# AURA: Agentic Diagnosis and Refinement for Production Recommender Systems at Scale

SungGeun Kim[1,‡], Abhinav Narain[1,†] and Daniel Nemirovsky[1,†]

[1]*The Walt Disney Company, San Francisco, CA, USA*

**Abstract**

How and why does a recommender system fail the users it serves? Oftentimes in recommender systems, practitioners are left to improve their algorithms based on a combination of feedback from stakeholder teams, domain expertise, and insights from data analyses. Yet, the nuances of how and where the recommendations are performing well or poorly for the end users are difficult to discern from aggregate quantitative metrics.

Whereas these metrics can provide a high-level and incomplete picture, further granularity into the quality of recommendations and their patterns requires reasoning with domain understanding and objectivity, at scale. In this paper, we contemplate this complex conundrum and describe a method and implementation that utilizes the latest AI agentic advances to provide actionable diagnoses and improvements for the production recommender systems.

We present AURA (Agentic Understanding and Refinement of recommender Algorithms), an end-to-end agentic system that performs qualitative evaluation at scale and can then generate improvements to our algorithms at the code level. Specialized agents read production engagement logs, from thousands of sessions to millions, and surface patterns and examples of how the recommender fails real users. The next step uses those diagnoses as well as context about the recommender's own code, data, and training pipeline to propose and implement refinements grounded in that codebase. We report the system design, initial tests on production data from two large consumer platforms at a major media-streaming company, safeguards, operational learnings, and early results toward a self-improving recommender system. Finally, the diagnostic gap AURA closes is not specific to streaming. The architecture is built to transfer and every domain-specific element enters through the configuration layer that already ported it between our two platforms. We map it concretely to e-commerce and online-retail recommendation.

**Keywords**

recommender systems, agentic AI, LLM agents, qualitative evaluation at scale, failure diagnosis, e-commerce recommendation, production systems

## 1. Introduction

Aggregate evaluation metrics (AUC, NDCG, precision/recall, diversity, coverage) are the standard quantitative toolkit for recommender systems [1, 2]. They tell teams whether a model is, on average, doing better or worse than a baseline. However, these metrics may obfuscate finer grained qualitative issues occurring in certain situations for items and/or users. For instance, even though a model can improve across aggregate metrics, it may quietly be surfacing undesirable or egregious recommendations, introducing diversity collapses, personalization regressions, or temporal staleness. McNee et al. argued that being accurate is not enough and accuracy-centric evaluation misses what makes recommendations useful to users [3]. More recently, Edizel et al. reported large offline improvements at Netflix that did not translate to online gains, because content-type effects do not show up in aggregate scores [4]. Broad aggregation can induce Simpson's paradox [5, 6] and mask severe degradation for specific demographic subgroups, tail items, or content producers [7, 8, 9, 10, 11].

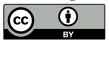
Finding the correct slices and segments for deeper eval analyses, or augmenting the eval metric set, requires acute domain knowledge and engineering effort, especially as these scopes evolve. This

*GenAIECommerce'26: The Third Workshop on Agentic and Generative AI for E-Commerce, co-located with RecSys, September 28, 2026, Minneapolis, MN, USA*

[†] All authors contributed equally to this research.

[‡] Work done while at The Walt Disney Company. Now at Intuit.

snugkim@gmail.com (S. Kim); abhinav.narain@disney.com (A. Narain); daniel.nemirovsky@disney.com (D. Nemirovsky)



diagnostic gap is not specific to any one domain. An e-commerce product feed that quietly over-promotes high-margin items to price-sensitive shoppers, or a retail search ranker that buries long-tail inventory for a subset of queries, exhibits the same structural failure: segment-level harm averaged away by a healthy top-line conversion or relevance score.

For industry practitioners this is an operational bottleneck. When a metric regresses offline or online, the cause could be a regional relevance mismatch, a feedback loop narrowing exposure, a deduplication failure across shelves, a cold-start spike, or upstream data corruption. For example, an engineer working on a regression on Platform B can read that NDCG dropped and still have no idea whether the issue is kids' profiles getting horror promos, sports fans getting reruns, or a deduplication bug across the homepage shelves. Top-line metrics tell us that something is wrong but they do not tell us why. A/B tests, the standard tool for causal evaluation, share the same diagnostic blind spot, and they are expensive, slow, and risky on high-impact surfaces [12]. Data scientists and ML engineers are tasked with maintaining and improving the recommender systems often measured by these aggregate quantitative metrics. Yet, without an understanding of the core issues the users are facing regarding their recommendations, the engineers are left with little insight into where the actual areas of concern and opportunity are. To account for this, practitioners typically complement quantitative metrics with manual, and hence small scale, qualitative analyses including slicing cohorts, writing ad-hoc queries, and watching individual sessions. It is this latter type of analysis, detecting issues and patterns from the engagement history and context using domain understanding, objectivity, and reasoning, that the rise of AI agentic systems newly enables performing at scale.

In this paper, we present AURA, an agentic system that automates the workflow from qualitative evaluation through diagnosis to resolution at production scale. Specialized agents read production sessions, can call tools to provide informative platform, user, and content context, and report how the system benefits and fails real users, with severity and evidence attached. They then read the recommender's own code, data, and training pipeline and propose refinements grounded in that codebase. We have implemented an initial version and run it on production engagement data from Platform A and Platform B at a major media-streaming company, producing diagnostics, technical proposals, and pull requests for our ML engineers to review. An engineer stays in the loop and the pipeline can run iteratively and learn from its prior experiments using a memory log. This work provides the following contributions:

1. **An end-to-end agentic design for diagnosing and refining production recommender systems.** To our knowledge, AURA is the first system to place an evidence-backed failure taxonomy between agentic evaluation at scale and code-level refinement. It derives the taxonomy from production session evidence, then works each finding back into the recommender's own codebase. We detail the design and implementation of AURA, a multi-stage agentic pipeline that selects sessions from production engagement data, diagnoses how the recommender benefits and fails users, generates root-cause hypotheses grounded in the recommender's own code and training pipeline, and produces technical proposals for resolution. The architecture is extensible to autonomous implementation, training, and evaluation across different domains.
2. **Implementation and production deployment at Platform A and Platform B.** We document the production implementation behind the design. An architecture-versus-prompt attribution methodology separates structural pipeline gains from prompt-edit gains across iterative refinement passes, lifting stage-level rubric scores from 15/25 to 23/25 on Platform A and from 17/25 to 24/25 on Platform B (§4.2). A cost-efficient multi-model orchestration sends the high-volume classification work to lightweight models and reserves frontier models for interpretation. Upstream per-session judging dominates end-to-end cost and scales with population ($321 over 96,801 Platform A sessions, $250 over 101,594 Platform B sessions), while AURA's downstream aggregate analysis adds a small fraction on top (§4.3). A human-in-the-loop workflow promotes findings to action and dismisses what does not survive scrutiny (§5).
3. **Operational learnings from running AURA on production data.** After the initial discovery phase, locking the category vocabulary clearly outperformed open-ended classification. Post-hoc

cleanup shrank to a single rename with zero merges on Platform B, and both platforms converged on closed taxonomies, with consolidation collapsing Platform A's 12 emergent tags to 8 categories with no diagnostic loss (§4.2). Engineer trust is built through structured evidence rather than narrative explanations (§5).

4. **Safeguards and accountability for agents touching production data and code.** Every agent runs read-only against analytical data lakes, code changes happen in a sandboxed clone, promotion requires a named engineer, and dismissal is a first-class outcome (§5). We catalog the defensive-validation layer this demanded in production, from fabricated-identifier sanitization to failure-taxonomy collapse. (§5).

## 2. Related Work

The LLM-as-Judge paradigm [13] and agentic evaluators [14] can evaluate evaluate recommendations without task-specific training. LLMs have enough competence for zero-shot ranking [15]. However, judges exhibit persistent biases including presentation order sensitivity [16], "style over substance" preferences [17], and unwarranted assertiveness [18]. Dedicated autoraters like FLAMe [19] improve reliability but require task-specific training, and all such systems ultimately require "human grounding" for trustworthy judgments [20]. Closest to our evaluation layer, Zhang et al. [14] remove humans from the loop to scale agentic evaluation of recommendations. AURA consumes that kind of per-session verdict as raw material rather than as the end product, aggregating verdicts into a failure taxonomy and following each finding into the recommender's own code. Because LLMs can interpret rich session context but remain sensitive to prompt framing and aggregation choices at scale, AURA uses them inside a staged diagnostic workflow with filtering, structured outputs, and verification rather than treating any single LLM judgment as final. Programmatic evaluation tools such as Slice Finder [21] and CheckList [22, 23] expose slice-specific or behavior-specific failures, but they do not produce root-cause hypotheses from recommendation-session evidence and domain context. AURA uses LLMs to generate it and to explain recurring failure categories after they have been surfaced.

A separate line evaluates recommenders by simulating users with LLM agents. These include generative user agents in Agent4Rec [24] and RecAgent [25]. Additionally, iEvaLM [26] uses LLM-based user simulators for conversational recommendation, and the broader RecAI toolkit [27] packages similar evaluation and explanation tooling. Whereas simulation provides some counterfactual control, AURA sees the real production sessions as the higher-fidelity evidence, and diagnoses from those.

Wang et al. [28] demonstrate that LLM agents can autonomously optimize recommendation models at YouTube scale through code generation and A/B testing ($O(100)$ experiments/week). Kim et al. [29] propose "directional feedback" co-evolution, though validation remains limited to offline datasets rather than live production environments. Broader agentic-ML benchmarks frame the wider landscape of LLM agents doing machine-learning experimentation and engineering [30, 31, 32]. These systems search over model space (architecture, loss formulation, training schedule, hyperparameters) and optimize against aggregate engagement signals. Critically, AURA starts from the other end. It first diagnoses user-facing failures from recommendation sessions, then grounds candidate refinements in the recommender's own code and training pipeline. The closed-loop extensions we sketch as future work would meet these systems in the middle.

## 3. System Architecture

### 3.1. Overview

AURA is an agentic pipeline that consumes production session logs and emits evidence-backed change proposals against the recommender's own code. The workflow runs as four sequential stages, with AI validation, engineer review, and memory threaded through all of them (Figure 1).

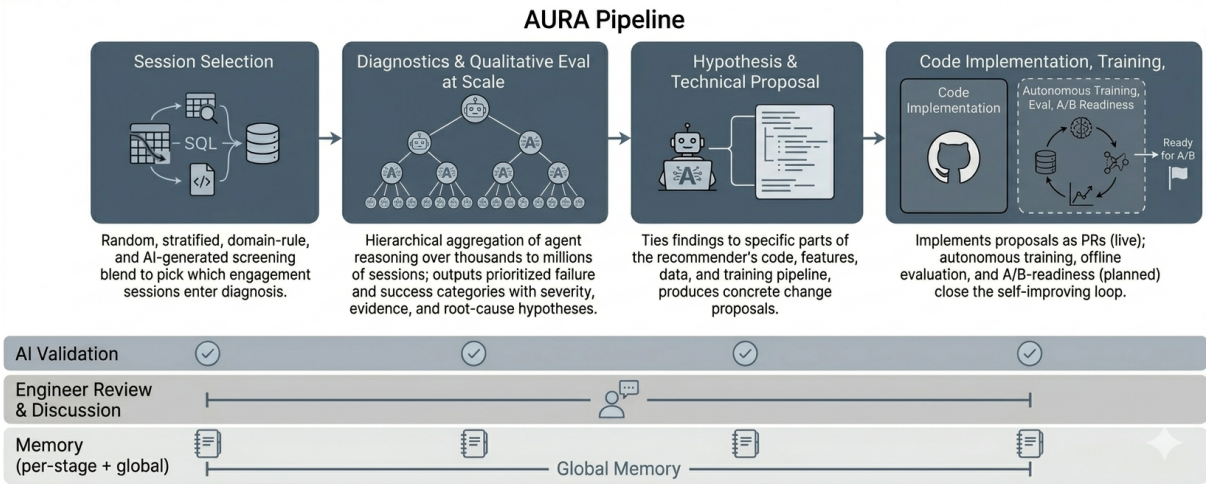



**Figure 1:** The AURA pipeline: Session Selection; Diagnostics & Qualitative Eval at Scale; Hypothesis & Technical Proposal; and Code Implementation, Training, Eval, A/B Readiness, with AI validation, engineer review, and memory running across all four stages. The dashed portion of the final stage is a planned extension. The current deployment delivers it as far as engineer-reviewed pull requests.

A selected stage chooses which sessions to examine. Selected sessions enter a diagnostic stage (§3.3) that scales LLM reasoning over large session volumes through hierarchical agent aggregation. The diagnostic stage applies configurable quality criteria to judge how recommender served users to find problematic patterns. It then emits failure and success categories, each with a severity, supporting evidence, and a root-cause hypothesis. These feed a hypothesis-and-proposal stage (§3.4) that reasons over the recommender's own code, features, and training pipeline. The technical proposals it produces then feed to a code-implementation stage (§3.5) that turns them into pull requests today, and (in the planned extensions) into training, offline evaluation, and A/B-ready artifacts that close the self-improving loop. Agents are given tools to gather additional information about the platform, users, items, and engagement where appropriate and available.

The system is designed for flexibility in that each stage can permit alternative instantiations. For example, session selection can blend random, stratified, domain-rule, and AI-driven mechanisms. The diagnostic aggregation can use a flat panel or a hierarchical tree mechanism and the hypothesis generation can read a narrow slice of the codebase or the full repository. Lastly, the code-change proposals can stop at natural-language descriptions or push to executable patches. A new platform is a configuration change, not a code change. The data sources, column mappings, segment definitions, and prompt overrides are supplied per platform, and platform isolation is enforced by a required parameter that threads through every component. Cost is managed by deterministic orchestrator by routing sessions at each stage to a model suited to its complexity, sending high-volume work (classification, consolidation) to lightweight models and reserving frontier models for analysis, verification, and code generation. We report concrete cost figures in §4.3.

## 3.2. Session Selection Stage

AURA's first stage selects which sessions enter deep diagnosis (Figure 1). This mechanism is configurable, blending random or stratified sampling with engineer-written domain SQL and AI agents. A profile pass summarizes the data distribution for a shared numerical baseline, then samples the segments whose statistics flag issues. The domain SQL catalog draws on recurring pain points from Platform A/B production experience (low NDCG, high watch time, repeat-content counts, sub-genre and content-tone mismatch, kid-profile safety) and research-literature failure modes (popularity bias, position bias, distribution shift, diversity collapse). Multiple independent LLM-powered screening agents then use tools to compose queries that identify which sessions to analyze.

The flagged segments go to $N$ agents that independently propose candidate problem cohorts in

parallel. A dedup pass merges overlapping proposals and consensus step ranks the survivors by majority vote across agent proposals. A surviving cohort might be, for example, kids' profiles in one region receiving horror promotions. Each surviving cohort's sessions are decorated with light per-session context: the judge verdict and reason, enrichment columns (editorial flags, user history, item timestamps, impression actions), and aggregated summaries for the next stage.

### 3.3. Diagnostic Stage

The diagnostic stage turns selected sessions into a prioritized set of failure and success categories, each with a severity, supporting evidence, a root-cause hypothesis, and a GOOD/BAD verdict with free-text reasons (§4.2). Verdicts follow an LLM-judge paradigm under configurable, extensible quality criteria.

The challenge is scale since production runs cover hundreds of thousands to millions of sessions, far beyond what any single agent can hold in context. The design solves this through hierarchical aggregation. At the leaves, parallel agents each consume a tractable slice (on the order of a thousand sessions, within context) under category-specific prompts that surface emergent good and bad patterns with evidence. Findings propagate to peer agents that reason about overlap and significance, consolidate near-duplicates, and surface a smaller, higher-confidence list. This repeats up the tree until a single layer produces the final prioritized list.

The current Platform A and Platform B deployment instantiates this hierarchy as a six-step funnel. *Classification* runs in parallel batches, classifying each session against an emergent or closed taxonomy and assigning a session type and short description. *Consolidation* is the first cross-batch aggregation. It is a single LLM pass that merges near-duplicate categories, renames noisy labels, and drops low-signal entries (in one Platform A run, "duplicate items across shelves" and "repetitive recommendations" merged into "cross-shelf deduplication failure"). *Sampling* retains only categories worth a frontier-model deep dive (via a top-$N$/count/percentage rule, e.g., the largest categories, at least roughly a hundred sessions, or five percent of the type's total) and draws a bounded random sample of sessions per retained category, while safety-critical categories (e.g., content/policy violations) bypass the threshold. *Analysis* runs in parallel across retained categories, one analyst worker per category producing a structured finding (description, severity, root-cause hypothesis, supporting evidence, actionable recommendations). *Verification* sits over the findings. One agent cross-checks consistency, validates claims against the sampled sessions, and corrects severity miscalibrations. *Synthesis* produces an executive summary that groups failures and successes, surfaces cross-cutting observations, and outputs prioritized recommendations.

### 3.4. Hypothesis and Technical Proposal Stage

Diagnostic findings highlight what is going wrong while the next stage proposes what to change and why. For each verified finding, the hypothesis-and-proposal stage pulls in the recommender's own context such as model code, feature definitions, training pipeline, label specifications, as well as data statistics and schemas, and reasons about which parts of the system could be producing the finding. The output is a structured technical proposal that includes a hypothesis tying the finding to specific components (for example, "the ranking model over-weights global popularity signals in the cold-start branch") and one or more candidate change suggestions targeted at those components.

We use the term "root-cause hypothesis" deliberately. AURA does not deliver formal causal inference. It produces explanatory hypotheses grounded in code, data, and training context. The surrounding evidence (sampled session excerpts, the diagnostic agent's reasoning, and schema and code references) is what makes those hypotheses tractable for engineers to verify before committing to a change.

### 3.5. Code Implementation, Training, Evaluation, and A/B Readiness Stage

The final stage takes technical proposals through to code. A generate-evaluate-refine loop [33] prompts a frontier model with the verified finding, the root-cause hypothesis, schema metadata for the relevant tables, and a code index of file paths and line counts (rather than full source, which would exceed context limits). A separate evaluator model scores each suggestion for correctness, feasibility, and side effects

and returns a pass/fail verdict with feedback. Failures, cycle back to the generator. A programmatic pass then checks every surviving suggestion against the actual codebase for nonexistent files, missing imports, or fabricated references.

The current deployment delivers this stage as far as the pull request. Surviving suggestions are written to the suggestions database and surfaced through the shared review interface (§3.6). Two extensions under active development would close the loop end-to-end. The first triggers an automated training run when a promoted suggestion is merged into the recommender's sandboxed repository, and the second runs offline evaluation on the trained artifact and packages an A/B-ready result. With both in place, AURA iterates autonomously between engineer-review checkpoints, proposing, implementing, evaluating, and refining until the offline evaluation clears.

### 3.6. Validation, Engineer Review, and Memory

Validation, engineer review, and memory run through every stage. Validation means that every LLM output boundary carries a programmatic check. Session-ID references are checked to exist in the input, and JSON-schema validation with a regex fallback catches malformed payloads. The diagnostic stage adds a verification agent that cross-checks consistency across findings. The code-implementation stage adds a validator confirming file paths and imports resolve against the actual repository. Parse failures and violations are logged to a structured data-quality flags array and surfaced to engineers.

Engineers, for their part, may interact with every stage through a shared web interface that pairs raw evidence with AURA's structured outputs. They can read sampled sessions, discuss findings with the relevant agents, vote, comment, promote items into team backlogs, or override AURA's calls. Human review is a property of the pipeline, not a final-stage gate.

Memory is a key consideration that closes the loop across runs. Each stage logs the categories surfaced, hypotheses tried, code changes proposed, and outcomes observed, which a shared memory layer aggregates across stages and runs. With this memory, each run builds on what earlier runs found and what already failed, so AURA moves forward instead of repeating itself.

## 4. Evaluation

We considered three main questions which drive our evaluation. Are the findings real? What does a diagnostic run cost? And does the same pipeline produce actionable findings on a platform it was never tuned for?

### 4.1. Experimental Setup

**Data.** We evaluate on production recommendation sessions from two streaming platforms within a large media enterprise: Platform A (family-oriented) and Platform B (general-audience). Platform A contributed 18,901 sessions flagged as BAD by the upstream per-session judge out of 96,801 total sessions evaluated (about 19.5%). Platform B contributed 4,154 BAD sessions out of 101,594 evaluated (about 4.1%). We do not read the 19.5% versus 4.1% gap as a quality ranking between platforms. Rather, judges are prompted and calibrated per platform, and Platform B's prompt was revised mid-stream, so the rates are not directly comparable. These represent all negatively-evaluated sessions from production recommendation serving over the evaluation window, not a sample. Sessions span diverse user cohorts, regions, devices, and content catalogs.

Because AURA's diagnostic quality depends on the upstream per-session judge, we separately validated the judge following the LLM-as-judge framework [13, 34], using a panel of three independent evaluators (Claude Opus 4.6, GPT-5.5, and Gemini 3.1 Pro) with majority-vote aggregation. On a stratified validation sample of $\sim$200 sessions on both platforms, the panel assessed whether each judge verdict was correct, flagging error patterns such as demographic overfit, position bias, and overconfidence. Majority-vote agreement (2+ evaluators judging the verdict correct) was 96.0% on Platform A (192/200) and 87.6% on Platform B (176/201), with full three-way agreement at 80% and 58% respectively. We

**Table 1**
Per-stage rubric evaluation results: baseline → after iterative improvement. Fractions indicate GOOD criteria out of total criteria for each stage.

| | Platform A | | Platform B | |
|---|---|---|---|---|
| Pipeline Stage | Before | After | Before | After |
| Classification | 4/5 | 5/5 | 4/5 | 5/5 |
| Consolidation | 1/4 | 3/4 | 2/4 | 4/4 |
| Category analysis | 3/5 | 5/5 | 4/5 | 5/5 |
| Verification | 2/3 | 3/3 | 2/3 | 3/3 |
| Synthesis | 4/4 | 4/4 | 4/4 | 4/4 |
| Code suggestions | 1/4 | 3/4 | 1/4 | 3/4 |
| Total | 15/25 | 23/25 | 17/25 | 24/25 |

treat the panel as a calibration check, not hard ground truth. At the high positive prevalence here (near 0.9) neither raw agreement nor chance-corrected agreement is decisive, and diagnostic-correctness is a genuinely hard, partly subjective judgment.[1]

We evaluate diagnostic quality through binary rubrics applied independently to each pipeline stage's output, since systemic recommendation failures lack a single definitive root cause and standard classification benchmarks do not apply. For each of the six scored stages we define 3 to 5 criteria (Table 4), each testing a single observable dimension with an unambiguous GOOD/BAD threshold. These six scored stages contribute 25 criteria (score is the count of criteria rated GOOD) in total and are the rubric's unit of evaluation (Classification, Consolidation, Category analysis, Verification, Synthesis, and Code suggestions). They differ from the diagnostic funnel (§3.3) in only two places: the funnel's Sampling step is not scored here, and the rubric adds Code suggestions from the code-implementation stage (§3.5). This per-stage, per-criterion evaluation mirrors the behavioral testing methodology of CheckList [22] and the recsys-specific analog RecList [23], which measure failure rates per capability rather than a single aggregate accuracy. Each criterion is evaluated by two independent LLM judges (Gemini 3.1 Pro and GPT-5.5) examining the pipeline's actual output, and inter-judge agreement determines the final verdict.

For each pipeline stage we follow a systematic improvement cycle: (1) evaluate the baseline output against rubrics for both platforms, (2) identify BAD criteria and their root causes, (3) apply targeted changes (architecture fix or prompt revision), (4) re-evaluate against the same rubrics. We keep a change when more criteria flip (BAD→GOOD) than GOOD→BAD. A criterion flipping the other way blocks the change, unless the two-judge disagreement finds a stricter judge rather than a worse output. This protocol separates architecture-level fixes (code changes affecting data flow) from prompt-level fixes (template changes affecting LLM instructions), so quality improvements can be attributed to their actual cause.

## 4.2. Diagnostic Quality

**Rubric-based evaluation results.** Table 1 reports the per-stage results on both platforms before and after the full improvement cycle. Six architecture fixes plus targeted prompt revisions moved both platforms to near-full GOOD across the 25 criteria of Table 4.

**Architecture vs. prompt attribution.** Most quality improvements came from architecture-level fixes, not prompt engineering. Of six architecture changes (display-name normalization, threshold tuning,

[1]Chance-corrected agreement is low by construction at this prevalence (Fleiss' $\kappa \approx 0.17$–$0.38$ across platforms and label granularities, "slight" to "fair"): when evaluators almost always agree, chance agreement is high and the residual is necessarily small. The informative figure is that a majority explicitly disagrees with the judge on only 3–4% of the validation sample, with non-trivial "unsure" rates (18% on Platform A, 36% on Platform B) that $\kappa$ captures and raw agreement does not. One caveat: Gemini-family models appear both upstream and downstream, which can inflate agreement through self-preference [35].

**Table 2**
Diagnostic output from a representative production run on each platform: failure categories with severity, session count, and share of the BAD population.

| Severity | Category | Sessions | % |
|---|---|---|---|
| *Platform A* | | | |
| Critical | Genre Pref. Mismatch | 9,135 | 48.3 |
| Critical | Sub-Genre Pref. Mismatch | 6,444 | 34.1 |
| High | Franchise Pref. Mismatch | 1,571 | 8.3 |
| High | Content Type Mismatch | 1,036 | 5.5 |
| High | Regional Pref. Mismatch | 978 | 5.2 |
| Medium | Age-Inappropriate Suggestions[†] | 533 | 2.8 |
| Low | Evaluator Label Mismatch[‡] | 250 | 1.3 |
| Low | User-Age Content Era Mismatch | 163 | 0.9 |
| *Platform B* | | | |
| Critical | Genre Pref. Mismatch | 1,911 | 46.0 |
| Medium | Tone & Sensibility Mismatch | 310 | 7.5 |
| Low | Content Format Mismatch | 93 | 2.2 |
| Low | Age-Inappropriate Suggestions[†] | 76 | 1.8 |
| Low | Franchise Affinity Mismatch | 25 | 0.6 |

[†] Safety-relevant; retained via top-$N$/count rule.

[‡] Meta-category: pipeline flagged the upstream judge's own verdict as likely incorrect.

Severity is threshold-derived from session % (Platform A: Critical $\geq$10%, High 5–10%, Medium 2–5%, Low $<$2%; Platform B: Critical $\geq$25%, High 10–24%, Med 3–9%, Low $<$3%). Platform A categories are non-exclusive (a session may carry multiple tags), so percentages sum to $>$100%; 17,996 of 18,901 BAD sessions (95.2%) were categorized. Platform B's higher thresholds retain fewer categories: 2,415 of 4,154 (58.1%) fall in a surfaced category. The rest fell below threshold or were non-diagnostic.

adaptive sample sizes, response-truncation limits, classification-context forwarding, and specialized verifier prompts), three resolved rubric failures that no prompt change could fix. For instance, the severity-data alignment failure in category analysis was not the LLM ignoring severity rules. The prompt template simply never received the category's session percentage, and adding it resolved the issue on both platforms simultaneously. Only two stages (classification and category analysis) needed prompt-level changes. The other four reached full GOOD through architecture fixes alone or needed none.

**Closed taxonomy as a classification technique.** Our most impactful prompt-level finding was that closed taxonomies (explicit category enumeration with a "DO NOT invent new categories" instruction) clearly outperform open-ended classification in production. On Platform B, the open-ended baseline produced 30+ overlapping categories. Adding format and consolidation rules cut this to 22 unique tags but did not eliminate proliferation. The final closed taxonomy (13 defined categories) reached 5/5 GOOD and cut post-hoc cleanup to almost nothing. Classification emitted 16 tags (the model occasionally falls back to off-taxonomy labels) and consolidation reduced to a single rename with no merges or deletions, versus the multi-merge cleanup the baseline required. Platform A converged independently on the same approach (a 13-entry enumeration with the same instruction). Its run produced 12 tags that consolidation collapsed to 8 categories (one merge of three synonymous tags, five renames, no deletions). The transferable insight is that database-populated category vocabularies from prior runs can be overridden by explicit prompt instructions, which requires careful coordination between the prompt template and the vocabulary enrichment system.

**Diagnostic output quality.** Table 2 shows the final diagnostic output from a representative production run on each platform. The pipeline produces a severity-stratified taxonomy in which categories are ordered by session share via configurable per-platform thresholds (Table 2). Severity here is a

**Table 3**
Global offline metrics: candidate-aware fix vs. the production-clone baseline, as a percentage delta on each of three independently-trained days. Every delta is within $\pm 0.1\%$ with no day regressing beyond noise.

| Metric (global) $\Delta$ | Day 1 | Day 2 | Day 3 |
|---|---|---|---|
| Click AUC | +0.03% | +0.06% | +0.01% |
| WM AUC | +0.04% | +0.08% | +0.00% |
| WM weighted AUC | +0.10% | +0.06% | +0.04% |
| NDCG@10 | −0.01% | +0.05% | +0.02% |

prevalence tier. It says how many sessions a failure touches, not how much harm each instance does. Frequency and harm are different axes, which is why safety-relevant categories carry a retention override (§3.3) and surface on both platforms despite low counts. Both platforms independently identify genre preference mismatch as a dominant failure mode, but the manifestations differ (sub-genre and franchise affinity for Platform A's family catalog vs. tone-and-sensibility mismatches on Platform B's general audience), suggesting the pipeline discovers platform-specific patterns rather than generic complaints. The Platform A run also turned on its own judge. In an Evaluator Label Mismatch category (250 sessions), the analysis stage, examining the full session evidence, concluded that the upstream per-session verdict was itself incorrect. Instead of inheriting the judge's mistakes silently, the pipeline caught the judge being wrong.

The two runs also surfaced different root causes: Platform A identified a systemic popularity bias, where globally trending content dominated across all failure categories, traced to the ranking model over-weighting popularity signals; Platform B, a content-sensitivity gap, where insufficient demographic filtering surfaced age-inappropriate recommendations misaligned with user profiles (its Age-Inappropriate Suggestions category).

**Individual stage contributions.** Several of these categories survive only because of the retention rule. Changing it from a percentage-only threshold to a top-$N$/count/percentage threshold surfaced Platform A's low-frequency categories (e.g., User-Age Content Era Mismatch, 163 sessions, 0.9%) and all five of Platform B's failure categories; a percentage-only ($\geq$5%) rule would have kept only Platform B's two largest and discarded the three smaller but still-actionable ones (Content Format, Age-Inappropriate Suggestions, Franchise Affinity). Downstream, the verification agent cross-checked severity calibration across categories, and the cross-stage validation layer (§3.6) stripped hallucinated session identifiers before findings reached engineers.

**Closed-loop fix validation.** The strongest test of whether a finding is real is whether a fix it motivates survives scrutiny. We screened AURA's candidate fixes offline, against the cohort they were meant to fix, before any could spend a production A/B slot. For the dominant Genre Preference Mismatch category, the remediation stage produced two plausible changes: (1) an explicit user-genre $\times$ candidate-genre cross feature, and (2) a candidate-aware attention mechanism pooling a user's genre-watch history conditioned on the candidate. Yet neither improved ranking on the diagnosed cohort (1,911 Platform B sessions), and both tracked the production baseline to within run-to-run noise on aggregate metrics ($\pm$0.1%, Table 3). Offline screening thus leaves the fix an open candidate: safe to iterate on, not yet demonstrated.

## 4.3. Cost-Effectiveness

AURA's diagnosis adds little to the cost of the per-session judging it builds on. Across both platforms, upstream judges account for the overwhelming majority of spend (92–99%), while AURA's aggregate analysis over BAD-verdict sessions is about 8% of the total on Platform A and about 1% on Platform B (Table 5). Multi-model routing is what keeps that share small: Gemini 3 Flash handles high-volume

**Table 4**
Rubric criteria per pipeline stage. Each criterion is binary (GOOD/BAD) with an unambiguous threshold observable from the stage's output.

| Pipeline Stage | Number of Criteria | Description |
|---|---|---|
| Classification | 5 | No catch-all categories; each category maps to one distinct fix; full session coverage; taxonomy compliance; descriptions contain specific pattern + code area |
| Consolidation | 4 | Merges combine genuinely synonymous categories; distinct failures remain separate; renames are more specific than originals; valid output structure |
| Category analysis | 5 | Root cause names specific model component; cites $\geq$2 session patterns; recommendations are engineer-implementable; severity matches session count; all fields present |
| Verification | 3 | Catches factual inconsistencies; preserves correct claims unchanged; flags contradictory recommendations across categories |
| Synthesis | 4 | Numbers match input data; priority justified with reasoning; $\geq$1 cross-cutting observation spans 2+ categories; no hallucinated categories |
| Code suggestions | 4 | All file paths exist in codebase; targets root cause from analysis; implementable as described; no destructive changes |
| Total | 25 | |

classification and consolidation, and the more expensive Gemini 3.1 Pro is reserved for the low-volume category analysis, verification, and synthesis stages.

Within AURA's own stages, classification consumes the majority of tokens (41.8M on Platform A vs. 369K on Platform B), and the rest operate on progressively smaller category-level aggregations. Platform A's higher aggregate cost is driven by larger, higher-context classification batches (757 calls at $\approx$55K input tokens vs. Platform B's 41 at $\approx$9K), which prompt caching partly offsets.

In absolute terms, aggregate diagnosis costs at most $5.38 per actionable finding on either platform, and $43–$50 per finding end-to-end once upstream judging is included. Runtime follows the same split: of the 1,554 and 965 minutes for an end-to-end run on Platform A and Platform B, AURA's own pipeline accounts for only 304 and 72 minutes. Because the end-to-end total is set by per-session judging, which scales linearly with the session population, the marginal cost of adding AURA to an existing evaluation stack is small.

## 4.4. Cross-Platform Generalizability

We ran AURA on production data from two streaming platforms with independent data schemas, content catalogs, recommendation models, and user populations, asking whether the same architecture produces meaningful findings on both without platform-specific tuning. The core pipeline code is identical across platforms; platform-specific configuration supplies data mappings, SQL fragments, prompt templates, taxonomies, and significance thresholds. Despite independent taxonomy development, both platforms converged on a closed 13-category enumeration sharing common concepts while differing in platform-specific categories (§4.2).

**From streaming to e-commerce.** The same portability argument extends across domains, because the architecture never touches domain semantics. The pipeline consumes sessions, a judge verdict, and a codebase, and every domain-specific element (data mappings, taxonomies, prompts, thresholds) arrives through the same configuration layer that ported Platform B to Platform A with no core-code changes. Table 6 makes the correspondence concrete. Each element of the deployed streaming instantiation has a direct conceptual e-commerce analog, so an e-commerce tenant can potentially onboard the way

**Table 5**
End-to-end cost breakdown for production diagnostic runs. Per-session judge costs are estimated from observed per-session rates applied to the full session population (GOOD + BAD). Code-suggestion generation cost is estimated per run from the model-code input context and the generated patches (Opus 4.6 rates), not from telemetry.

| | | Platform A (96,801 sess.) | | Platform B (101,594 sess.) | |
|---|---|---|---|---|---|
| Stage | Model | Cost | % | Cost | % |
| Upstream: Session judges | Flash | $320.98 | 91.9 | $250.21 | 99.0 |
| Classification | Flash | $26.29 | 7.5 | $0.65 | 0.3 |
| Consolidation | Flash | $0.03 | 0.0 | $0.00 | 0.0 |
| Category analysis | Pro | $0.49 | 0.1 | $0.17 | 0.1 |
| Verification | Pro | $0.08 | 0.0 | $0.05 | 0.0 |
| Synthesis | Pro | $0.03 | 0.0 | $0.01 | 0.0 |
| Code suggestions | Opus 4.6 | $1.36 | 0.4 | $1.76 | 0.7 |
| AURA subtotal | | $28.28 | 8.1 | $2.64 | 1.0 |
| Total (end-to-end) | | $349.26 | | $252.85 | |

**Table 6**
The deployed streaming instantiation and its e-commerce analog. Onboarding is designed to be a configuration change.

| Streaming (deployed) | E-commerce analog |
|---|---|
| Session: impressions, clicks, watch time | Session: impressions, add-to-cart, purchase, return signals |
| GOOD/BAD judge over engagement evidence | GOOD/BAD judge over conversion and satisfaction evidence |
| Genre / sub-genre preference mismatch | Category and price-affinity mismatch |
| Franchise affinity mismatch | Brand affinity mismatch |
| Kid-profile safety (age-inappropriate content) | Age-restricted item compliance |
| Post-launch content staleness | Out-of-stock and seasonal staleness |
| Regional preference mismatch | Regional assortment and shipping mismatch |
| Cross-shelf deduplication failure | Duplicate listings across carousels |
| Ranker code, features, training pipeline | Ranker code, features, training pipeline (unchanged) |

Platform A did, through configuration.

## 5. Deployment and Lessons

We are integrating AURA into the ML engineering workflow alongside existing A/B testing and metrics dashboards. The port from Platform B to Platform A required no core-pipeline changes (§4.4). Database-backed prompt templates support versioning and rollback, enabling prompt iteration without redeployment, while engineers review findings through a shared web interface. Model-version changes can affect prompt formatting and baseline distributions, so we treat prompts as versioned artifacts and re-evaluate them against historical baselines before upgrades.

LLM agents with access to production data and code need explicit safeguards at every boundary. Every agent runs read-only against analytical data lakes, and the code-implementation stage operates on a cloned, sandboxed repository. A named human owns every change that ships. AURA generates pull requests, and every finding carries the raw session evidence (session IDs, severity, root-cause hypotheses) the engineer validates independently. Because a model-suggested change that breaks production lands on a named owner, engineer review is a safety mechanism rather than a convenience layer, and dismissal is a first-class outcome.

Beyond the diagnostic results of §4.2, four operational choices proved decisive for adoption. We closed the category vocabulary after an initial discovery phase, and the taxonomy stopped drifting across runs. Engineers trusted findings precisely because they could check them, so LLM-authored prose became packaging rather than the deliverable. Per-step cost went on the screen instead of in a billing dashboard, which made each run easy to justify. And the mandatory platform parameter that looked redundant caught cross-platform data-mixing bugs at development time and let Platform A onboard with almost no new code.

The dominant lesson, and the hardest, was that every LLM output boundary needs a defensive layer, implemented as the cross-stage validation layer of §3.6. Each failure below surfaced in a baseline run and is now caught programmatically:

- **Fabricated identifiers.** On a baseline Platform B run (prior to the fixes in §4.2), the category-analysis stage populated session-ID fields with 904 fabricated UUIDs dressed up with real collection names to look plausible (e.g., `37a9...5147::Top 15 Today`). Post-verification sanitization now strips any ID absent from the input.
- **Severity miscalibration.** Over-inflated severity scores traced to the LLM inferring session counts. Passing counts and percentages into the prompt directly resolved it on both platforms simultaneously.
- **Schema and reference violations.** Malformed JSON is caught by defensive validation with a regex fallback. Code suggestions that referenced nonexistent files are now validated against the actual codebase.
- **Taxonomy coordination failures.** Closed taxonomies are brittle. For instance, when Platform A's category database was wiped, restrictive prompt instructions collapsed all sessions into two categories until the taxonomy was re-embedded.

## 6. Future Work

Two directions define our next steps: stronger evidence and a closed loop. On evidence, a held-out human audit of diagnostic quality and comparisons against external baselines remain for future work. On closing the loop, candidate training and A/B testing remain manual behind the engineer checkpoint. Automating them so discovered failure categories feed back into model improvements is the natural extension of our closed-loop validation. In parallel, we are scaling agent orchestration and cross-run memory so that each run reuses what earlier runs discovered and avoids re-diagnosing known failures.

We further plan to give the judges richer context, including agentic tools, knowledge-graph representations, and global and local engagement signals, and to expand from a single judge to a committee that reduces the biases of any one LLM. Finally, we aim to develop adaptive judges that condition on persona, with success criteria tuned per ranker and retriever.

## 7. Conclusion

Aggregate metrics tell recommender teams whether a model is improving on average, not what is failing or for whom. That knowledge is a gap in the existing process that recent AI agentic technology can help close. AURA closes that gap. It reads thousands to millions of production sessions and returns a ranked taxonomy of the ways the system fails real users. Then it reads the recommender's own code and says what to change, and why.

Run on production data from both platforms, AURA produces platform-specific taxonomies, runs at low end-to-end cost, and surfaces findings engineers can check against raw evidence. Two lessons stood out: locking the category vocabulary after initial discovery beat open-ended classification, and engineer trust came from structured evidence, not LLM authority. Offline closed-loop validation adds a cheap check that rejects fixes before they reach a live experiment. In its first application, the most useful thing AURA did was refuse to flatter us: the two genre-targeted fixes an engineer would have

tried first did not resolve the diagnosed failure, and learning that took an offline screen rather than an A/B test.

## References


[1] A. Gunawardana, G. Shani, Evaluating recommender systems, in: F. Ricci, L. Rokach, B. Shapira (Eds.), Recommender Systems Handbook, 2 ed., Springer, 2015, pp. 265–308. doi:10.1007/978-1-4899-7637-6_8.

[2] E. Zangerle, C. Bauer, Evaluating recommender systems: Survey and framework, ACM Computing Surveys 55 (2022) Article 170. doi:10.1145/3556536.

[3] S. M. McNee, J. Riedl, J. A. Konstan, Being accurate is not enough: How accuracy metrics have hurt recommender systems, in: CHI Extended Abstracts, 2006. doi:10.1145/1125451.1125659.

[4] B. Edizel, T. Sweetser, A. Chandrashekar, K. Ahmadi, P. Das, Towards understanding the gaps of offline and online evaluation metrics: Impact of series vs. movie recommendations, in: Proceedings of the 18th ACM Conference on Recommender Systems (RecSys '24), ACM, 2024. doi:10.1145/3640457.3688056.

[5] A. H. Jadidinejad, C. Macdonald, I. Ounis, The simpson's paradox in the offline evaluation of recommendation systems, ACM Transactions on Information Systems 40 (2021) Article 4. doi:10.1145/3458509.

[6] F. Prost, B. Packer, J. Chen, L. Wei, P. Kremp, N. Blumm, S. Wang, T. Doshi, T. Osadebe, L. Heldt, E. H. Chi, A. Beutel, Simpson's paradox in recommender fairness: Reconciling differences between per-user and aggregated evaluation, in: arXiv preprint arXiv:2210.07755, 2022. arXiv:2210.07755.

[7] M. D. Ekstrand, M. Tian, I. M. Azpiazu, J. D. Ekstrand, J. Anztor, A. Kishi, P. M. Soledad, All the cool kids, how do they fit in?: Popularity and demographic biases in recommender evaluation and effectiveness, in: Conference on Fairness, Accountability, and Transparency (FAT*), volume 81 of *Proceedings of Machine Learning Research*, 2018.

[8] R. Mehrotra, J. McInerney, H. Bouchard, M. Lalmas, F. Diaz, Towards a fair marketplace: Counterfactual evaluation of the trade-off between relevance, fairness & satisfaction in recommendation systems, in: Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM), 2018. doi:10.1145/3269206.3272027.

[9] M. Mitchell, S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji, T. Gebru, Model cards for model reporting, in: Conference on Fairness, Accountability, and Transparency (FAccT), 2019.

[10] G. Shani, A. Gunawardana, Evaluating recommendation systems, in: F. Ricci, L. Rokach, B. Shapira, P. B. Kantor (Eds.), Recommender Systems Handbook, 1 ed., Springer, 2011, pp. 257–297. doi:10.1007/978-0-387-85820-3_8.

[11] S. Yao, B. Huang, Beyond parity: Fairness objectives for collaborative filtering, in: Advances in Neural Information Processing Systems (NeurIPS), 2017.

[12] R. Kohavi, D. Tang, Y. Xu, Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing, Cambridge University Press, 2020. doi:10.1017/9781108653985.

[13] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, I. Stoica, Judging llm-as-a-judge with mt-bench and chatbot arena, in: Advances in Neural Information Processing Systems (NeurIPS), 2023. arXiv:2306.05685.

[14] T. Zhang, K. Yao, L. Ma, J. Chen, R. Y. Maragheh, K. Zhao, J. Xu, E. Korpeoglu, S. Kumar, K. Achan, No-human in the loop: Agentic evaluation at scale for recommendation, arXiv preprint arXiv:2511.03051 (2025). arXiv:2511.03051.

[15] Y. Hou, J. Zhang, Z. Lin, H. Lu, R. Xie, J. McAuley, W. X. Zhao, Large language models are zero-shot rankers for recommender systems, in: Proceedings of the European Conference on Information Retrieval (ECIR), 2024. arXiv:2305.08845.

[16] P. Wang, L. Li, L. Chen, Z. Cai, D. Zhu, B. Lin, Y. Cao, Q. Liu, T. Liu, Z. Sui, Large language models are not fair evaluators, arXiv preprint arXiv:2305.17926 (2023). arXiv:2305.17926.

[17] M. Wu, A. F. Aji, Style over substance: Evaluation biases for large language models (2023). arXiv:2307.03025.
[18] T. Hosking, P. Blunsom, M. Bartolo, Human feedback is not gold standard, in: International Conference on Learning Representations (ICLR), 2024. arXiv:2309.16349.
[19] T. Vu, K. Krishna, S. Alzubi, C. Tar, M. Faruqui, Y.-H. Sung, Foundational autoraters: Taming large language models for better automatic evaluation (2024). arXiv:2407.10817.
[20] M. Krumdick, C. Lovering, V. Reddy, S. Ebner, C. Tanner, No free labels: Limitations of llm-as-a-judge without human grounding, arXiv preprint arXiv:2503.05061 (2025). arXiv:2503.05061.
[21] Y. Chung, T. Kraska, N. Polyzotis, K. H. Tae, S. E. Whang, Slice finder: Automated data slicing for model validation, in: Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE), IEEE, 2019, pp. 1550–1553. doi:10.1109/ICDE.2019.00088.
[22] M. T. Ribeiro, T. Wu, C. Guestrin, S. Singh, Beyond accuracy: Behavioral testing of nlp models with checklist, in: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL), 2020. doi:10.18653/v1/2020.acl-main.442.
[23] P. J. Chia, J. Tagliabue, F. Bianchi, C. He, B. Ko, Beyond NDCG: Behavioral testing of recommender systems with RecList, in: Companion Proceedings of the Web Conference (WWW '22 Companion), ACM, 2022. doi:10.1145/3487553.3524215. arXiv:2111.09963.
[24] A. Zhang, Y. Chen, L. Sheng, X. Wang, T.-S. Chua, On generative agents in recommendation, in: Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '24), ACM, 2024. arXiv:2310.10108.
[25] L. Wang, J. Zhang, H. Yang, Z. Chen, J. Tang, Z. Zhang, X. Chen, Y. Lin, R. Song, W. X. Zhao, J. Xu, Z. Dou, J. Wang, J.-R. Wen, User behavior simulation with large language model based agents, arXiv preprint arXiv:2306.02552 (2023). arXiv:2306.02552.
[26] X. Wang, X. Tang, W. X. Zhao, J. Wang, J.-R. Wen, Rethinking the evaluation for conversational recommendation in the era of large language models, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2023. arXiv:2305.13112.
[27] J. Lian, Y. Lei, X. Huang, J. Yao, W. Xu, X. Xie, RecAI: Leveraging large language models for next-generation recommender systems, in: Companion Proceedings of the ACM Web Conference (WWW '24 Companion), ACM, 2024. doi:10.1145/3589335.3651242. arXiv:2403.06465.
[28] H. Wang, Y. Wu, D. Chang, L. Wei, L. Heldt, Self-evolving recommendation system: End-to-end autonomous model optimization with llm agents, arXiv preprint arXiv:2602.10226 (2026). arXiv:2602.10226.
[29] S. Kim, S. Park, H. Kang, W. Kim, J. Seo, Y. In, K. Yoon, C. Park, Self-evolverec: Self-evolving recommender systems with llm-based directional feedback, arXiv preprint arXiv:2602.12612 (2026). arXiv:2602.12612.
[30] Q. Huang, J. Vora, P. Liang, J. Leskovec, MLAgentBench: Evaluating language agents on machine learning experimentation, arXiv preprint arXiv:2310.03302 (2023). arXiv:2310.03302.
[31] J. S. Chan, N. Chowdhury, O. Jaffe, J. Aung, D. Sherburn, E. Mays, G. Starace, K. Liu, L. Maksin, T. Patwardhan, L. Weng, A. Mądry, MLE-Bench: Evaluating machine learning agents on machine learning engineering, arXiv preprint arXiv:2410.07095 (2024). arXiv:2410.07095.
[32] Z. Jiang, D. Schmidt, D. Srikanth, D. Xu, I. Kaplan, D. Jacenko, Y. Wu, AIDE: AI-driven exploration in the space of code, arXiv preprint arXiv:2502.13138 (2025). arXiv:2502.13138.
[33] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, P. Clark, Self-refine: Iterative refinement with self-feedback, in: Advances in Neural Information Processing Systems (NeurIPS), 2023. arXiv:2303.17651.
[34] P. Verga, S. Hofstatter, S. Althammer, Y. Su, A. Piktus, A. Arkhangorodsky, M. Xu, N. White, P. Lewis, Replacing judges with juries: Evaluating LLM generations with a panel of diverse models, arXiv preprint arXiv:2404.18796 (2024). arXiv:2404.18796.
[35] A. Panickssery, S. R. Bowman, S. Feng, LLM evaluators recognize and favor their own generations, arXiv preprint arXiv:2404.13076 (2024). arXiv:2404.13076.