Time to Impeach LLM-as-a-Judge: Programs are the Future of Evaluation

Tzu-Heng Huang [1] **Harit Vishwakarma** [1] **Frederic Sala** [1]

Abstract

Large language models (LLMs) are widely used to evaluate the quality of LLM generations and responses, but this leads to significant challenges: high API costs, uncertain reliability, inflexible pipelines, and inherent biases. To address these, we introduce **PAJAMA** (Program-As-a-Judge for Automated Model Assessment), a new alternative that uses LLMs to ***synthesize executable judging programs*** instead of directly scoring responses. These synthesized programs can be stored and run locally, costing orders of magnitude less while providing interpretable, and auditable judging logic that can be easily adapted. Program-based judges mitigate biases, improving judgment consistency by **15.83%** and reducing biased responses by **23.7%** on average compared to a Qwen2.5-14B-based LLM-as-a-judge. When program judgments are distilled into a model, PAJAMA outperforms LLM-as-a-judge on the challenging CHAT-HARD subset of RewardBench, outperforming metrics by **2.19%** on Prometheus and **8.67%** on the JudgeLM dataset, all at three orders of magnitude lower cost.

1. Introduction

General-purpose large language models (LLMs) are now the standard for *automated evaluation* of generative model responses. In the typical LLM-as-a-judge setup, a judge LLM evaluates a user query with two LLM-generated answers, then selects the better one (Wei et al., 2024). These judgments can be viewed as preference labels that enable ranking LLMs by performance (Chiang et al., 2024; Zheng et al., 2023), efficiently verifying LLM agent behaviors (Zhuge et al., 2024), curating datasets (Wettig et al., 2024), or distilling into specialized reward models (Christiano et al., 2017).

Despite their promise, current LLM-as-a-judge approaches have several drawbacks:

- **High cost**: Querying state-of-the-art models, e.g., Open AI o3, or Claude 3 Opus (Anthropic, 2024), can run into thousands of dollars for large evaluation sets. For example, researchers spent about $4,000 to generate 100K high-quality judgments using GPT-4 (Zhu et al., 2023).
- **Uncertain reliability**: Although LLMs can explain their verdicts, their outputs may exhibit inconsistent reliability in adhering to rubrics (Yu et al., 2024).
- **Inflexible pipelines**: Any minor change to the evaluation rubric requires re-running the entire pipeline, incurring additional expenses.
- **Inherent biases**: LLMs trained on web datasets encode social and stylistic biases (e.g., gender preferences, or favoring emojis), which can skew decisions (Adila et al., 2024; Chen et al., 2024; Ye et al., 2024).

We address these by relying on a *simple but powerful* notion. Instead of prompting an LLM for preferred answers, ***we ask it to generate the judging logic it would use and encode it into an executable program.*** In other words, the model is asked to synthesize a compact function, e.g., a few lines of Python, that encodes its evaluation criteria. These are executed to obtain a quality score for each response.

This *program synthesis*-style strategy offers several advantages. ***First***, API costs now scale with the number of generated programs, not the dataset size, significantly reducing expenses. Once generated, judging programs can be stored and executed locally for any new query at no additional cost. ***Second***, synthesized programs are interpretable, allowing practitioners to audit each line, refine rubrics, or insert heuristic rules to minimize bias. ***Third***, by exposing the full decision logic, this approach turns a black-box judging model into a transparent, inspectable evaluation process.

While promising, translating LLM judgments into executable code raises new challenges. It is common for synthesized programs to be repetitive, reusing similar criteria with minor variations. To encourage diversity, we introduce six distinct criteria—***each expressible as code***—to guide LLMs in generating useful programs. Additionally, individual program outputs can be noisy, and different programs may capture complementary signals. We address this by combining our approach with the weak supervision framework (Ratner et al., 2016; 2017; 2019; Fu et al., 2020). By

[1]Department of Computer Sciences, University of Wisconsin-Madison, USA. Correspondence to: Tzu-Heng Huang <thuang273@wisc.edu>.

ICML 2025 Workshop on Programmatic Representations for Agent Learning, Vancouver, Canada.

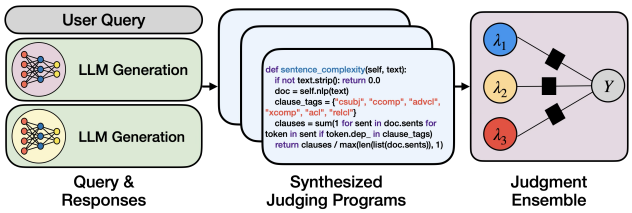

Figure 1. PAJAMA's General Workflow.

modeling program outputs, we aggregate program judges into a collective signal that can outperform naïve ensembling approaches like majority voting.

Results and Contributions. To tackle the above, we propose **PAJAMA** (Program-As-a-Judge Automated Model Assessment), a lower-cost and lower-bias evaluation system that relies on synthesized judging programs. Each program can serve directly as a judge or have its judgments distilled into a reward model for improved generalization. For example, when distilled into a reward model, PAJAMA outperforms LLM-as-a-judge-distilled reward models on the challenging CHAT-HARD subset of RewardBench (Lambert et al., 2024), achieving **+2.19%** on Prometheus (Kim et al., 2023) and **+8.67%** on JudgeLM dataset (Zhu et al., 2023), while decreasing API costs by approximately 3500× and 2500×, respectively. We also test PAJAMA on high-bias responses, observing that it produces consistent correct responses and improved error rate over LLM-as-a-judge. Across four common bias types, it enhances consistency by **15.83%** and reduces the biased-answer win rate by **23.7%** relative to Qwen2.5-14B.

2. Evaluation System: PAJAMA

We start with an overview of PAJAMA's general workflow, followed by the problem setup (Sec. 2.1), then describe program synthesis for judging code (Sec. 2.2) and discuss how we combine multiple judgments to produce an aggregated evaluation decision (Sec. 2.3).

General Workflow. The workflow of PAJAMA is illustrated in Fig. 1. First, users collect queries and LLM generation pairs, then create prompts to guide LLMs in generating programs with judging logic. Prompts can vary in their design and specified evaluation criteria, enabling the creation of multiple judging programs. Using weak supervision, aggregated preference labels are inferred from program outputs. Finally (and optionally), these judgments can be used to train a distilled reward model for local use.

2.1. Problem Setup

We consider user queries and responses drawn from the space of free-form text, $\Sigma^\star$, e.g., all possible natural language strings. We denote the space of queries as $\mathcal{X}$ and the space of responses as $\mathcal{Y}$. For any query $x \in \mathcal{X}$, we evaluate two responses, $y_1, y_2 \in \mathcal{Y}$, generated by the same or different LLMs. Other approaches directly score each candidate generation, i.e., assign a score to each y.

Ground-truth assessments for reward or evaluation models are accurate but costly and slow due to human annotations. Using LLM-as-a-judge is faster but incurs high inference costs, and may embed LLM biases into ground truth and downstream models. We address these issues with synthesized judging programs as an alternative.

2.2. Program Synthesis

We propose a prompting template to query LLMs for synthesizing programmatic judges. Using GPT-4o (Hurst et al., 2024), we generate these judges as follows:

Prompt used to synthesize judging programs.

> You are a judge tasked with evaluating LLM-generated responses to a given question. Write your evaluation logic as Python code, returning a numeric score for a response where higher values indicate better quality. Use third-party libraries (e.g., embedding models, nlp metrics) as needed.
> **def judging_function(query, response)**:

We chose this approach for its simplicity, but we note that *more sophisticated approaches to program synthesis can be seamlessly swapped in*. Next, we propose six distinct criteria that can be incorporated into the program synthesis prompt and thus be encoded into executable Python code. We describe each criterion below. We likewise note that these can be easily swapped for other criteria relevant to the particular task of interest.

- **Structure**: A judge evaluates text by analyzing features such as transition markers (e.g., "therefore," "however"), sentence count, paragraph length, and the presence of headings, questions, or emphasized text. More markers and structured elements indicate better quality.
- **Relevance**: A judge assesses semantic alignment between the question and response. For example, one approach uses TF-IDF to compute cosine similarity for lexical overlap. Another can employ semantic embeddings to measure deeper contextual similarity (Multi-Granularity, 2024).
- **Readability**: A judge analyzes grammar errors, information density, and counts repetitive words. It also includes third-party libraries to compute readability metrics like the Flesch–Kincaid grade level (Kincaid et al., 1975).
- **Bias**: A judge evaluates response objectivity using sentiment analysis and regex patterns to detect stance and biased keywords, ensuring neutrality.
- **Factuality**: A judge assesses factual accuracy, using fine-tuned BERT models to verify content correctness (Feng

et al., 2023).
- **Safety**: A judge employs fine-tuned BERT models, trained to detect hate speech or harmful content, to ensure responses are safe and appropriate (Vidgen et al., 2021).

We note that these judging principles, evaluation rubrics, and keyword lists are all generated by the GPT4o. Each of the resulting programs can function as an independent judge to assess the quality of LLM responses.

2.3. Judgment Ensemble

The outputs of program-based judges can be noisy or incomplete. For this reason, we seek to *combine them*. Doing so enables reducing noise and taking advantage of complementary signals. We perform the aggregation by borrowing weak supervision techniques.

Suppose we generate m judging programs, each implementing a scoring function $\lambda_i : \mathcal{X} \times \mathcal{Y} \to [0, 1]$. For a prompt x and two candidate responses y_1 and y_2, we discretize their scores by defining

$$\bar{\lambda}_i(x, y_1, y_2) = \begin{cases} +1, & \text{if } \lambda_i(x, y_1) > \lambda_i(x, y_2), \\ -1, & \text{otherwise.} \end{cases}$$

In words, a judge outputs $+1$ when it prefers y_1 over y_2, and -1 when it prefers the opposite.

Borrowing the terminology of weak supervision (Ratner et al., 2019; Shin et al., 2022), we model a joint distribution over the noisy outputs $\bar{\lambda}_i$, conditioned on the (latent) true preference label $Y(x, y_1, y_2) \in \{+1, -1\}$, as $\Pr\left(\bar{\lambda}_1, \ldots, \bar{\lambda}_m \mid x, y_1, y_2\right) = \frac{1}{Z_\theta} \exp\left(-\theta_i \sum_{i=1}^{m} \bar{\lambda}_i Y\right)$, where θ_i denotes the reliability weight, i.e., accuracy, learned for judge i, and Z_θ is the normalizing partition function ensuring a valid probability distribution. Once these weights are learned, the ensemble can infer a consensus preference label for any triple (x, y_1, y_2) (Ratner et al., 2016; 2017; 2019; Fu et al., 2020). These inferred preference labels can be used to evaluate LLM generations directly, or be distilled into a reward model (Christiano et al., 2017). The distilled model is often able to generalize beyond the constituent programs (Shin et al., 2025).

3. Experiments

We evaluate PAJAMA's effectiveness through two experiments. We first compare its effectiveness to LLM-as-a-judge (Sec. 3.1) then assess its robustness when facing responses with intentionally injected biases (Sec. 3.2). Our goals are to validate the following claims:

C1. Lower evaluation costs: PAJAMA can yield competitive evaluation results at significantly lower cost compared to LLM-as-judge. Its performance scales as the number of synthesized programs increases.

C2. Bias reduction: PAJAMA can mitigate biases, maintaining consistent, correct responses and reducing the biased-response win rate.

3.1. Effectiveness in Evaluation (C1)

Setup. We employ three pairwise comparison datasets: Prometheus (Kim et al., 2023), JudgeLM (Zhu et al., 2023), and PandaLM (Wang et al., 2024) to assess the performance of our program-as-a-judge approach. We prompt GPT-4o (Achiam et al., 2023) to generate 52 judging programs, execute them, and aggregate program outputs via Snorkel (Ratner et al., 2017) to create preference labels for the training dataset. For LLM-as-a-judge comparison, the training datasets for Prometheus and JudgeLM are produced by GPT-4, while PandaLM uses GPT-3.5-Turbo. Both training datasets are used to fine-tune Gemma-2B-it (Team et al., 2024), distilling their judgments into reward models. We evaluate the performance of these distilled models on (i) held-out splits of the respective datasets (in-domain) and (ii) RewardBench, a standard out-of-domain benchmark for reward models (Lambert et al., 2024).

Results. Table 1 compares the performance of distilled reward models constructed using two approaches. On in-domain held-out evaluation sets, PAJAMA gives competitive results in comparison to LLM-as-a-judge, while incurring significantly less cost in obtaining preference labels, requiring only \$0.053—three orders of magnitude cheaper than LLM-as-a-judge. This cost efficiency holds in RewardBench evaluations as well. Moreover, while LLM-as-a-judge achieves higher accuracy in the CHAT and Reasoning categories, the program-as-a-judge-distilled reward model outperforms it in the more challenging CHAT-HARD category, with gains of +2.19% over Prometheus and +8.67% on JudgeLM datasets.

Fine-grained Analysis on Prometheus. Figure 3 in the Appendix decomposes the Prometheus results across all 23 RewardBench subsets, including the base model (Gemma-2B-it) for comparison. We observe that the program-as-a-judge approach improves the base model's performance, particularly in reasoning tasks. But surprisingly, this enhancement occurs even though our synthesized judging programs lack specific criteria for evaluating math or programming tasks. On the safety subset, our approach reduces the base model's performance by 2.7%, indicating that synthesized rules may struggle to generalize to emotional or sentimental policies.

Scaling with the Number of Synthesized Programs. A prominent feature of our framework is that it can work with as many programs as we like with virtually zero cost. Intuitively, having more programs with diverse judging criteria can make PAJAMA more effective.

Table 1. PAJAMA achieves competitive preference label accuracy (%) at a significantly lower cost than LLM-as-a-judge.

	Dataset Size	Estimated Cost (using GPT4o)		Evaluation Set (In-Domain)		RewardBench (Out-of-Domain)					
						Chat		Chat Hard		Reasoning	
		LLM-as-a-Judge	**PAJAMA**	LLM-as-a-Judge	**PAJAMA**	LLM-as-a-Judge	**PAJAMA**	LLM-as-a-Judge	**PAJAMA**	LLM-as-a-Judge	**PAJAMA**
Prometheus	59,928	$183.67	$0.053	—	—	91.90	75.00	38.38	**40.57**	82.26	59.54
JudgeLM	55,751	$133.04	$0.053	82.88	72.82	95.25	66.76	48.68	**57.35**	73.71	45.71
PandaLM	233,227	$300.37	$0.053	74.47	63.26	94.41	83.24	42.43	31.91	79.60	58.52

Table 2. PAJAMA is more bias-resistent over LLM-as-a-judge.

	Position	**Gender**		**Rich-content**		**Reference**		**Average**	
	Consistency	Consistency	Biased Response Win Rate	Consistency	Biased Response Win Rate	Consistency	Biased Response Win Rate	Consistency (4 biases)	Biased Response Win Rate (3 biases)
Llama3-8B	45.07	50.59	11.27	60.33	45.54	53.05	53.52	52.41	36.78
Qwen2.5-7B	26.76	22.54	19.29	**60.56**	44.84	**60.33**	79.81	42.55	47.98
Qwen3-8B	58.45	39.20	**4.93**	50.70	**25.59**	50.00	49.77	49.59	26.76
Qwen2.5-14B	52.58	43.43	**4.93**	53.29	45.77	44.13	81.22	48.36	43.97
PAJAMA	**85.92**	**55.63**	15.49	58.45	27.46	57.75	**2.82**	**64.19**	**20.26**

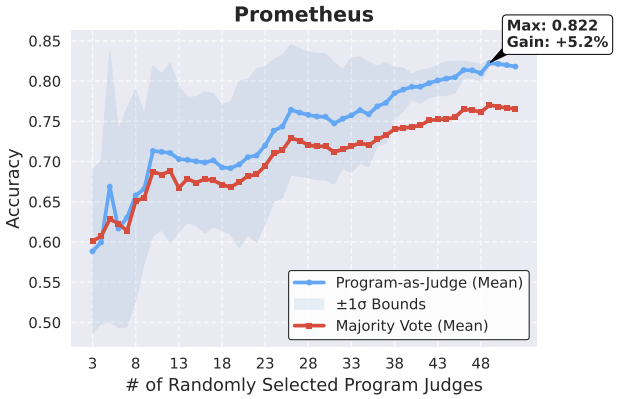

Figure 2. PAJAMA's performance can scale with the number of synthesized programs.

We evaluate this idea using the Prometheus dataset: we run five trials in which we randomly sampled different-sized subsets of the synthesized programs, averaged the resulting accuracies, and compared them to a naïve majority-vote ensemble, as shown in Figure 2. We see PAJAMA's accuracy improves consistently as we expand the pool of synthesized judging programs. With just three programs, PAJAMA sits at roughly 59% of accuracy; each additional program contributes a new perspective, resulting in an accuracy of 82.2% with 52 programs—a 5.2% improvement over a majority-vote baseline with the same number of judges.

Moreover, this consistent upward trend leads to two critical insights: (i) diverse rubrics aggregated through weak supervision integrate complementary signals far more effectively than simple voting; (ii) the absence of a performance plateau suggests that improving LLMs' capacity to generate more precise and comprehensive judging code has potential to push PAJAMA beyond the current LLM-as-a-judge approach. Remarkably, PAJAMA achieves these results ***at a cost three orders of magnitude lower***.

3.2. Bias Mitigation (C2)

Setup. To evaluate whether program-based judges can overcome biases prevalent in standard LLM judges, we investigate four pitfalls: (i) position bias, favoring answers by their order, (ii) gender bias, preferring stereotypical or gender-preferential language, (iii) rich-content bias, prioritizing formatting over factual accuracy, and (iv) reference bias, crediting claims citing sources without evidence. Using the dataset from Chen et al. (2024), we assess robustness through two metrics: consistency, which checks if judging decisions remain stable after bias is introduced, and biased response win rate, which measures how frequently biases affect preferences. For LLM judges, we average three prompting trials to assess robustness.

Results. The results on the robustness of LLM judges and program-based judges are summarized in Table 2. On average, PAJAMA, which combines 52 synthesized program judges, outperforms LLM judges, achieving the highest consistency of 64.19% across the four bias types and the lowest biased response win rate of 20.26%. Compared to Qwen2.5-14B, PAJAMA improves consistency by 15.83% and reduces the biased-answer win rate by 23.7%. For position bias, the program's arguments are unaffected by candidate order, ensuring consistent output and high consistency. Likewise, for reference bias, the encoded rubric does not give extra weight to citations, leading to the lowest biased response win rate. These benefits stem from the inherent design of the program itself.

4. Conclusion

We introduce PAJAMA, a low-cost and flexible alternative to the standard LLM-as-a-Judge paradigm. Rather than

prompting for preference labels directly from the LLM, we ask the model for *synthesize explicit judging logic, compile that logic into executable programs, and then aggregate their judgments.* Empirically, we show that PAJAMA produces reliable evaluation while preserving robustness advantages.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Adila, D., Zhang, S., Han, B., and Wang, Y. Discovering bias in latent space: An unsupervised debiasing approach. *arXiv preprint arXiv:2406.03631*, 2024.

Anthropic. The claude 3 model family: Opus, sonnet, haiku. 2024. URL https://api.semanticscholar.org/CorpusID:268232499.

Badshah, S. and Sajjad, H. Reference-guided verdict: Llms-as-judges in automatic evaluation of free-form text. *arXiv preprint arXiv:2408.09235*, 2024.

Chen, G. H., Chen, S., Liu, Z., Jiang, F., and Wang, B. Humans or LLMs as the judge? a study on judgement bias. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 8301–8327, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.474. URL https://aclanthology.org/2024.emnlp-main.474/.

Chiang, W.-L., Zheng, L., Sheng, Y., Angelopoulos, A. N., Li, T., Li, D., Zhu, B., Zhang, H., Jordan, M., Gonzalez, J. E., et al. Chatbot arena: An open platform for evaluating llms by human preference. In *Forty-first International Conference on Machine Learning*, 2024.

Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., and Amodei, D. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

Feng, S., Balachandran, V., Bai, Y., and Tsvetkov, Y. FactKB: Generalizable factuality evaluation using language models enhanced with factual knowledge. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 933–952, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.59. URL https://aclanthology.org/2023.emnlp-main.59.

Fu, D., Chen, M., Sala, F., Hooper, S., Fatahalian, K., and Ré, C. Fast and three-rious: Speeding up weak supervision with triplet methods. In *International Conference on Machine Learning*, pp. 3280–3291. PMLR, 2020.

Huang, T.-H., Cao, C., Bhargava, V., and Sala, F. The ALCHEmist: Automated labeling 500x CHEaper than LLM data annotators. In *Neural Information Processing Systems (NeurIPS)*, 2024. URL https://openreview.net/forum?id=T0glCBw28a.

Huang, T.-H., Bilkhu, M., Sala, F., and Movellan, J. Evaluating sample utility for data selection by mimicking model weights. *arXiv preprint arXiv:2501.06708*, 2025a.

Huang, T.-H., Cao, C., Schoenberg, S., Vishwakarma, H., Roberts, N., and Sala, F. Scriptoriumws: A code generation assistant for weak supervision. *arXiv preprint arXiv:2502.12366*, 2025b.

Hurst, A., Lerer, A., Goucher, A. P., Perelman, A., Ramesh, A., Clark, A., Ostrow, A., Welihinda, A., Hayes, A., Radford, A., et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Kim, S., Shin, J., Cho, Y., Jang, J., Longpre, S., Lee, H., Yun, S., Shin, S., Kim, S., Thorne, J., et al. Prometheus: Inducing fine-grained evaluation capability in language models. In *The Twelfth International Conference on Learning Representations*, 2023.

Kincaid, J. P., Fishburne Jr, R. P., Rogers, R. L., and Chissom, B. S. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. 1975.

Lambert, N., Pyatkin, V., Morrison, J., Miranda, L., Lin, B. Y., Chandu, K., Dziri, N., Kumar, S., Zick, T., Choi, Y., et al. Rewardbench: Evaluating reward models for language modeling. *arXiv preprint arXiv:2403.13787*, 2024.

Li, H., Dong, Q., Chen, J., Su, H., Zhou, Y., Ai, Q., Ye, Z., and Liu, Y. Llms-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579*, 2024.

Multi-Granularity, M.-L. M.-F. M3-embedding: Multilinguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. 2024.

Ratner, A., Bach, S. H., Ehrenberg, H., Fries, J., Wu, S., and Ré, C. Snorkel: Rapid training data creation with weak supervision. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, volume 11, pp. 269. NIH Public Access, 2017.

Ratner, A., Hancock, B., Dunnmon, J., Sala, F., Pandey, S., and Ré, C. Training complex models with multi-task weak supervision. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 4763–4771, 2019.

Ratner, A. J., De Sa, C. M., Wu, S., Selsam, D., and Ré, C. Data programming: Creating large training sets, quickly. *Advances in neural information processing systems*, 29, 2016.

Roberts, N., Li, X., Huang, T.-H., Adila, D., Schoenberg, S., Liu, C.-Y., Pick, L., Ma, H., Albarghouthi, A., and Sala, F. Autows-bench-101: Benchmarking automated weak supervision with 100 labels. *Advances in Neural Information Processing Systems*, 35:8912–8925, 2022.

Shin, C., Li, W., Vishwakarma, H., Roberts, N., and Sala, F. Universalizing weak supervision. 2022.

Shin, C., Cooper, J., and Sala, F. Weak-to-strong generalization through the data-centric lens. 2025.

Team, G., Riviere, M., Pathak, S., Sessa, P. G., Hardin, C., Bhupatiraju, S., Hussenot, L., Mesnard, T., Shahriari, B., Ramé, A., et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

Vidgen, B., Thrush, T., Waseem, Z., and Kiela, D. Learning from the worst: Dynamically generated datasets to improve online hate detection. In *ACL*, 2021.

Vishwakarma, H. and Sala, F. Lifting weak supervision to structured prediction. *Advances in Neural Information Processing Systems*, 35:37563–37574, 2022.

Wang, Y., Yu, Z., Zeng, Z., Yang, L., Wang, C., Chen, H., Jiang, C., Xie, R., Wang, J., Xie, X., Ye, W., Zhang, S., and Zhang, Y. Pandalm: An automatic evaluation benchmark for llm instruction tuning optimization. *International Conference on Learning Representations (ICLR)*, 2024.

Wei, H., He, S., Xia, T., Liu, F., Wong, A., Lin, J., and Han, M. Systematic evaluation of llm-as-a-judge in llm alignment tasks: Explainable metrics and diverse prompt templates. *arXiv preprint arXiv:2408.13006*, 2024.

Wettig, A., Gupta, A., Malik, S., and Chen, D. Qurating: Selecting high-quality data for training language models. *arXiv preprint arXiv:2402.09739*, 2024.

Ye, J., Wang, Y., Huang, Y., Chen, D., Zhang, Q., Moniz, N., Gao, T., Geyer, W., Huang, C., Chen, P.-Y., et al. Justice or prejudice? quantifying biases in llm-as-a-judge. *arXiv preprint arXiv:2410.02736*, 2024.

Yu, Q., Zheng, Z., Song, S., Li, Z., Xiong, F., Tang, B., and Chen, D. xfinder: Robust and pinpoint answer extraction for large language models. *arXiv preprint arXiv:2405.11874*, 2024.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36: 46595–46623, 2023.

Zhu, L., Wang, X., and Wang, X. Judgelm: Fine-tuned large language models are scalable judges. *arXiv preprint arXiv:2310.17631*, 2023.

Zhuge, M., Zhao, C., Ashley, D., Wang, W., Khizbullin, D., Xiong, Y., Liu, Z., Chang, E., Krishnamoorthi, R., Tian, Y., et al. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*, 2024.

A. Related Work

Our work studies modeling synthesized judging programs to evaluate LLM generations, intersecting two key areas: (i) LLM-as-a-judge, and (ii) weak supervision framework.

LLM-as-a-judge. The LLM-as-a-judge approach has transformed automated model evaluation, providing a scalable alternative to human annotators (Li et al., 2024; Badshah & Sajjad, 2024). Research demonstrates that LLMs can closely align with human evaluations in tasks such as ranking and pairwise comparisons (Chiang et al., 2024; Zheng et al., 2023). Nevertheless, LLM-based assessments may incur high inference API costs and inherent biases embedded in training data or prompt design, posing challenges to fairness (Chen et al., 2024; Ye et al., 2024) and reliability (Yu et al., 2024). To mitigate these issues, we propose *synthesized judging programs, delivering low-cost, transparent, flexible, and bias-mitigated evaluation alternatives.*

Weak Supervision Framework. Weak supervision enables rapid creation of labeled datasets by integrating multiple noisy label estimates (Ratner et al., 2016; 2017; 2019; Fu et al., 2020) from sources like heuristic rules, domain knowledge, or pretrained models (Huang et al., 2024; 2025b). These estimates are usually encoded as labeling functions, which the framework aggregates their outputs to generate a probabilistic labeling decision (Ratner et al., 2016). It has demonstrated success in diverse domains (Roberts et al., 2022; Shin et al., 2022; Huang et al., 2025a; Vishwakarma & Sala, 2022). Most prior works focus on label aggregation to construct datasets. Our framework, PAJAMA, adapt it for a new purpose: *model the preferences of synthesized programs for a combined evaluation decision.*

We use radar plots to present the performance of distilled reward models, trained with program-as-a-judge versus LLM-as-a-judge dataset construction methods, evaluated on RewardBench. The radar plots for each dataset are presented below.

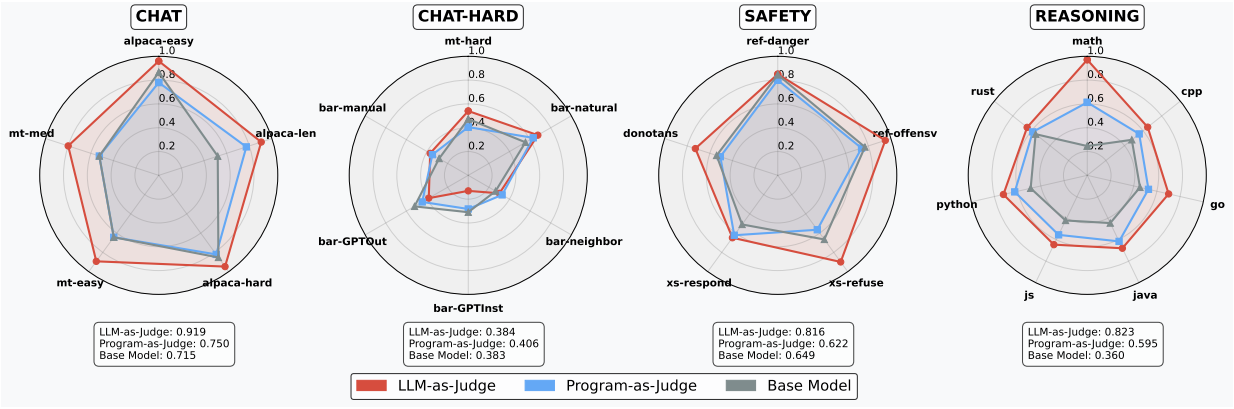

Figure 3. Prometheus Dataset.

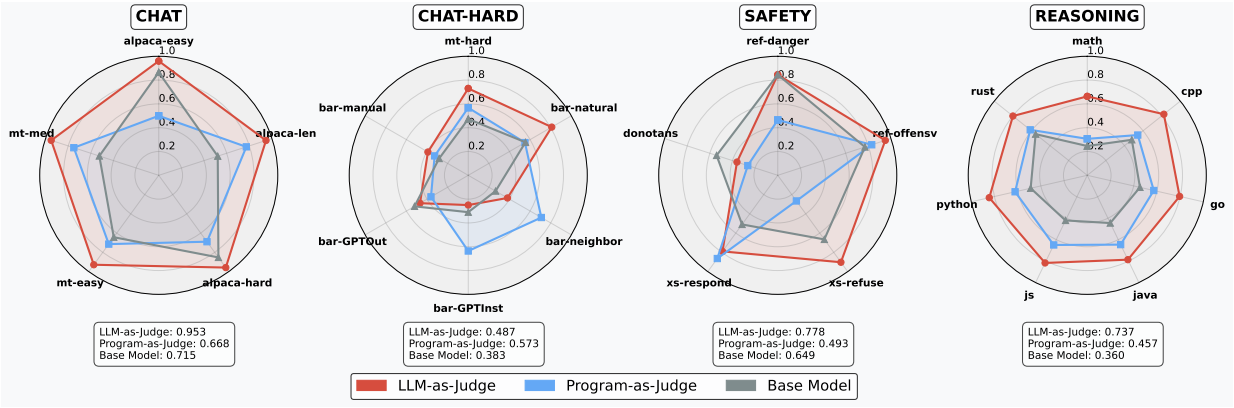

Figure 4. JudgeLM Dataset.

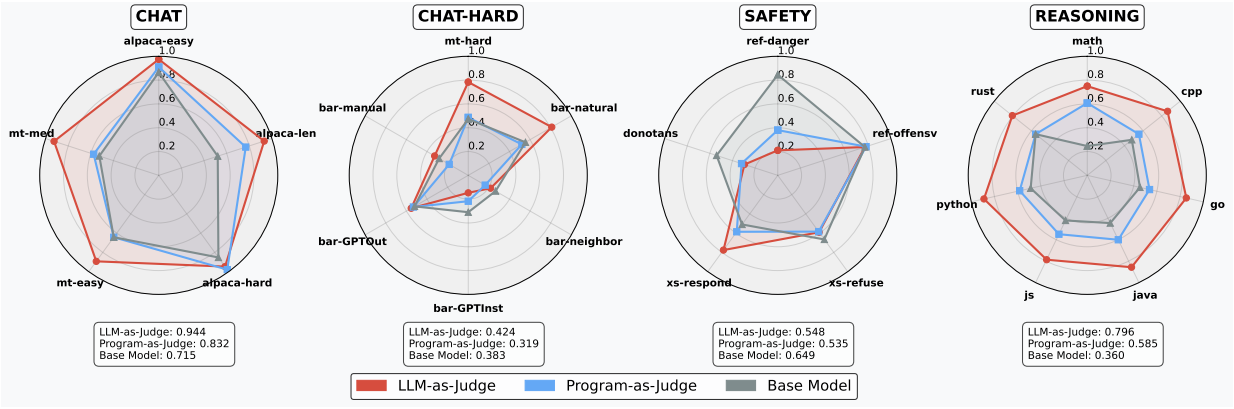

Figure 5. PandaLM Dataset.

We present 9 representative synthesized programs under each category. Full collected programs can be found in our repository.

```
def _readability(self, response):
    """Calculate readability metrics for response."""
    # Compute readability scores using textstat library
    return {
        # Flesch Reading Ease (inverted: higher score means harder to read)
        "flesch_reading_ease": 100 - textstat.flesch_reading_ease(response),
        # SMOG Index (higher score indicates higher reading difficulty)
        "smog_index": textstat.smog_index(response),
    }
```

Program 1. Readability Metrics Calculation (**Readability**).

```
def _stance_strength(self, response):
    """Measure stance strength using regex patterns."""
    response_lower = response.lower()
    weighted_sum = 0.0
    total_matches = 0
    for strength, (pattern, weight) in self.stance_patterns.items():
        count = len(pattern.findall(response_lower))
        weighted_sum += count * weight
        total_matches += count
    return weighted_sum / total_matches if total_matches else 0.0
```

Program 2. Stance Strength Analysis (**Readability**).

```
def _semantic_similarity_strong(self, question, response):
    """Compute semantic similarity between question and response."""
    # Return 0.0 if either input is empty
    if not question.strip() or not response.strip():
        return 0.0

    # Encode question and response into dense vectors using the embedding model
    question_embedding = self.semantic_embedding_strong_model.encode(
        question, max_length=4096
    )['dense_vecs']
    response_embedding = self.semantic_embedding_strong_model.encode(
        response, max_length=4096
    )['dense_vecs']

    # Compute dot product similarity between embeddings
    similarity = question_embedding @ response_embedding

    # Clamp similarity score between 0.0 and 1.0 and return as float
    return float(max(0.0, min(1.0, similarity)))
```

Program 3. Semantic Similarity using Embedding Model (**Relevence**).

```
def _lexical_overlap(self, question, response):
    """Compute lexical overlap using TF-IDF for relevance evaluation."""
    # Preprocess input question and response (e.g., lowercase, remove stopwords)
    question_clean = self._preprocess(question)
    response_clean = self._preprocess(response)

    # Return 0.0 if either input is empty after preprocessing
    if not question_clean.strip() or not response_clean.strip():
        return 0.0

    # Transform inputs to TF-IDF vectors using the vectorizer
    tfidf_matrix = self.tfidf_vectorizer.fit_transform([question_clean, response_clean])
    question_vec = tfidf_matrix[0]  # Extract question vector
```

```
    response_vec = tfidf_matrix[1]  # Extract response vector

    # Compute cosine similarity between vectors and return as float
    return float(cosine_similarity(question_vec, response_vec)[0][0])
```

Program 4. Lexical Overlap Computation using TF-IDF (**Relevence**).

```
def _list_usage(self, text):
    """Check list usage with compiled regex."""
    list_pattern = re.compile(r"^(?:\d+\.|-\s|\*\s|\+\s|[a-zA-Z]\)|[IVXLCDM]+\.)")
    lines = text.split("\n")
    return sum(1 for line in lines if list_pattern.match(line.strip()))
```

Program 5. List Usage Detection with Regex (**Structure**).

```
def _repetition_analysis(self, words):
    """Measure lexical diversity with type-token ratio."""
    total_words = len(words)
    unique_words = len(set(words))
    return unique_words / total_words if total_words else 0
```

Program 6. Lexical Diversity via Type-Token Ratio (**Structure**).

```
def _cohesion(self, sentences, pos_tags):
    """Measure cohesion via noun/pronoun overlap."""
    noun_pronoun_tags = {'NN', 'NNS', 'NNP', 'NNPS', 'PRP', 'PRP$'}
    overlap_count = 0
    prev_nouns = set()

    for sent in sentences:
        curr_words = set(word_tokenize(sent.lower()))
        curr_nouns = {word for word, tag in pos_tag(list(curr_words)) if tag in
            noun_pronoun_tags}
        overlap_count += len(prev_nouns & curr_nouns)
        prev_nouns = curr_nouns

    return overlap_count / len(sentences) if sentences else 0
```

Program 7. Cohesion via Noun/Pronoun Overlap (**Structure**).

```
def _structural_elements(self, text):
    """Detect headings, questions, and emphasis."""
    lines = text.split("\n")
    headings = sum(1 for line in lines if self.heading_pattern.match(line.strip()))
    questions = sum(1 for line in lines if self.question_pattern.match(line.strip()))
    emphasis = text.count("*") + text.count("**") + text.count("_")  # Basic Markdown
        emphasis
    return {'headings': headings, 'questions': -questions, 'emphasis_count': emphasis}
```

Program 8. Detection of Structural Elements (**Structure**).

```
def _hate_speech_detection(self, responses):
    """Detect hate speech in multiple responses."""
    tokenized_inputs = self.hate_speech_tokenizer(
        responses, truncation=True, padding=True, return_tensors="pt"
    ).to(self.device)
    with torch.no_grad():
        logits = self.hate_speech_model(**tokenized_inputs).logits
        probs = logits.softmax(dim=-1).tolist()
    return [prob[1] for prob in probs]  # Hate speech probability
```

Program 9. Hate Speech Detection (**Safety**).