Metix AI

ENGINEERING BLOG

Agent Evaluation, Done Right

Notes from building Mira, our recruiting agent

Dr. Zhilin Wang

Co-founder & CTO, Metix AI

July 2026

People tend to credit the architecture, or the raw capability of the model. In my experience, what actually decides whether an agent system runs reliably, day after day, is the evaluation system behind it. Unlike traditional software, an agent is non-deterministic by nature; bug-free code alone cannot hold it up. In this post I want to share what building Mira, our recruiting agent, taught me about making an agent system solid through rigorous evaluation.

01

Why you need an evaluation system

Traditional software quality rests on an assumption so familiar we rarely notice it: the system is deterministic. The same input always produces the same output, so once a unit test passes, that path is correct forever. Agents break this. Run the same résumé and the same job description through twice and you may get different reasoning, sometimes a different conclusion. “The code has no bugs” and “the system behaves correctly” become two separate claims.

Agents also fail quietly. When traditional software breaks, it throws an exception and leaves a stack trace. An agent hands you a perfectly plausible answer that happens to be wrong, with a straight face. Early on, Mira ran into a whole class of these: a candidate’s résumé said they had attended a training course on some framework, and the generated summary said they had hands-on experience with it. Nothing errored. The formatting was immaculate. Unless a recruiter checked the summary against the original line by line, they would never notice. You cannot catch this kind of failure by eyeballing outputs.

Nor can you predict the blast radius of a change. Tweak one line of the system prompt, bump a model version, adjust a tool’s return format, and something entirely unrelated shifts. We once added an instruction to fix the parsing of English résumés and quietly degraded timeline extraction for Chinese ones. Without an evaluation system, teams verify changes by running a few familiar cases and seeing how they feel: the vibe check. That works while the system is small. Once the number of scenarios, tools, and prompts grows, the coverage of a vibe check rounds to zero.

Agents also make decisions in sequence, and sequences amplify error. At 95% accuracy per step, a ten-step chain lands around 60% end to end. And when it

fails, you need to know where it went off the rails. Bad parsing? A retrieval miss? A weak ranking prompt? End-to-end results alone cannot answer that; only process-level metrics can.

Then there is the plain fact that everything you depend on keeps moving. Models get upgraded, repriced, deprecated; swapping models is a matter of when, not if. Production inputs are far wilder than anything you imagined during development: résumés as scanned images, designer PDFs set in two columns, careers described in a mix of Chinese and English, projects inflated beyond recognition. With an evaluation system, a model swap means running the suite and reading the numbers. Without one, it means shipping and praying.

Finally, the business. In recruiting, mistakes carry real cost: a strong candidate overlooked, a hiring decision made on bad grounds, bias creeping into screening. When the business asks "can this version ship?", the answer needs to be "key metrics pass on the evaluation suite, no regressions", not "we tried a few cases and it felt fine".

02

Test like a researcher, not a software engineer

Once you are convinced you need evaluation, the next question is how to build it. Here is where I have watched many engineering teams take a wrong turn: an evaluation system should not be modeled on traditional software testing. It is much closer to the experimental method of machine learning research.

Unit testing has an assertion-based worldview. Given this input, the output must equal that expectation; one failure is a bug; the suite passes 100% before anything merges. That worldview simply does not hold for agents. Agent output is stochastic, you will never reach 100%, and a single failing run of a single case may mean nothing. Force the unit-test model onto an agent and you end up in one of two places: assertions so loose the tests test nothing, or a team drowning in noise-driven "failures" until nobody looks anymore.

The research worldview is different: treat the agent as a model. You hold a test set (your benchmark, your held-out data), a handful of metrics (accuracy, pass rate, ranking quality), and every prompt change or model swap is an experiment to be compared against a baseline. What you watch is how the metrics move across the whole distribution, not whether one case passed. A single failure is a data point, not an incident; the real signal is a pass rate dropping from 92% to 85%, or one category of cases degrading systematically. Fixing changes accord-

ingly. Instead of patching case by case, you do error analysis: lay the failures out, find what they share, name the pattern, fix the pattern.

This mindset carries some discipline that research has stress-tested for decades. A test set cannot be used both for tuning and for reporting; that is leakage, and it inflates your numbers, so keep a dev set for iteration and a test set you only touch at milestones. Conclusions need sample size and variance behind them: rerun important cases several times, and treat a two-point difference on a small sample as noise. And every experiment needs a record. Prompt version, model version, dataset version, and results, bound together, or a month later nobody can reconstruct why a change was made. At Mira we eventually managed every prompt change exactly like an experiment, with a baseline, a metric comparison, and a log entry. Once it becomes habit, it feels no different from tuning a model.

Everything in the rest of this post, the datasets, the scoring, the regressions, the canaries, sits on top of this mindset.

03

What exactly are you evaluating?

Before writing any evaluation code, answer one question: what is the unit of evaluation? I split it into three levels.

Component-level evaluation targets a single atomic capability: one résumé parse, one intent classification, one retrieval, one tool call. The input and output boundaries are clean, it is the easiest level to automate, and it is the smallest unit for localizing a problem.

Trajectory-level evaluation targets the agent's full execution path. Did it pick the right tool, pass the right arguments, avoid spinning in circles? When a tool errored, did it try another route or fabricate an answer? This level is unique to agents, and it is the one that teams with a traditional testing background most often miss.

Outcome-level evaluation targets end-to-end task quality and business metrics. "Schedule an interview with this candidate for next week": did the interview actually get scheduled? Of the candidates recommended to recruiters, how many were accepted?

You need all three. Outcome-only, and you know something broke without knowing where. Component-only, and you meet the strange case where every

step is right and the whole is wrong, small deviations compounding along the chain, or modules disagreeing about interface semantics. At Mira, component metrics drive day-to-day iteration, trajectory metrics drive the diagnosis of agent behavior, and outcome metrics drive release decisions and reporting to the business.

04

Building the dataset

The dataset sets the ceiling for the whole evaluation system. However refined the method, a bad dataset yields precisely wrong conclusions.

Do not aim big at the cold start. We began with roughly a hundred hand-built cases, but they covered the trunk of every core scenario: résumés in the common formats, candidate-job matching on typical job descriptions, the standard interview-scheduling flow, each with an explicit expected result or scoring rubric. That was our first golden set. Its value was never the count; it was that "did this change make things worse?" became, for the first time, an answerable question.

What keeps a dataset alive is growth driven by bad cases. Every failure in production gets anonymized and folded into the regression set: never again. A few months in, the set reads as a record of every hole the system has ever fallen into, more honest than any document.

The long tail and the adversarial cases have to be gathered deliberately. Recruiting has a far longer tail than you would guess: scanned résumés, two-column designer layouts, traditional Chinese characters, pure English, bilingual careers. The adversarial side includes inflated project claims and training courses dressed up as work experience. None of this walks into your golden set on its own. Collect it on purpose, and synthesize what you cannot collect.

Two engineering details, finally. Version the dataset, so that every conclusion maps to the exact data it was drawn from. And layer it: a smoke set of a few dozen cases that runs on every commit, and a full set that runs nightly or before release. If evaluation is slow, nobody runs it. If nobody runs it, the system is dead.

05

Scoring: rules, judges, humans

My scoring stack has three tiers: a wide base of rule checks, a middle tier of LLM-as-a-judge, and a thin layer of human review on top.

Never use a model where a rule will do. Rule checks are cheap, stable, and zero-variance. Is the output valid JSON? Are required fields present? Are the tool arguments legal? Did the date parse correctly? Can every fact cited in a summary be traced back to the original résumé? At Mira, plain rule checks alone catch a large share of our regressions, at essentially no cost.

LLM-as-a-judge handles the subjective dimensions rules cannot reach: whether an outreach message reads as professional, whether a candidate summary is faithful. A few things matter here. Make the rubric concrete; do not ask for a score from one to ten, define bands with examples for each, so the judge is classifying rather than feeling. For version comparisons, pairwise beats pointwise: "is A or B better" is far more consistent than absolute scores. And the point most often skipped: the judge itself must be evaluated. Periodically have humans label a sample and measure agreement with the judge; a judge whose agreement rate has never been validated produces numbers that mean nothing. Watch for the known failure modes too, position bias and self-preference, with the usual hedge of swapping A and B and running both orders.

Human evaluation is the most expensive tier, so spend it in exactly two places: calibrating judges, and regular spot checks on high-stakes scenarios. Write a real annotation guideline, use multiple annotators, and measure agreement between them. Low agreement means the guideline itself is ambiguous; fix the guideline before labeling more.

06

The agent-specific layer: trajectory evaluation

Everything so far applies to any LLM application; this part belongs to agents alone. One precondition first: no full tracing, no trajectory evaluation. Every step of every run, model inputs and outputs, tool calls and returns, intermediate decisions, must be recorded and replayable. Build this early. It serves evaluation, debugging, and production monitoring at once, and it is the highest-leverage infrastructure investment you will make.

On trajectories we watch four kinds of signal. Tool-call correctness: when the candidate database should have been queried, was it, and with the right arguments? Efficiency: how many steps, how many tokens, any pointless ping-ponging between two tools; getting the same thing right in six steps is not the same as getting it right in twenty. Recovery: after a timeout or a tool error, does the agent try another route, tell the user honestly, or quietly make something up? That last behavior deserves special hunting in evaluation. And assertions on key intermediate states: some business rules can be written as hard checkpoints directly on the trajectory. In Mira, “deduplicate the candidate before recommending them” is one such assertion. Skip that step and the run fails, no matter how good the final answer looks.

07

After launch: online evaluation

An offline suite, however good, is one sample of the real world. Once you ship, evaluation continues in a different form.

The most honest metrics are implicit user behavior: how heavily recruiters edit what the agent produced, the acceptance rate of its recommendations, the human-takeover rate, mid-flow abandonment. A user may never click thumbs-down, but silently rewriting your entire interview invitation is the loudest one-star review there is. Track these as first-class quality metrics, permanently.

Bring the offline scoring online as well. Sample a slice of production traffic, run the same rule checks and judges over it, and watch the quality distribution for drift. A model provider quietly changing behavior, a new résumé format suddenly spiking: only an online patrol catches these in time. Wire it to alerts, so that a sudden drop can be matched quickly to a specific change or an external shift.

And treat every change as a release. Prompt edits and model swaps go through canary and A/B: small traffic first, confirm the online numbers agree with the offline conclusion, then ramp. Offline evaluation says the new version is probably better. The canary says it is actually better on the real distribution. You need both.

08

Wiring evaluation into development

The most common way an evaluation system fails is that it gets built and then nobody uses it. The fix is to wire it into the development process until it is impossible to route around.

The core is one loop. A bad case surfaces in production; you reproduce and localize it in the trace; anonymize it into the evaluation set; fix it; run the full regression to confirm the fix has not broken something else; canary the release; verify with online metrics. Every bad case walks the whole pipeline, and quality accumulates one case at a time.

Mechanically, evaluation belongs in CI. Any change to prompts, tools, or retrieval logic must pass the smoke set, and a key metric falling past its threshold blocks the merge, exactly the way a failing unit test would. Note that the gate is a metric threshold, not per-case assertions; section two explains why. Bind the prompt version, model version, dataset version, and results into one archived record, so historical conclusions stay reproducible and any release can be rolled back.

Underneath it all, this is a team habit. On the Mira team, “did this change go through evals?” carries the same weight as “did you write tests?”

Optimization without evaluation is not optimization.
It is gambling.

09

The pitfalls

Overfitting the evaluation set. The moment a metric becomes a target it starts to lie; Goodhart’s law is unusually punctual here. A team grinding against one fixed set watches the score climb while the real experience stands still. Rotate in fresh production data on a schedule, and let online metrics be the court of final appeal.

Ignoring randomness. One passing run of one case proves nothing. Rerun the important cases and look at pass rates, and compare versions on samples large

enough to mean something. Whether a team will spend compute to shrink variance tells you how seriously it takes evaluation.

Trusting an uncalibrated judge. This is the sneakiest trap. The judge emits scores with perfect confidence and everything looks scientific, but if agreement with humans was never measured, you have simply outsourced gut feeling to another model.

Metrics detached from the business. Parsing accuracy is superb and recruiters still will not use the product; you are probably measuring the wrong thing. Maybe what shapes the experience is latency, or how readable the recommendation rationale is, or the fact that your one percent of errors lands precisely on the highest-value candidates. Calibrate the metric system against real user complaints, regularly.

10

Closing thoughts

An evaluation system is not a project you finish; it grows up alongside the agent. If I had to lay out a path: build full tracing first, so every run can be replayed. Then a small, precise golden set, so “did it get worse?” has an answer. Then rule checks, to bank all the cheap determinism. Then calibrated judges for the subjective dimensions. After launch, implicit behavioral metrics and an online patrol. Finally, wire it all into CI and the release process, closing the loop from bad case to regression set.

The model decides how good an agent system can be. The evaluation system decides whether, after every single change, you are still standing on ground you have already won. For a production system meant to run and evolve for years, the second thing is usually the harder one, and the more valuable.

ABOUT METIX AI

Metix AI builds Mira, an AI recruiting agent that reads résumés, matches candidates to roles, drafts outreach, and keeps interviews moving, so hiring teams spend their time on people, not pipelines.

TRUSTED BY

ByteDance Alibaba Cloud Tencent

Dyna Robotics Deel

SAN FRANCISCO, CALIFORNIA