Literature-Grounded Generation of README Files with Large Language Models: A Dual Human–LLM Evaluation

Allan Assis de Andrade
Universidade do Estado do Rio de Janeiro
Rio de Janeiro, Brazil
allan-bdc@hotmail.com

Talita Vieira Ribeiro
Universidade do Estado do Rio de Janeiro
Rio de Janeiro, Brazil
talita.ribeiro@ime.uerj.br

Abstract

README files are the primary entry point to a software project, yet they are frequently incomplete or outdated. Large Language Models (LLMs) can automate their creation, but existing generators either leave the document structure to the model or fix it in templates with no documented basis in studies of README content. This paper presents README-Gen, a generation approach whose README structure is derived from studies of README content and from the Theory of Robust API Knowledge (ATRAK). We apply it, backed by GPT-4.1-mini, to twelve GitHub repositories across three languages, and assess the output with a dual protocol pairing a human assessment with an automated evaluator (Claude Fable 5) driven by a rule-based per-section rubric; the two agree substantially (Krippendorff's $\alpha = 0.90$). Structured prompting reliably enforces the expected structure (98% completeness) and full ATRAK adherence, and README-Gen outscores README-AI, under the same model and rubric, on the user-oriented dimension that rubric measures (78% vs. 48% correctness). Correctness, however, is stratified by repository popularity (99% vs. 28%, human evaluation), which we report as an exploratory association with how much information about a project exists outside its repository.

Keywords

Software documentation, API knowledge, README generation, large language models, LLM-as-a-judge

1 Introduction

README files are among the most recognizable artifacts in software documentation and typically the first element a developer consults when visiting a repository: they shape the initial impression of a project and influence the decision to use it or contribute to it [5, 12]. Despite this importance, documentation is usually deprioritized. Developers rarely keep it synchronized with the code [6], and in GitHub's large-scale Open Source Survey [7], roughly 93% of respondents considered incomplete or outdated documentation a pervasive problem.

Large Language Models (LLMs) are a natural fit for generating or updating READMEs automatically and have already been applied to README creation, as examples there are LARCH [11], the agent-based RMGenie [3], and the open-source tool README-AI [14]. In these tools, however, the structure of the generated document is either implicitly decided by the model, delegated to agent decisions, or fixed in templates whose empirical basis is not documented, and their output quality is rarely assessed with reproducible criteria.

We propose README-Gen, a structured prompting approach in which the README template is derived from studies of README content [8, 12] and each section is mapped to one of the three Knowledge Elements of the Theory of Robust API Knowledge (ATRAK) [16]: Domain Concepts, Execution Facts, and Usage Patterns. We apply the approach, backed by GPT-4.1-mini, to twelve GitHub repositories spanning JavaScript, Python, and Bash/Shell, mixing popular and obscure projects. The generated files are assessed by a dual evaluation protocol: a human assessment and an automated evaluator (Claude Fable 5) that applies a rule-based per-section rubric, allowing us to cross-check the two judges. To determine whether this approach improves README generation, we compare the README files generated by README-Gen with those generated by README-AI, both configured with the same underlying model and scored with the same rubric. The study is guided by two main research questions:

RQ1. Can a structured prompting strategy, operationalizing ATRAK's three Knowledge Elements, generate README files that are (i) structurally complete, (ii) fully ATRAK-adherent, and (iii) factually correct for software projects with a programmatic or command-line interface?

RQ1.1. *How does the quality of these generated files compare to an existing LLM-based generator (README-AI)?*

RQ2. How does the correctness of LLM-generated READMEs vary with repository popularity and with the amount of information about the project available *outside* its own artifacts?

RQ2 is motivated by the fact that models memorize substantial portions of their training data, and that memorization grows with a document's frequency in the corpus [2]: documentation of popular repositories is well represented on the web, so it is more likely both to have been seen during training and to be retrievable at inference time. We treat popularity as a coarse proxy for the availability of external information and report an *association* with correctness, without attributing it to exposure, retrieval, or documentation quality individually.

2 Background and Related Work

2.1 README Content and Quality

Empirical studies show recurring structural patterns in READMEs. Ikeda et al. [8] mined thousands of JavaScript repositories and found *Usage* (60.8%), *Installation* (59.4%), and *License* (36.2%) to be the most frequent sections, followed by *API/Options* documentation. Liu et al. [12] analyzed Java repositories, found most READMEs organized around the same small set (project name, installation, usage examples, license), and reported an association between usage and license sections containing code snippets and repository popularity; Prana et al. [13] categorized README content to improve navigability. Together these studies suggest that READMEs primarily help

users understand *what* a project does and *how* to install and use it, rather than document internal development processes.

2.2 The Theory of Robust API Knowledge

The Theory of Robust API Knowledge [16] — which we abbreviate ATRAK for brevity — models how technical knowledge about an API is communicated through documentation, identifying three Knowledge Elements that a good documentation should present: **Domain Concepts**, the entities and abstractions of the problem domain the software addresses; **Execution Facts**, how the software behaves at run time (inputs, outputs, dependencies, configuration, and constraints); and **Usage Patterns**, typical ways the software is applied, usually expressed through examples and tutorials. When the three elements are clearly represented, documentation supports both learning and practical use.

ATRAK was formulated for APIs, not repositories, so applying it here needs an explicit bridge. We restrict the scope (RQ1) to repositories that *expose* an API in ATRAK's sense — a library's public functions, or a tool's commands and flags — and treat the README as the entry point to that interface rather than as the project's complete documentation. The theory then applies to the subset of the repository the README is about, and the three Knowledge Elements map onto sections without strain: what the software is about (Overview), how it behaves when run (API Reference), and how it is typically applied (Usage). We claim no extension to prose-only, dataset, or application repositories that expose no such interface.

2.3 LLMs for Documentation Generation

The work closest to ours in this line is Doan et al. [4], who apply transformer-based models to summarize existing GitHub READMEs — a summarization task over documentation that already exists, whereas we generate it from scratch. Closer to our goal, LARCH [11] generates READMEs with LLMs and heuristics; the open-source tool README-AI [14] generates READMEs from repository metadata; and RMGenie [3] employs LLM-based agents with external tool invocation to generate READMEs, evaluating output completeness and factual accuracy. In all of these tools, the resulting document structure is implicitly determined by the model, by ad-hoc heuristics, or by agent decisions; none derives it from evidence about what a README should contain.

LLMs are also increasingly used as evaluators of generated text: Zheng et al. [18] report that strong judges reach agreement with human raters comparable to human–human agreement, while cautioning that they exhibit systematic biases (position, verbosity, self-enhancement). We take this dual message — useful agreement, quantifiable biases — as the reason to pair the automated judge with an independent human assessment rather than replace it.

3 The README-Gen Approach

README-Gen receives a GitHub repository and produces a README file whose structure is fixed *a priori*, instead of being left to the model. Two properties of LLMs motivate this design: they are prone to hallucinated content [9], and their behavior is highly sensitive to prompt design, with few-shot competence depending on carefully constructed demonstrations [1]. Fixing the structure removes one degree of freedom from the model and makes the output checkable section by section. The approach has two components: a README *template* and a *structured prompting* strategy that instructs the LLM to fill that template.

Table 1: The README template. Frequencies are from Ikeda et al. [8].

Section	Content requested by the prompt	Knowledge Element	Evidence
Title	Project or tool name	—	[12]
Overview	Purpose, goal, functionality	Domain Concepts	9.6% [8]
Installation	Setup steps (methods, package managers, OSes)	—	59.4% [8]
Usage	Runnable snippets and outputs; why and how to use it	Usage Patterns	60.8% [8, 12]
API Ref.	Main functions, classes, or endpoints with parameters	Execution Facts	24.3% [8]
License	License, checked against `LICENSE`	—	36.2% [8, 12]

3.1 The README Template

The template combines two criteria: empirical frequency and theoretical coverage. For frequency, we rely on the two content studies most applicable to our setting — Ikeda et al. [8] and Liu et al. [12], with Prana et al. [13] corroborating. Theoretical coverage ensures that every ATRAK Knowledge Element is carried by one section, which explains the inclusion of Overview despite its low frequency (9.6%) in the primary studies: it is the natural carrier of Domain Concepts, without which the theory would be only partially covered. Table 1 shows the resulting structure.

Four frequent sections were intentionally excluded: *Release History* (release notes usually live in a dedicated file [17]); *Contributing* and *Test Instructions* (contributor-facing; they belong in `CONTRIBUTING.md`, whose presence is associated with project activity [10]); and *TODO lists* (incomplete development tasks, not user-facing documentation).

3.2 Structured Prompting Strategy

The prompt has two parts: (i) a *contextual prompt* with the ATRAK-grounded template, a short description of what each section should contain (e.g., "focus on Domain Concepts" for Overview), and three few-shot examples of well-formed output; and (ii) a *task instruction* identifying the target repository by URL, permitting the model to search the web. No repository artifacts (source, documentation, or configuration) are injected: the model must produce the README from its own prior knowledge and whatever its provider retrieves at inference time, never from the artifacts themselves. Generation uses `gpt-4.1-mini-2025-04-14`; the full prompt is in the replication package.

4 Study Design

We conducted an exploratory study, designed as follows.

4.1 Repository Selection

Twelve GitHub repositories were selected across three of the most used programming languages in 2024: JavaScript (JS), Python (Py), and Bash/Shell (Sh) [15]. HTML and SQL were excluded as non-general-purpose, and TypeScript for its similarity to JavaScript.

Table 2: The twelve repositories; stars as of July 2026.

Repository	L	Stars	Core Functionality
Axios	JS	109k	Execute HTTP requests
jQuery	JS	59.8k	Query HTML elements
Moment	JS	47.9k	Parse and manipulate dates
uri	JS	26	Parse URIs
NumPy	Py	32.4k	Create and manipulate arrays
Rich	Py	56.8k	Enhance Python console output
Scikit-learn	Py	66.7k	Train machine learning models
SnakeMD	Py	48	Generate README files
Git	Sh	62k	Create and manage commits
jq	Sh	35.1k	Parse JSON data
Notes-cli	Sh	263	Create and manage notes
Command-launcher	Sh	45	Install packages

Within each language, we varied two dimensions: *documentation* (with vs. without user-oriented usage instructions — five of the twelve have them: Axios, `uri`, Rich, SnakeMD and Notes-cli) and *popularity*, which defines the *high-popularity* group (the eight repositories above 30,000 stars) and the *low-popularity* group (the four below 300). This exposes the model to different knowledge-availability conditions: popular projects may benefit from prior exposure, while for obscure ones little project-specific knowledge is expected to be available.

4.2 Procedure and Evaluation Dimensions

To account for output variability, generation was repeated three times per repository; every score reported is the arithmetic mean of the three runs. Each README was manually evaluated along three dimensions:

Structural Completeness. Presence of seven elements: the six template sections plus *Core Functionality* — whether the README documents the repository's primary purpose, defined per repository in Table 2. An element counts as present only when the section carries the expected information, not merely when a heading exists; *completeness* is the percentage of the seven present.

Correctness. In the *human* evaluation, three aspects were assessed independently: *(i) textual correctness (T)*: the five textual sections (title, overview, installation *text* instructions, usage *text* instructions, license) were each classified against the official documentation as *Correct*, *Incomplete*, or *Incorrect*; the first two score 1 and the last 0, and T is their mean; *(ii) code-snippet correctness (C)*: every snippet was manually executed and passes only if it runs without modification (except omitted imports) and produces the described behavior, so snippets that fail or run partially score 0; C is the fraction passing. *(iii) API Reference correctness (A)*: cross-checked against the official API documentation as a single *Correct/Incomplete*-vs-*Incorrect* judgment, scoring 1 or 0. To be explicit: in T and A, *Correct* and *Incomplete* score the same and only *Incorrect* scores 0. An incomplete but accurate description still serves the user, whereas an incorrect one misleads — but it makes T and A lenient by construction, one reason we report the graded automated rubric alongside them. In C the opposite holds: *Incomplete* is a failure, because a snippet that almost runs does not run. Code instructions are thus captured exclusively by C, via execution, never by reading. Repository correctness weights the three equally, $correctness = (T + C + A)/3$.

ATRAK Adherence. Binary presence of each Knowledge Element *anywhere in the README*, averaged into a percentage. Unlike completeness, this check is not tied to a section — Execution Facts count wherever the file states dependencies, installation steps or runtime behavior — and it assesses presence, not correctness or depth: hallucinated content counts as present, as does an element carried without the rationale ATRAK associates with it.

4.3 Automated LLM-Based Evaluator

Since the human assessment was performed by a single evaluator, we codified the criteria into a rule-based per-section rubric and instructed *Claude Fable 5* (Anthropic) to apply it, producing scores in the same CSV format as the human evaluation. Completeness follows the same presence check as the human procedure; correctness is decomposed into explicit verification rules per section (Table 3). Each rule is a binary check V_i; the section score is the percentage of satisfied rules (for Usage and API Reference, averaged over snippets/elements), and the overall correctness is the unweighted mean of the six section scores.

Operation of the Judge. The execution-dependent rules are not answered by reading. The judge ran as a tool-using agent with shell and web access: on a macOS 14 (arm64) host with Python 3.14 and Node v24.12.0, it built a throwaway environment per repository (a fresh `venv`, or a clean `npm` directory), installed the project as the README documents it, and executed each snippet there, recording the command, its output, and the verdict per rule. API elements were checked by introspection against the installed artifact (`inspect.signature`, `hasattr`) and the official docs, not by recall; every reasoning file names the sources cross-checked. Each (tool, repository) pair was one invocation, each README scored individually, and repairing a broken snippet before scoring it was forbidden. Three controls we did *not* impose: ground truth is the released artifact (e.g., `numpy 2.5.1`) and the default branch at evaluation time, not a pinned commit; no execution timeout was fixed; and the judge ran at the model's then-current rolling version rather than a pinned dated snapshot, with sampling parameters at provider defaults and one invocation per artifact — its decisions rest on binary rules, so we expect limited run-to-run variance, but did not measure it. Where execution evidence is unobtainable the judge may fall back on textual plausibility, a limitation of all LLM-based evaluators and a further reason we pair it with, rather than replace it by, the human assessment. Per-rule outputs and execution logs are in the replication package.

4.4 Baseline Comparison Protocol

Isolating the contribution of the structure (RQ1) requires holding the underlying LLM constant, and README-AI [14] is the only candidate that allows it: LARCH [11] supports Python only (excluding eight of our repositories) and is bound to the discontinued `text-davinci-003`; RMGenie [3] exposes no public artifact with model control, and its agentic pipeline would confound the comparison with retrieval capability; closed-source generators permit neither model control nor pipeline inspection. README-AI is also the most widely adopted open-source generator (≈2.9k GitHub stars vs. 17 for LARCH, June 2026), so the comparison gauges the state of practice.

Table 3: Correctness rubric. For Usage and API Reference a snippet/element scores 1 only if *all* its rules hold.

Section	#	Rules (condensed)
Project Title	3	Name matches the repository; no different project described; no hallucinated terminology.
Overview	5	Functionality and domain correct; claims backed by repository artifacts; no unsupported features; terminology matches.
Installation	5	Isolated environment: dependencies declared; commands run unmodified; no errors; environment and artifact correct.
Usage & Examples	5 / snip.	Clean environment: executes unmodified; imports documented; output as documented; no exceptions; matches the text.
API Reference	6 / elem.	Against the source: element exists; names, types and returns correct; behavior consistent; no deprecated API.
License	3	Matches the `LICENSE` file; valid identifier; no conflicting licensing in the README.

README-AI (v0.6.0rc1) was run on the same twelve repositories with the same model (`gpt-4.1-mini-2025-04-14`) and an identical command line (full invocation in the package). All outputs were scored by the evaluator of Section 4.3 with the identical rubric.

Asymmetries of the Comparison. The comparison is asymmetric in four ways that all favor README-Gen; we state them so the reader can discount it accordingly, rather than read it as a like-for-like benchmark. *(i) Rubric:* completeness and ATRAK adherence are enforced by our own prompt, so we win those two dimensions by construction — they measure whether the prompt works, not whether our READMEs are better. *(ii) Repetition:* our score is the mean of three runs, README-AI's a single draw; we mitigate this by also comparing against our *worst* run, though we did not measure README-AI's own variance. *(iii) Human evaluation:* only README-Gen was scored by the human evaluator, so every cross-tool number here comes from the automated judge alone. *(iv) Purpose:* README-AI assembles a broad, repository-derived scaffold (project structure, per-file index, badges, contribution workflow), whereas README-Gen emits only the six user-oriented sections the content studies identify as the README's core [8, 12, 16]. A maintainer wanting an onboarding map of the codebase is better served by README-AI, and our rubric gives it no credit for that. The comparison measures fitness for the user-oriented purpose — not overall tool quality.

5 Results

5.1 Completeness and ATRAK Adherence

README-Gen achieved a mean structural completeness of **98.0%** (human) and **97.6%** (automated), with ten of twelve repositories at 100% under both. The two exceptions, SnakeMD (90.5%/85.7%) and Command-launcher (85.7%/85.7%), are low-popularity projects failing the same check — *Core Functionality* — the model not knowing what they are for; the judges differ only on SnakeMD, credited by the human in one of three runs and by the automated evaluator in none. ATRAK adherence was **100%** across all repositories and runs: every README carried all three Knowledge Elements. Neither result is surprising, and we do not present it as a discovery: the prompt enforces the sections the rubric checks. What they establish is that the model *reliably complies* with an imposed structure across languages and popularity levels — a precondition for the correctness analysis below.

Scored by the same rubric, README-AI reaches 65.5% completeness and 75.0% ATRAK adherence; the two diverge because ATRAK credits an element wherever it appears, so README-AI's build and dependency content carries Execution Facts even where an API Reference section is absent. The gap in *level* is expected, since our prompt enforces the sections being checked; the difference in *shape* is not. README-Gen is flat at or near 100% everywhere, whereas README-AI swings between 42.9% and 85.7% without tracking popularity: its best results include one popular repository (Axios, 85.7%) and one obscure one (`uri`, 85.7%), while jQuery — with 59.8k stars — is its worst at 42.9%. Where it loses ground is consistent rather than random: API Reference is missing from all twelve outputs and Usage from eight. Section 5.4 returns to what that implies.

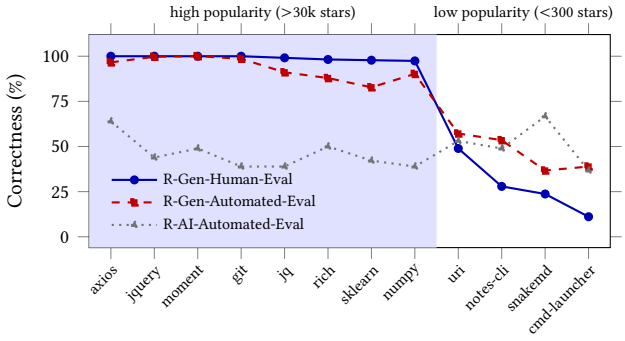

Figure 1: Per-repository correctness. README-Gen: mean of three runs, under both evaluators; README-AI: single run, automated evaluator only (Krippendorff's $\alpha = 0.90$ between evaluators).

5.2 Correctness: Human vs. Automated

Figure 1 reports overall correctness per repository under both evaluators (mean of three runs); averages are 75.3% (human) and 77.8% (automated), though the sample combines two strongly distinct popularity groups.

In the human evaluation, textual correctness averaged 87.2% while code-snippet correctness averaged 69.3%: executable content is substantially less reliable than prose. In the automated evaluation, Title and License scored 100% everywhere (factual matching only), while Usage (57.2%) and API Reference (64.5%) concentrated the variability.

Agreement Between the Two Protocols. The protocols do not score identically — the human rubric treats *Incomplete* as a pass, the automated one awards graded partial credit — so we measure how far apart their *absolute* scores fall, with Krippendorff's α under the interval difference function. Correlation would not answer this: it measures covariation, so two judges could rank the twelve repositories identically while disagreeing on every value. We obtain $\alpha = 0.902$ for correctness and $\alpha = 0.963$ for completeness, both

Table 4: Automated evaluation (%).

Metric	README-Gen	README-AI
Structural completeness	97.6	65.5
ATRAK adherence	100.0	75.0
Correctness (all repos)	77.8	47.6
Correctness (high-popularity)	93.4	45.7
Correctness (low-popularity)	46.6	51.3

above the 0.80 conventionally treated as reliable, with a mean absolute deviation of 10.0 points. Cohen's κ does not apply, as both protocols emit continuous percentages rather than nominal categories. Reported separately, as association and not agreement, the rankings are near-identical ($\rho = 0.95$, $p < 0.001$; $\tau = 0.86$).

The residual disagreement is systematic and one-directional: on the eight popular repositories the automated evaluator is stricter (−5.7 points), its sub-criteria catching partial failures the binary human rubric absorbed as "Correct or Incomplete"; on the four low-popularity ones it is more lenient (+18.7), awarding partial credit where the human rubric scored the section entirely wrong. The three largest cases show the mechanism: Command-launcher (11.1 vs. 38.9) and Notes-cli (27.9 vs. 53.5), where the automated rubric credited a correct Title and License while the human registered the README as wrong overall; and scikit-learn (97.8 vs. 82.8), where per-snippet scoring penalized two Usage examples the human passed as "Incomplete". Both effects stem from granularity, not from the judges reading the artifacts differently: the scores are related but not interchangeable.

5.3 Correctness by Popularity and Language

Popular repositories (>30k stars) achieved a mean correctness of **99.0%** (human) / **93.4%** (automated), against **27.9%** / **46.6%** for low-popularity ones — a lower bound, since obscure projects also have sparser reference documentation (Section 6). Both groups ran the same method and prompt, differing only in the repository named, so the gap isolates the repository rather than the technique; RQ2 discusses what varies with it. The per-language means (JavaScript 87.2%/88.4%, Python 79.3%/74.4%, Shell 59.5%/70.5%) track the same split: the lower Shell figure comes from its two low-popularity repositories, not from the language.

5.4 Baseline Comparison with README-AI

Table 4 summarizes the contrast. README-Gen leads on completeness and ATRAK adherence — the two dimensions Section 4.4 flagged as ours to win — and on correctness overall. The correctness lead comes entirely from the popular group; on the low-popularity group **the ranking inverts**. Our three-run mean beats README-AI in 11 of 12 repositories, but our worst run in only 9, losing on `uri` and SnakeMD and tying exactly on Command-launcher — so the margin depends partly, though not mainly, on our variability.

Inspecting the outputs suggests why. README-AI emits user-facing headers, but the content behind them is often generic scaffolding: its *API Reference* scored 0 in all twelve repositories and its *Usage* section averaged 10.4%, with unresolved placeholders (`python {entrypoint}`, `INSERT-RUN-COMMAND-HERE`) or inapplicable run commands — `npm start` as the usage of jQuery, a browser library. Its generative effort goes into repository-structure content (per-file index, build, CI details), valuable to maintainers but outside a rubric built on the user-oriented core [8, 12]. Emitting the expected sections is therefore not sufficient to produce verifiable, project-specific content.

Its correctness is also flat across the two popularity groups and varies far less across repositories than ours (standard deviation 9.4 vs. 23.2 points), matching the popularity-independent completeness profile described above. This is what a repository-parsing pipeline predicts: content extracted from the artifacts gains nothing from prior exposure, so quality is bounded by the pipeline rather than by familiarity with the project — the mirror image of README-Gen, bounded by that familiarity precisely because it reads no artifacts. Consistently, the one repository README-AI wins outright is SnakeMD (66.7% vs. 36.7%), where our exposure deficit was largest, and that single margin is what flips the low-popularity group mean. All of this is consistent with artifact injection compensating for missing external information, but it is not a controlled test: the two tools differ in pipeline as well as in inputs.

5.5 Answering the Research Questions

RQ1. The strategy reliably produces well-formed, fully ATRAK-adherent READMEs and outscores README-AI under the same model and rubric — subject to the four asymmetries of Section 4.4, and with both structural results enforced by the prompt rather than discovered. The substantive finding is the one the prompt did not enforce: correctness holds for popular repositories and degrades for low-popularity ones. Structure guarantees form and adherence, not content correctness.

RQ2. Since README-Gen supplies no repository artifacts, popularity stands in for how much information about a project exists *outside* its repository. The stratification is strong (99% vs. 28%, human) and supports an *association* with that external availability — not the claim that memorization causes it, which the confounds of Section 1 prevent establishing here. Two observations are consistent with the association: README-AI, which injects artifacts, shows no comparable stratification, and the one repository where it beats us is the one where our exposure deficit was largest. Disentangling exposure, retrieval, and documentation quality would require controlling the retrieved context and the training corpus.

6 Threats to Validity

Internal Validity. The human evaluation was performed by a single evaluator. We mitigated this with a rule-based rubric and by cross-checking against an independent automated evaluator, whose absolute scores agree substantially with the human ones (Krippendorff's $\alpha = 0.902$, interval). The automated evaluator is itself an LLM and may share biases with the generator, but uses a different model family (Claude vs. GPT) and explicit verification rules; its systematic divergence is a more lenient scoring of the low-popularity group (+18.7 points, against −5.7 on the popular one), consistent with the general concern that LLM judges carry quantifiable biases [18]. Because that divergence is confined to sections the human rubric scored as entirely wrong, it inflates the low-popularity scores rather than the popularity gap itself, which is the finding it could have threatened.

Construct Validity. Beyond the confounds that keep RQ2 an association (Section 1), popularity co-varies with the richness of the official documentation we use as *ground truth*, which is a confound in the measurement itself: for obscure projects that documentation is sparse, so part of the low score may reflect a thinner reference rather than a worse README. We therefore read the low-popularity numbers as a lower bound. A second construct gap is that completeness and ATRAK adherence measure compliance with an imposed structure, not usefulness to a reader; establishing the latter needs a comprehension or task-performance study, which we did not run.

Conclusion Validity. The two protocols do not score identically, so we report chance-corrected agreement (α) for absolute scores and Spearman's ρ only as rank association. With twelve repositories no coefficient reported here carries much precision, and we draw no inference from small differences between groups.

External Validity. Twelve repositories, one generation model, and one judge model are a small basis for generalization; the study is exploratory — a reference point rather than a statistical claim. The baseline contrast is further bounded by the four asymmetries of Section 4.4.

7 Conclusion and Future Work

This paper proposed a README generation approach whose structure is derived from studies of README content and from the Theory of Robust API Knowledge, together with a dual human–LLM evaluation protocol. Structured prompting reliably enforced the expected structure and full ATRAK adherence, and the automated evaluator agreed substantially with the human one ($\alpha = 0.90$), supporting its use as a reproducible consistency check on a single-rater assessment. Content correctness, however, collapses for repositories with little presence outside their own code (28% vs. 99%, human): given no repository artifacts, current LLM-based generation leans on whatever is known about the project elsewhere rather than on the artifacts themselves. Structure alone cannot fix this.

Three directions follow. The most direct is to inject repository artifacts at inference time and measure how much of the dependency they remove; disentangling injection from pipeline differences requires varying the inputs of one tool, not comparing two. Second, and most important for the claim we could not make here, a user study on whether ATRAK-complete READMEs improve comprehension or task completion: we measured structural alignment and factual correctness, never understanding. Third, a stronger baseline protocol — repeated runs for the competing tool, a human evaluation of a sample of its outputs, and a usefulness criterion independent of any document structure — would remove the asymmetries of Section 4.4.

Artifact Availability

The generation and evaluation prompts, the READMEs produced by both tools in every run, the per-run human and automated scores behind every number reported, and the script that recomputes them are in the replication package at https://doi.org/10.6084/m9.figshare.33201684.

Acknowledgements

The authors used AI-based tools to improve the clarity and grammar of author-written passages and to cross-check the reported numbers; all content, analyses, and conclusions are the sole responsibility of the authors.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, et al. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[2] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramèr, and Chiyuan Zhang. 2023. Quantifying Memorization Across Neural Language Models. In *The Eleventh International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=TatRHT_1cK

[3] Xing Cui, Jingzheng Wu, Zhiyuan Li, Tianyue Luo, and Xiang Ling. 2025. RMGenie: An LLM-Based Agent Framework for Open Source Software README Generation. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 505–516.

[4] Thu T. H. Doan, Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. 2023. Too long; didn't read: Automatic summarization of GitHub README.MD with Transformers. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering* (Oulu, Finland) *(EASE '23)*. Association for Computing Machinery, New York, NY, USA, 267–272. doi:10.1145/3593434.3593448

[5] Karl Fogel. 2017. *Producing Open Source Software: How to Run a Successful Free Software Project* (second ed.). O'Reilly Media, Sebastopol, CA, USA. http://www.producingoss.com/.

[6] Andrew Forward and Timothy C. Lethbridge. 2002. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering (DocEng '02)*. Association for Computing Machinery, New York, NY, USA, 26–33. doi:10.1145/585058.585065

[7] GitHub. 2017. Open Source Survey. https://opensourcesurvey.org/2017/. Accessed: 2026-07-10.

[8] Shohei Ikeda, Akinori Ihara, Raula Gaikovina Kula, and Kenichi Matsumoto. 2019. An Empirical Study of README contents for JavaScript Packages. *IEICE Transactions on Information and Systems* E102.D, 2 (Feb. 2019), 280–288. doi:10.1587/transinf.2018EDP7071

[9] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *Comput. Surveys* 55, 12, Article 248 (2023), 38 pages. doi:10.1145/3571730

[10] Naoki Kobayakawa and Kenichi Yoshida. 2017. How GitHub Contributing.md Contributes to Contributors. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 694–696. doi:10.1109/compsac.2017.139

[11] Yuta Koreeda, Terufumi Morishita, Osamu Imaichi, and Yasuhiro Sogawa. 2023. LARCH: Large Language Model-based Automatic Readme Creation with Heuristics. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23)*. ACM, New York, NY, USA, 5066–5070. doi:10.1145/3583780.3614744

[12] Yuyang Liu, Ehsan Noei, and Kelly Lyons. 2022. How ReadMe files are structured in open source Java projects. *Information and Software Technology* 148 (2022), 106924. doi:10.1016/j.infsof.2022.106924

[13] Gede Artha Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. 2019. Categorizing the Content of GitHub README Files. *Empirical Software Engineering* 24, 3 (June 2019), 1296–1327. doi:10.1007/s10664-018-9660-3

[14] Eli Salamie. 2023. README-AI: A README File Generator, Powered by AI. https://github.com/eli64s/readme-ai. Accessed: 2026-06-18.

[15] Stack Overflow. 2024. 2024 Developer Survey: Technology. https://survey.stackoverflow.co/2024/technology. Accessed: 2026-07-10.

[16] Kyle Thayer, Sarah E. Chasins, and Amy J. Ko. 2021. A Theory of Robust API Knowledge. *ACM Trans. Comput. Educ.* 21, 1, Article 8 (Jan. 2021), 32 pages. doi:10.1145/3444945

[17] Jianyu Wu, Hao He, Wenxin Xiao, Kai Gao, and Minghui Zhou. 2022. Demystifying Software Release Note Issues on GitHub. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 602–613. doi:10.1145/3524610.3527919

[18] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, et al. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*. https://arxiv.org/abs/2306.05685