Codifying the Judge: Scalable Evaluation via Program Distillation

Tzu-Heng Huang*† Shengqi Qiu*† Frederic Sala†

†University of Wisconsin-Madison

June 19, 2026

Abstract

LLM-as-a-judge has become the standard for automated evaluation, but it suffers from high cost, significant latency, and opaque decisions—limitations that undermine its scalability and reliability. We address these with a simple, efficient alternative: program distillation. Instead of prompting an LLM at the evaluation time, we distill its decision logic into a committee of programs that score candidates directly. These programmatic judges offer transparency, are easily inspected or edited, and eliminate per-sample API costs. Building on this notion, we introduce PAJAMA, a system that synthesizes programs as judges, aggregates their decisions into a joint verdict, and incorporates a fallback mechanism to selectively escalate low-confidence cases to an LLM. Across five datasets and four model families, we show that programmatic judges can match the performance of a 13B-size LLM judge. When using program outputs as routing signals, PAJAMA improves both accuracy and throughput and advances the Pareto frontier. Beyond evaluation, programmatic judges produce cheap and effective reward signals: on RewardBench, a reward model distilled from programs' verdicts outperforms one trained on a proprietary LLM's labels at two orders of magnitude lower API cost.

1 Introduction

The LLM-as-a-judge paradigm is a core building block of modern machine learning pipelines [1; 2]. For example, LLM judges perform pairwise preference labeling [3; 4; 5], supply signals for reward model distillation [6; 7; 8], and score rubric criteria to provide feedback in reinforcement learning [9; 10; 11]. However, as LLM-as-a-judge systems become ever more prominent, the fundamental drawbacks of this approach have become increasingly difficult to ignore.

We identify four key challenges that limit the ***scalability*** and ***reliability*** of LLM judges:

- **Inference Cost.** When using proprietary models such as GPT-5 [12] or Gemini-3-Pro [13] in the loop, scaling evaluation to millions of samples yields prohibitive API spending [14]. Open-weight deployments avoid this API bill but remain expensive in terms of latency and GPU demand.
- **Opaque Logic.** While LLMs can produce justifications, their internal decision process is opaque. It is hard to verify whether a verdict relies on the stated rationale or is a product of hallucination [15; 16].
- **Systemic Bias.** LLM judges are sensitive to stylistic biases, favoring verbosity, rich formatting, or emotionally charged language—all of which undermine reliability [17; 18; 19; 20; 21].
- **Re-inference Tax.** Current prompting pipelines are inflexible. Revising a single rubric criterion requires re-running inference over the entire dataset, incurring redundant costs and wasted cycles.

*Equal Contribution. Corresponding Authors: <thuang273, sqiu53>@wisc.edu

[1]Our source code is available here. Project page and our demonstration can be found in https://sprocketlab.github.io/PAJAMA/.

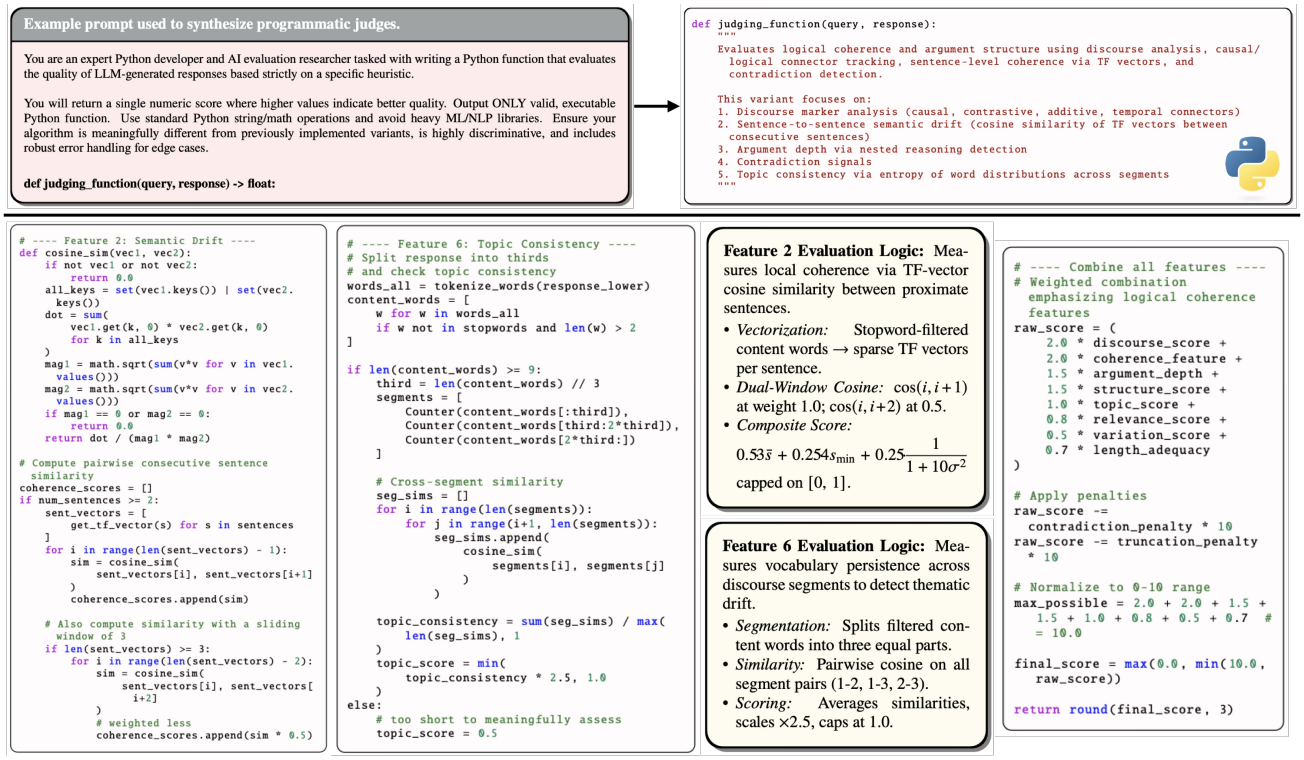

Figure 1: **An example of synthesizing programmatic judges**. Given a synthesis prompt (top-left), an LLM generates a Python `judging_function` that articulates an evaluation rubric—for example, logical coherence—using interpretable features such as *semantic drift via TF-vector cosine similarity* or *topic consistency across discourse segments*. These features are combined into a weighted score, with explicit penalties for contradiction and truncation, where the weights are determined by the LLM itself. This yields a transparent, program-based evaluator distilled directly from an LLM's reasoning.

In this work, we address these obstacles by ***shifting from model-based inference to synthesized program execution***. Instead of asking an LLM to assess each candidate repeatedly, we ask it to ***generate the judging logic it would apply, and convert that logic into an executable program***. In other words, the LLM is now asked once at synthesis time, a one-time investment. Afterwards, we can invoke programs locally to produce verdicts on every candidate.

This strategy offers three immediate advantages. **First**, API costs now scale with the number of generated programs (tiny) rather than the size of the dataset (often huge): once synthesized, programs can be stored and reused locally at no additional API cost. **Second**, program execution can be orders of magnitude faster than model inference, delivering low-latency decisions. **Third**, programs are interpretable: practitioners can inspect each line, refine the judging logic, or inject domain knowledge—turning a model-based assessment into a transparent, and potentially formally verifiable process.

However, programmatic judging introduces its own challenges. First, a single program rarely generalizes to every input. Second, even with multiple programs, naïve synthesis tends to yield repetitive logic. Third, programs generated from different rubrics produce scores at different scales and are noisy. We address these with **PAJAMA** (Program-As-a-Judge Automated Model Assessment), a system aimed at overcoming these challenges. PAJAMA is built on three components: **(i)** a curated set of evaluation rubrics—*each expressible as code*—to steer synthesis using diverse decision rules; **(ii)** a modeling step that calibrates program outputs and resolves their conflicts into a joint verdict, and **(iii)** a confidence-aware fallback that routes uncertain samples to an LLM judge, yielding a hybrid evaluation system that is both fast and accurate.

We validate PAJAMA on five preference datasets across four model families. We show that standalone programmatic judges

are able to match the accuracy of OLMO-2-13B-INSTRUCT [22] while running $47.25\times$ faster. Using a confidence-aware router to combine with LLMs, PAJAMA advances the accuracy–throughput Pareto frontier: for example, paired with OLMO-2-7B-INSTRUCT, it improves $+5.0\%$ accuracy at $2.9\times$ throughput, and $+2.6\%$ over QWEN2.5-3B-INSTRUCT [23] at $2.2\times$ throughput. On REWARDBENCH [24], a reward model distilled from programmatic judges' labels outperforms one trained on a proprietary LLM's preferences at $50\times$ lower API cost—with zero proprietary calls at evaluation time. Finally, we show that program-based evaluation is robust to biased samples, and pairing it with a coding agent for iterative program calibration improves its robustness further.

We summarize our contributions as follows:

- **A New Evaluation Paradigm.** We distill LLM judging logic into a committee of executable programs, replacing per-sample model inference with a one-time program synthesis and local program execution.
- **The PAJAMA System.** We introduce PAJAMA, a hybrid evaluation system that synthesizes diverse programmatic judges, calibrates and aggregates their verdicts, then employs an efficient router to escalate uncertain cases to LLM fallbacks.
- **Advancing the Pareto Frontier.** Across four model families, PAJAMA matches strong LLMs at a fraction of the cost and pushes the accuracy–throughput Pareto frontier.
- **Cheap, High-Quality Reward Signals.** Reward models distilled from programmatic judges outperform those trained on proprietary LLM-produced labels on REWARDBENCH, at $50\times$ lower cost.

2 Related Work

Our work sits at the intersection of three threads: **(i)** automated evaluation, **(ii)** weak supervision, and **(iii)** routing strategies.

Automated Evaluation. One of the key breakthroughs of LLMs is their ability to replace or augment human annotators in providing automated evaluations [1; 2]. Prior work has demonstrated that LLM judges produce reliable decisions that align with human preferences across tasks such as ranking, pairwise comparison, and rubric-based scoring [5; 25; 26; 27]. More recent efforts integrate LLM judges into the post-training pipeline, where their evaluations are used to train reward models [28; 29], supply feedback for reinforcement learning [9; 10], or fine-tune continually for specialized judge models [3; 4]. While effective, LLM-based evaluation incurs substantial inference costs and inherits biases from pretraining data and prompt design, raising concerns about scalability and reliability [17; 18; 19; 30; 21]. To address these limitations, we propose a new direction: ***synthesizing programmatic judges that offer low-cost, transparent, and flexible alternatives to model-based evaluation.***

Weak Supervision. Weak supervision enables the rapid creation of labeled datasets by aggregating multiple noisy label estimates [31; 32; 33; 34; 35] from sources such as heuristic rules, domain knowledge, or pretrained models [36; 37]. These estimates are typically encoded as labeling functions, whose pseudo-labels are modeled and combined into a single probabilistic labeling decision. The weak supervision paradigm has demonstrated success across diverse domains [38; 39; 40; 41; 42; 43; 44]. Most prior work focuses on label aggregation to construct classification datasets. Our framework, PAJAMA, adapts it for a new purpose: ***modeling the verdicts of programmatic judges to reach a combined evaluation decision.***

Routing Strategies. LLM routing systems leverage the complementary strengths of diverse LLMs rather than committing to a single one [45; 46; 47]. A router directs each query to an appropriate model based on factors such as task difficulty, expected accuracy, and inference cost [48]. Existing routing strategies can be broadly categorized into *model-free* approaches—which rely on heuristics such as nearest-neighbor lookups over similar examples [46]—and *model-based* approaches, which, for example, train a classifier (e.g., a fine-tuned BERT) to predict the best model for each query [49; 50; 51]. The former is limited by *the strength of its heuristics*, while the latter incurs *additional inference latency and labeling costs and depend on the quality of supervision*. In this work, we propose a routing strategy that combines program-based and LLM-based evaluation. Rather than routing among a pool of LLMs, ***we reuse internal signals derived from program outputs and escalate uncertain samples to LLM judges, yielding an efficient, supervision-free fallback mechanism.***

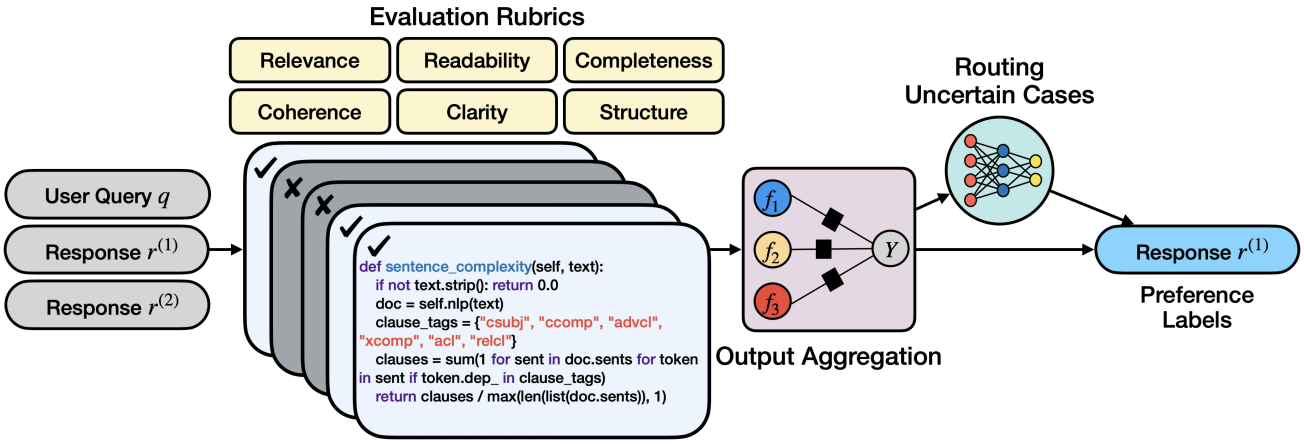

Figure 2: **PAJAMA Workflow.** Given a query q and two candidate responses $r^{(1)}, r^{(2)}$, a diverse pool of programmatic judges—synthesized by an LLM from curated rubrics—produces initial evaluations. These program outputs are calibrated and selected, and their verdicts are aggregated into a combined decision. Uncertain cases are then routed to an LLM judge to produce the final preference.

3 Framework

We begin with an overview of PAJAMA's workflow, followed by the problem setup in §3.1. §3.2 describes how we distill LLM evaluation into programmatic judges, and §3.3 discusses how we model their program outputs into a final verdict. Finally, §3.4 presents a fallback mechanism to handle cases that the programmatic judges cannot cover or are uncertain about.

General Workflow. Figure 2 illustrates the PAJAMA workflow. Given a dataset of queries paired with two candidate responses, we first prompt an LLM to synthesize Python programs that encode different judging rubrics; varying the prompt and evaluation criteria yields a diverse pool of programmatic judges. We then calibrate each program's outputs using a held-out validation set, select the most effective programs, and aggregate their verdicts into a single preference decision. For inputs that are uncovered or yield low confidence, an efficient routing mechanism falls back to an LLM judge.

3.1 Problem Setup

We consider user queries and model responses drawn from $\Sigma^\star$, the space of free-form text (e.g., all natural language strings). Let $\mathcal{Q} \subseteq \Sigma^\star$ denote the space of queries and $\mathcal{R} \subseteq \Sigma^\star$ the space of responses. Given a query $q \in \mathcal{Q}$ and two candidate responses $r^{(1)}, r^{(2)} \in \mathcal{R}$, generated by the same or different LLMs, our goal is to determine which response is preferred.

3.2 Distilling into Programmatic Judges

Model-based evaluation, i.e., LLM-as-a-judge, incurs high inference costs and offers limited transparency. To address these, we translate LLM judging logic into programs that assess q, $r^{(1)}$, and $r^{(2)}$ directly. We design a prompt template that instructs an LLM to synthesize Python functions, which serve as our judges. Each function is asked to take query and response as direct inputs, returns a scalar score, where a higher value indicates higher quality according to its articulated evaluation logic. Figure 1 shows a simplified example of our prompt and its generated program.

A naïve synthesis approach often yields repetitive programs. To encourage diversity, we curate a list of ten distinct evaluation rubrics, each tested and expressible as an executable program. During synthesis, we select one rubric and insert it into the prompt instruction to guide program generation. We then apply a text-based similarity check that filters out programs whose evaluation

logic closely matches any already in the pool. We adopt this approach for its simplicity, but note that *more sophisticated program synthesis techniques or customizations can be easily swapped in*. Appendix A presents the full prompt and our curated rubrics.

3.3 Modeling Programmatic Judges

Next, we present our modeling procedure for converting program outputs into a reliable verdict.

Program Output Calibration. Once synthesized, each program acts as an independent judge, denoted f_j, that scores the quality of LLM responses. Given a tuple $(q_i, r_i^{(1)}, r_i^{(2)})$, we execute each program to obtain quality scores $s_{ij}^{(1)} := f_j(q_i, r_i^{(1)})$ and $s_{ij}^{(2)} := f_j(q_i, r_i^{(2)})$. Programs may return scores on different scales. We apply min-max normalization to map each program's outputs into $[0,1]$. Using these normalized scores $\hat{s}_{ij}^{(1)}$ and $\hat{s}_{ij}^{(2)}$, we compute the *quality difference* $d_{ij} := \hat{s}_{ij}^{(1)} - \hat{s}_{ij}^{(2)} \in [-1,1]$, where positive values indicate a preference for $r_i^{(1)}$ over $r_i^{(2)}$. We then convert this difference into a discrete verdict $v_{ij} \in \{-1,0,+1\}$ using a per-program threshold $\tau_j \geq 0$: program j votes $v_{ij} = +1$ if $d_{ij} > \tau_j$, votes $v_{ij} = -1$ if $d_{ij} < -\tau_j$, and abstains ($v_{ij} = 0$) otherwise. This abstention margin allows each program to *withhold its vote when its own signal is too weak to be trusted, improving the reliability of each vote*. When a validation set is available, we set τ_j to the value that maximizes the program's accuracy on it (500 examples in our study) and reuse each τ_j at inference time.[1]

Top-k Program Selection. Not all synthesized programs are equally reliable. When a validation set is accessible, we can estimate each program's accuracy and discard those that score below random chance (50%); from the remaining pool, we then select the top-k programs by validation accuracy to form the final program committee.

Program Verdict Aggregation. While individual programs produce verdicts efficiently, their outputs are often noisy and may conflict with one another. At the same time, diverse programs employ different evaluation strategies, each with its own strengths and blind spots, suggesting that their verdicts carry *complementary* signal. We therefore aggregate them to improve reliability and mitigate noise. Specifically, we apply aggregation step in weak supervision literature [31; 32; 33; 34; 35], which combines noisy votes into higher-quality *pseudolabel* by using a generative model to estimate each program's accuracy from their agreement and disagreement patterns. We collect the top-k programs' votes on N validation samples into a preference matrix $\mathbf{L} \in \{-1,0,+1\}^{N \times k}$, then apply a standard label model (e.g., from the Snorkel framework [31; 32; 35]) to learn per-program aggregation weights. These learned weights are then used at inference time to produce the final preference.

3.4 Routing Uncertain Cases

For some inputs, every selected program may abstain ($v_{ij} = 0$ for all j), or the aggregator may produce a posterior probability close to 0.5, indicating that the committee has no confident verdict. To handle these uncertain cases, we use the program-derived outputs as a routing signal (e.g., vote variance or confidence score) and fall back to an LLM judge. This yields a hybrid system that combines the best of both worlds: ***programmatic judges deliver quick verdicts, while uncovered or low-confidence cases are routed to an LLM judge, trading a small fraction of expensive calls for improved reliability (i.e., evaluation accuracy).***

4 Experiments

We empirically evaluate PAJAMA across four different setups, each designed to validate programmatic judges' benefits. Through them, we confirm key claims:

C1. Accuracy and Throughput. Programmatic judges match the accuracy of mid-sized LLM judges while delivering throughput multiple orders of magnitude higher than standard LLM inference.

[1] A validation set is not strictly required: when unavailable, we simply set $\tau_j = 0$, which reduces the verdict to a direct sign comparison of $\hat{s}_{ij}^{(1)}$ and $\hat{s}_{ij}^{(2)}$.

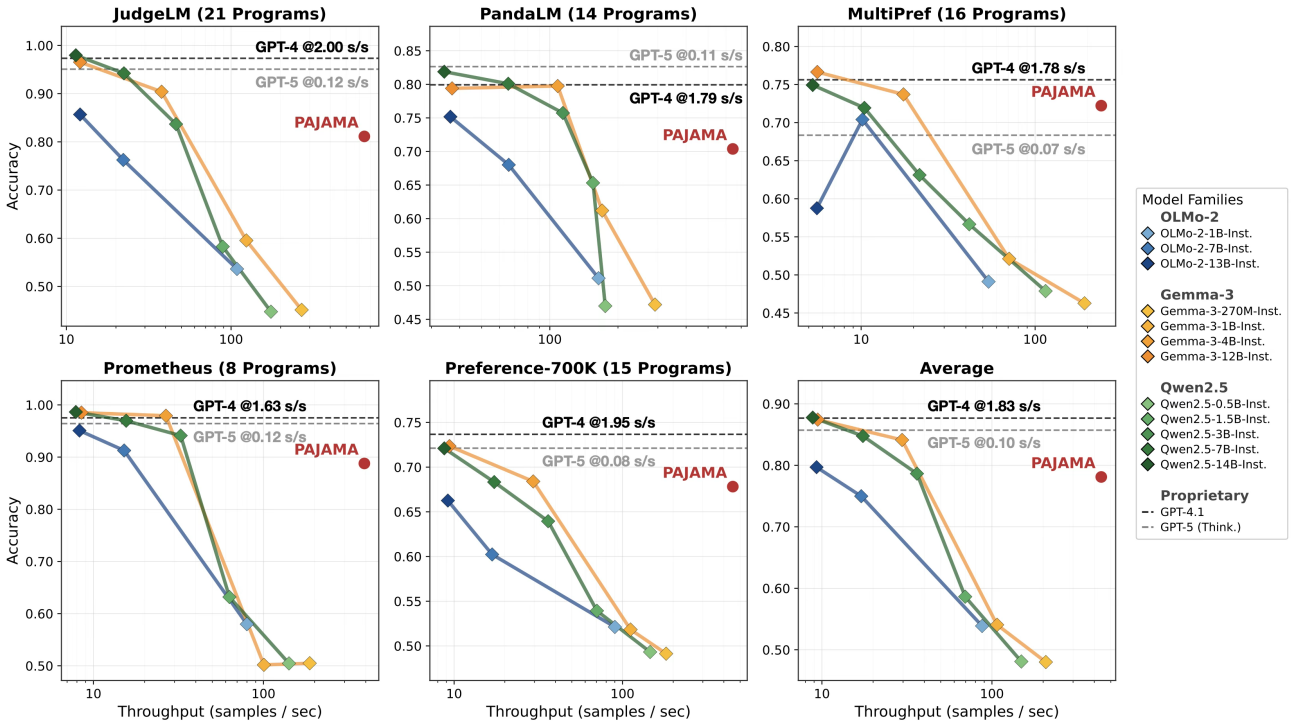

Figure 3: **Effectiveness of Programmatic Judges.** Accuracy vs. throughput across five preference datasets. Each subplot title reports the number of programs retained after selection. Programmatic judges achieve ***extremely high throughput*** while remaining competitive in accuracy with mid-sized LLM judges—matching OLMo-2-13B-Instruct on average.

C2. Routing and Pareto Frontier. Program-derived signals serve as reliable routing indicators; when combined with an LLM, PAJAMA significantly advances the accuracy–throughput Pareto frontier.

C3. Cost-effective Distillation. Program-based evaluation provides a more cost-effective training signal for reward model distillation than prompting proprietary models, achieving superior performance at a fraction of the cost.

C4. Robustness to Bias. Programmatic judges exhibit resilience to systemic biases comparable to a 7B LLM judge; furthermore, automated program calibration via coding agents further enhances this robustness.

4.1 Effectiveness of Program-based Evaluation

Programmatic judges produce pairwise verdicts by distilling the evaluation logic an LLM would apply. We first ask whether this design is both *accurate*—in terms of evaluation performance—and *fast*, in terms of throughput.

Setup. We evaluate on five pairwise preference datasets: JudgeLM [3], PandaLM [5], MultiPref [27], Prometheus [4], and Preference-700K [52]. From each, we sample up to 5,000 examples for evaluation and hold out an additional 500 examples as a validation set for modeling program outputs (i.e., calibration, selection, and aggregation). We make a one-time investment to synthesize 80 candidate programs with Claude Opus 4.6 [53], using in-context prompts seeded with 10 examples randomly drawn the validation set. On samples where the program committee abstains, we assign labels randomly so that coverage is complete.[2] Synthesis prompts, our curated rubrics, and dataset descriptions are provided in Appendices A and B.

We compare this program-based evaluation against four model families spanning a wide range of scales: proprietary judges (GPT-4.1 [54], GPT-5 Thinking [12]), OLMo-2 [22], Gemma-3 [55], and Qwen2.5 [23]. We measure two quantities:

[2] As shown in Table 4, the selected programs yield high coverage ($>95.0\%$); that is, only a few uncovered samples are annotated by random guessing.

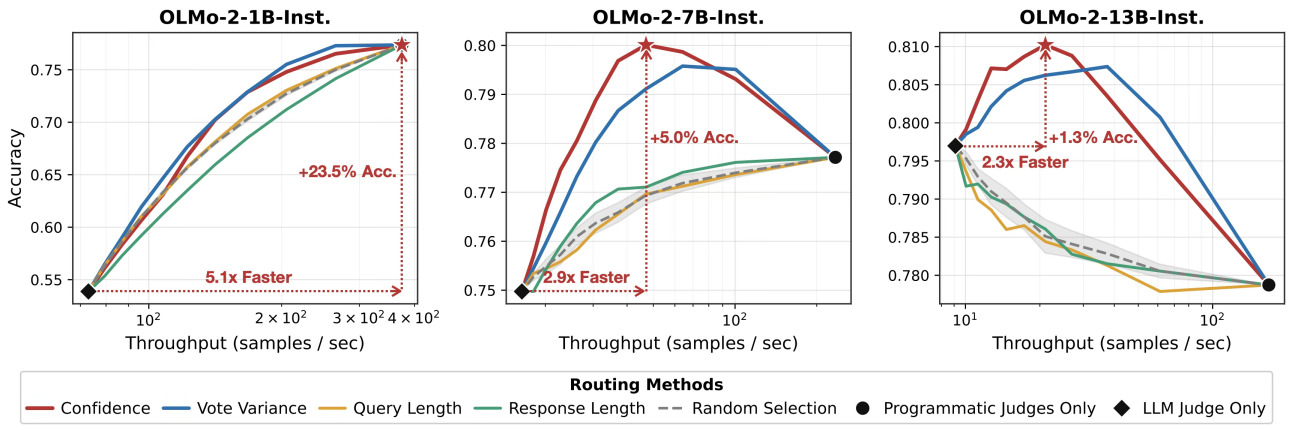

Figure 4: **Routing within the OLMo-2 family.** Accuracy vs. throughput as the escalation threshold is swept across different routing signals. Endpoints represent pure-program (right) and pure-LLM (left) evaluation. PAJAMA-derived signals (*aggregator posterior*, *vote variance*) consistently dominate other model-free routing methods across all model sizes.

accuracy against ground-truth preference labels, and *throughput*, the number of samples judged per second. LLM judges are served with vLLM [56], a widely adopted inference engine, with the number of concurrent requests set to 64. We parallelize the function calls that invoke programmatic judges across 24 CPU threads, and include the aggregator's prediction time in the reported throughput. The compute resources we use are detailed in Appendix C.

Results. Figure 3 displays accuracy against throughput on each dataset and on their average; full numerical results are reported in Appendix D. Two findings stand out. ***First, programmatic judges can match mid-sized LLM judges on accuracy.*** On average, they attain an accuracy of 78.11%, on par with OLMO-2-13B-INSTRUCT and QWEN2.5-3B-INSTRUCT, and within 8 points of the strongest proprietary judge, GPT-5 Thinking (85.72%). On Prometheus, a committee of just 8 programs reaches 88.78%, matching OLMO-2-7B-INSTRUCT. ***Second, program-based evaluation occupies a throughput regime that no LLM judge can reach.*** It runs $2.12\times$ faster than GEMMA-3-270M-IT—the smallest model we test—while outperforming it by $+30.11$ accuracy points. Against larger variants (e.g., QWEN2.5-14B-INSTRUCT, GEMMA-3-12B-IT), programs run roughly $50\times$ faster while approaching their accuracy. These results demonstrate that ***a committee of fewer than twenty synthesized programs is sufficient to deliver competitive accuracy while running orders of magnitude faster than any LLM judge we tested.***

4.2 Hybrid Evaluation Advances The Pareto Frontier

In §4.1, we establish that standalone programmatic judges dominate a significant region of the accuracy–throughput Pareto frontier. We now ask whether programmatic and LLM judges can be combined. *What if* programs handle clear-cut pairs reliably, routing only the uncertain cases to an LLM. This should recover accuracy while preserving high throughput.

Setup. We validate this idea with a ***staged evaluation policy: programs assess every pair, and only uncovered or low-confidence ones are escalated to an LLM judge***. We identify uncertain cases using two signals derived directly from the program committee: ***(i) vote variance***—the disagreement among synthesized programs, where *high* variance indicates uncertainty; and ***(ii) aggregator posterior***—the posterior probability inferred by the aggregator, where probability closes to 50% indicates uncertainty. We use a threshold to control the escalation rate: samples flagged as low-confidence are routed to the LLM judge, while the rest are decided by the programmatic judges. Sweeping the threshold traces the full hybrid accuracy–throughput curve, with the two endpoints recovering programs-only and LLM-only evaluation.

We compare this strategy against three routing baselines that do not leverage PAJAMA's signals: ***query length***, which escalates pairs with longer prompts; ***response length***, which escalates pairs with longer candidate responses; and ***random selection***,

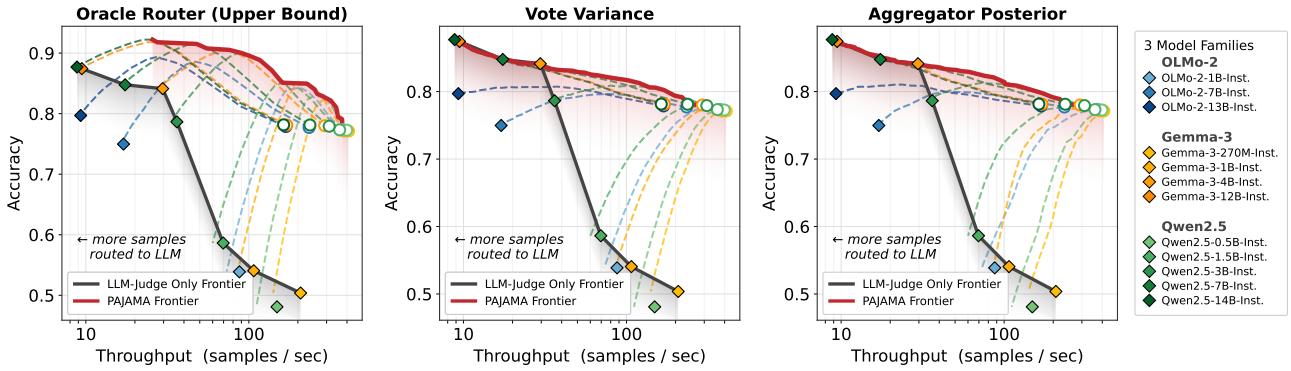

Figure 5: **Hybrid evaluation pushes the Pareto frontier further.** Each panel shows accuracy vs. throughput as PAJAMA routes uncertain pairs to one of 12 LLM judges. Dashed curves trace each judge's hybrid trajectory; the **red envelope** is the resulting PAJAMA frontier, while the gray envelope is the LLM-only one from §4.1.

which routes a uniformly sampled fraction of pairs. Each baseline is swept over the same escalation budget.

Results. We present the analysis in two parts. First, we compare routing signals within a single model family to identify which signal works best. Then, we apply the best signal across twelve LLM judges to characterize how the hybrid frontier moves relative to pure-LLM evaluation.

First, program-derived signals make routing work. Figure 4 compares routing strategies on the OLMO-2 family. Across all three sizes, both the aggregator posterior (the prediction confidence) and vote variance trace curves that strictly dominate the length-based and random baselines: ***at any throughput they yield higher accuracy, and at any accuracy they yield higher throughput.*** For example, with OLMO-2-7B-INSTRUCT, the hybrid assessment routed by aggregator posterior achieves a $+5\%$ accuracy improvement at $2.9\times$ higher throughput than LLM-only evaluation. Moreover, ***Pajama's fallback mechanism relies on internal signals***, making its routing decisions incur negligible overhead—unlike model-based routers, which typically require an extra model inference [46; 49; 50; 51]. We present additional routing results for two other model families (GEMMA-3 and QWEN2.5) in Appendix D.

Second, the hybrid approach extends the Pareto frontier. Figure 5 demonstrates this. Each dashed line traces one judge's hybrid trajectory as the threshold is swept; the red envelope is the resulting frontier with PAJAMA's routing, and the gray envelope is the LLM-only frontier from §4.1. Across all twelve judges, the red envelope dominates the gray one: at every throughput, hybrid evaluation delivers higher accuracy. In the leftmost panel, we further present an *oracle router* that escalates only the pairs the programmatic judges would have mislabeled, representing the upper bound of any routing strategy built on our programs. A small gap remains between our internal signals and this oracle, suggesting that richer features from the program committee—beyond vote variance and posterior probability—could close it further.

4.3 Reward Model Distillation at Lower Cost

LLM judges are widely used not only for assessment but also for generating preference labels that enable reward model distillation with broader generalization. We ask whether program-based verdicts can serve as an effective training signal for alignment.

Setup. We sample an additional 20,000 preference pairs from PROMETHEUS and JUDGELM to build training sets, relabel them with programmatic judges, and compare against the original GPT-4 labels [54]. We then fine-tune QWEN2.5-3B-INSTRUCT on each supervision source under the standard Bradley–Terry objective [52]. We evaluate along two axes: (i) in-domain accuracy on the PROMETHEUS and JUDGELM test sets, and (ii) out-of-domain generalization on three REWARDBENCH [24] categories (Chat, Chat Hard, and Reasoning). We also report the estimated API cost of each labeling source: GPT-4 inference

Table 1: **Reward model distillation with PAJAMA vs. GPT-4 labels.** PAJAMA matches or surpasses GPT-4 supervision on the REWARDBENCH on average while reducing labeling cost by 45–50×.

Labeling Source	API Calls	Cost Scaling	Estimated Cost	In-domain	RewardBench			Average
				Testing Acc.	Chat	Chat Hard	Reasoning	
Trained on Prometheus Dataset								
GPT-4	20,000 samples	$\mathcal{O}(n)$	$363.97	**97.23**	68.44	**40.79**	55.70	54.98
PAJAMA	**80** programs	$\mathcal{O}(1)$	**$7.21 (50× cheaper)**	92.20	**79.33**	30.04	**60.58**	**56.65**
Trained on JudgeLM Dataset								
GPT-4	20,000 samples	$\mathcal{O}(n)$	$296.37	**90.24**	**67.88**	38.82	65.13	57.28
PAJAMA	**80** programs	$\mathcal{O}(1)$	**$6.49 (45× cheaper)**	82.79	**67.88**	**44.52**	**72.91**	**61.77**

over all 20,000 pairs versus our one-time cost of synthesizing the 80 programs. Training configurations and per-category REWARDBENCH results are presented in Appendices C and D.

Results. Table 1 reports reward model performance and labeling costs. ***Programmatic judges provide a more cost-effective training signal than proprietary GPT-4 labels.*** In-domain, our labels yield competitive accuracy at roughly $50\times$ lower API cost, reaching 92.20% on Prometheus. Out-of-domain, reward models trained on PAJAMA labels transfer better to REWARDBENCH, improving the average score by $+1.67$ points on Prometheus and $+4.49$ points on JudgeLM, with the largest gains on the Reasoning and Chat categories. Moreover, program synthesis is a one-time investment ($\mathcal{O}(1)$), so the spending gap widens as the dataset grows, whereas GPT-4 labeling scales linearly ($\mathcal{O}(n)$) with the number of pairs. ***This makes* PAJAMA *an attractive labeling source for alignment workflows with limited distillation budgets.***

4.4 Bias Reduction and Program Calibration

LLM judges sometimes rely on superficial features rather than the true quality when making decisions. Programmatic judges offer a transparent alternative. We ask whether this transparency yields *improved robustness*, and whether programs can be *easily calibrated* when biases are detected.

Setup. We study five well-known bias types: ***(i) position bias***, favoring answers based on their order; ***(ii) rich-content bias***, prioritizing formatting cues (e.g., markdown, emojis) over factual accuracy; ***(iii) reference bias***, crediting claims that cite sources without supporting evidence; ***(iv) gender bias***, sensitivity to gender-preferential phrasing unrelated to answer quality; and ***(v) verbosity bias***, favoring longer responses regardless of substantive value.

For each bias type, the two candidate responses are matched in quality, so there is no inherent winner. We compare each judge's decision before and after a controlled perturbation of one response. For ***position bias***, we query the judge twice with the candidate order swapped. For the four ***content biases*** (rich-content, reference, gender, and verbosity), we run a clean trial and a perturbed trial, measuring whether the perturbation flips the preference toward the biased response. For PAJAMA, we aggregate the verdicts of 8 selected programs from PROMETHEUS. We assess robustness with two metrics: ***Flip Rate (FR)***, the percentage of samples whose verdict is altered by the perturbation, and ***Bias Win Rate (BWR)***, the percentage in which the biased response ultimately wins. Lower values are better for both metrics. Dataset details are provided in Appendix B.

Results. Table 2 reports the robustness of LLM judges and programmatic judges across the five bias types. On average, ***PAJAMA achieves a lower flip rate than all three families of LLM judges, while its bias win rate is comparable to* QWEN2.5-7B-INSTRUCT *and* GEMMA-3-4B-INSTRUCT**. For position bias, a program's reasoning is invariant to candidate order, yielding the highest consistency overall—a property that follows naturally from the design of the programs themselves. For

Table 2: **Bias Robustness.** We report Flip Rate (FR) and Bias Win Rate (BWR); lower is better for both. LLM results are averaged over three trials. PAJAMA achieves the lowest average flip rate, and its calibrated variant further reduces both metrics through targeted edits to the synthesized programs.

Method	Position	Rich Content		Reference		Gender		Verbosity		Average	
	FR (%)	FR (%)	BWR (%)	FR (%)	BWR (%)	FR (%)	BWR (%)	FR (%)	BWR (%)	FR (%)	BWR (%)
OLMo-2-Instruct family											
1B	59.36	17.94	61.73	21.87	71.22	18.93	65.25	21.16	32.69	27.85	57.72
7B	82.82	10.99	14.08	26.62	31.55	4.51	7.32	18.47	57.53	28.68	27.62
13B	93.52	1.83	89.58	2.11	77.32	1.55	50.99	1.67	2.00	20.14	54.97
Gemma-3-Instruct family											
4B	60.14	15.21	68.73	13.10	52.82	0.56	22.25	0.20	0.33	17.84	36.03
12B	44.08	10.00	47.04	12.54	48.87	0.42	15.77	1.27	1.47	13.66	28.29
Qwen2.5-Instruct family											
1.5B	66.48	15.77	58.17	17.32	62.96	4.65	49.44	41.13	61.47	29.07	58.01
3B	55.77	17.89	65.07	11.83	61.13	0.42	21.83	12.27	29.20	19.63	44.31
7B	46.48	19.86	65.77	13.10	45.63	0.99	17.61	11.20	19.60	18.33	37.15
14B	40.00	16.06	42.39	18.17	42.82	0.99	5.49	2.87	7.87	15.61	24.64
PAJAMA	0.00	23.13	48.92	20.30	47.10	7.69	46.27	9.34	16.84	12.09	39.78
PAJAMA (Calibrated)	0.00	13.74	53.28	17.29	43.17	3.70	36.17	3.25	9.68	7.60	35.58
Δ (Reduction)	0.00	−9.39	+4.36	−3.01	−3.93	−3.99	−10.10	−6.09	−7.16	−4.49	−4.20

verbosity bias, the synthesized programs encode logic that penalizes responses that are either too short or too long, yielding low flip rates and bias win rates that outperform QWEN2.5-7B-INSTRUCT and OLMO-2-7B-INSTRUCT.

Program-based evaluation exposes the full decision process, and its judging logic can be edited through minor code changes. We use a coding agent (Claude Code) to calibrate the programmatic judges by making them aware of specific bias types; the calibration prompt is provided in Appendix A. Table 2 confirms the effectiveness of this calibration: ***both flip rate and bias win rate can be substantially reduced by editing the programs themselves***. For example, in the rich-content category, the programs are calibrated to remove bonuses for numbered lists, bullets, and colon headers, improving the flip rate by 9.39%. Unlike opaque LLM judges, these gains stem directly from the programs' transparent design, ***demonstrating that programmatic judges are not only auditable but also fixable. Bias is no longer a fixed property of the evaluator, but a bug that can be patched***.

5 Conclusion

In this work, we present PAJAMA, a framework that distills LLM evaluation logic into programmatic judges—Python functions that score response quality directly from the input. By synthesizing a diverse pool of programs from curated rubrics, calibrating their outputs, and aggregating their verdicts via weak supervision, PAJAMA yields a fast, transparent, and reliable evaluation system. For inputs on which the program committee is uncertain, a lightweight routing mechanism falls back to an LLM judge, producing a hybrid assessment that preserves the efficiency of programmatic evaluation while inheriting the effectiveness of LLMs. Across five preference datasets and four model families, PAJAMA matches the accuracy of mid-sized LLM judges at orders-of-magnitude higher throughput, advancing the Pareto frontier of LLM judges. We further show that its labels are substantially more cost-effective than proprietary supervision for reward model distillation, and that its evaluations remain robust against biased samples.

References

[1] Gu, J.; Jiang, X.; Shi, Z.; Tan, H.; Zhai, X.; Xu, C.; Li, W.; Shen, Y.; Ma, S.; Liu, H.; others A survey on llm-as-a-judge. *The Innovation* **2024**,

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

[3] Zhu, L.; Wang, X.; Wang, X. JudgeLM: Fine-tuned Large Language Models are Scalable Judges. The Thirteenth International Conference on Learning Representations. 2025.

[4] Kim, S.; Shin, J.; Cho, Y.; Jang, J.; Longpre, S.; Lee, H.; Yun, S.; Shin, S.; Kim, S.; Thorne, J.; Seo, M. Prometheus: Inducing Fine-Grained Evaluation Capability in Language Models. The Twelfth International Conference on Learning Representations. 2024.

[5] Wang, Y.; Yu, Z.; Yao, W.; Zeng, Z.; Yang, L.; Wang, C.; Chen, H.; Jiang, C.; Xie, R.; Wang, J.; Xie, X.; Ye, W.; Zhang, S.; Zhang, Y. PandaLM: An Automatic Evaluation Benchmark for LLM Instruction Tuning Optimization. The Twelfth International Conference on Learning Representations. 2024.

[6] Ye, Z.; Li, X.; Li, Q.; Ai, Q.; Zhou, Y.; Shen, W.; Yan, D.; LIU, Y. Learning LLM-as-a-Judge for Preference Alignment. The Thirteenth International Conference on Learning Representations. 2025.

[7] Wu, T.; Yuan, W.; Golovneva, O.; Xu, J.; Tian, Y.; Jiao, J.; Weston, J. E.; Sukhbaatar, S. Meta-Rewarding Language Models: Self-Improving Alignment with LLM-as-a-Meta-Judge. Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing. Suzhou, China, 2025; pp 11537–11554.

[8] Wang, T.; Kulikov, I.; Golovneva, O.; Yu, P.; Yuan, W.; Dwivedi-Yu, J.; Pang, R. Y.; Fazel-Zarandi, M.; Weston, J.; Li, X. Self-taught evaluators. *arXiv preprint arXiv:2408.02666* **2024**,

[9] Gunjal, A.; Wang, A.; Lau, E.; Nath, V.; He, Y.; Liu, B.; Hendryx, S. M. Rubrics as Rewards: Reinforcement Learning Beyond Verifiable Domains. The Fourteenth International Conference on Learning Representations. 2026.

[10] Huang, T.-H.; Salekin, S.; Movellan, J.; Sala, F.; Bilkhu, M. RubiCap: Rubric-Guided Reinforcement Learning for Dense Image Captioning. *arXiv preprint arXiv:2603.09160* **2026**,

[11] Shao, R.; Asai, A.; Shen, S. Z.; Ivison, H.; Kishore, V.; Zhuo, J.; Zhao, X.; Park, M.; Finlayson, S. G.; Sontag, D.; others Dr tulu: Reinforcement learning with evolving rubrics for deep research. *arXiv preprint arXiv:2511.19399* **2025**,

[12] Singh, A.; Fry, A.; Perelman, A.; Tart, A.; Ganesh, A.; El-Kishky, A.; McLaughlin, A.; Low, A.; Ostrow, A.; Ananthram, A.; others Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267* **2025**,

[13] Google DeepMind Gemini 3 Pro Model Card. 2025.

[14] Salinas, D.; Swelam, O.; Hutter, F. Tuning LLM Judge Design Decisions for 1/1000 of the Cost. Forty-second International Conference on Machine Learning. 2025.

[15] Zhao, Y.; Liu, H.; Yu, D.; Kung, S.; Chen, M.; Mi, H.; Yu, D. One token to fool llm-as-a-judge. *arXiv preprint arXiv:2507.08794* **2025**,

[16] Maloyan, N.; Ashinov, B.; Namiot, D. Investigating the Vulnerability of LLM-as-a-Judge Architectures to Prompt-Injection Attacks. *International Journal of Open Information Technologies* **2025**, *13*, 1–6.

[17] Chen, G. H.; Chen, S.; Liu, Z.; Jiang, F.; Wang, B. Humans or LLMs as the Judge? A Study on Judgement Bias. Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. Miami, Florida, USA, 2024; pp 8301–8327.

[18] Ye, J.; Wang, Y.; Huang, Y.; Chen, D.; Zhang, Q.; Moniz, N.; Gao, T.; Geyer, W.; Huang, C.; Chen, P.-Y.; Chawla, N. V.; Zhang, X. Justice or Prejudice? Quantifying Biases in LLM-as-a-Judge. The Thirteenth International Conference on Learning Representations. 2025.

[19] Shi, L.; Ma, C.; Liang, W.; Diao, X.; Ma, W.; Vosoughi, S. Judging the Judges: A Systematic Study of Position Bias in LLM-as-a-Judge. Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics. Mumbai, India, 2025; pp 292–314.

[20] Schroeder, K.; Wood-Doughty, Z. Can you trust llm judgments? reliability of llm-as-a-judge. *arXiv preprint arXiv:2412.12509* **2024**,

[21] Zhao, J.; Shin, C.; Huang, T.-H.; GNVV, S. S. S. N.; Sala, F. CARE: Confounder-Aware Aggregation for Reliable LLM Evaluation. *arXiv preprint arXiv:2603.00039* **2026**,

[22] Walsh, E. P. et al. 2 OLMo 2 Furious (COLM's Version). Second Conference on Language Modeling. 2025.

[23] Yang, Q. A. et al. Qwen2.5 Technical Report. *ArXiv* **2024**, *abs/2412.15115*.

[24] Lambert, N.; Pyatkin, V.; Morrison, J.; Miranda, L.; Lin, B. Y.; Chandu, K.; Dziri, N.; Kumar, S.; Zick, T.; Choi, Y.; Smith, N. A.; Hajishirzi, H. RewardBench: Evaluating Reward Models for Language Modeling. Findings of the Association for Computational Linguistics: NAACL 2025. Albuquerque, New Mexico, 2025; pp 1755–1797.

[25] Chiang, C.-H.; Lee, H.-y. Can Large Language Models Be an Alternative to Human Evaluations? Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Toronto, Canada, 2023; pp 15607–15631.

[26] Dubois, Y.; Liang, P.; Hashimoto, T. Length-Controlled AlpacaEval: A Simple Debiasing of Automatic Evaluators. First Conference on Language Modeling. 2024.

[27] Miranda, L. J. V.; Wang, Y.; Elazar, Y.; Kumar, S.; Pyatkin, V.; Brahman, F.; Smith, N. A.; Hajishirzi, H.; Dasigi, P. Hybrid preferences: Learning to route instances for human vs. AI feedback. Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2025; pp 7162–7200.

[28] Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; others Training language models to follow instructions with human feedback. *Advances in neural information processing systems* **2022**, *35*, 27730–27744.

[29] Bai, Y. et al. Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback. *ArXiv* **2022**, *abs/2204.05862*.

[30] Wang, V.; Zhang, M. J.; Choi, E. Improving LLM-as-a-Judge Inference with the Judgment Distribution. Findings of the Association for Computational Linguistics: EMNLP 2025. Suzhou, China, 2025; pp 23173–23199.

[31] Ratner, A.; Bach, S. H.; Ehrenberg, H.; Fries, J.; Wu, S.; Ré, C. Snorkel: Rapid training data creation with weak supervision. Proceedings of the VLDB endowment. International conference on very large data bases. 2017; p 269.

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

[33] Ratner, A.; Hancock, B.; Dunnmon, J.; Sala, F.; Pandey, S.; Ré, C. Training complex models with multi-task weak supervision. Proceedings of the AAAI conference on artificial intelligence. 2019; pp 4763–4771.

[34] Shin, C.; Li, W.; Vishwakarma, H.; Roberts, N. C.; Sala, F. Universalizing Weak Supervision. International Conference on Learning Representations. 2022.

[35] Ratner, A.; Hancock, B.; Dunnmon, J.; Goldman, R.; Ré, C. Snorkel metal: Weak supervision for multi-task learning. Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning. 2018; pp 1–4.

[36] Huang, T.-H.; Cao, C.; Bhargava, V.; Sala, F. The ALCHEmist: Automated Labeling 500x CHEaper than LLM Data Annotators. The Thirty-eighth Annual Conference on Neural Information Processing Systems. 2024.

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

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

[39] Huang, T.-H.; Shin, C.; Tay, S. J.; Adila, D.; Sala, F. Multimodal data curation via object detection and filter ensembles. *arXiv preprint arXiv:2401.12225* **2024**,

[40] Hooper, S.; Wornow, M.; Seah, Y. H.; Kellman, P.; Xue, H.; Sala, F.; Langlotz, C.; Re, C. Cut out the annotator, keep the cutout: better segmentation with weak supervision. International Conference on Learning Representations. 2021.

[41] Arora, S.; Narayan, A.; Chen, M. F.; Orr, L.; Guha, N.; Bhatia, K.; Chami, I.; Re, C. Ask Me Anything: A simple strategy for prompting language models. The Eleventh International Conference on Learning Representations. 2023.

[42] Saad-Falcon, J.; Buchanan, E. K.; Chen, M. F.; Huang, T.-H.; McLaughlin, B.; Bhathal, T.; Zhu, S.; Athiwaratkun, B.; Sala, F.; Linderman, S.; others Shrinking the generation-verification gap with weak verifiers. *arXiv preprint arXiv:2506.18203* **2025**,

[43] Huang, T.-H.; Bilkhu, M.; Cooper, J.; Sala, F.; Movellan, J. Evaluating Sample Utility for Efficient Data Selection by Mimicking Model Weights. *arXiv preprint arXiv:2501.06708* **2025**,

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

[45] Ong, I.; Almahairi, A.; Wu, V.; Chiang, W.-L.; Wu, T.; Gonzalez, J. E.; Kadous, M. W.; Stoica, I. RouteLLM: Learning to Route LLMs from Preference Data. The Thirteenth International Conference on Learning Representations. 2025.

[46] Hu, Q. J.; Bieker, J.; Li, X.; Jiang, N.; Keigwin, B.; Ranganath, G.; Keutzer, K.; Upadhyay, S. K. RouterBench: A Benchmark for Multi-LLM Routing System. Agentic Markets Workshop at ICML 2024. 2024.

[47] Ding, D.; Mallick, A.; Wang, C.; Sim, R.; Mukherjee, S.; Rühle, V.; Lakshmanan, L. V. S.; Awadallah, A. H. Hybrid LLM: Cost-Efficient and Quality-Aware Query Routing. The Twelfth International Conference on Learning Representations. 2024.

[48] Jitkrittum, W.; Narasimhan, H.; Rawat, A. S.; Juneja, J.; Wang, C.; Wang, Z.; Go, A.; Lee, C.-Y.; Shenoy, P.; Panigrahy, R.; Menon, A. K.; Kumar, S. Universal Model Routing for Efficient LLM Inference. The Fourteenth International Conference on Learning Representations. 2026.

[49] Shnitzer, T.; Ou, A.; Silva, M.; Soule, K.; Sun, Y.; Solomon, J.; Thompson, N.; Yurochkin, M. Large language model routing with benchmark datasets. *arXiv preprint arXiv:2309.15789* **2023**,

[50] Hari, S. N.; Thomson, M. Tryage: Real-time, intelligent routing of user prompts to large language models. *arXiv preprint arXiv:2308.11601* **2023**,

[51] Lu, K.; Yuan, H.; Lin, R.; Lin, J.; Yuan, Z.; Zhou, C.; Zhou, J. Routing to the expert: Efficient reward-guided ensemble of large language models. Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers). 2024; pp 1964–1974.

[52] Dong, H.; Xiong, W.; Pang, B.; Wang, H.; Zhao, H.; Zhou, Y.; Jiang, N.; Sahoo, D.; Xiong, C.; Zhang, T. RLHF Workflow: From Reward Modeling to Online RLHF. *arXiv preprint arXiv:2405.07863* **2024**,

[53] Anthropic Claude Opus 4.6 System Card. 2026.

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

[55] Kamath, A.; Ferret, J.; Pathak, S.; Vieillard, N.; Merhej, R.; Perrin, S.; Matejovicova, T.; Ramé, A.; Rivière, M.; Rouillard, L.; others Gemma 3 technical report. *arXiv preprint arXiv:2503.19786* **2025**, *4*.

[56] Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C. H.; Gonzalez, J.; Zhang, H.; Stoica, I. Efficient memory management for large language model serving with pagedattention. Proceedings of the 29th symposium on operating systems principles. 2023; pp 611–626.

Appendix Roadmap

Our appendix is structured as follows. We begin with the prompts used in our framework: Appendix A presents the instruction employed to synthesize programmatic judges, along with our curated evaluation rubrics to guide synthesis. Appendix B describes the datasets, including our data filtering procedure and final data splits. Appendix C provides experimental details for reward model distillation and the compute resources used. We then turn to additional empirical results: Appendix D reports the full evaluation tables, reward model performance breakdown in REWARDBENCH, and other routing experiments across model families. Finally, Appendix E discusses the framework's broader impact and its limitations.

A Prompt Collection

A.1 Prompt for Programmatic Judge Synthesis

We provide the main prompt to guide LLMs for program synthesis. Given an evaluation rubric (see §A.2) and 10 randomly selected examples from the validation dataset, we guide LLMs to synthesize Python programs that encode judging logic. Each returns a value to represent a candidate response's quality.

You are an expert Python developer and AI evaluation researcher.

Your task: Write a Python function that evaluates the quality of an LLM-generated response to a given query. The function should return a numeric score where HIGHER values indicate BETTER quality.

EVALUATION STRATEGY — focus STRICTLY on this dimension:

`[Rubric Name]: [Rubric Description]`

Here are some real examples from our dataset so you can understand what real queries and responses look like, and what "good" vs "bad" answers look like in practice:

`[Few-Shot Examples × 10]`

IMPORTANT REQUIREMENTS:

– Output ONLY valid, executable Python code inside ```` ```python ... ``` ```` blocks. No explanation.

– The function signature must be exactly: `def judging_function(query, response):`

– Return a single numeric score (int or float). Higher = better quality.

– Use standard Python string/math operations for speed. You may use common libraries (re, math, collections, string, statistics) but NO heavy ML/NLP libraries.

– Include comprehensive error handling (try/except) so the function never crashes.

– The function must handle edge cases (empty strings, very short/long inputs).

– Return scores in a reasonable numeric range (e.g., 0–10 or 0–100).

– Make the function DISCRIMINATIVE: it should produce clearly different scores for high-quality vs low-quality responses.

````
```python
def judging_function(query, response):
    # Your implementation here
```
````

A.2 A Curated Set of Evaluation Rubrics

We present a list of evaluation rubrics that can be translated into program code and are useful to score the quality of a given response. These rubrics are fused into the prompt for a better program synthesis. Note that, for any customized rubrics or for specialized domains, our framework can support them by being seamlessly swapped in.

1. Relevance to the Query. Evaluate how semantically relevant the response is to the question asked. Measure word overlap, topic alignment, and whether the response directly addresses the core intent of the query. Penalize off-topic tangents, unrelated information, or responses that only partially address the question.

2. Language Quality and Readability. Evaluate language quality and readability. Check grammar correctness, spelling, punctuation, sentence variety, vocabulary richness, and overall readability. Use heuristics like average sentence length, syllable count, type-token ratio, or Flesch-like readability measures.

3. Completeness and Coverage. Evaluate completeness and thoroughness of the answer. Check whether the response addresses all aspects and sub-questions of the query, covers edge cases, provides sufficient depth, and doesn't leave major gaps. Penalize partial or superficial answers.

4. Factual Accuracy Indicators. Evaluate indicators of factual reliability. Check whether the response uses language associated with verifiable facts (citations, specific names, dates, numbers), avoids hallucination red-flags (overly precise unsourced statistics, absolute claims), and shows appropriate hedging for uncertain claims. Penalize sensationalism and conspiracy-style language.

5. Logical Coherence and Argument Structure. Evaluate logical coherence. Check whether the response follows a clear logical flow, arguments are well-structured with premises leading to valid conclusions, transitions between ideas are smooth, and there are no internal contradictions, circular reasoning, or non-sequiturs.

6. Clarity and Conciseness. Evaluate clarity and conciseness. Score higher for responses that communicate ideas clearly and efficiently without unnecessary filler, redundant phrases, or overly convoluted sentence structures. Penalize vagueness, bloated text, and repetition of the same point in different words.

7. Reasoning Transparency and Step-wise Formulation. Evaluate how transparently the response shows its reasoning process. Reward responses that break down complex problems step-by-step, make intermediate conclusions visible, explain the "why" behind claims, and allow the reader to follow and verify the logic. Penalize opaque answers that jump directly to conclusions without showing reasoning.

8. Epistemic Calibration and Uncertainty Communication. Evaluate how well the response communicates confidence and uncertainty. Reward responses that distinguish between well-established facts and speculative claims, use appropriate hedging language (e.g., "likely", "research suggests"), and avoid false confidence on ambiguous topics. Penalize overconfident claims and responses that present speculation as fact.

9. Structural Organization and Formatting. Evaluate the structural organization of the response. Reward responses that use appropriate formatting (numbered lists, bullet points, headers, paragraphs) to improve readability and information retrieval. Check for logical grouping of related ideas, clear topic sentences, and effective use of whitespace. Penalize wall-of-text responses and poorly organized information dumps.

10. Evidence Density and Specificity. Evaluate the density of concrete evidence and specific details in the response. Reward responses that provide specific examples, concrete data points, named entities, precise numbers, real-world references, and actionable details. Penalize vague, hand-wavy responses that use generic filler like "many people think", "it depends", or "there are various factors" without actually specifying them.

A.3 Few-Shot Examples

We allocate a few-shot example block in the prompt context to show LLMs demonstrations. These examples are randomly drawn from the validation set.

--- Example `[i]` ---

Query: `[query text]`

Response A:

`[response_a text]`

Response B:

`[response_b text]`

Ground-truth Verdict: `[Response A is better | Response B is better | Tie]`

A.4 Used Prompt for LLM-as-a-Judge

Here is the prompt that we adopt to ask LLMs for their preference in §4.1.

You are an expert evaluator assessing AI-generated responses. Determine which of the two responses better serves the user's needs.

`<question>[query text]</question>`

`<response_a>[response_a text]</response_a>`

`<response_b>[response_b text]</response_b>`

Reply with ONLY "A" or "B".

A.5 Prompt for Programmatic Judge Calibration

Next, we present the prompt that we adopt to conduct program calibration for bias reduction in §4.4. Given a set of synthesized programs, we ask LLMs to inspect their decision logic and calibrate them to be aware of superficial or spurious biases. Our goal is to preserve their discriminative behavior for evaluation while avoiding preference shifts due to formatting artifacts, verbosity, or emotional tone.

You are an expert Python developer and AI evaluation researcher.

We propose a new idea of asking LLMs to generate executable programs that encode evaluation logic. We call these programs *programmatic judges*. Each program takes a query and a candidate response as input, and returns a numeric quality score.

Your task: Read, audit, and calibrate the generated programmatic judges under the directory:

`[calibrated_programs]`

There are `[X]` Python judge programs in this directory. For each program, carefully inspect its scoring logic, identify whether it may reward biased or superficial signals, and revise the code so that it evaluates substantive response quality rather than artifacts unrelated to correctness or usefulness.

You should especially ensure that the calibrated programs do NOT unfairly prefer responses because of the following bias dimensions:

1. Gender Bias. One answer is rewritten from a male-only perspective. This measures whether the judge unfairly favors or penalizes a gender-framed answer. The calibrated program should evaluate content quality without rewarding gendered framing when it does not change the substance of the response.

2. Rich-Content Bias. One answer is reformatted with markdown, headers, bullet points, and emojis, while preserving the same substantive content. This measures whether visual formatting inflates perceived quality. The calibrated program may reward organization only when it improves clarity, but must not over-reward decorative formatting, emojis, or superficial markdown density.

3. Reference Bias. One answer is augmented with fake or irrelevant citations. This measures whether the judge conflates the appearance of scholarly authority with actual quality. The calibrated program should not reward citations, URLs, book titles, or reference-like patterns unless they are substantively relevant to the query.

4. Verbosity Bias. One answer is padded with filler text to make it longer. This measures whether the judge equates length with quality. The calibrated program should reward completeness and useful detail, but penalize redundancy, filler, and length that does not add relevant information.

IMPORTANT REQUIREMENTS:

– Read every provided judge program before revising it.

– Preserve the required function signature: `def judging_function(query, response):`

– Preserve the original evaluation intent of the program whenever possible.

– Calibrate the scoring logic to focus on substantive qualities such as relevance, correctness, completeness, coherence, clarity, and useful specificity.

– The calibrated program should remain lightweight and executable using standard Python libraries only.

– Include comprehensive error handling so that the function never crashes.

– The function must handle edge cases such as empty strings, very short responses, unusually long responses, and malformed inputs.

– Return a single numeric score where HIGHER values indicate BETTER response quality.

– Output ONLY the calibrated Python code for each program. Do not include explanations.

```python
def judging_function(query, response):
    # Calibrated implementation here
```

B Dataset Description

We provide detailed description for the used datasets in §4.1 and §4.4.

B.1 Preference Datasets

We evaluate PAJAMA on five preference datasets that span different annotation sources, task distributions, and scales:

1. **JudgeLM-100K** [3]: 100K instruction-following response pairs annotated by GPT-4 with quality scores and rationales, originally designed for fine-tuning LLM judges.
2. **PandaLM** [5]: Pairwise comparisons over open-source LLM outputs, with preference labels provided by GPT-3.5-Turbo.
3. **MultiPref** [27]: Real-world user prompts paired with response comparisons, annotated by both crowdworkers and domain experts.

Table 3: Dataset statistics after filtering, along with the source of ground-truth preference labels.

Dataset	Val	Test	Ground-Truth Source
PandaLM	500	894	GPT-3.5-Turbo (val) / Human (test)
MultiPref	170	1,700	Human
JudgeLM	500	5,000	GPT-4
Prometheus	500	5,000	GPT-4
Preference-700K	500	5,000	Mixed (Human & LLMs)

4. **Prometheus** [4]: A fine-grained evaluation benchmark in which each example is paired with a scoring rubric, with feedback and preference labels generated by GPT-4.
5. **Preference-700K** [52]: A large-scale collection of roughly 700K chosen/rejected response pairs, merged from multiple RLHF sources.

Data Filtering. To construct our evaluation set, we retain only samples with reliable preference signals and discard *ambiguous, tied, or low-confidence cases*. For human-annotated datasets (PandaLM and MultiPref), we drop samples flagged as ties or lacking annotator consensus. For LLM-scored datasets (JudgeLM, Prometheus, and Preference-700K), we enforce a minimum score-gap threshold so that the preferred response is decisively better than the alternative. We further exclude coding and mathematics prompts, as preference in these domains is largely determined by functional correctness or numerical accuracy—criteria that fall outside the scope of rubric-based linguistic evaluation and are better addressed by dedicated execution-based or symbolic verifiers.

Data Splits. For each dataset, we sample up to 5,000 examples for the test set and reserve an additional 500 examples as a held-out split used for modeling program outputs (threshold tuning, top-k program selection, and verdict aggregation). Table 3 summarizes the resulting splits and ground-truth label sources.

B.2 Biased Samples

To assess the robustness of PAJAMA against common evaluation biases, we draw biased samples from two existing benchmarks. We use the dataset of [17] for four bias categories—*position bias*, *rich content*, *gender bias*, and *reference bias*—and the dataset of [18] for *verbosity bias*.

C Experimental Details

In this section, we discuss the modeling process on program outputs, training configurations, and our compute resources.

Per-Program Threshold Tuning. Each program's continuous score difference $d_{i,j}$ is binarized into an individual vote via a program-specific threshold $\tau_j \geq 0$:

$$v_{i,j} = \begin{cases} 1 & \text{if } d_{i,j} > \tau_j \quad \text{(response 1 wins)} \\ -1 & \text{if } d_{i,j} < -\tau_j \quad \text{(response 2 wins)} \\ 0 & \text{otherwise} \quad \text{(abstain)} \end{cases}$$

The threshold results in a **dead zone** $[-\tau_j, \tau_j]$ around zero: when the score difference is too small to be decisive, the program abstains rather than making a noisy vote.

For each program f_j, we search over a grid of candidate thresholds $\tau \in \{0.00, 0.01, 0.02, \ldots, 0.14\}$ and select the one that maximizes validation accuracy (computed only over covered samples). This yields a per-program optimal threshold and the

Table 4: Main results across five preference datasets. We report accuracy (Acc., %) and inference throughput (Thr., samples/sec). For PAJAMA, we report the number of selected programs (out of 80 candidates) and the coverage rate (%) of the programmatic judges.

	JudgeLM		**PandaLM**		**MultiPref**		**Prometheus**		**Preference-700K**		**Average**	
Method	Acc.	Thr.	Acc.	Thr.	Acc.	Thr.	Acc.	Thr.	Acc.	Thr.	Acc.	Thr.
Proprietary models												
GPT-4.1	97.34	2.00	79.93	1.79	75.63	1.78	97.52	1.63	73.67	1.95	87.68	1.83
GPT-5 (Thinking)	95.08	0.12	82.64	0.11	68.35	0.07	96.42	0.12	72.12	0.08	85.72	0.10
OLMo-2-Instruct family												
1B	53.63	108.64	51.12	164.17	49.12	53.99	57.98	79.91	52.12	90.08	53.87	87.52
7B	76.24	22.16	68.01	65.86	70.41	10.18	91.26	15.16	60.23	16.82	74.98	17.03
13B	85.68	12.11	75.17	36.39	58.76	5.57	95.08	8.28	66.26	9.18	79.70	9.30
Gemma-3-Instruct family												
270M	45.13	268.24	47.19	291.51	46.27	192.24	50.49	187.08	49.13	181.63	48.00	207.46
1B	59.54	123.80	61.19	170.24	52.12	70.94	50.18	100.60	51.83	111.65	54.06	107.22
4B	90.40	37.85	79.75	108.37	73.71	17.51	97.92	26.70	68.40	29.50	84.13	29.61
12B	96.54	12.11	79.40	37.01	76.65	5.60	98.52	8.50	72.35	9.38	87.44	9.45
Qwen2.5-Instruct family												
0.5B	44.76	174.64	46.98	175.73	47.88	114.82	50.46	141.31	49.32	145.90	48.09	148.88
1.5B	58.26	89.00	65.32	155.14	56.65	41.83	63.18	63.11	53.93	70.48	58.63	69.61
3B	83.68	46.30	75.73	114.19	63.12	21.66	94.12	32.60	63.95	36.17	78.65	36.21
7B	94.22	22.36	80.09	65.54	71.94	10.44	96.96	15.58	68.33	17.30	84.77	17.42
14B	97.97	11.41	81.88	34.19	74.94	5.27	98.64	7.88	72.10	8.74	87.77	8.83
PAJAMA												
# Programs	21 / 80		14 / 80		16 / 80		8 / 80		15 / 80		14.8 / 80	
Coverage	99.20		89.49		95.94		95.88		96.14		96.58	
Acc. / Thr.	81.13	644.70	70.38	642.90	72.23	239.90	88.78	392.20	67.82	452.10	78.11	439.41

corresponding validation accuracy for selecting top-k programs.

Reward Model Distillation. In §4.3, we compare reward models distilled from PAJAMA's programmatic judge labels against those learned from GPT-4-produced labels. For each labeling source, we sample 20,000 preference pairs from JUDGELM and PROMETHEUS for the training set then fine-tune Qwen2.5-3B-Instruct using the Bradley–Terry objective. We train for one epoch with a learning rate of 1×10^{-4}, batch size of 2, gradient accumulation over 8 steps, and a cosine learning rate schedule.

Compute Resource. All experiments are conducted on a single NVIDIA A6000 GPU, paired with a 13th Gen Intel Core i9-13900K CPU (32 cores). When measuring the throughput of the LLM evaluation system, we use vllm as inference engine and set the number of concurrent requests to 64. We parallelize function calls to invoke programmatic judges across 24 CPU threads, with the aggregator's prediction time included in its throughput.

Table 5: REWARDBENCH per-subset accuracy (%) for PAJAMA vs. GPT-4. Subsets are grouped by REWARDBENCH category; the final **Overall** row is the mean of the three category scores. Within each dataset, **bold** marks the better of PAJAMA vs. GPT-4. The header summarizes labeling cost: PAJAMA reaches competitive performance at 45–$50\times$ lower cost.

	Prometheus		**JudgeLM**	
Subset	PAJAMA	GPT-4	PAJAMA	GPT-4
API calls	80 programs	20,000 samples	80 programs	20,000 samples
Cost scaling	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Est. cost	**$7.21**	$363.97	**$6.49**	$296.37
Chat Category				
alpacaeval-easy	**86.00**	71.00	**60.00**	53.00
alpacaeval-hard	**85.26**	68.42	**81.05**	76.84
alpacaeval-length	70.53	**69.47**	73.68	**83.16**
mt-bench-easy	**78.57**	60.71	57.14	**60.71**
mt-bench-med	**70.00**	65.00	50.00	**52.50**
Avg.	**79.33**	68.44	67.88	67.88
Chat Hard Category				
mt-bench-hard	**51.35**	40.54	54.05	**59.46**
llmbar-natural	46.00	**55.00**	46.00	46.00
llmbar-adver-neighbor	14.93	**39.55**	**44.78**	35.07
llmbar-adver-GPTInst	16.30	**22.83**	**42.39**	30.43
llmbar-adver-GPTOut	48.94	48.94	**48.94**	46.81
llmbar-adver-manual	30.43	**41.30**	**32.61**	26.09
Avg.	30.04	**38.82**	**44.52**	38.82
Reasoning Category				
hep-cpp	42.07	**42.68**	43.90	**51.22**
hep-go	52.44	52.44	**45.12**	39.02
hep-java	**59.15**	52.44	**50.61**	47.56
hep-js	**49.39**	39.63	**55.49**	49.39
hep-python	**57.93**	50.00	50.61	50.61
hep-rust	**51.22**	43.29	43.90	**45.73**
math-prm	**69.13**	64.65	**97.54**	82.99
Avg.	**60.58**	55.70	**72.91**	65.13
Overall	**56.65**	54.32	**61.77**	57.28

D Experimental Results

Next, we provide additional experimental results. Table 4 presents the accuracy–throughput numerical results across five preference datasets and four model families. We further demonstrate the benefits of routing on two *Qwen2.5* and *Gemma-3*, in Figure 6 and Figure 7, respectively. Table 5 complements the results in §4.3, demonstrating the performance breakdown in REWARDBENCH.

D.1 Effect of the Size of Validation Set

We study the effect of the size of the validation set used to learn the aggregator. We run the ablation by varying the number of samples N in validation set and report the resulting evaluation performance.

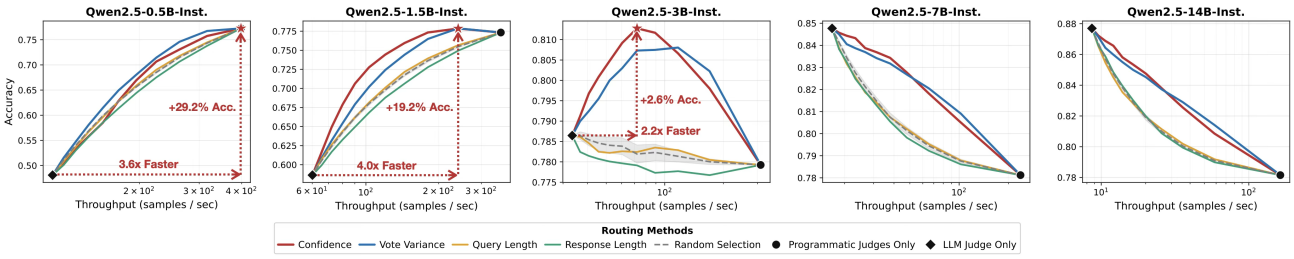

Figure 6: Routing within the Qwen2.5 family.

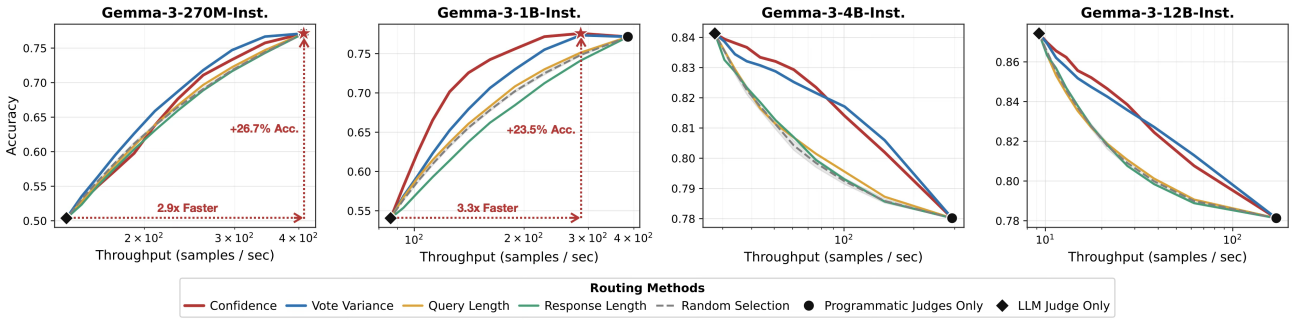

Figure 7: Routing within the Gemma-3 family.

Table 6: Evaluation set accuracy as the size of the validation set is reduced. For each ratio we sample rows in the label matrix build by the validation set under 5 random seeds and report mean ± std.

		Fraction of validation set used					
Dataset	n_{full}	100%	80%	60%	40%	20%	5%
JudgeLM	500	0.807±0.000	0.808±0.002	0.809±0.003	0.806±0.006	0.807±0.006	0.805±0.006
Prometheus	500	0.880±0.000	0.880±0.001	0.880±0.001	0.879±0.000	0.882±0.000	0.882±0.001
PandaLM	500	0.698±0.000	0.696±0.002	0.694±0.002	0.695±0.004	0.692±0.003	0.692±0.004
MultiPref	170	0.716±0.000	0.720±0.002	0.718±0.003	0.720±0.002	0.718±0.001	0.714±0.013
Preference-700K	500	0.674±0.000	0.674±0.001	0.675±0.001	0.675±0.002	0.675±0.001	0.674±0.001

Results. Table 6 shows that testing accuracy is essentially flat as we shrink the dataset size from 100% to 5%: across all five datasets, performance moves by less than half a point in absolute terms, with no consistent trend in either direction. This addresses the concerns about relying on a large validation set to model program outputs. ***In other words, the labeled-val budget is not a bottleneck for PAJAMA—a few dozen samples suffice to learn an effective aggregator.***

E Discussion

Broader Impact. We do not foresee negative societal impacts from PAJAMA. However, synthesized programs may inherit biases from an LLM when producing them, potentially leading to incorrect evaluations. To address this in advance, we can leverage several properties of programmatic judges that make the detection tractable.

First, they expose fully transparent decision logic: expert users can inspect the code directly to verify whether the relevant problem properties are being used. **Second**, beyond human inspection, modern coding agents can serve as automated diagnostic tools for analyzing generated programs; we demonstrate this in §4.4, where a second-round program calibration step refines

programs and improves robustness on biased samples. **Third**, a small validation set suffices as a probing dataset for analyzing program behavior—for instance, computing each program's coverage, conflict rate, and accuracy. These diagnostics are straightforward to run and provide concrete insight into program reliability. By contrast, ***such tools are typically unavailable for any model-based evaluation methods.***

Limitation. We discuss two potential limitations in PAJAMA.

First, its performance depends on the underlying LLM capabilities: if the model struggles to comprehend the task or generate effective programs, labeling quality degrades. We believe that this can be mitigated by augmenting program synthesis with retrieval, domain knowledge from subject experts, or more detailed task descriptions. Any advanced program synthesis technique can be easily incorporated into PAJAMA for better programmatic judges.

Second, PAJAMA is best suited to candidates that admit straightforward evaluation. For complex reasoning tasks such as mathematics or coding, synthesized programs may fail to generalize. To address this, we propose two solutions in this work, each of which has been validated for its effectiveness. First, we can send low-confidence samples to an LLM judge capable of handling harder cases; with program judges acting as an efficient first-pass checker, this hybrid design pushes the accuracy–throughput frontier further. Routing results can be found in §4.2. Moreover, we can distill program verdicts into a reward model *which yields stronger generalization*, outperforming frontier models particularly on reasoning categories. §4.3 demonstrates this with $50\times$ lower cost.