Appendix B. Reported precision is computed over *live* labels — violations the current tool version actually produces on the current corpus — not over the union of everything any historical version ever reported; an earlier accounting bug of exactly that kind made two already-fixed rules look permanently imprecise, and the fix (tracking each violation’s last-seen run) is part of the harness.

The 97.2% figure is a ratio of *labels*, and this paper’s own thesis is that labels are fallible measurements, so we apply that thesis to our own headline rather than exempt it. Three readings bracket the truth. The label-set estimate is 97.2% (all labels at face value). Restricted to the 1,773 human labels alone, precision is 93.8%. Between the two sits a labeler-error-propagated band: taking the human and auto labels at face value but discounting the AI-issued true positives by the model true-positive-error rate e measured on the gold set, adjusted precision is

$$\frac{\text{TP} - e \cdot \text{TP}_{\text{AI}}}{\text{TP} + \text{FP}},$$

which for e ranging over the worst and best measured model error (Bonsai 13.7%, Qwen3 6.9%) yields 85.4%–91.2%. This band is deliberately conservative: the gold set is enriched in hard, skip-listed, and disputed cases, so its measured TP-error overstates the error on the full label population, and propagating it therefore leans the band toward a lower bound. The honest one-line summary is that 97.2% is the label-set estimate, 93.8% is the human-only subset, and 85.4%–91.2% is the error-propagated bracket; we quote all three and never let 97.2% stand alone as “the tool’s precision.”

6.3. Recall

Precision alone cannot distinguish a useful checker from a timid one: a tool that never fires has perfect precision. Recall is measured against a separate ground-truth corpus of 17 manuscripts in which violations were hand-annotated (“planted”) at the source level, partly by seeding known violations and partly by exhaustive manual annotation of the manuscripts’ existing violations, cross-checked against — but never auto-accepted from — linter output. Deterministic rules’ annotations are pre-filled mechanically; judgment rules were annotated and repeatedly re-audited by the human author, and annotation errors discovered during audits (including stale plants left by corpus-fetch changes and mislabeled edge cases) were corrected at source with the correction history retained.

At the pinned recall run, the tool recovers 1,587 of the 1,967 planted violations across 53 rules: a per-instance (micro) recall of 80.7% and a per-rule average (macro) recall of 88.4%. We report per-rule recall only for the 26 rules with at least 10 plants — 8 of which recover every plant — and pool the thinner rules into a single row in Appendix B, because a recall estimate on two plants is noise. The two figures differ substantially, and the micro figure is the honest headline: the macro average is buoyed by many thin, easy rules, whereas the largest gaps concentrate in exactly the high-volume rules whose false-positive analysis also revealed inherent ambiguity. The four lowest-recall thick rules, worst first, are the house-style comma

after e.g.,/i.e., ([JSS-HOUSE-001](#)), inline-code markup ([JSS-MARKUP-003](#)), the expectation and probability operators ([JSS-OPER-004](#)), and the missing-DOI advisory ([JSS-REFS-003](#)); each recovers between roughly half and two-thirds of its plants, with the exact figures in Table 9 of Appendix B. These are the rules discussed next.

6.4. Two case studies and the accuracy ceiling

Retirement: the caption-case rule. A rule requiring captions in sentence style reached only moderate precision after five successive fix rounds. The residual false positives were capitalized words that are proper nouns only to a domain reader — method names, R class names, place names. Every syntactic gate the loop tested either left the false positives or sacrificed true positives at a poor ratio; the honest options were a curated whitelist (unbounded maintenance) or retirement, and we retired the rule. The catalogue records it, and this paper reports it, because a silent removal would misrepresent the tool’s history.

Exemption: single-letter language names. The rule wrapping programming-language names in `\proglang` is accurate for multi-character names but faces irreducible ambiguity on the single-letter names of R and C, which also serve as panel labels, statistical factors, and parts of disease names and other compounds. After the structurally fixable clusters were eliminated, the residue is inherent; the rule is exempt from the gate at its measured precision, with the rationale recorded in the reviewed exemption file, rather than silently tolerated or retired outright — it remains far too useful on the unambiguous cases. A parallel scope narrowing governs the capitalization rules, whose sentence style asks for a capital after both a colon and a hyphen: the tool enforces only the colon, since a literal capital-after-hyphen would demand compound modifiers like `Model-Based`, so that half is documented as a deliberate exception rather than enforced.

The same boundary appears on the recall side: the rule with the most planted violations, inline-code markup ([JSS-MARKUP-003](#)), misses plants that no syntactic signal can separate from their false-positive twins — `\texttt` wrapping an ordinary word versus a function name, path-shaped strings that are prose in one manuscript and code in another. We consider this the tool’s accuracy ceiling: past it, improvement requires domain knowledge, and the design response is confidence tiers and human review rather than more guards.

The loop does not end. Writing this manuscript extended the corpus in kind and promptly surfaced a new false-positive class: the code-spacing rule ([JSS-CODE-003](#)), which reads hyphens in inline code as unspaced subtraction, flagged every command-line option this paper documents — R vignettes, the corpus’s dominant genre, simply never typeset strings like `--fix` or `cargo install jsslint-cli`. The fix (skipping token sequences shaped like flags, names, and paths) went through the standard cycle: implemented against failing tests in both engines, re-scanned, and source-audited — it silenced six corpus violations previously labeled true positives, of which five turned out to be label debt of exactly this class and one a genuine single-letter subtraction, the same inherent token ambiguity the rule has always accepted. In candor, that re-adjudication of the six rows was performed by the same automated workflow that authored the fix, so it was not an independent check; a human spot-check of those six specific rows, recorded in the follow-ups, has since confirmed the re-adjudication: five false positives, one true positive. The episode is the methodology in miniature: heuristics over tokens have a measurable boundary, new manuscript genres move it, and the loop prices every guard in both currencies before it ships.

A related limitation of the tool’s self-check surfaced while preparing this manuscript’s appendix. The captions-below-floats rule ([JSS-TYP0-004](#)) inspects `figure` and `table` floats but does not see `longtable`, so a top-caption on a `longtable` passes the linter silently — the appendix tables of this paper were corrected by hand, not by the tool. The blindness is disclosed here and tracked as a follow-up rather than papered over, for the same reason we report retirements.

6.5. Reproducibility

Every number, table body, and listing in this paper is generated — none is typed by hand. A generator script queries the pinned evaluation state (recall run 2026-07-19 10:30:02 UTC, ground-truth corpus hash 70951d5371df0734, and the precision iteration labeled `new-additions-through-resolve-input`) together with the rule catalogue, corpus manifest, and benchmark records, and emits the macros and table bodies this manuscript inputs; a drift check fails the build if the committed values differ from regeneration. The regeneration gate `regenerate.sh` performs the full sequence — regenerate, compare, lint this manuscript with `jss-lint` itself (any violation fails the build), and recompile the PDF. The replication script `replicate.sh`, shipped with the submission materials, needs no repository: it installs the released tool at the version stated here, re-runs every command demonstrated in this paper, and verifies that each output matches the listings and the reviewer-mode table as printed. The history database is append-only, so the pinned queries return byte-identical results indefinitely, and the evaluated software itself is archived: every tagged release is deposited on Zenodo (<https://doi.org/10.5281/zenodo.21441932>), independent of the repository’s continued existence. What replication establishes is *consistency* — that every figure faithfully transcribes the pinned evaluation state and that the manuscript is self-compliant — not the *validity* of the underlying labels; label validity is a separate question, which is why the labels are shipped for independent audit in `eval/labels-export.csv.gz` together with the gold-set export, so a reader can re-adjudicate any figure at its source.

7. Using jss-lint

The engine reaches users through the four channels of Table 4, one per ecosystem. A naming note, because the channels differ: the original pure-Python reference implementation is published as `jss-style-checker` and installs a `jss-lint` command; the Rust-backed wheel is `jsslint`; the native binary crate is `jsslint-cli` and installs a `jsslint` executable. The channels are published to their public registries as version 1.1.0 — the crates on crates.io, both Python packages on the Package Index, the WebAssembly build on npm, the editor extension on the VS Code Marketplace and Open VSX, and the R package `jsslintr` on CRAN. All components are released under the MIT license. Development takes place in the repository at <https://github.com/kollerma/jss-style-checker>, and each tagged release is archived on Zenodo — the concept DOI 10.5281/zenodo.21441932 resolves permanently to the latest archived version.

From R, the ecosystem JSS serves most directly, the package works directly on the files of a manuscript project: `jsslint()` lints every `.tex/.Rnw/.Rmd/.bib` file that `jss_files()` discovers in the working directory (or any explicitly named files), the `summary()` method aggregates the result by rule and category, and `jssfix()` applies the safe automatic fixes in place, with a `dry_run` mode that previews them as a diff. The session below was run in a

Channel		Install	What you get
CLI binary		<code>cargo install jsslint-cli</code>	the native <code>jsslint</code> executable with the full command line, <code>--fix</code> , <code>explain</code> , and the language server
Python	(reference)	<code>pip install jss-style-checker</code>	the <code>texlint</code> module and the <code>jss-lint</code> command
Python	(Rust wheel)	<code>pip install jsslint</code>	a PyO3 extension with the same Python API
R		<code>install.packages("jsslintr")</code>	<code>jsslint()</code> and <code>jssfix()</code> over project files, plus the low-level <code>render()</code>
Browser / npm		<code>jsslint-wasm</code>	a client-side checker that needs no installation and no network

Table 4: Distribution channels of **jss-lint**. All install commands are live.

directory holding the manuscript of Section 2.2:

```
R> library("jsslintr")
R> res <- jsslint()
R> summary(res)
```

```
JSS style check (journal "jss"): demo.tex
Compliance: 62.5% -- 7 issues (1 error, 6 warnings; 4 auto-fixable)
```

Files:

file	error	warning	info	total
demo.tex	1	6	0	7

Issues by rule:

rule_id	severity	issues	fixable	guide_section
JSS-CAP-002	warning	1	0	§6.2 Capitalisation
JSS-CAP-004	warning	1	0	§6.2 Capitalisation
JSS-CITE-003	warning	1	1	§3.2 Citations
JSS-MARKUP-001	warning	1	1	§4.1 Markup
JSS-OPER-001	warning	1	1	§4.4 Mathematical notation
JSS-PRE-002	error	1	0	§2.1 Preamble
JSS-STRUCT-005	warning	1	1	§2.2 Structure

Categories:

category	status	rules_passed	issues
Preamble	FAIL	7/8	1
Structure	FAIL	5/6	1
Markup	FAIL	3/4	1
Citations	FAIL	2/3	1

References	PASS	6/6	0
BibTeX	PASS	5/5	0
Naming	PASS	2/2	0
Capitalization	FAIL	1/3	2
Typography	PASS	4/4	0
Abbreviations	PASS	1/1	0
Code style	PASS	3/3	0
Code width	PASS	1/1	0
Operators	FAIL	3/4	1
Cross-references	PASS	7/7	0
House style	PASS	3/3	0
Project	PASS	2/2	0

```
R> jssfix("demo.tex", dry_run = TRUE)
```

```
--- demo.tex
+++ demo.tex
@@ -1,11 +1,11 @@
\documentclass[article]{jss}
-\author{Ann Author\and Ben Butcher}
+\author{Ann Author\And Ben Butcher}
\title{A Package for Robust Estimation in R}
\abstract{We present a package for robust estimation.}
\keywords{Robust Statistics, R}
\begin{document}
\section{Our New Method}
-The package is implemented in R and builds on lme4
-(\cite{bates2015}). The estimator attains a smaller p-value
+The package is implemented in \proglang{R} and builds on lme4
+\citep{bates2015}. The estimator attains a smaller  $p$ -value
than the classical approach; see the table below.
\end{document}
```

Dry run: 4 fixes would be applied to 1 file. Re-run with `dry_run = FALSE` to write.

A lower-level `render()` mirrors the other bindings' in-memory interface — file contents in, rendered report out — and a getting-started vignette walks through the workflow.

7.1. Author workflow

Running `jss-lint` on the manuscript of Section 2.2 prints the report of Figure 1: the seven violations as a table with positions, messages, and suggestions. The mechanical subset can be repaired directly; `--fix --dry-run` previews the repairs as a unified diff without touching the file, printed ahead of the unchanged violation table. The listing below shows the diff portion, with `--no-resolve` keeping the file name in the diff header relative rather than resolved to an absolute path:

```
$ jss-lint --fix --dry-run --no-resolve demo.tex
--- demo.tex
+++ demo.tex
@@ -1,11 +1,11 @@
\documentclass[article]{jss}
-\author{Ann Author\and Ben Butcher}
+\author{Ann Author\And Ben Butcher}
\title{A Package for Robust Estimation in R}
\Abstract{We present a package for robust estimation.}
\Keywords{Robust Statistics, R}
\begin{document}
\section{Our New Method}
-The package is implemented in R and builds on lme4
-(\cite{bates2015}). The estimator attains a smaller p-value
+The package is implemented in \proglang{R} and builds on lme4
+\citep{bates2015}. The estimator attains a smaller  $p$ -value
  than the classical approach; see the table below.
\end{document}
```

Applying `--fix` rewrites the file atomically and re-verifies the result. Fixes are only generated where they are provably safe; judgment calls (the section heading's case, the keyword list) are deliberately left to the author. Each rule documents itself, including the style-guide clause it enforces:

```
$ jss-lint explain JSS-CITE-003
JSS-CITE-003 (WARNING)
  Category: citations
  Authority: style_guide (#what-are-the-different-cite-citet-citep-commands-about)
  JSS guide: §3.2 Citations
  See: https://www.jstatsoft.org/about/submissions#citations

Avoid bracket-in-bracket citation forms like (\cite{...}); use \citep{...} instead.
```

Rule selection is controlled by a per-project configuration file (`.jss-lint.toml`) or by the `--ignore-rules` and `--min-confidence` options; suppressions are explicit and reportable, never silent.

7.2. Reviewer workflow and continuous integration

Reviewer mode aggregates findings into a pass/fail summary per category with an overall compliance percentage. Table 5 shows the summary for the demonstration manuscript, exactly as produced by the JSON output of `--mode reviewer` (the terminal renderer displays the same data as a formatted table).

The same interface serves automation. On a clean manuscript the tool prints nothing and exits 0, so it composes quietly in a pipeline; otherwise it exits 1 when any violation reaches the `--fail-on` severity and 2 when it cannot complete (bad arguments, an unreadable file). The `--fail-on` threshold defaults to `warning`, so info-severity advisories — the missing-DOI check, for instance — are reported but do not by themselves fail the build; a stricter gate

Category	Status	Applied	Passed
Preamble	FAIL	8	7
Structure	FAIL	6	5
Markup	FAIL	4	3
Citations	FAIL	3	2
References	PASS	6	6
BibTeX	PASS	5	5
Naming	PASS	2	2
Capitalization	FAIL	3	1
Typography	PASS	4	4
Abbreviations	PASS	1	1
Code style	PASS	3	3
Code width	PASS	1	1
Operators	FAIL	4	3
Cross-references	PASS	7	7
House style	PASS	3	3
Project	PASS	2	2
Overall compliance		62.5%	

Table 5: Reviewer-mode compliance summary for `examples/demo.tex`, transcribed from the tool’s JSON output and re-verified by the replication script.

sets `--fail-on info` (any finding fails), a looser one `--fail-on error` (only errors do). In continuous integration, the provided GitHub Action lints the manuscript on every push and uploads SARIF so violations annotate the diff; the exit status is the gate. While editing, the language server surfaces the diagnostics and safe fixes inside any LSP-capable editor, with a packaged extension for VS Code. For bibliography hygiene, the opt-in `--crossref` mode queries the Crossref REST API (Crossref 2026) to verify and fill in missing DOIs; it is the only network-touching feature and is excluded from the deterministic replication path.

8. Comparison with existing tools

General-purpose L^AT_EX linters — **chktex** (Berger 2026), **lacheck** (Thorup and Abrahamsen 2012), and the **textlint** ecosystem (azu *et al.* 2026) — check language-level hygiene: mismatched delimiters, spacing around punctuation, deprecated constructs. They know nothing of any journal’s requirements; running **chktex** on the corrected demonstration manuscript of Section 2.2 — the `--fix` output, which still carries the residual judgment items the auto-fixer deliberately leaves — produces advice about L^AT_EX idioms and none about JSS style. At the other end, commercial AI writing assistants such as **Writefull** (Digital Science 2026) give probabilistic language feedback: valuable for prose, but non-deterministic, unspecific to journal rules, and requiring manuscript upload to a third party. Table 6 summarizes the comparison. The comparison is not adversarial: the tools operate at different layers, and a JSS author can productively run **chktex** for L^AT_EX hygiene alongside **jss-lint** for journal compliance.

Two adjacent categories are worth naming even though neither targets JSS manuscript style.

	chktex	lacheck	textlint	Writefull	jss-lint
Journal-specific rules	–	–	–	–	yes
Deterministic	yes	yes	yes	–	yes
.Rnw/.Rmd aware	–	–	–	–	yes
BibTeX rules	–	–	–	–	yes
Auto-fix	–	–	partial	–	yes
Measured precision/recall	–	–	–	–	yes
Fully local	yes	yes	yes	–	yes

Table 6: Feature comparison with general L^AT_EX linters and AI-based writing assistance. “Measured precision/recall” means the project publishes accuracy figures against a real-manuscript corpus.

Within statistical computing itself, **lintr** (Hester *et al.* 2026) and **styler** (Müller and Walthert 2026) lint and format R source; they act on code rather than on manuscripts, but they establish the community’s own expectation that a linter and a formatter are ordinary parts of the workflow, and the methodological claim of this paper — that a rule set is an empirical object to be measured against real inputs — transfers directly from their setting to ours. On the publisher side, several journals and preprint systems run internal submission checkers for structural completeness and metadata. What distinguishes **jss-lint** is thus not that it stands alone — it does not — but that it publishes precision and recall against a real-manuscript corpus, which the general-purpose and publisher-side tools do not report and largely cannot, because measuring those quantities requires journal-specific ground truth that a language-level or metadata-level checker never constructs. To our knowledge, **jss-lint** is the first checker to encode a statistical journal’s style guide as an executable rule set and report its accuracy this way.

9. Summary and outlook

jss-lint makes the editorial form of a JSS submission checkable in the same sense that its computational results are reproducible: by a deterministic tool, before a human referee spends time on it. The rule set is not merely plausible — it is measured, with a label-set precision of 97.2% over 20,060 adjudicated instances (bracketed by the human-only 93.8% and the error-propagated 85.4%–91.2% of Section 6.2), 80.7% micro recall against hand-annotated ground truth, a documented exemption, and 4 public retirements. The development methodology — specifications for cross-cutting work, AI assistance under adversarial measurement, deterministic auto-labeling where judgment is not required, and per-rule gates — is, we believe, as much of a contribution as the tool, and the self-referential replication materials demonstrate both together: this manuscript passes the checker it describes, and one script regenerates everything shown here.

The style guide is not exhausted. Known checks we have not yet implemented include the Sweave header declaration for .Rnw-based manuscripts and sentence-style capitalization of table row and column headers; the catalogue’s future-work list tracks these. Beyond JSS, the rule engine is journal-agnostic — rules are data, and a catalogue for another journal’s guide is the intended extension path, now cheaper to distribute through the four-channel core. Our

broader hope is that measured, deterministic style checking becomes a normal part of the submission pipeline, so that editors, reviewers, and authors can spend their attention where it is actually needed.

Acknowledgments

Large language models of the Claude family (Anthropic) wrote the software, the tests, the evaluation tooling, and this manuscript, and labeled the majority of violations under the audited scheme of Section 5.5. The human author directed the work and corrected its course when it strayed, annotated the recall ground truth, reviewed the interactive label queue, and verified behavior rather than code — and takes sole responsibility for the software, the empirical results, and the text.

References

- Anthropic (2026). “Claude [Large Language Model].” Model versions Opus 4.6–4.8, Sonnet 4.6–5, Fable 5, URL <https://www.anthropic.com/claude>.
- azu, *et al.* (2026). “**textlint**: The Pluggable Linting Tool for Text and Markup.” URL <https://textlint.github.io/>.
- Bates D, Mächler M, Bolker B, Walker S (2015). “Fitting Linear Mixed-Effects Models Using **lme4**.” *Journal of Statistical Software*, **67**(1), 1–48. doi:10.18637/jss.v067.i01.
- Berger J (2026). “**ChkTeX**: A L^AT_EX Semantic Checker.” URL <https://www.nongnu.org/chktex/>.
- Crossref (2026). “Crossref REST API.” Accessed 2026-07-11, URL <https://api.crossref.org/>.
- Daly PW (2010). “Natural Sciences Citations and References.” URL <https://ctan.org/pkg/natbib>.
- Digital Science (2026). “**Writefull**: AI-Based Language Feedback for Academic Writing.” Accessed 2026-07-11, URL <https://www.writefull.com/>.
- ExtendR Project Developers (2026). “**extendr**: Rust Extensions for R.” Accessed 2026-07-11, URL <https://extendr.github.io/>.
- Faist P (2025). “**pylatexenc**: LaTeX Parsing and Encoding in Python.” URL <https://github.com/phfaist/pylatexenc>.
- Hester J, *et al.* (2026). “**lintr**: A Linter for R Code.” URL <https://CRAN.R-project.org/package=lintr>.
- Journal of Statistical Software (2026a). “Instructions for Authors.” Accessed 2026-07-11, URL <https://www.jstatsoft.org/pages/view/authors>.

- Journal of Statistical Software (2026b). “Style Guide for Authors.” Accessed 2026-07-11, URL <https://www.jstatsoft.org/style>.
- Leisch F (2002). “**Sweave**: Dynamic Generation of Statistical Reports Using Literate Data Analysis.” *Compstat 2002: Proceedings in Computational Statistics*, pp. 575–580. doi:[10.1007/978-3-642-57489-4_89](https://doi.org/10.1007/978-3-642-57489-4_89).
- Lenth RV, Højsgaard S (2007). “**SASweave**: Literate Programming Using SAS.” *Journal of Statistical Software*, **19**(8), 1–20. doi:[10.18637/jss.v019.i08](https://doi.org/10.18637/jss.v019.i08).
- Müller K, Walthert L (2026). “**styler**: Non-Invasive Pretty Printing of R Code.” URL <https://CRAN.R-project.org/package=styler>.
- Noack M, *et al.* (2026). “**bibtexparser**: A Python Library for Parsing BibTeX Files.” URL <https://github.com/sciunto-org/python-bibtexparser>.
- PyO3 Project Developers (2026). “**PyO3**: Rust Bindings for Python.” Accessed 2026-07-11, URL <https://pyo3.rs/>.
- Rust and WebAssembly Working Group (2026). “**wasm-bindgen**: Facilitating High-Level Interactions Between Rust and JavaScript.” Accessed 2026-07-11, URL <https://github.com/rustwasm/wasm-bindgen>.
- Rust Project Developers (2026). “The Rust Programming Language.” Accessed 2026-07-11, URL <https://www.rust-lang.org/>.
- Thorup KK, Abrahamsen P (2012). “**lacheck**: A Simple Syntax Checker for L^AT_EX.” URL <https://ctan.org/pkg/lacheck>.
- Xie Y (2014). “**knitr**: A Comprehensive Tool for Reproducible Research in R.” *Implementing Reproducible Computational Research*, pp. 3–32. doi:[10.1201/9781315373461](https://doi.org/10.1201/9781315373461).

A. The rule catalogue

Table 7 lists every active rule with its measured precision at the pinned evaluation iteration (“—” marks rules with no adjudicated firing on the corpus). The table is generated from the machine-readable catalogue; rule identifiers referenced in the text link [here](#).

Rule	Requirement	Precision
Preamble		
JSS-PRE-001	Document class must be jss with a valid class option (article, codesnippet, bookreview, softwarereview)	100.0%
JSS-PRE-002	Preamble defines <code>\Address{}</code> with author affiliation and contact	100.0%
JSS-PRE-003	When <code>\title{}</code> contains LaTeX markup, preamble also defines <code>\Plaintitle{}</code> with the markup-free form	100.0%
JSS-PRE-004	<code>\Abstract{}</code> is present and overrides the sentinel placeholder from jss.cls	100.0%
JSS-PRE-005	<code>\Keywords{}</code> is present and overrides the sentinel placeholder from jss.cls	100.0%
JSS-PRE-006	<code>\Plaintitle</code> , <code>\Plainauthor</code> , <code>\Plainkeywords</code> contain no LaTeX markup (PDF metadata must be plain text)	100.0%
JSS-PRE-007	When <code>\author{}</code> contains LaTeX markup, preamble also defines <code>\Plainauthor{}</code> with the markup-free form	100.0%
JSS-PRE-008	When <code>\Keywords{}</code> contains LaTeX markup, preamble also defines <code>\Plainkeywords{}</code> with the markup-free form	—
Structure		
JSS-STRUCT-001	Document ends with a summary / discussion section before the bibliography	100.0%
JSS-STRUCT-002	Acknowledgments section uses American spelling (not “Acknowledgements”)	100.0%
JSS-STRUCT-003	Appendix sections have proper titles instead of a bare “Appendix”	100.0%
JSS-STRUCT-004	References are declared via <code>\bibliography{}</code> rather than a hand-written thebibliography environment	100.0%
JSS-STRUCT-005	<code>\author{}</code> separates authors with <code>\And</code> or <code>\AND</code> (not lowercase <code>\and</code>)	100.0%
JSS-STRUCT-006	Appendix follows the bibliography with a <code>\newpage</code> (or <code>\clearpage</code>) separator	100.0%
Markup		
JSS-MARKUP-001	Programming-language names in prose are wrapped in <code>\proglang{}</code>	85.0%
JSS-MARKUP-002	Software-package names in prose are wrapped in <code>\pkg{}</code>	99.6%
JSS-MARKUP-003	Inline function, argument, command names, and R sentinel values are wrapped in <code>\code{}</code>	93.7%

(continued on next page)

Rule	Requirement	Precision
JSS-MARKUP-004	Section titles containing markup supply a plain-text shim via <code>\section[plain]{markup}</code>	100.0%
Citations		
JSS-CITE-002	First occurrence of a software package has a citation within the same paragraph	91.1%
JSS-CITE-003	Avoid bracket-in-bracket citation forms like <code>(\cite{...})</code> ; use <code>\citep{...}</code> instead	100.0%
JSS-CITE-004	Citations use natbib commands (<code>\cite</code> , <code>\citete</code> , <code>\citep</code> , <code>\citealp</code>) rather than hardcoded author-year text	100.0%
References		
JSS-REFS-001	BibTeX entries carry a year field so natbib author-year citations render correctly	100.0%
JSS-REFS-003	BibTeX entries include a doi field where one is available (advisory)	96.3%
JSS-REFS-004	BibTeX titles use JSS markup (<code>\proglang</code> , <code>\pkg</code> , <code>\code</code>) for language and package names	100.0%
JSS-REFS-005	Journal titles in BibTeX entries are not abbreviated	100.0%
JSS-REFS-006	BibTeX titles are in title style — loose heuristic (flags lowercase first word or unusual mixed case)	98.9%
JSS-REFS-007	Journal titles in BibTeX entries are in title case	100.0%
Bibtex		
JSS-BIBTEX-001	Every BibTeX entry has a non-empty citation key	—
JSS-BIBTEX-002	BibTeX citation keys are unique within the database	100.0%
JSS-BIBTEX-003	BibTeX entries carry the fields required for their entry type (article, book, inproceedings, ...)	100.0%
JSS-BIBTEX-004	Entries with 6+ authors use <code>\shortcites{}</code> or the short-names class option is enabled	100.0%
JSS-BIBTEX-005	No BibTeX field key is repeated within a single entry	100.0%
Naming		
JSS-NAME-001	Programming-language names use their canonical capitalization	100.0%
JSS-NAME-002	Publisher and journal names follow JSS conventions (e.g., “Springer-Verlag”, “The Annals of Statistics”)	100.0%
Capitalization		
JSS-CAP-001	<code>\title{}</code> is in title style (principal words capitalized)	100.0%
JSS-CAP-002	Section titles are in sentence style (first word capitalized; others lowercase except proper names)	98.4%

(continued on next page)

Rule	Requirement	Precision
JSS-CAP-004	<code>\Keywords{}</code> is comma-separated and in sentence case	100.0%
Typography		
JSS-TYPO-001	Figure and table captions end with a period	100.0%
JSS-TYPO-002	Figure / table captions avoid emphasis macros wrapping the whole caption (<code>\emph</code> , <code>\textbf</code> , <code>\textit</code> on full caption)	100.0%
JSS-TYPO-003	Tables do not use footnote-style annotations; annotations go in the caption	100.0%
JSS-TYPO-004	<code>\caption{}</code> appears after the figure / table content, not before	100.0%
Abbreviations		
JSS-ABBR-001	Abbreviations are in uppercase without periods or additional formatting	100.0%
Code style		
JSS-CODE-001	Verbatim / <code>CodeInput</code> blocks do not contain comments; comments belong in the surrounding LaTeX text	100.0%
JSS-CODE-002	R library() and data() calls quote their first argument	100.0%
JSS-CODE-003	Code samples use spaces around operators and after commas	99.1%
Code width		
JSS-WIDTH-001	Code input / output inside <code>Sinput</code> / <code>CodeInput</code> / <code>CodeOutput</code> environments fits within the configured column limit	100.0%
Operators		
JSS-OPER-001	Symbol-plus-noun constructs like p-value and t-statistic are typeset as <code>\$p\$~value</code> and <code>\$t\$~statistic</code> (tie, no hyphen)	100.0%
JSS-OPER-002	Transpose is typeset with <code>\top</code> rather than a superscript prime or literal T	98.8%
JSS-OPER-003	Display equations have no blank lines immediately before or after (use % to suppress paragraph breaks)	100.0%
JSS-OPER-004	Expectation / variance / covariance / probability use jss.cls shortcuts <code>\E</code> , <code>\VAR</code> , <code>\COV</code> , <code>\Prob</code>	96.6%
Crossrefs		
JSS-XREF-001	Figures and tables are referenced via <code>\ref{}</code> rather than by manual numbering	95.0%
JSS-XREF-002	Equation references prefer <code>Equation~\ref{...}</code> (capitalized) over bare (<code>\ref{...}</code>) or <code>\eqref{...}</code>	100.0%
JSS-XREF-003	Cross-references to subsections use “Section x.y” rather than “Subsection x.y”	—

(continued on next page)

Rule	Requirement	Precision
JSS-XREF-004	Numbered equations carry <code>\label{}</code> and are referenced from the text	100.0%
JSS-XREF-005	Figures and tables carry <code>\label{}</code> and are referenced from the text	99.6%
JSS-XREF-006	Figure and table floats carry a <code>\caption{}</code>	100.0%
JSS-XREF-007	Cross-reference nouns are spelled out (Figure/Section/Table), not abbreviated (Fig./Sec./Tab.)	100.0%
House style		
JSS-HOUSE-001	“e.g.” and “i.e.” are followed by a comma so LaTeX does not treat the period as a sentence end	100.0%
JSS-HOUSE-002	Book editions are indicated as 2nd, 3rd, etc., not as “second” or “2e”	100.0%
JSS-HOUSE-003	Preamble avoids loading LaTeX packages that jss.cls already provides (graphicx, xcolor, ae, fancyvrb, hyperref)	100.0%
Project		
JSS-PROJECT-001	A cycle exists in the <code>\input/\include/\subfile/\bibliography</code> reference graph	—
JSS-PROJECT-002	A <code>\input/\include/\subfile/\bibliography</code> target could not be found	—

Table 7: Active rules in version 1.1.0 with measured precision.

B. Per-rule precision and recall

Table 8 reports per-rule precision for every rule with at least one adjudicated firing; Table 9 reports per-rule recall for rules with at least 10 planted ground-truth violations, pooling thinner rules.

Rule	TP	FP	Precision	Gate
JSS-ABBR-001	40	0	100.0%	pass
JSS-BIBTEX-002	11	0	100.0%	pass
JSS-BIBTEX-003	63	0	100.0%	pass
JSS-BIBTEX-004	135	0	100.0%	pass
JSS-BIBTEX-005	9	0	100.0%	pass
JSS-CAP-001	118	0	100.0%	pass
JSS-CAP-002	669	11	98.4%	pass
JSS-CAP-004	23	0	100.0%	pass
JSS-CITE-002	296	29	91.1%	pass
JSS-CITE-003	271	0	100.0%	pass
JSS-CITE-004	20	0	100.0%	pass

(continued on next page)

Rule	TP	FP	Precision	Gate
JSS-CODE-001	463	0	100.0%	pass
JSS-CODE-002	280	0	100.0%	pass
JSS-CODE-003	2,442	21	99.1%	pass
JSS-HOUSE-001	641	0	100.0%	pass
JSS-HOUSE-002	37	0	100.0%	pass
JSS-HOUSE-003	65	0	100.0%	pass
JSS-MARKUP-001	1,165	206	85.0%	exempt
JSS-MARKUP-002	241	1	99.6%	pass
JSS-MARKUP-003	1,771	120	93.7%	pass
JSS-MARKUP-004	152	0	100.0%	pass
JSS-NAME-001	10	0	100.0%	pass
JSS-NAME-002	195	0	100.0%	pass
JSS-OPER-001	93	0	100.0%	pass
JSS-OPER-002	85	1	98.8%	pass
JSS-OPER-003	441	0	100.0%	pass
JSS-OPER-004	564	20	96.6%	pass
JSS-PRE-001	71	0	100.0%	pass
JSS-PRE-002	6	0	100.0%	pass
JSS-PRE-003	1	0	100.0%	pass
JSS-PRE-004	6	0	100.0%	pass
JSS-PRE-005	6	0	100.0%	pass
JSS-PRE-006	24	0	100.0%	pass
JSS-PRE-007	6	0	100.0%	pass
JSS-REFS-001	2	0	100.0%	pass
JSS-REFS-003	3,589	138	96.3%	pass
JSS-REFS-004	688	0	100.0%	pass
JSS-REFS-005	56	0	100.0%	pass
JSS-REFS-006	1,214	14	98.9%	pass
JSS-REFS-007	182	0	100.0%	pass
JSS-STRUCT-001	99	0	100.0%	pass
JSS-STRUCT-002	36	0	100.0%	pass
JSS-STRUCT-003	2	0	100.0%	pass
JSS-STRUCT-004	9	0	100.0%	pass
JSS-STRUCT-005	20	0	100.0%	pass
JSS-STRUCT-006	9	0	100.0%	pass
JSS-TYPO-001	267	0	100.0%	pass
JSS-TYPO-002	1	0	100.0%	pass
JSS-TYPO-003	10	0	100.0%	pass
JSS-TYPO-004	76	0	100.0%	pass
JSS-WIDTH-001	520	0	100.0%	pass
JSS-XREF-001	38	2	95.0%	pass
JSS-XREF-002	907	0	100.0%	pass
JSS-XREF-004	1,021	0	100.0%	pass
JSS-XREF-005	236	1	99.6%	pass

(continued on next page)

Rule	TP	FP	Precision	Gate
JSS-XREF-006	16	0	100.0%	pass
JSS-XREF-007	78	0	100.0%	pass

Table 8: Per-rule precision at the pinned iteration (new-additions-through-resolve-input).

Rule	Plants	Recovered	Recall
JSS-BIBTEX-004	16	13	81.2%
JSS-CAP-002	90	70	77.8%
JSS-CITE-002	12	8	66.7%
JSS-CITE-003	44	44	100.0%
JSS-CODE-001	57	56	98.2%
JSS-CODE-002	72	70	97.2%
JSS-CODE-003	228	203	89.0%
JSS-HOUSE-001	60	32	53.3%
JSS-HOUSE-003	10	10	100.0%
JSS-MARKUP-001	44	38	86.4%
JSS-MARKUP-003	282	168	59.6%
JSS-NAME-002	23	21	91.3%
JSS-OPER-002	19	14	73.7%
JSS-OPER-003	71	55	77.5%
JSS-OPER-004	78	47	60.3%
JSS-REFS-003	279	189	67.7%
JSS-REFS-004	74	66	89.2%
JSS-REFS-006	108	105	97.2%
JSS-REFS-007	26	26	100.0%
JSS-TYPO-001	37	32	86.5%
JSS-TYPO-004	20	20	100.0%
JSS-WIDTH-001	19	13	68.4%
JSS-XREF-002	42	42	100.0%
JSS-XREF-004	139	139	100.0%
JSS-XREF-005	19	19	100.0%
JSS-XREF-007	25	25	100.0%
27 rules below 10 plants (pooled)	73	62	84.9%

Table 9: Per-rule recall at the pinned run against the hand-annotated ground-truth corpus.

C. Engine parity evidence

The byte-parity claim of Section 3.2 is continuously enforced by the test suites; the listing below is the evidence for the demonstration manuscript, regenerated with every build of this

paper. Both engines — the Python reference and the Rust port, loaded in-process through the **jsslint** wheel — render `examples/demo.tex` in all four output formats, and the renderings are compared byte for byte:

```
Byte-parity of the two engines on examples/demo.tex
reference: jss-style-checker 1.1.0 (Python)
port:      jsslint 1.1.0 (Rust, in-process wheel)

terminal    5022 bytes  IDENTICAL
  sha256 896c5bbac511b0faa088bd53c7c2e54bec31a33093c0b6551f945d0013a9a2c4
json        6163 bytes  IDENTICAL
  sha256 7a52d8ddcf55ec42f29ccec503fc814f329b37885e0ef436688469a23d65290f
sarif       53711 bytes  IDENTICAL
  sha256 01470b1abce5ce78c4c722fc0be66c5afdf1691a66d1936a651f8dce45569d6f
html        4294 bytes  IDENTICAL
  sha256 7e94949b93dbac3846dae5c5b93604415357b08e496b46f637c7638256dae1c

4/4 output formats byte-identical.
```

Affiliation:

Manuel Koller

Zürich, Switzerland

E-mail: kollerma@protonmail.com