AutoChecklist: Composable Pipelines for Checklist Generation and Scoring with LLM-as-a-Judge

Karen Zhou **Chenhao Tan**

University of Chicago

{karenzhou, chenhao}@uchicago.edu

Code Website

Abstract

Checklists have emerged as a popular approach for interpretable and fine-grained evaluation, particularly with LLM-as-a-Judge. Beyond evaluation, these structured criteria can serve as signals for model alignment, reinforcement learning, and self-correction. To support these use cases, we present AutoChecklist, an open-source library that unifies checklist-based evaluation into composable pipelines. At its core is a taxonomy of five checklist generation abstractions, each encoding a distinct strategy for deriving evaluation criteria. A modular *Generator* → *Refiner* → *Scorer* pipeline connects any generator with a unified scorer, and new configurations can be registered via prompt templates alone. The library ships with ten built-in pipelines implementing published approaches and supports multiple LLM providers (OpenAI, OpenRouter, vLLM). Beyond the Python API, the library includes a CLI for off-the-shelf evaluation and a web interface for interactive exploration. Validation experiments confirm that these checklist methods significantly align with human preferences and quality ratings, and a case study on ICLR peer review rebuttals demonstrates flexible domain adaptation. AutoChecklist is publicly available at https://github.com/ChicagoHAI/AutoChecklist.

1 Introduction

Checklists decompose quality into individually verifiable criteria, with simple yes/no answers that bypass position bias of pairwise comparisons and subjectivity of scalar metrics with LLM-as-a-Judge (Ye et al., 2024). As a result, checklists offer interpretable and fine-grained evaluation of text quality, making them an effective tool for evaluation with LLM evaluators (Cook et al., 2024; Viswanathan et al., 2025; Wei et al., 2025; Zhou et al., 2025).

Several checklist generation methods have been proposed in recent work (see §2), each with distinct codebases, prompting strategies, and scoring mechanisms. These approaches differ along several axes: scope of checklist questions, inclusion of a reference response in checklist generation, unweighted vs. weighted scoring, and question refinement steps, if any. No existing toolkit provides a unified interface across these methods, making it difficult to compare or extend them for new tasks without significant re-implementation.

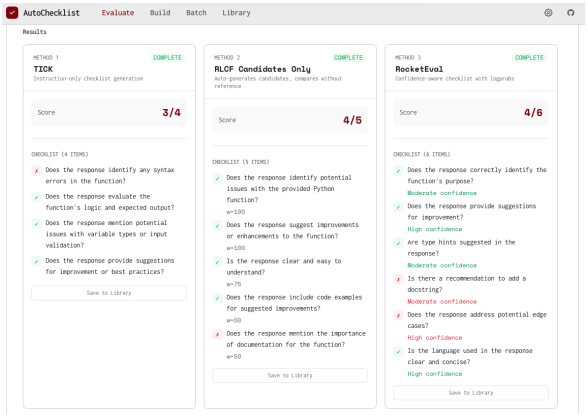


Figure 1: In the *Compare* page of AutoChecklist's UI, users select methods and enter an input to generate side-by-side checklists, enabling direct comparison of generation strategies on the same task.

To address this, we develop AutoChecklist, an open-source Python library that consolidates common checklist approaches into composable pipelines. We additionally validate the utility of the library (§5) and demonstrate its effectiveness in a new domain (§6). Our contributions are as follows:

- A **taxonomy of five generator abstractions** (direct, contrastive, inductive, deductive, and interactive) that organizes checklist methods by their strategies for producing evaluation criteria.
- A framework of **composable pipelines**: ten built-in configurations implementing published methods, compatible with a unified scorer that consolidates three scoring strategies from the literature. Users can customize these pipelines to new tasks

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 515–525

via Markdown prompt templates alone.

- A **`pip`-installable Python package** with multiple usage modes, including a CLI for off-the-shelf evaluation with pre-defined pipelines.
- **Multi-provider LLM backend** support for OpenAI, OpenRouter, and vLLM (including `VLLMOfflineClient` for local GPU inference), and automatic handling of structured output.
- A **locally hosted user interface** supporting most package functionality, allowing for interactive prompt customization, pipeline configuration and comparison (e.g., Figure 1), and batch evaluation.

2 Related Work

Checklist Generation & Evaluation. Checklist-based evaluation with LLM-as-a-Judge has been leveraged for a variety of tasks and benchmarks (Lin et al., 2024; Que et al., 2024; Qin et al., 2024). A common evaluation setup is to generate a checklist per input-output pair, such as for a user query and LLM response. Checklist generation can either use zero- to few-shot prompting to generate yes/no questions from the input alone, or by incorporating a reference and/or alternative response samples (Cook et al., 2024; Wei et al., 2025; Viswanathan et al., 2025). Other methods aim to derive a shared checklist from corpus-wide signals. These signals are either pre-defined by humans and then processed into granular criteria, or extracted from brainstorming or feedback items into structured checklists (Lee et al., 2025; Chu et al., 2025; Zhou et al., 2025). These methods often involve multiple steps to refine the initial checklist questions, such as through deduplication, filtering, or elaboration.

Despite their differences, all of these methods share a common structure: generate criteria from an input, optionally refine them, then score a target response. `AutoChecklist` builds on this shared structure. We identify five generator abstractions, each encoding a distinct reasoning strategy, and factor them into interchangeable generators, refiners, and scorers within a single library. To our knowledge, this is the first unified toolkit for LLM-based checklist generation.

Utility of Checklists. Beyond evaluation, structured criteria serve as signals for alignment and reinforcement learning (Viswanathan et al., 2025; Siro et al., 2026; Dineen et al., 2025; Yang et al., 2026) and self-correction (Wan et al., 2026). However, Furuhashi et al. (2025) find that checklists are not consistently useful and highlight the need to more clearly define objective evaluation criteria. `AutoChecklist` aims to lower the barrier for both human and LLM judges to define and select such criteria.

Unified Frameworks. judges is a similar library to use and create LLM-as-a-Judge evaluators, curating a set of LLM evaluators in a low-friction format that can be used off-the-shelf or customized. Numerous other open-source packages exist for evaluating and testing LLM systems (DeepEval, RAGAS, OpenAI Evals). While the generated checklists from `AutoChecklist` are primarily used with LLM-as-a-Judge, their usefulness is not limited to automatic evaluation; there is also potential for such checklists to help further align human annotators in subjective tasks (e.g., essay grading, clinical note assessment). Furthermore, `AutoChecklist` features unified criteria *generation*; most other libraries focus on scoring.

3 System Details

A *checklist* $C = \{q_1, \ldots, q_n\}$ is a set of yes/no questions designed to assess a target text. Each question q_i may carry an optional importance weight $w_i \in [0, 100]$. Given a target response, a *scorer* assigns each item an answer $a_i \in \{\text{YES}, \text{NO}\}$ and optionally a confidence $c_i \in [0, 1]$. The *pass rate* for a single input is $|\{a_i = \text{YES}\}|/|C|$, or $\sum w_i \cdot \mathbf{1}[a_i{=}\text{YES}] \,/\, \sum w_i$ when importance weights are available.

Since different methods are originally implemented with specific tasks in mind (e.g., evaluating LLM instruction-following or quality of a summary), we further define the following abstractions: a checklist *input* is the instruction, query, or task being evaluated (e.g., "Write a haiku about autumn" or "Summarize: {article}"); the checklist or evaluation *target* is the output being scored against the checklist (e.g., the haiku or summary).

Design and Architecture. As depicted in Figure 2, generators fall into two categories. *Instance-level generators* produce a checklist per input (e.g., per prompt-response pair). `DirectGenerator` (DIRECT) implements direct inference: a single LLM call that converts the input into checklist questions. `ContrastiveGenerator` (CONTRASTIVE) implements counterfactual reasoning; it first generates candidate responses of varying quality (usually from weaker models), then derives discriminative criteria by contrasting them. If the user provides

Generator	Level	Strategy	Pipelines
DIRECT	Instance	Single-step generation: given the input (and optionally a reference), directly prompt the LLM to produce a checklist	TICK (Cook et al., 2024), RLCF Direct (Viswanathan et al., 2025), RocketEval (Wei et al., 2025)
CONTRASTIVE	Instance	Counterfactual reasoning: generate candidate responses of varying quality, then derive criteria by contrasting good vs. bad responses	RLCF Candidate-based (Viswanathan et al., 2025), CRG (Liu et al., 2026)
INDUCTIVE	Corpus	Bottom-up induction: convert reviewer feedback or observations into general evaluation criteria, with built-in deduplication and selection	Zhou et al. (2025)
DEDUCTIVE	Corpus	Top-down decomposition: convert expert-defined evaluation dimensions into specific checklist questions, with optional augmentation	Lee et al. (2025)
INTERACTIVE	Corpus	Starting with defined dimensions, extract criteria from human and LLM think-aloud evaluation sessions via a multi-stage pipeline of clustering and question generation	Chu et al. (2025)

Table 1: Five checklist generation abstractions in `AutoChecklist`. Each represents a distinct strategy for deriving evaluation criteria. The rightmost column lists built-in pipelines that instantiate each generator with paper-specific prompt templates and scoring strategies.

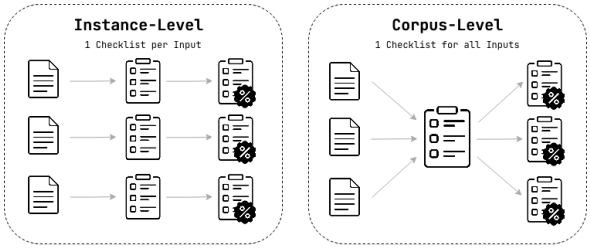


Figure 2: *Instance-level* generators produce one checklist per input; criteria are tailored to each specific task (target). *Corpus-level* generators produce one checklist for an entire dataset of targets, capturing general quality patterns derived from higher-level signals.

Pipeline	Generator	Score	Ref?
`tick`	DIRECT	pass	No
`rocketeval`	DIRECT	norm.	Yes
`rlcf_direct`	DIRECT	weight.	Yes
`rlcf_cand.`	CONTRASTIVE	weight.	Yes
`rlcf_cand._only`	CONTRASTIVE	weight.	No
`or_pairwise`	CONTRASTIVE	pass	No
`or_listwise`	CONTRASTIVE	pass	No
`checkeval`	DEDUCTIVE	pass	—
`interacteval`	INTERACTIVE	pass	—
`feedback`	INDUCTIVE	pass	—

Table 2: Built-in pipelines. Each pipeline pairs a generator with a default `ChecklistScorer` configuration and paper-specific prompt templates. Score metrics: pass rate, normalized, or weighted (see §3).

preference pairs, the generator can skip candidate generation and directly derive criteria by contrasting chosen and rejected responses. *Corpus-level generators* derive a shared checklist from corpus-wide signals. `InductiveGenerator` (INDUCTIVE) creates checklists "bottom-up" from unstructured observations, such as user feedback or reviews. `DeductiveGenerator` (DEDUCTIVE) performs top-down decomposition from expert-defined evaluation dimensions. `InteractiveGenerator` (INTERACTIVE) extracts criteria from simulated think-aloud protocols, wherein participants verbalize their thoughts during tasks. Table 1 shows how several papers ultimately share similar underlying steps for checklist generation.

Refiners. Refiners are optional post-processing steps composed in sequence before scoring. The `Deduplicator` merges semantically redundant questions via embedding similarity and LLM-based cluster merging. The `Tagger` applies additional filtering of items by a specified quality (e.g., generality, specificity). The `UnitTester` validates that each item is "LLM enforceable" (Zhou et al., 2025). The `Selector` uses beam search to optimize an objective function against checklist length. For most of the corpus-level generators, refinement steps are built into the generator classes; future work can refactor them into more modular steps.

Scorer. A unified `ChecklistScorer` class consolidates scoring strategies from three different papers into a single configurable interface. It operates in two modes: *batch* mode scores all checklist questions in a single LLM call, while *item* mode scores one question per call. It can configure use of chain-of-thought reasoning (`capture_reasoning`) and logprob-derived confidence scores (`use_logprobs`). The scorer computes three aggregate metrics: `pass_rate` (fraction of YES answers), `weighted_score` ($\sum w_i \cdot \mathbf{1}[a_i{=}\text{YES}] / \sum w_i$, using importance weights as in Viswanathan et al. (2025)), and `normalized_score` (calibrated from logprob confidence as in Wei et al. (2025)). A

`primary_metric` parameter designates which metric the `primary_score` attribute returns; each built-in pipeline sets this to match its original paper's scoring method. For a corpus of targets, both macro-average scores and Decomposed Requirements Following Ratio (DRFR), i.e., micro-average pass rate (Qin et al., 2024), are calculated.

Pipelines. Each stage in a pipeline is its own unit; any generator can be paired with any scorer, and refiners compose in sequence. For instance, a user can generate checklists with the INTERACTIVE strategy but score them with the normalized scorer, a previously unsupported combination.

Table 2 shows how the five generator abstractions are instantiated as ten built-in pipelines, i.e., pre-set configurations. Each row pairs a generator with a default scorer configuration and paper-specific prompt templates, but any generator can be combined with any scorer. Users can add new pipelines by registering custom prompt templates, without modifying library code. Base classes are also available for ease of developing more advanced custom components.

4 Deployment and Usage

`AutoChecklist` is `pip`-installable and supports three usage modes with increasing customization: a CLI for off-the-shelf evaluation, a web interface for interactive exploration and prompt iteration, and a Python API for full pipeline control. The library supports multiple LLM backends via a shared `LLMClient` protocol: OpenAI API, OpenRouter, and vLLM (both as an online server and via `VLLMOfflineClient` for local GPU inference). Structured output is handled via provider-enforced JSON schema (`response_format`), or appended format instructions and parsing as a fallback.

```
autochecklist run --pipeline tick \
  --data eval_data.jsonl \
  --output results.jsonl \
  --generator-model openai/gpt-5-mini \
  --scorer-model openai/gpt-5-mini
```

Figure 3: CLI: end-to-end evaluation with a built-in pipeline.

Command-Line Interface. The CLI provides three subcommands for use with registered components: `run` (generate checklists and score in one step), `generate` (produce checklists without scoring), and `score` (score targets against a pre-existing checklist). A fourth subcommand, `list`, discovers available generators, scorers, and refiners. Users specify a built-in pipeline (e.g., `--pipeline tick`), a custom generator prompt file (`--generator-prompt my_eval.md`), or a saved pipeline config JSON. Runs are resumable after interruption, with incremental saving of results.

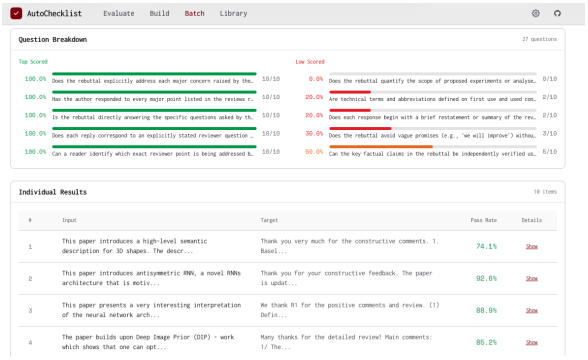


Figure 4: The Batch results page showing per-item score breakdowns and aggregate statistics for a dataset evaluation run.

User Interface. A locally hosted interface built with FastAPI and Next.js provides interactive access to most library functionality without writing code, easily launched with `autochecklist ui`.

The main page **Evaluate** offers three tabs. *Custom Eval* provides full prompt editing: users choose a generator class, edit generator and scorer prompts in tabbed editors, load from saved prompt templates, select an output format and scorer type, then run an evaluation and view checklist and score results. The *Compare* tab enables side-by-side comparison of built-in methods and custom pipelines on the same input, with results displayed as horizontally scrollable method cards (see Figure 1). *Reference* includes descriptions for all generators, scorer options, and pipelines. The **Build** interface supports `Deductive` corpus-level checklist generation for downstream use.

Configurations from these workflows can be saved to the **Library**, which stores *Checklists* for saved checklists (clicking in allows for edits), *Prompt Templates* for reusable prompts with placeholders, and *Pipelines* for saving complete evaluation configurations that bundle generator class, prompts, scorer settings, and output format.

To process more data samples at once, the **Batch** page supports CSV/JSON data upload, evaluation mode with saved Library items, progress tracking, and a results page with basic aggregate statistics and per-item score breakdowns (such as in Figure 4). There is also the option to export the experiment in Python code to run the same pipeline

Pipeline	Generator	W/L/T	Δ	Cohen's d	t-test p
`tick`	`gpt-5-mini`	**75**/10/15	**+0.286**	**0.919**	6.3e-15
	`sonnet-4.6`	67/11/22	+0.259	0.846	3.0e-13
`rlcf`	`gpt-5-mini`	70/16/14	+0.228	0.785	4.9e-12
	`sonnet-4.6`	54/11/35	+0.203	0.696	4.4e-10

Table 3: Preference discrimination on RewardBench pairs using instance-level pipelines. Win = chosen scores higher than rejected, and W/L/T = ratio of Win/Loss/Tie scores (%).

through a script, which is preferred for very large datasets and for more fine-grained customization.

Python API. The Python API offers full control over the pipeline, ideal for large-scale checklist evaluation with finalized generator and scorer configurations. The `pipeline()` factory is the recommended entry point: it resolves a pipeline name to a pre-configured generator and scorer, accepts optional refiners, and returns a callable that generates and scores in one step. For finer-grained control, `ChecklistPipeline` allows explicit composition of independently configured generators, refiners, and scorers. Components can also be used outside of any pipeline (e.g., calling a generator and scorer directly). Batch results provide both macro-average scores and Decomposed Requirements Following Ratio (DRFR, i.e., a micro-average pass rate) (Qin et al., 2024), and can be exported as DataFrames or JSONL files. Users can register custom pipelines via `register_custom_pipeline()` and share configurations as JSON files, making new evaluation setups reusable without modifying library code. See Appendix B for code examples.

5 Validation

We validate that our pipeline implementations produce useful and effective checklists for two benchmarks: RewardBench (Lambert et al., 2025) for instance-level methods and SummEval (Fabbri et al., 2021) for corpus-level methods. All experiments use `gpt-5` as the scorer; to test robustness of the pipelines with different checklist generators, we use both `gpt-5-mini` and `claude-sonnet-4.6`.

Instance-Level. We test whether instance-level checklist scores can discriminate between preferred and rejected responses. From RewardBench, we sample 100 preference pairs and generate a checklist for each pair's input prompt using the `tick` (DIRECT, reference-free) and `rlcf_candidate_only` (CONTRASTIVE, reference-free) pipelines. Each method scores both the chosen and rejected response, and we measure win ratio (fraction where the chosen response scores higher), mean score delta, and effect size.

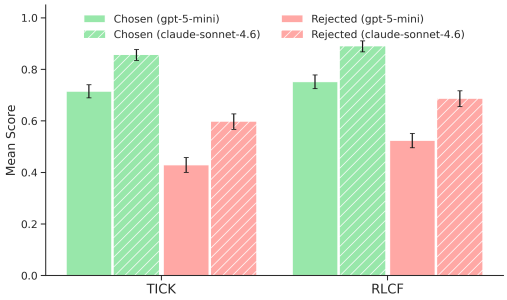
		`gpt-5-mini`		`sonnet-4.6`	
Pipeline	Dimension	ρ_s	MAE	ρ_s	MAE
`checkeval`	Coherence	0.723	1.104	0.733	1.041
	Consistency	0.756	0.925	0.804	**0.627**
	Fluency	**0.819**	**0.829**	**0.805**	**0.863**
	Relevance	0.571	**0.719**	**0.639**	**0.652**
`interacteval`	Coherence	**0.755**	**0.703**	**0.775**	**0.639**
	Consistency	**0.835**	**0.565**	**0.808**	**0.627**
	Fluency	0.670	0.980	0.759	1.047
	Relevance	**0.612**	0.910	0.587	0.838

Table 4: Corpus-level checklist validation on SummEval, for generator models `gpt-5-mini` and a `claude-sonnet-4.6`. ρ_s: Spearman correlation with human scores; MAE on rescaled 1–5 scores; bold is best per dimensions. All ρ_s are significant ($p < .001$).

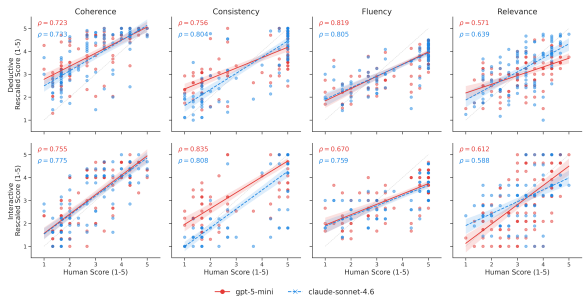
Table 3 shows that both pipelines significantly discriminate preferred responses (also in Figure 5a). For `gpt-5-mini`, `tick` achieves a 75% win rate with a large effect size ($d = 0.919$, $p < .001$), and `rlcf_candidate_only` achieves 70% ($d = 0.785$, $p < .001$). `claude-sonnet-4.6` has a smaller win to tie ratio, but still produces checklists that give significantly scores to preferred responses. These results confirm that the library's instance-level pipelines produce checklist criteria that align with human preference judgments of quality.

Corpus-Level. We validate corpus-level methods by measuring correlation with expert quality judgments on the SummEval benchmark, which provides human scores on a 1–5 scale across four dimensions: coherence, consistency, fluency, and relevance. We generate corpus-level checklists using `checkeval` (DEDUCTIVE) and `interacteval` (INTERACTIVE), then score 100 summaries against each checklist. We map checklist scores to the 1–5 scale (pass rate $\times$ 4 + 1).

Both methods achieve strong correlations across all four dimensions, across generator models (Table 4). All correlations are significant at $p < .001$ (Figure 5b). `interacteval` achieves the highest dimensional correlations on coherence ($\rho = 0.775$ with `claude-sonnet-4.6`) and consistency ($\rho = 0.835$ with `gpt-5-mini`), outperforming `checkeval` on both attributes in Mean Absolute Error (MAE) from human ratings as well. `checkeval` beats `interacteval` on correlation with fluency ($\rho > 0.805$), also with lowest MAE on this dimension. MAE for relevance is lowest for `checkeval` methods, though correlation strength is more mixed



(a) Mean checklist pass rates from `tick` and `rlcf` on RewardBench pairs.



(b) Rescaled checklist pass rates against human assigned scores on SummEval, across four dimensions.

Figure 5: Validation of built-in pipelines on RewardBench and SummEval.

depending on generator model. These results confirm that the library's corpus-level pipelines produce checklists whose pass rates align with expert judgments, a conclusion that holds across different generator models/providers.

6 Case Study: Author Rebuttal Checklists

To demonstrate how `AutoChecklist`'s composable design enables application to new domains, we apply checklist-based evaluation to peer review rebuttals, a setting where no existing checklist method has been tested. The adaptation required only prompt modifications and the library's existing generator and scorer infrastructure.

Review-rebuttal dynamics of peer review have been studied extensively, uncovering their effects on final scores and successful rebuttal strategies (Gao et al., 2019; Dershowitz and Verma, 2023; He et al., 2026; Ma et al., 2026). While there are efforts to evaluate review quality and usefulness (Ryu et al., 2025; Sadallah et al., 2025), no prior study has applied checklist-based evaluation to automatically assess *rebuttal* quality. Checklists offer a natural fit: each item can target a specific concern raised by the reviewer, yielding interpretable, fine-grained assessment of rebuttal responses.

Setup. We collect 100 review–rebuttal pairs via the OpenReview API and evaluate them with four generator configurations spanning both instance-level and corpus-level approaches. We select pairs from ICLR 2019, the latest year with post-rebuttal rating change data.

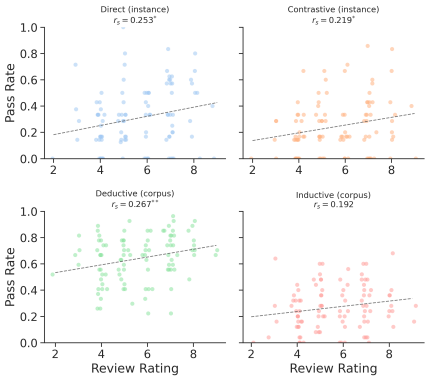
We register custom generator prompts specific to the peer review context. Additionally, DEDUCTIVE uses three dimensions from Ma et al. (2026)—Relevance, Argumentation Quality, and Communication Quality—each with sub-dimensions (e.g., Semantic Alignment, Evidence Support, Constructiveness). INDUCTIVE takes 1,000 labeled reviewer sentences related to "weaknesses" as input, generates candidate checklist items that capture common patterns of concern across the corpus, and then refines the checklist with deduplication, tagging, and optimal selection.

	RATING	ACCEPTANCE		RATING CHANGE	
Generator	r_s	$\lvert d\rvert$	$\Delta\bar{x}$	r_{pb}	AUC
DIRECT	0.253*	**0.296***	**+0.124**	0.143	0.587
CONTRASTIVE	0.219*	0.270*	+0.094	0.084	0.531
DEDUCTIVE	**0.267****	0.282*	+0.094	**0.242***	**0.668**
INDUCTIVE	0.192	0.240*	+0.068	0.172	0.630

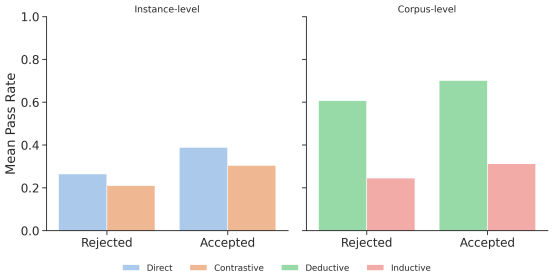
Table 5: Evaluation of checklist-based rebuttal scoring on ICLR 2019. RATING: Spearman r_s between pass rate and reviewer rating. ACCEPTANCE: rank-biserial effect size $|d|$ and pass rate difference $\Delta\bar{x} = \bar{x}_{acc} - \bar{x}_{rej}$. RATING CHANGE: point-biserial r_{pb} and ROC-AUC for predicting whether the reviewer changed their rating post-rebuttal. $^*p < .05$, $^{**}p < .01$, $^{***}p < .001$.

Results. We evaluate each pipeline's checklist pass rates against three external signals of rebuttal quality: (1) RATING: Spearman correlation between pass rate and the reviewer's numeric rating; (2) ACCEPTANCE: rank-biserial effect size between accepted and rejected papers; and (3) RATING CHANGE: point-biserial correlation and ROC-AUC for predicting whether the reviewer changed their rating post-rebuttal.

Table 5 reveals the patterns across generator types. DIRECT shows the strongest acceptance discrimination and moderate rating correlation. CONTRASTIVE shows similar patterns with slightly lower effect sizes. DEDUCTIVE achieves the highest rating correlation, and both DEDUCTIVE and INDUCTIVE are the only methods with significant rating-change prediction, i.e., corpus-level checklists may better capture the signals in rebuttals that convince reviewers to update their score. Figure 6 shows each pipeline's trends against reviewer ratings and acceptance outcomes.



(a) Checklist pass rate vs. reviewer rating on review–rebuttal pairs; dashed lines show linear fits.



(b) All four generators assign higher pass rates to rebuttals for accepted papers vs. rejected papers.

Figure 6: Author rebuttal checklists generated from ICLR 2019 reviews.

This case study illustrates two features of the library's design. First, the custom prompt API allows adapting any pipeline to a new domain without modifying library code, as the existing generator and scorer infrastructure is reusable with custom prompts. Second, the composable pipeline makes it straightforward to compare instance-level and corpus-level approaches on the same data, revealing their complementary strengths.

7 Conclusion and Future Work

We present `AutoChecklist`, a composable Python library that unifies LLM-based checklist evaluation. Beyond consolidating existing methods, the library contributes a taxonomy of five generator abstractions that organizes the design space of checklist generation by reasoning strategy, as well as a unified `ChecklistScorer` that consolidates three scoring strategies. These utilities, combined with the composable *Generator* $\rightarrow$ *Refiner* $\rightarrow$ *Scorer* architecture, enable systematic comparison and rapid extension of checklist evaluation methods.

Validation on RewardBench and SummEval confirms that library-generated checklists align with human judgments. The peer review rebuttal case study (§6) demonstrates that adapting to a new domain requires only new custom prompt templates and no library code changes. Future directions include integrating new checklist methods as they emerge, supporting human-in-the-loop generation and refinement for corpus-level checklists, and providing more library features through the UI.

Limitations. We aim to faithfully implement the built-in pipelines based on their papers and code, but we make some changes for standardization (e.g., enforcing JSON structured outputs) and generalization (e.g., modifying original prompts). We provide default prompts calibrated to the original papers, but users adapting to new domains should expect to iterate on prompt templates, especially between models. Using OpenAI or OpenRouter providers also incurs API costs that users should account for. Finally, there is not yet full end-to-end support for corpus-level pipelines that require multi-stage human-in-the-loop workflows (e.g., INDUCTIVE's refinement pipeline and an interface to collect human and LLM data for INTERACTIVE's input attributes).

Ethical Considerations

Automatic checklist-based evaluation is imperfect, especially with multiple LLM-based steps. We caution that checklist scores should not replace human judgment in high-stakes settings without careful validation.

The library is released under the Apache 2.0 license. Built-in pipelines are adapted based on publicly available papers and code. Validation experiments use open-source benchmarks (RewardBench, SummEval) and publicly accessible OpenReview data. No private or personally identifiable data is used. Users do not expose API keys or data through using the package or UI.

Acknowledgments

We thank the anonymous reviewers for their feedback, as well as the Chicago Human+AI Lab for helpful discussions during this work. This work is supported in part by a University of Chicago UU fellowship and NSF grants IIS-2126602, 2302785.

References

Seong Yeub Chu, Jong Woo Kim, and Mun Yong Yi. 2025. Think together and work better: Combining humans' and llms' think-aloud outcomes for effective text evaluation. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–23.

Jonathan Cook, Tim Rocktäschel, Jakob Nicolaus Foerster, Dennis Aumiller, and Alex Wang. 2024. TICKing all the boxes: Generated checklists improve LLM evaluation and generation. In *Language Gamification - NeurIPS 2024 Workshop*.

Nachum Dershowitz and Rakesh M. Verma. 2023. Rebutting rebuttals. *Commun. ACM*, 66(9):35–41.

Jacob Dineen, Aswin Rrv, Qin Liu, Zhikun Xu, Xiao Ye, Ming Shen, Zhaonan Li, Shijie Lu, Chitta Baral, Muhao Chen, and Ben Zhou. 2025. QA-LIGN: Aligning LLMs through constitutionally decomposed QA. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 20619–20642, Suzhou, China. Association for Computational Linguistics.

Alexander R. Fabbri, Wojciech Kryściński, Bryan McCann, Caiming Xiong, Richard Socher, and Dragomir Radev. 2021. SummEval: Re-evaluating summarization evaluation. *Transactions of the Association for Computational Linguistics*, 9:391–409.

Momoka Furuhashi, Kouta Nakayama, Takashi Kodama, and Saku Sugawara. 2025. Are checklists really useful for automatic evaluation of generative tasks? In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 10641–10664.

Yang Gao, Steffen Eger, Ines Hanselowski, and Iryna Gurevych. 2019. Does my rebuttal matter? insights from a major NLP conference. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1274–1290. Association for Computational Linguistics.

Zhitao He, Yiran Lyu, and Pascale Fung. 2026. Dancing in chains: Strategic persuasion in academic rebuttal via theory of mind. In *The Fourteenth International Conference on Learning Representations*.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. 2025. RewardBench: Evaluating reward models for language modeling. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1755–1797, Albuquerque, New Mexico. Association for Computational Linguistics.

Yukyung Lee, JoongHoon Kim, Jaehee Kim, Hyowon Cho, Jaewook Kang, Pilsung Kang, and Najoung Kim. 2025. CheckEval: A reliable LLM-as-a-judge framework for evaluating text generation using checklists. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Bill Yuchen Lin, Yuntian Deng, Khyathi Chandu, Faeze Brahman, Abhilasha Ravichander, Valentina Pyatkin, Nouha Dziri, Ronan Le Bras, and Yejin Choi. 2024. Wildbench: Benchmarking llms with challenging tasks from real users in the wild. *Preprint*, arXiv:2406.04770.

Tianci Liu, Ran Xu, Tony Yu, Ilgee Hong, Carl Yang, Tuo Zhao, and Haoyu Wang. 2026. Openrubrics: Towards scalable synthetic rubric generation for reward modeling and llm alignment. *Preprint*, arXiv:2510.07743.

Qianli Ma, Chang Guo, Zhiheng Tian, Siyu Wang, Jipeng Xiao, Yuanhao Yue, and Zhipeng Zhang. 2026. Paper2rebuttal: A multi-agent framework for transparent author response assistance. *Preprint*, arXiv:2601.14171.

Yiwei Qin, Kaiqiang Song, Yebowen Hu, Wenlin Yao, Sangwoo Cho, Xiaoyang Wang, Xuansheng Wu, Fei Liu, Pengfei Liu, and Dong Yu. 2024. InFoBench: Evaluating instruction following ability in large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13025–13048, Bangkok, Thailand. Association for Computational Linguistics.

Haoran Que, Feiyu Duan, Liqun He, Yutao Mou, Wangchunshu Zhou, Jiaheng Liu, Wenge Rong, Zekun Moore Wang, Jian Yang, Ge Zhang, Junran Peng, Zhaoxiang Zhang, Songyang Zhang, and Kai Chen. 2024. Hellobench: Evaluating long text generation capabilities of large language models. *Preprint*, arXiv:2409.16191.

Hyun Ryu, Doohyuk Jang, Hyemin S. Lee, Joonhyun Jeong, Gyeongman Kim, Donghyeon Cho, Gyouk Chu, Minyeong Hwang, Hyeongwon Jang, Changhun Kim, Haechan Kim, Jina Kim, Joowon Kim, Yoonjeon Kim, Kwanhyung Lee, Chanjae Park, Heecheol Yun, Gregor Betz, and Eunho Yang. 2025. Reviewscore: Misinformed peer review detection with large language models. *Preprint*, arXiv:2509.21679.

Abdelrahman Sadallah, Tim Baumgärtner, Iryna Gurevych, and Ted Briscoe. 2025. The good, the bad and the constructive: Automatically measuring peer review's utility for authors. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 28991–29021, Suzhou, China. Association for Computational Linguistics.

Clemencia Siro, Pourya Aliannejadi, and Mohammad Aliannejadi. 2026. Learning to judge: LLMs designing and applying evaluation rubrics. In *Findings of the European Chapter of the Association for Computational Linguistics*.

Vijay Viswanathan, Yanchao Sun, Xiang Kong, Meng Cao, Graham Neubig, and Tongshuang Wu. 2025.

Checklists are better than reward models for aligning language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Yuxuan Wan, Tianqing Fang, Zaitang Li, Yintong Huo, Wenxuan Wang, Haitao Mi, Dong Yu, and Michael R. Lyu. 2026. Inference-time scaling of verification: Self-evolving deep research agents via test-time rubric-guided verification. *arXiv preprint arXiv:2601.15808*.

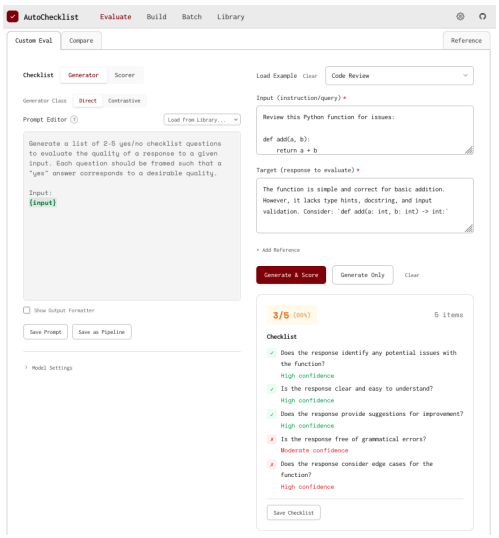
Tianjun Wei, Wei Wen, Ruizhi Qiao, Xing Sun, and Jianghong Ma. 2025. Rocketeval: Efficient automated LLM evaluation via grading checklist. In *The Thirteenth International Conference on Learning Representations*.

Zhichao Yang, Sepehr Janghorbani, Dongxu Zhang, Jun Han, Qian Qian, Andrew Ressler II, Gregory D. Lyng, Sanjit Singh Batra, and Robert E. Tillman. 2026. Health-SCORE: Towards scalable rubrics for improving health-LLMs. *arXiv preprint arXiv:2601.18706*.

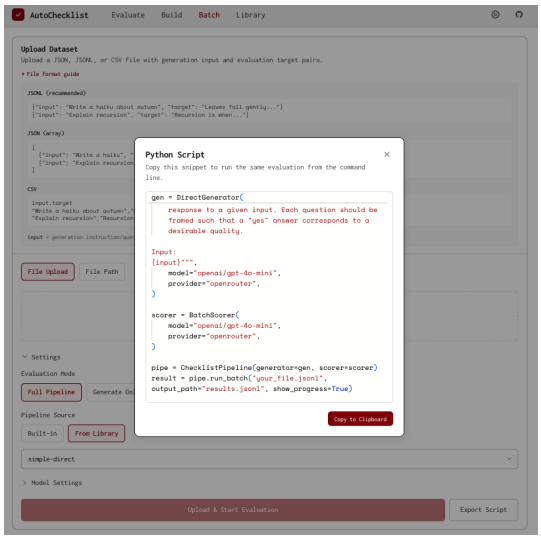
Jiayi Ye, Yanbo Wang, Yue Huang, Dongping Chen, Qihui Zhang, Nuno Moniz, Tian Gao, Werner Geyer, Chao Huang, Pin-Yu Chen, and 1 others. 2024. Justice or prejudice? quantifying biases in llm-as-a-judge. *arXiv preprint arXiv:2410.02736*.

Karen Zhou, John Michael Giorgi, Pranav Mani, Peng Xu, Davis Liang, and Chenhao Tan. 2025. From feedback to checklists: Grounded evaluation of AI-generated clinical notes. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*. Association for Computational Linguistics.

A Additional Figures



(a) The *Custom Eval* tab, where users edit generator and scorer prompts, select an output format, and run single-input evaluations.



(b) The *Batch* page, where users can use built-in and custom pipelines to generate checklists and/or score a dataset, along with an option to export the pipeline as a script.

Figure 7: Additional UI pages to support prompt iteration and dataset evaluation.

B Code Samples

```python
from autochecklist import pipeline

# Built-in pipeline: generate checklist + score in one call
pipe = pipeline("tick", generator_model="openai/gpt-5-mini", scorer_model="openai/
    gpt-5-mini")
result = pipe(input="Write a haiku about autumn.", target="Leaves fall gently down
    ...")
print(f"Pass rate: {result.pass_rate:.0%}")

# Evaluation over a dataset
data = [{"input": "Write a haiku", "target": "Leaves fall..."}, ...]
result = pipe.run_batch(data, show_progress=True)
print(f"Macro pass rate: {result.macro_pass_rate:.0%}")
result.to_dataframe()  # or result.to_jsonl("results.jsonl")
```

Figure 8: Python API: basic usage

```python
from autochecklist import DirectGenerator, ChecklistScorer, pipeline
from autochecklist.refiners import Deduplicator, Tagger
from pathlib import Path

# Custom generator prompt (inline or from a Markdown file)
gen = DirectGenerator(custom_prompt=Path("prompts/my_eval.md"), model="openai/gpt-5-mini")
checklist = gen.generate(input="Write a haiku about autumn.")

# Mix and match: chain refiners, then score with a different scorer
checklist = Deduplicator(model="openai/gpt-5-mini").refine(checklist)
checklist = Tagger(model="openai/gpt-5-mini").refine(checklist)
scorer = ChecklistScorer(mode="item", primary_metric="weighted", model="openai/gpt-5-mini")
score = scorer.score(checklist, target="Leaves fall gently down...")

# Or compose everything through the pipeline factory
pipe = pipeline("tick", generator_model="openai/gpt-5-mini",
                refiners=["deduplicator", "tagger"], scorer="weighted")

# Switch LLM provider: use a local vLLM server instead of OpenRouter
pipe = pipeline("tick", provider="vllm", base_url="http://localhost:8000/v1",
                generator_model="meta-llama/Llama-3-8B")

# Register and save custom pipelines for reuse
from autochecklist import register_custom_pipeline, save_pipeline_config
register_custom_pipeline("my_eval", generator_prompt=Path("prompts/my_eval.md"), scorer="weighted")
save_pipeline_config("my_eval", "my_eval.json")   # share as a portable JSON config

pipe = pipeline("my_eval", generator_model="openai/gpt-5-mini", scorer_model="openai/gpt-5-mini")
```

Figure 9: Python API: custom prompts, providers, and composition