AutoJudge: Automatic Generation of LLM-Based Judges from Execution Traces

Alina Zhidkovskaya [1] **Kirill Rapatskikh** [1] **Grigorii Barakhsin** [1] **Jerzy Kamiński** [1] **Alexander Serebryakov** [1] **Roman Dolgoshei** [1] **Julia Belikova** [2] **Maksim Makarenko** [2 1] **Andrey Savchenko** [2 1] **Anna Kalyuzhnaya** [1] **Nikolay Nikitin** [1]

Abstract

Evaluating multi-agent systems from LLM execution traces is challenging due to long-context inputs, diverse failure modes, and the lack of adaptable evaluation pipelines. Existing methods rely on fixed, benchmark-specific judges or manual prompt engineering, limiting generality and scalability.

We introduce **AutoJudge**, a framework that automatically constructs LLM-based evaluation pipelines from execution traces. Given a trace, an evaluation taxonomy, and an output schema, a MetaAgent synthesizes a set of intermediate judges and a final aggregation judge, enabling input-adaptive evaluation without task-specific training. To handle long traces, we propose a dual strategy combining full-trace evaluation with a summary-based mode and controlled step-level access to the original trace.

The system uses a hierarchical model design that separates pipeline construction from large-scale judging, improving both efficiency and robustness.

Experiments across multiple benchmarks show that AutoJudge matches or outperforms prior fixed-judge baselines, while generalizing across heterogeneous evaluation taxonomies and maintaining strong performance under both full-trace and summarized settings. These results suggest that evaluation of complex agent systems can be formulated as a dynamic, inference-time pipeline generation problem.

Our code is available at
https://github.com/ITMO-NSS-team/AutoJudge

1. Introduction

The deployment of multi-agent systems (MAS) for complex problem-solving has accelerated rapidly, yet their evaluation remains challenging. Unlike single-agent systems and standalone LLM applications, MAS exhibit behaviors that emerge from inter-agent coordination, role specialization, and dynamic task decomposition. As a result, end-to-end task metrics alone do not reveal whether a final outcome reflects robust planning and tool use or fragile intermediate decisions simultaniously. In the setting we study here, the practical evaluation problem is broader still: for a new benchmark, trace format, target output schema, or agent architecture, one must decide not only how to score trajectories, but also how to design the judge itself. Our central premise is that this judge should not be fixed in advance: it should be constructed dynamically, adapting its evaluation pipeline to the task, the system structure, and the failure modes exposed by the trace.

Recent trace-based methods like TRAIL (Deshpande et al., 2025) , Who&When (Zhang et al., 2025b), Aegis (Kong et al., 2025), AgentErrorBench (Zhu et al., 2025), PIPA (Kim et al., 2025) and AgentRewardBench (Lù et al., 2025) address specific failure attribution and categorization in agent traces]. Benchmark suites including AgentBench (Liu et al., 2023),, TravelPlanner (Xie et al., 2024),, and τ-bench (Yao et al., 2024) span diverse agent tasks and interaction patterns. Yet these judge designs remain tied to particular benchmarks, architectures, or annotation schemes, requiring manual redesign for new evaluation settings.

We introduce AutoJudge, a framework for automatic generation of LLM-based judges from execution traces. Given a trace (or summary), taxonomy, and output schema, a MetaAgent constructs a dynamic pipeline of intermediate judges plus final aggregation-adapting to the input rather than reusing static templates. The framework supports full-trace evaluation, summary mode with step-level retrieval, and both per-trace and dataset-level generation. Per-trace pipelines outperform dataset-level reuse on Who&When. AutoJudge matches or beats prior judges across attribution, localization, error categorization, and reward benchmarks, motivating dynamic pipeline construction over fixed judges.

2. Related Work

2.1. Benchmark-Driven Evaluation

Standard benchmarks establish agent baselines. SWE-bench (Jimenez et al., 2023) tests real-world software engineering fixes on GitHub issues. GAIA (Mialon et al., 2023) evaluates general AI assistants through tool-using tasks across modalities. MMLU-Pro (Wang et al., 2024) measures multitask knowledge and reasoning under stricter conditions. These deliver end-to-end success rates but mask intermediate failures, coordination issues, and error modes in multi-agent traces.

2.2. Production Evaluation Frameworks

Production frameworks provide LLM-as-judge metrics like faithfulness (hallucination detection), conciseness, correctness, toxicity, and tool call checks, often via configurable rubrics or templates. Arize Phoenix (Arize AI, 2026) offers faithfulness, tool selection/invocation, refusal detection; DeepEval (Confident AI, 2026) supports G-Eval and custom criteria; PydanticAI (Pydantic Services Inc., 2026) enables schema checks and LLM rubrics; Maxim AI (Maxim AI, 2026) and Langfuse (Langfuse, 2026) handle session/trace-level evals for quality and coherence. Users define rubrics for subjective traits (helpfulness, relevance) or use presets, but lack built-in multi-agent system-level metrics like inter-agent attribution or pipeline failure analysis-focusing instead on single-response or component scores. All operate through static, user-specified pipelines.

2.3. Trace Benchmarks and Judge Architectures

Recent work increasingly evaluates multi-agent systems through execution traces rather than only end-to-end task success (Table 4). Importantly, most benchmarks are coupled with their own task-specific judge architectures, evaluation taxonomies, and output protocols.

Who&When studies ReAct-style tool-augmented reasoning pipelines and introduces prompt-based judges for agent-level and step-level failure attribution. TRAIL focuses on web navigation, coding, and environment-interaction agents, using hierarchical prompting to localize planning, environment, and tool-use failures. AgenTracer (Zhang et al., 2025a) further studies causal trajectory decomposition and first-failure localization using retrieval-augmented judging.

Several benchmarks emphasize coordination and multi-agent interaction quality. Aegis evaluates heterogeneous MAS settings spanning QA, mathematical reasoning, coding, and debate systems through rubric-decomposed judges assessing competence, coordination, and validation correctness. AgentFail similarly studies system-level failure attribution using hierarchical evaluators and specialized judge ensembles. AgentErrorBench frames evaluation as taxonomy-driven error classification over memory, planning, reflection, action, and tool/API failures in interactive agents.

Another line of work formulates evaluation as reward-style trajectory assessment. AgentRewardBench evaluates web and workstation agents using scalar or binary reward-style judges over complete trajectories, while PIPA proposes unified evaluation protocols for interactive planning agents across heterogeneous tasks. Pumpkin (Zhidkovskaya et al., 2026) explores ensemble-based judging for heterogeneous multi-agent pipelines, combining 10 independent LLM judges with a final aggregation judge for binary trajectory-level classification.

Despite their diversity, these approaches largely rely on fixed prompts, manually designed rubrics, or benchmark-specific evaluation pipelines. As a result, judges often remain tightly coupled to a particular trace format, taxonomy, or MAS architecture and require substantial redesign when transferred to new settings. Existing systems also struggle with ultra-long execution traces: truncation loses important intermediate reasoning steps, while full-context evaluation becomes prohibitively expensive for large-scale multi-agent coordination traces. More broadly, current approaches lack mechanisms for dynamically adapting evaluation logic to emerging failure modes, heterogeneous interaction structures, or input-specific reasoning requirements.

3. Proposed Approach

AutoJudge removes the need for manual prompt engineering by synthesizing evaluation logic directly from traces.

We introduce a framework that automatically constructs multi-judge evaluation pipelines for LLM-based agent systems. Given an execution trace, a MetaAgent generates a structured set of judges that evaluate different aspects of the trajectory and aggregate their outputs into a final decision. This design enables scalable and flexible evaluation of complex agent behaviors with minimal user input.

3.1. Problem Setup and Design Principles

We consider the problem of evaluating execution traces produced by LLM-based agent systems. Each trace is a sequence of steps representing agent actions, tool calls, and intermediate reasoning.

Formally, the input to our system consists of: **Trace** T (full execution trace or summarized representation), **Taxonomy** $\mathcal{C}$ (evaluation dimensions such as task completion, reasoning quality, or tool usage), and **Output schema** $\mathcal{S}$ (structured specification of the desired output format, e.g., JSON fields and labels).

The goal is to construct an evaluation function

$$f : (T, \mathcal{C}, \mathcal{S}) \to y,$$

where y is a structured evaluation output conforming to schema $\mathcal{S}$.

Instead of manually designing this function, we propose to automatically generate it as a composition of multiple LLM-based judges organized into an evaluation pipeline.

Design Principles. Our framework follows four principles:

- **Minimal user input:** the user specifies only the evaluation taxonomy $\mathcal{C}$ and output schema $\mathcal{S}$, without writing prompts or designing judges.
- **Automatic pipeline construction:** evaluation logic is generated dynamically as a multi-judge pipeline rather than manually specified.
- **Schema-driven evaluation:** the output schema $\mathcal{S}$ defines the structure of the final decision, enabling compatibility across benchmarks without changing evaluation logic.
- **MetaAgent-driven judge construction:** judge prompts and roles are generated dynamically by the MetaAgent after analyzing the input trace.

3.2. System Overview

We propose a framework for automatic construction of multi-judge evaluation pipelines based on execution traces. Given a trace as input, the system dynamically generates a structured pipeline of LLM-based judges that collaboratively produce a final evaluation. The system consists of three core components: (1) an input preparation layer that processes the trace, (2) a MetaAgent that generates the evaluation pipeline, and (3) an execution layer that instantiates and runs the generated judges.

At a high level, the workflow proceeds as follows. First, the input trace is optionally preprocessed. Then, the MetaAgent analyzes the trace and produces a structured specification of a multi-judge pipeline. This pipeline consists of several intermediate judges followed by a final aggregation judge. During execution, intermediate judges evaluate different aspects of the trace and produce structured outputs. These outputs are then consumed by the aggregation judge, which synthesizes them into the final decision.

3.3. Input Preparation Layer (Full Trace vs Summary Mode)

The framework supports two modes of trace processing: full trace mode and summary mode.

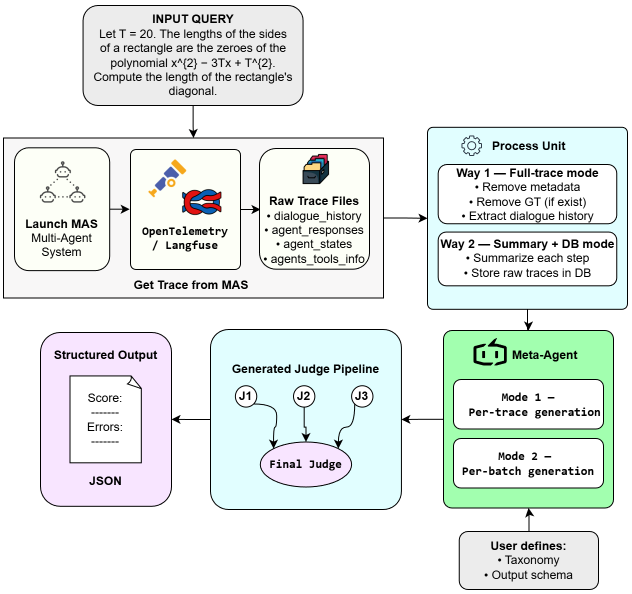

Figure 1. Overview of the AutoJudge pipeline.

Modern execution traces in LLM-based systems can reach extreme lengths, often exceeding millions of tokens (e.g., 2M+ tokens), making it infeasible to process them directly within standard context windows. This creates a fundamental bottleneck for trace-based evaluation and necessitates specialized mechanisms for scalable handling of long-context inputs.

In the full trace mode, the raw execution trace is passed directly to the MetaAgent and all generated judges. While this mode preserves maximal information fidelity, it becomes impractical for long traces due to context length limitations and computational cost.

To address this challenge, we introduce a summary mode, designed specifically for ultra-long traces. In this mode, each step of the trace is compressed into a concise representation, while the full trace is stored in an external database. The generated judges primarily operate on the summarized trace but are equipped with a retrieval tool that enables step-level access to the original data. By specifying a step identifier, a judge can retrieve the corresponding raw content of that step. Importantly, unrestricted access to the full trace is disallowed, enforcing controlled and localized retrieval.

This design allows the system to scale to traces far beyond the context limits of current LLMs while preserving access to fine-grained information when required. In contrast to existing approaches such as Who&When and TRAIL, which assume that the full trace fits within the model context or rely on static truncation strategies, our framework explicitly supports structured compression combined with on-demand retrieval.

3.4. MetaAgent: Automatic Judge Generation

The MetaAgent is responsible for constructing the evaluation pipeline, i.e., generating a structured set of LLM-based judges and defining their interaction pattern. Given input traces, the MetaAgent produces a pipeline consisting of multiple intermediate judges followed by a final aggregation judge, which outputs the final evaluation.

Pipeline Structure. The generated pipeline includes: (1) **intermediate judges**, each responsible for evaluating a specific aspect of the trace, and (2) an **aggregation judge**, which consumes intermediate outputs and produces the final decision. This design decomposes evaluation into complementary perspectives finalized through a dedicated reasoning step.

Operating Modes. We consider two modes of MetaAgent operation:

- **Per-trace pipeline generation (default):** the MetaAgent generates a separate evaluation pipeline for each trace using either the full trace or its summarized representation, allowing both judges and aggregation logic to adapt dynamically.
- **Dataset-level pipeline generation (budget mode):** the MetaAgent generates a single reusable pipeline conditioned on a small random subset of traces (three in our implementation) to capture coarse-grained dataset characteristics. This significantly reduces computational cost but may decrease evaluation quality.

A schematic comparison is shown in Figure 2.

Prompting Interface. The MetaAgent operates using a fixed prompt template with two user-defined components: (1) **taxonomy** — evaluation dimensions such as correctness, reasoning quality, or tool usage; and (2) **output schema** — the desired structure and format of the final output.

These components allow users to control evaluation criteria and output format without adjusting to the benchmark. The MetaAgent uses them to generate judges aligned with the desired evaluation objectives.

An example of a full pipeline is illustrated in Figure 1.

3.5. Fixed Prompting and Non-Trainable Evaluation Logic

AutoJudge implements evaluation through fixed prompts rather than benchmark-specific fine-tuning or trainable reward models. The prompting scheme separates three components: a general evaluation instruction, few-shot diagnostic examples, and a benchmark-specific output schema. This enables a stable evaluation policy across tasks while adapting only the required output format.

The general instruction defines universal principles for evaluating multi-agent traces, including task completion, reasoning quality, tool usage, and environment or API failures. Few-shot examples demonstrate how these criteria should be applied in practice using reusable diagnostic categories rather than benchmark-specific answer patterns, allowing the same prompting strategy to generalize across heterogeneous benchmarks and MAS architectures without retraining or manual redesign.

Benchmark adaptation is handled through the output schema, which specifies the required structure, labels, and metadata of the final prediction while leaving the underlying evaluation policy unchanged. As a result, AutoJudge avoids supervised training and benchmark-specific classifiers, relying instead on the reasoning capabilities of the underlying LLM together with a structured evaluation protocol. This design improves portability and transparency, since adapting to new benchmarks requires only modifying the output schema while the evaluation logic remains directly inspectable through natural-language instructions and examples.

3.6. Aggregation and Final Decision

The final evaluation is produced by an aggregation judge that jointly reasons over three sources of information: (1) the execution trace (either full or summarized with retrieval access), (2) the outputs of all intermediate judges, and (3) the user-defined output schema. Unlike intermediate judges, which focus on localized aspects of the trajectory, the aggregation judge performs global synthesis over all available signals.

The aggregation judge does not operate as a simple voting mechanism over intermediate outputs. Instead, it performs schema-conditioned reasoning that integrates structured judgments with evidence from the original trace. This enables the model to resolve inconsistencies between intermediate judges by grounding its decisions in the underlying execution trajectory.

Formally, the aggregation function takes as input the trace T, the set of intermediate judge outputs $\{o_i\}_{i=1}^{N}$, and the output schema $\mathcal{S}$, and produces a final structured output y such that y conforms to $\mathcal{S}$. The schema defines both the format and semantic organization of the output, while the reasoning process remains grounded in both the trace and intermediate evaluations.

This design separates local evidence extraction from global decision making, while still allowing the final judge to revisit the raw trajectory when necessary. As a result, the

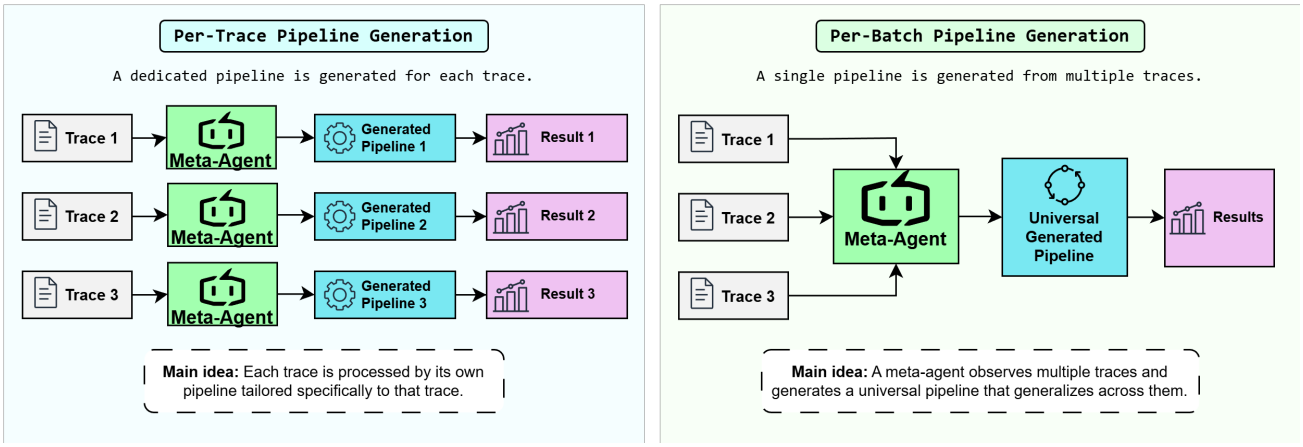

Figure 2. Comparison of per-trace and per-batch pipeline generation.

aggregation stage serves as a unifying reasoning component that ensures consistency between fine-grained evaluations and the final structured decision.

3.7. Implementation

AutoJudge is implemented as a modular LLM-based evaluation system built on PydanticAI, which provides structured orchestration and strict schema validation for all LLM outputs. Every LLM component in the system is executed as a schema-constrained call with automatic retries on format violations, ensuring robust and consistent structured outputs at inference time.

We adopt a hierarchical model allocation strategy. The **MetaAgent** is instantiated using Gemini 3 Flash Preview, selected for its stronger reasoning capabilities required for dynamic judge pipeline synthesis. All evaluation judges, including intermediate and aggregation components, are powered by Gemini 2.5 Flash, enabling efficient large-scale inference under constrained cost.

For long execution traces, we introduce a preprocessing **summarization module**, implemented using Gemini 2.5 Flash, which compresses raw traces into structured summaries before evaluation. This module is strictly decoupled from the judging process and is not part of the evaluation logic. Full traces are stored in a **PostgreSQL** database, enabling controlled step-level retrieval during evaluation via a constrained API. Judges operating in summary mode can query individual trace steps, while unrestricted access to the full trace is explicitly disallowed to enforce localized reasoning.

The system is executed as a dynamic runtime graph: the MetaAgent first generates a structured multi-judge pipeline, which is then instantiated and executed via PydanticAI. All judges operate as stateless LLM calls conditioned on the trace (or its summary), evaluation taxonomy, and output schema. Intermediate outputs are validated and passed to a final aggregation judge, which produces the structured evaluation result.

4. Experimental Studies

We evaluate AutoJudge across multiple benchmarks of LLM-based agent execution traces, each defined by its own evaluation taxonomy and scoring. A key goal of our experiments is to demonstrate that AutoJudge can automatically adapt to different evaluation criteria without task-specific prompt engineering or retraining.

In contrast to prior approaches that rely on fixed, benchmark-specific judges, our framework dynamically constructs evaluation pipelines conditioned on the input trace and the provided taxonomy. This allows us to directly compare performance across datasets while respecting their native definitions of correctness and error categories.

Our experimental study is structured around three questions: (1) how different design choices affect evaluation quality (analyzed on the Who&When dataset), (2) how well AutoJudge generalizes across datasets with different taxonomies and metrics, and (3) how robust the framework is to variations in model configuration and sampling.

4.1. Design Space Evaluation on W&W Dataset

4.1.1. Judge Pool Generation Strategy

We compare two strategies for constructing evaluation pipelines: (1) per-trace judge generation and (2) dataset-level judge pool generation.

Per-trace generation. In this setting, the MetaAgent generates a dedicated evaluation pipeline for each input trace. The pipeline is conditioned on the trace (in summary mode) and adapts both the set of judges and the aggregation logic

to the specific example.

Dataset-level pool. In this setting, a single judge pool is generated once using a small random subset of traces (three in our setup) and reused across the entire dataset. This significantly reduces computational cost, but removes per-trace adaptation.

Comparison. As shown in Table 6, per-trace generation consistently outperforms the dataset-level approach across both Who&When subsets. The performance gap is especially pronounced on the algorithmic subset, where agent behaviors are more diverse and less structured, making per-trace adaptation more important.

The final prompting configuration used for all experiments is selected based on a separate ablation over judge ensembles, optimizing for agent-level accuracy (see Appendix B).

Overall, these results highlight a clear trade-off: dataset-level pool generation is more efficient, but leads to a noticeable drop in evaluation quality, while per-trace generation provides more accurate and reliable assessments at a higher computational cost.

4.1.2. Selected Configuration

Based on preliminary experiments, we select two configurations of our framework: (1) **AutoJudge** (per-trace pipeline generation) and (2) **AutoJudge (summ + DB)**, which operates in summary mode with database-backed trace retrieval.

Unless otherwise stated, these configurations are used consistently across all benchmarks.

4.2. Cross-Dataset Evaluation

We evaluate AutoJudge across multiple benchmarks of agent execution traces, each defined by its own error taxonomy and evaluation protocol. All datasets consist of LLM-based agent traces, but differ in annotation schemes and metrics.

We compare three setups: (1) **AutoJudge (full trace)**, a high-cost configuration that operates on the complete trace, (2) **AutoJudge (summ + DB)**, a more efficient variant that uses summarized traces with database-backed access to the full trace, and (3) **PaperJudge**, i.e., judge implementations reported in prior work.

Results. Across benchmarks, both AutoJudge variants are competitive with or outperform judges from prior work, including the PaperJudge baselines whose underlying models are detailed in Table 5 (see Appendix C). As shown in Table 1, the full-trace configuration generally achieves the strongest performance, particularly on datasets requiring detailed reasoning over the entire trajectory (e.g., WW Algo from Who&When, Pumpkin, and Aegis).

At the same time, the summary-based variant (summ + DB) remains competitive and in some cases performs best (e.g., WW Hand), while offering a significantly more efficient evaluation setup.

Analysis. The results highlight a clear trade-off between evaluation quality and cost. Access to the full trace provides the highest accuracy, but the summary-based approach retains most of the performance while reducing context size and computational requirements.

Importantly, both variants generalize across datasets with different taxonomies and metrics, indicating that AutoJudge can adapt evaluation pipelines without dataset-specific tuning. At the same time, certain metrics (e.g., some Aegis and AgentErrorBench settings) remain challenging, where specialized judges from prior work still have an advantage.

4.3. Comparison with Stronger Judge Backbones

We further compare AutoJudge against the strongest reported configuration of the Who&When (WW) judge. Since WW achieves its best performance when using GPT-4o as the backbone, we run AutoJudge with the same GPT-4o backbone to ensure a controlled comparison under the strongest available setting for this baseline. As shown in Table 2, AutoJudge substantially outperforms WW on both evaluation levels: 67.24% vs. 55.17% in Agent Accuracy and 17.24% vs. 7.02% in Step Accuracy. These results show that AutoJudge's gains are not simply due to using a stronger LLM backbone. Even when both methods are evaluated with GPT-4o, the best-performing WW configuration remains considerably less accurate, especially at the step level. This suggests that AutoJudge's evaluation procedure provides a more effective framework for MAS evaluation, improving both agent-level judgment and fine-grained assessment of intermediate reasoning steps.

4.4. Choice of Model Configuration

Based on the results in Table 7, we configure AutoJudge with `Gemini 3 Flash Preview` as the MetaAgent and `Gemini 2.5 Flash` as the judge backbone. This combination achieves strong agent-level accuracy (59.19%) with moderate step-level accuracy (12.07%) at a reasonable cost (3.9 USD), while processing all samples successfully (Lost = 0).

Using `Gemini 3 Flash Preview` as the MetaAgent leverages its enhanced reasoning and instruction-following capabilities to generate higher-quality judge pipelines and aggregation logic. At the same time, keeping the judges on `Gemini 2.5 Flash` ensures low inference cost and high throughput, since each trace is evaluated by multiple judges and thus consumes many more tokens than the MetaAgent call. The experiments show that this asymmetric

Table 1. Comparison of AutoJudge variants across multiple benchmarks.

Dataset / Metric	AutoJudge (summ + DB)	AutoJudge (full)	PaperJudge
WW hand (Agent Acc)	**61.49**	59.19	55.17
WW hand (Step Acc)	**17.25**	12.07	7.02
WW algo (Agent Acc)	55.55	**63.23**	54.33
WW algo (Step Acc)	41.23	**43.38**	25.51
TRAIL (GAIA) (F1)	21.35	21.26	**24.16**
TRAIL (GAIA) (Loc Acc)	26.55	**27.68**	17.05
Pumpkin small (F1)	64.52	**66.13**	64.30
Pumpkin large (F1)	64.58	**72.00**	65.70
Aegis (Pair mF1)	7.0	**8.9**	6.99
Aegis (Agent mF1)	51.93	**53.31**	42.02
Aegis (Error mF1)	22.5	**24.17**	23.47
Aegis (Pair MF1)	2.51	2.66	**2.76**
Aegis (Agent MF1)	17.1	**17.67**	16.45
Aegis (Error MF1)	12.69	14.59	**19.85**
AgentErrorBench (S Acc)	8.78	**26.93**	20.33
AgentErrorBench (S+M Acc)	2.06	7.77	**8.00**
AgentRewardBench AxTree (Precision)	**77.92**	72.73	68.5

Table 2. Comparison with stronger judge backbones. Both AutoJudge and Who&When use GPT-4o as the judge backbone.

Judge	Agent Acc.	Step Acc.
AutoJudge	**67.24**	**17.24**
WW judge	55.17	7.02

configuration outperforms settings where both components are shifted to more expensive models (e.g., `GPT-4o` or `Claude Sonnet 4.5`) without a clear improvement in evaluation quality.

We note that for `Qwen 3 32b` and `GPT-4o`, several samples could not be processed due to context length limitations. In this experiment, evaluation is performed in the full trace mode without summarization, leading to input sequences that exceed the model's maximum context size. This highlights the importance of summary-based processing for models with limited context capacity. The chosen configuration with `Gemini 2.5 Flash` as the judge is robust even under full-trace processing, while still allowing efficient execution in summary + DB mode. Therefore, we adopt `Gemini 3 Flash Preview` as the MetaAgent backbone and `Gemini 2.5 Flash` as the default judge model for AutoJudge.

4.5. Robustness and Statistical Analysis

4.5.1. Temperature Stability

We analyze the effect of sampling temperature on both the judge pipeline and the MetaAgent. To ensure reliable conclusions, each configuration was evaluated over three independent runs, resulting in a total of 36 experiments.

Judge temperature. We first fix the MetaAgent temperature to 0.3 and vary the judge temperature. The results are reported in Table 8.

We observe that temperatures in the low range (0.0-0.3) yield comparable agent-level performance, indicating robustness of the evaluation pipeline to small variations in sampling. However, temperature 0.1 provides the best balance between accuracy and stability: it achieves the highest step-level accuracy while maintaining low variance across runs. In contrast, higher temperatures lead to both performance degradation and reduced consistency, particularly for step-level evaluation.

MetaAgent temperature. Next, we fix the judge temperature to 0.1 and vary the MetaAgent temperature. The results are shown in Table 9.

We find that moderate temperatures (around 0.3-0.5) provide the most reliable performance. In particular, temperature 0.3 consistently achieves the best results on both metrics while maintaining low variance. Lower temperatures reduce diversity in generated judge pipelines, while higher temperatures introduce instability and increased variance across runs.

Summary. Overall, the results demonstrate that AutoJudge is robust to temperature variations within a reasonable range. Performance remains stable for low-to-moderate temperatures, while extreme values primarily affect consistency rather than causing catastrophic degradation.

Based on this analysis, we select temperature 0.1 for the judge pipeline and 0.3 for the MetaAgent. This configuration provides a principled trade-off: low temperature for judges ensures stable and consistent evaluation, while moderate temperature for the MetaAgent enables sufficient diversity in pipeline generation without sacrificing reliability.

4.5.2. Sampling Robustness

To verify that the AutoJudge evaluation pipeline produces stable results regardless of the particular data sample, we conducted a bootstrap study on the *algo* subset of the Who&When dataset. The pipeline was configured with Gemini 2.5 Flash as the agent backbone and Gemini 3 Flash Preview as the pool generator. We performed three independent bootstrap runs, each time resampling the evaluation set with replacement and computing both agent-level and step-level accuracy. The results are summarized in Table 10. Agent accuracy remains highly consistent across all three runs, ranging from 68.25% to 69.05% (std = $\pm$ 0.38). Step accuracy exhibits moderately greater variability, spanning from 43.09% to 51.61% (std = $\pm$ 3.49), which is expected given the finer granularity of step-level evaluation. The low variance across bootstrap runs confirms that the scores produced by AutoJudge are robust to sampling fluctuations and can be reliably used for comparing different pipeline configurations.

5. Conclusions

In this work, we introduced **AutoJudge**, a framework for automatic evaluation of LLM-based multi-agent systems that constructs evaluation pipelines dynamically rather than relying on fixed, benchmark-specific judges. The key idea is to shift evaluation from static prompt design to on-the-fly pipeline generation, where a MetaAgent decomposes the evaluation task into multiple specialized judges and an aggregation stage.

We addressed a central challenge in agent evaluation: the extreme length and structural complexity of execution traces. To this end, we proposed a dual input processing strategy that combines full-trace evaluation with a scalable summary + DB tool mode. This design enables AutoJudge to handle traces that exceed standard context limits while preserving access to fine-grained information when needed.

A second core contribution is the use of a fixed, non-trainable prompting strategy. Instead of training reward models or designing benchmark-specific judges, AutoJudge separates (1) a general evaluation policy implemented via prompts and few-shot examples, and (2) two user-controlled components: the taxonomy (evaluation criteria) and the output schema. This design allows the framework to adapt to different benchmarks by aligning with their native definitions of errors and required output formats, without modifying the underlying evaluation logic or requiring retraining.

Our experiments demonstrate that this design leads to strong and robust performance across a wide range of benchmarks. In particular: (1) per-trace pipeline generation consistently outperforms dataset-level reuse, highlighting the importance of input-adaptive evaluation; (2) AutoJudge generalizes across datasets with different taxonomies by directly aligning with their evaluation criteria, achieving competitive or superior results without task-specific tuning; (3) the framework remains stable under different model configurations and sampling settings, and maintains a favorable trade-off between accuracy and computational cost through summary-based processing.

At the same time, our results reveal several limitations. Although AutoJudge performs well across diverse benchmarks, full-trace evaluation remains challenging for extremely long execution traces, where context management without summarization can degrade reasoning over complete trajectories. The summary-based retrieval mechanism improves scalability but may occasionally omit subtle details from the original trace. In addition, highly domain-specific metrics can still benefit from manually designed judges. Finally, despite the introduced optimizations, full-trace evaluation remains computationally expensive for large-scale settings.

Overall, AutoJudge demonstrates that flexible, prompt-driven evaluation pipelines can serve as a viable alternative to static or trainable judge designs. By combining dynamic pipeline generation, scalable input handling, and benchmark-aligned evaluation via taxonomy and output schema, the framework provides a general and practical solution for evaluating complex multi-agent systems.

Future work may explore more advanced retrieval strategies, tighter integration between intermediate judges, and hybrid approaches that combine automatic pipeline generation with lightweight adaptation to domain-specific evaluation criteria.

Acknowledgements

This work was supported by the Ministry of Economic Development of the Russian Federation (IGK 000000C313925P4C0002), agreement No. 139-15-2025-010.

Impact Statement

This work introduces AutoJudge, a framework for scalable evaluation of LLM-based multi-agent systems. It can improve benchmarking and reproducibility in AI research, but may also simplify optimization of capable agents, includ-

ing in potentially risky settings. The framework inherits limitations of LLM-based judgments and summary-based trace processing, so it should be used with care, especially in high-stakes applications.

References

Arize AI. Phoenix. https://phoenix.arize.com/, 2026. Source code available at https://github.com/Arize-ai/phoenix; accessed 2026-05-06.

Confident AI. Deepeval. https://deepeval.com/, 2026. Source code available at https://github.com/confident-ai/deepeval; accessed 2026-05-06.

Deshpande, D., Gangal, V., Mehta, H., Krishnan, J., Kannappan, A., and Qian, R. Trail: Trace reasoning and agentic issue localization. *arXiv preprint arXiv:2505.08638*, 2025.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. Swe-bench: Can language models resolve real-world github issues? In *The twelfth international conference on learning representations*, 2023.

Kim, T., Singh, J., Mehri, S., Acikgoz, E. C., Mukherjee, S., Bozdag, N. B., Shashidhar, S., Tur, G., and Hakkani-Tür, D. Pipa: A unified evaluation protocol for diagnosing interactive planning agents. *arXiv preprint arXiv:2505.01592*, 2025.

Kong, F., Zhang, R., Yin, H., Zhang, G., Zhang, X., Chen, Z., Zhang, Z., Zhang, X., Zhu, S.-C., and Feng, X. Aegis: Automated error generation and attribution for multi-agent systems. *arXiv preprint arXiv:2509.14295*, 2025.

Langfuse. Langfuse. https://langfuse.com/, 2026. Source code available at https://github.com/langfuse/langfuse; accessed 2026-05-06.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Lù, X. H., Kazemnejad, A., Meade, N., Patel, A., Shin, D., Zambrano, A., Stańczak, K., Shaw, P., Pal, C. J., and Reddy, S. Agentrewardbench: Evaluating automatic evaluations of web agent trajectories. *arXiv preprint arXiv:2504.08942*, 2025.

Maxim AI. Maxim ai. https://www.getmaxim.ai/, 2026. Accessed 2026-05-06.

Mialon, G., Fourrier, C., Wolf, T., LeCun, Y., and Scialom, T. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Pydantic Services Inc. Pydantic ai. https://ai.pydantic.dev/, 2026. Source code available at https://github.com/pydantic/pydantic-ai; accessed 2026-05-06.

Wang, Y., Ma, X., Zhang, G., Ni, Y., Chandra, A., Guo, S., Ren, W., Arulraj, A., He, X., Jiang, Z., et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290, 2024.

Xie, J., Zhang, K., Chen, J., Zhu, T., Lou, R., Tian, Y., Xiao, Y., and Su, Y. Travelplanner: A benchmark for real-world planning with language agents. *arXiv preprint arXiv:2402.01622*, 2024.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Zhang, G., Wang, J., Chen, J., Zhou, W., Wang, K., and Yan, S. Agentracer: Who is inducing failure in the llm agentic systems? *arXiv preprint arXiv:2509.03312*, 2025a.

Zhang, S., Yin, M., Zhang, J., Liu, J., Han, Z., Zhang, J., Li, B., Wang, C., Wang, H., Chen, Y., et al. Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems. *arXiv preprint arXiv:2505.00212*, 2025b.

Zhidkovskaya, A., Rapatskikh, K., Kaminski, J., Barakhsin, G., Kalyuzhnaya, A. V., Druzhinin, A., Savchenko, A., Belikova, J., Polev, K., and Nikitin, N. O. Towards design of an automated judge for multi-agent systems. In *Trustworthy Agentic AI Workshop at the AAAI Conference on Artificial Intelligence (AAAI)*, 2026. URL `https://trustagenticai.github.io/AAAI2026/AAAI-Workshop/66.pdf`.

Zhu, K., Liu, Z., Li, B., Tian, M., Yang, Y., Zhang, J., Han, P., Xie, Q., Cui, F., Zhang, W., et al. Where llm agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025.

Appendix

A. Prompts

A.1. MetaAgent prompt

You are an AI judge pool generator specialized in creating evaluation pipelines for multi-agent systems. Your goal is to design a team of specialized judges that detect problems, errors, and quality issues in system execution.

DESIGN PRINCIPLES:
- START SIMPLE: Create the minimum number of judges needed to detect key problems
- Prefer 3-9 specialized judges that cover different error domains
- Add more judges only when:
* Independent problem categories can be assessed in parallel (e.g., API errors vs environment setup)
* Different quality dimensions need separate evaluation (correctness, efficiency, reliability)
* Specific failure modes require dedicated detection logic
- Avoid over-engineering: one comprehensive problem detector > multiple narrow similar judges

YOU CAN FOLLOW NEXT TAXONOMY:
{taxonomy}

EXAMPLES:
{examples}

RULES:
- Ensure all judge names are unique and descriptive (end with _JUDGE)!
- You should always ask judges to use tool (get_content_tool) to recive a context (1 or 2 times per judge)!
- Instructions must include explicit scoring criteria (ideal/fair/poor)
- Each judge returns JSON with justification + score
- Focus judges on detecting specific problem categories: task failures, API errors, setup issues, tool misuse, etc.
- Include TOOL_SELECTION_JUDGE or TOOL_PERFORMANCE_JUDGE when evaluating tool-based systems
- Always include FINAL_AGGREGATOR as the final judge that synthesizes all findings into binary score (poor/ideal) and justification

CRITICAL: TOOLS:
Force the court to use tools! Be sure to specify in the prompt that they should call the tool!!! But, FINAL_AGGREGATOR should not use tools!

ATTENTION CRITICAL: FINAL_AGGREGATOR NAME:
FINAL_AGGREGATOR must be named exactly "FINAL_AGGREGATOR" (case-sensitive)

ATTENTION CRITICAL: FINAL_AGGREGATOR OUTPUT FORMAT:
FINAL_AGGREGATOR must include these exact instructions at the end (it is important that it has the same output format as indicated below):
{judge_output_format}

B. Few-Shot Prompting Ablation

To study the effect of few-shot prompting design, we conducted an ablation over six empirically selected combinations of metric-specific judges. The experiment was performed on 58 long handcrafted traces and evaluated using two complementary metrics: *Agent Accuracy*, which measures whether the final trajectory-level outcome is classified correctly, and *Step Accuracy*, which measures agreement at the level of individual intermediate steps.

Table 3. Ablation study of judge ensembles on handcrafted traces (58 long traces).

Ensemble	Agent Acc. (%)	Step Acc. (%)
MAS_TASK_COMPLETION_JUDGE, MAS_COMPLEXITY_JUDGE, TOOL_PERFORMANCE_JUDGE, MAS_ENVIRONMENT_SETUP_JUDGE, MAS_API_ISSUES_JUDGE, TOOL_SELECTION_JUDGE	**59.19**	3.51
MAS_TASK_COMPLETION_JUDGE, MAS_COMPLEXITY_JUDGE, TOOL_PERFORMANCE_JUDGE, TOOL_SELECTION_JUDGE	56.90	6.90
MAS_TASK_COMPLETION_JUDGE, MAS_COMPLEXITY_JUDGE	56.14	10.53
MAS_COMPLEXITY_JUDGE, MAS_ROLE_COVERAGE_JUDGE	58.62	8.77
MAS_TASK_COMPLETION_JUDGE, MAS_COMPLEXITY_JUDGE, TOOL_PERFORMANCE_JUDGE	58.62	8.62
MAS_TASK_COMPLETION_JUDGE, GUIDELINE_CONSISTENCY_JUDGE	55.17	3.51

The primary objective of this study was to identify a prompting configuration that provides reliable agent-level evaluation. Accordingly, *Agent Accuracy* was treated as the primary selection criterion, as the judge is ultimately used to determine whether an agent successfully completes the task, rather than to provide dense supervision for each intermediate step.

The results are presented in Table 3. The highest Agent Accuracy is achieved by the largest ensemble, consisting of MAS_TASK_COMPLETION_JUDGE, MAS_COMPLEXITY_JUDGE, TOOL_PERFORMANCE_JUDGE, MAS_ENVIRONMENT_SETUP_JUDGE, MAS_API_ISSUES_JUDGE, and TOOL_SELECTION_JUDGE, reaching 59.19%. Although this configuration yields relatively low Step Accuracy, this trade-off is acceptable given the focus on trajectory-level correctness.

These results suggest that broad diagnostic coverage is beneficial for evaluating long agent traces. Failures may arise from multiple sources, including incomplete task execution, excessive interaction complexity, incorrect tool usage, environment setup issues, API failures, or suboptimal tool selection. Smaller ensembles tend to capture only a subset of these failure modes, whereas larger ensembles provide a more comprehensive diagnostic signal, improving agent-level evaluation.

Based on this analysis, we select the six-judge ensemble as the final few-shot prompting configuration. While it is less suitable for fine-grained step-level annotation, it is better aligned with the primary objective of reliable trajectory-level evaluation.

Table 4. Benchmarks for evaluating LLM-based judges: MAS trace sources, judge architectures, and evaluation signals.

Benchmark	MAS architecture (trace source)	Judge architecture	Evaluation objective	N-traces
Who & When	ReAct-style tool-augmented QA agents; single- and multi-agent reasoning pipelines	Single LLM judge; prompt-based rubric; no SFT; optional self-consistency sampling	Agent-level attribution of failure; step-level correctness and error localization	176
TRAIL	Web navigation agents; SWE-style coding agents; environment-interaction agents (browser / API / sandbox)	Single LLM judge; hierarchical prompting; no training; decomposition into sub-criteria	Detection of planning, environment, and tool-use failures in execution traces	148
Aegis	Heterogeneous MAS: QA, math reasoning (GSM8K/MATH), coding (HumanEval), and multi-agent debate systems	Multi-aspect prompt-based LLM judge; rubric decomposition; no SFT	Assessment of (1) agent competence, (2) inter-agent coordination, (3) validation correctness	9533
AgentRewardBench	Web agents (WebArena / VisualWebArena), workstation agents (WorkArena), assistant-style task agents	Single LLM reward-style judge; scalar/binary scoring; prompt-only (no training)	Global task completion judgment (success / partial / failure)	1302
AgentErrorBench	Goal-driven interactive agents (ALFWorld, WebShop, GAIA subsets)	LLM-based error classifier; taxonomy-driven prompting; no SFT	Fine-grained failure categorization (memory, planning, reflection, action, tool/API, environment)	200
AgenTracer	Reasoning and coding agents (MBPP+, KodCode, GSM8K, MATH, structured puzzles)	LLM judge with trajectory-level decomposition and retrieval; prompt-only	Localization of first failure step and causal action attribution	2500
AgentFail	General multi-agent systems (synthetic and real-world deployments)	Hierarchical LLM judge; optional ensemble of specialized prompt-based evaluators	System-level failure detection and categorization (agent / system / environment)	307
Pumpkin	Multi-agent pipeline traces with heterogeneous reasoning and tool-use patterns	Ensemble of 10 independent LLM judges followed by a final aggregator; prompt-only (no SFT)	Binary success/failure classification of full trajectory	328

C. Additional Details

Table 5. Expanded comparison of AutoJudge variants across multiple benchmarks. PaperJudge uses GPT-4o for WW benchmarks and Gemini 2.5 Flash for all others. PaperBest shows the highest score reported in the original paper regardless of the model (WW hand and algo - GPT-4o; TRAIL - Gemini 2.5 pro-preview; Pumpkin - GPT-5; Aegis - Aegis-SFT based on Qwen2.5-14B-Instruct; AgentErrorBench - GPT-4.1; AgentRewardBench - Functional).

Dataset / Metric	AutoJudge (summ + DB)	AutoJudge (full)	PaperJudge	PaperBest
WW hand (Agent Acc)	61.49	59.19	55.17	55.17
WW hand (Step Acc)	17.25	12.07	7.02	7.02
WW algo (Agent Acc)	55.55	63.23	54.33	54.33
WW algo (Step Acc)	41.23	43.38	25.51	25.51
TRAIL (GAIA) (F1)	21.35	21.26	24.16	38.9
TRAIL (GAIA) (Loc Acc)	26.55	27.68	17.05	54.6
Pumpkin small (F1)	64.52	66.13	64.30	74.2
Pumpkin large (F1)	64.58	72.00	65.70	68.9
Aegis (Pair mF1)	7.0	8.9	6.99	16.62
Aegis (Agent mF1)	51.93	53.31	42.02	76.53
Aegis (Error mF1)	22.5	24.17	23.47	27.53
Aegis (Pair MF1)	2.5113	2.66	2.76	9.99
Aegis (Agent MF1)	17.1	17.67	16.45	47.97
Aegis (Error MF1)	12.69	14.59	19.85	27.66
AgentErrorBench (S Acc)	8.78	26.93	20.33	42.0
AgentErrorBench (S+M Acc)	2.06	7.77	8.00	32.0

Table 6. Comparison of judge pool generation strategies in summary mode (Who&When dataset).

	Per-trace	Dataset-level	WW judge
Hand (WW subset)			
Agent Acc. (%)	61.49	57.81	55.17
Step Acc. (%)	17.25	13.50	7.02
Algo (WW subset)			
Agent Acc. (%)	55.55	42.65	54.33
Step Acc. (%)	41.23	26.59	25.51

Table 7. Comparison of MetaAgent and Judge model combinations.

MetaAgent model	Judge model	Agent Acc.	Step Acc.	Lost	Price ($)
Gemini 3 Flash Preview	Gemini 2.5 Flash	59.19	12.07	0	3.90
Gemini 3 Flash Preview	Qwen 3 32b	62.50	3.57	2	2.32
Gemini 3 Flash Preview	Kimi k2	53.45	**17.24**	0	5.57
Gemini 3 Flash Preview	GPT-4o	60.00	7.27	3	20.19
Gemini 3 Flash Preview	Gemini 3 Flash Preview	56.90	8.62	0	5.54
Claude Sonnet 4.5	Gemini 2.5 Flash	51.72	7.02	0	7.71
Gemini 2.5 Flash	Gemini 2.5 Flash	**63.79**	8.62	0	3.62
Qwen 3 32b	Gemini 2.5 Flash	52.38	4.76	8	3.26
Kimi k2	Gemini 2.5 Flash	42.11	7.14	0	3.90
GPT-4o	Gemini 2.5 Flash	53.45	3.45	0	6.75

Table 8. Effect of judge temperature on agent and step accuracy (mean ± std)
(MetaAgent temperature fixed at 0.3).

Judge Temp.	Agent Acc. (%)	Step Acc. (%)
0.0	59.50 ± 6.01	7.00 ± 4.66
0.1	$\mathbf{59.19 \pm 1.62}$	$\mathbf{12.07 \pm 1.44}$
0.3	58.09 ± 4.28	5.81 ± 2.55
0.5	54.66 ± 1.05	3.55 ± 0.08
0.7	57.60 ± 3.67	5.85 ± 1.02
1.0	52.87 ± 1.00	7.60 ± 1.01

Table 9. Effect of MetaAgent temperature on agent and step accuracy (mean ± std)
(Judge temperature fixed at 0.1).

MA Temp.	Agent Acc. (%)	Step Acc. (%)
0.0	55.17 ± 0.00	9.77 ± 4.34
0.1	56.32 ± 1.00	8.71 ± 1.66
0.3	$\mathbf{59.19 \pm 1.62}$	$\mathbf{12.07 \pm 1.44}$
0.5	58.05 ± 0.99	8.10 ± 1.04
0.7	57.47 ± 3.59	3.46 ± 1.71
1.0	57.22 ± 4.41	9.32 ± 2.10

Table 10. Results on the bootstrapped WW algo dataset

Run	Agent Acc. (%)	Step Acc. (%)
1	68,25	47,58
2	68,25	51,61
3	69,05	43,09