# CODE-EXECUTABLE JUDGE: MITIGATING ARITHMETIC HALLUCINATIONS FOR WALLET-AGENT EVALUATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Large-language-model-driven wallet agents often generate answers derived from thousands of billing records, requiring heavy numerical verification. Conventional LLM-as-Judge suffers severe arithmetic hallucinations, leading to inconsistent and unreliable evaluation results. To tackle this problem, we present Code-Executable Judge: instead of letting LLMs directly compute numeric results for judgement, our approach dynamically generates Python verification code and validates answers via program execution. We conduct extensive experiments on wallet-agent benchmarks. Compared with vanilla LLM-as-Judge baselines, our method achieves substantially higher agreement with human annotations. We further systematically analyze failure modes of code-driven judging and discuss its practical limitations.

## 1 INTRODUCTION

Wallet agents increasingly rely on large language models (LLMs) to answer numerical questions over users' financial records. A seemingly simple query such as "How much did I spend on dining last month?" may require inspecting hundreds or thousands of transactions, selecting records within the requested time range, excluding invalid transactions, identifying those belonging to the requested semantic category, and finally aggregating their amounts. Reliably evaluating such answers is therefore itself a data-intensive numerical task.

Conventional LLM-as-a-Judge (Zheng et al., 2023; Liu et al., 2023) is poorly matched to this setting. For numerical claims, the judge implicitly performs an entire verification pipeline:

$$\text{understand} \rightarrow \text{select evidence} \rightarrow \text{extract values} \rightarrow \text{compute} \rightarrow \text{judge}.$$

Failures anywhere in this pipeline may manifest as arithmetic hallucinations, especially when transaction traces range from tens to thousands of entries while the textual representation visible to the judge is long or truncated.

A natural remedy is a calculator. This externalizes arithmetic, but cannot guarantee that the operands are correct. The LLM must still determine which transactions satisfy the query, extract their values, and exclude irrelevant records. A calculator can therefore *precisely compute the wrong evidence*.

We introduce **Code-Executable Judge**. Rather than asking the LLM to directly derive the numerical result, we ask it to synthesize a verification program that executes in a restricted Python environment with access to original structured tool outputs. Evidence retrieval, semantic filtering, aggregation, and comparison are performed explicitly over the underlying data. The LLM primarily reasons about *what should be verified and how to verify it*, while execution performs the actual computation.

Generated verification code can itself be wrong, and an automatic evaluator cannot rely on a human to debug it. We therefore combine executable verification with a bounded **probe–verify–reflect** loop and separate reasoning context from execution evidence. The LLM receives a compact, structure-aware view of large traces, whereas generated programs retain lossless access to complete records: **compact context for reasoning, complete evidence for execution**.

We evaluate the method on **471 real-world wallet-agent interactions** sampled proportionally to production traffic and double-annotated against complete execution traces. Under the same GLM-



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

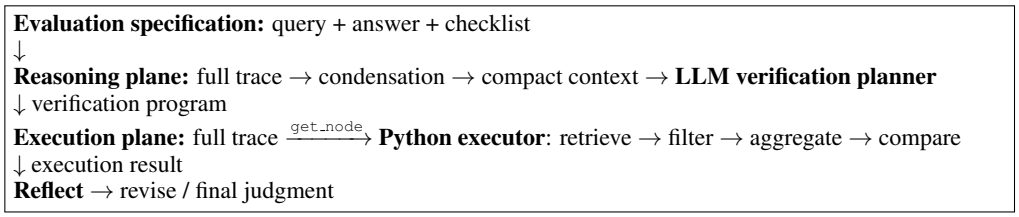


Figure 1: **Code-Executable Judge.** Compact evidence supports reasoning while verification executes against the complete structured trace.

5 backbone, Code-Executable Judge achieves **93.8% accuracy and 86.0% Bad Case F1**, versus 61.0% accuracy and 46.6% F1 for vanilla LLM-as-a-Judge. Our contributions are: (1) numerical judging as **verification-program synthesis over structured evidence**; (2) an autonomous framework combining constrained synthesis, lossless evidence access, evidence condensation, and bounded self-correction; and (3) a real-world human-annotated evaluation showing a 39.4-point absolute F1 improvement under the same judge backbone.

## 2 Related Work

**LLM and agent evaluation.** LLMs are increasingly used as automatic evaluators for open-ended outputs. MT-Bench and Chatbot Arena established LLM-as-a-Judge as a scalable alternative to human preference evaluation while identifying systematic reliability limitations (Zheng et al., 2023). G-Eval improves alignment with human judgments through structured criteria and chain-of-thought reasoning (Liu et al., 2023). Agent-as-a-Judge extends evaluation toward agentic systems (Zhuge et al., 2024). Our setting instead requires recomputing numerical quantities over structured execution evidence.

**Program-aided reasoning.** Toolformer studies language-model use of external tools (Schick et al., 2023), while PAL separates semantic reasoning from computation by generating programs executed by a Python interpreter (Gao et al., 2023). We use programs for *verification rather than answer generation*: code is synthesized independently by the evaluator to verify another agent's natural-language claims and derives operands directly from structured execution evidence.

## 3 Code-Executable Judge

### 3.1 Problem formulation

Each instance is $x = (q, a, C, L)$, where $q$ is the user query, $a$ the wallet-agent answer, $C$ a verification checklist, and $L$ the structured execution trace. We focus on numerical claims whose correctness depends on counting, summation, averaging, or aggregation under semantic constraints. A conventional judge predicts $\hat{y} = J_\theta(q, a, C, \tilde{L})$. We instead use

$$P = G_\theta(q, a, C, \tilde{L}), \qquad R = \mathrm{Exec}(P, L), \qquad \hat{y} = D(R, C, a).$$

The verifier is synthesized independently of numerical reasoning produced by the evaluated agent; the answer is a claim to verify, not a trusted source of operands.

### 3.2 Executable numerical verification

For claim $c$ with claimed value $v_a$, semantic scope $s_c$, and aggregation $f_c$,

$$E_c = \mathrm{Filter}(L; s_c), \quad v_e = \mathrm{Aggregate}(E_c; f_c), \quad r_c = \mathrm{Compare}(v_e, v_a).$$

For example, verifying "1,200 RMB on dining last month" requires independently selecting transactions in the target month, restricting them to valid dining expenses, aggregating their amounts, and only then comparing the result with 1,200.

**Algorithm 1** Autonomous executable verification

$\tilde{L} \leftarrow \mathrm{Condense}(L)$; extract targets $\mathcal{T}$ from $(a, C)$
$H \leftarrow \emptyset,\ V \leftarrow \emptyset$
**for** $t = 1, \ldots, T_{\max}$ **do**
  $\mathcal{U} \leftarrow \mathrm{Unverified}(\mathcal{T}, V)$
  **if** $\mathcal{U} = \emptyset$ **then**
    **break**
  **end if**
  $P_t \leftarrow G_\theta(q, a, \mathcal{U}, \tilde{L}, H)$
  $R_t \leftarrow \mathrm{Exec}(P_t, L)$
  $V \leftarrow \mathrm{Update}(V, R_t)$
  $z_t \leftarrow \mathrm{Reflect}(\mathcal{U}, P_t, R_t)$
  $H \leftarrow H \cup \{P_t, R_t, z_t\}$
  **if** $z_t =$ VERIFIED **then**
    **break**
  **end if**
**end for**
**return** $V$ and final decision

We constrain synthesis in three ways. **Execution-only computation:** supported aggregations must be produced by executed code. **Independent evidence construction:** semantic scope is reconstructed from the trace rather than records listed in the evaluated answer. **No hard-coded operands:** values visible in textual logs cannot be copied into code as operands. Disagreement with the answer is not itself a reason to alter evidence scope. These constraints prevent execution from degenerating into calculator-assisted reasoning over LLM-extracted or answer-fitted values.

### 3.3 COMPACT CONTEXT, COMPLETE EVIDENCE

A billing node may contain hundreds or thousands of transactions. Supplying every record to the LLM is costly and can obscure structure; truncation removes evidence. We therefore expose two views: $\tilde{L} = \mathrm{Condense}(L)$ for reasoning and complete $L$ for execution. Programs retrieve full nodes through `get_node(node_id)` even when only a small sample is visible to the LLM.

For large nodes, the condensed view contains distributions over transaction type, status, category, merchant, payment method, and time, plus representative records. Summaries guide synthesis only; numerical results are recomputed from original records. Multi-node tasks can merge, deduplicate, and persist processed evidence for reuse. This yields **compact context for reasoning; complete evidence for execution**.

### 3.4 AUTONOMOUS PROBE–VERIFY–REFLECT

Execution removes implicit arithmetic but introduces program-synthesis failures. Because evaluation runs without human debugging, recovery must be autonomous and bounded. **Probe** inspects schemas, distributions, and representative records when context is insufficient. **Verify** retrieves complete evidence, implements semantic scope, aggregates, and compares. **Reflect** checks coverage, scope, and execution validity and revises unsuccessful attempts from execution feedback.

The loop reuses intermediate evidence, rejects exact duplicate executions, consolidates probing, and can parallelize independent checks after an initial strategy succeeds. Early development variants frequently required more than ten rounds; the resulting system requires approximately three rounds per instance on average.

Table 1: Results on 471 real-world wallet-agent interactions (%).

| Method | Backbone | Acc. | Prec. | Rec. | F1 |
|---|---|---|---|---|---|
| Vanilla LLM Judge | GLM-5 | 61.0 | 41.5 | 53.1 | 46.6 |
| Calculator Judge | Qwen3-235B | 81.1 | 61.2 | 64.9 | 63.0 |
| **Code-Executable** | **GLM-5** | **93.8** | **85.6** | **86.4** | **86.0** |

# 4 EXPERIMENTS

## 4.1 DATASET AND HUMAN ANNOTATION

We construct the benchmark from real-world wallet-agent traffic using proportional sampling to preserve deployed traffic composition rather than artificially balancing labels. The dataset contains **471 interactions**, each with a query, agent answer, and MCP execution trace; billing outputs range from tens to thousands of transactions.

Each instance is labeled Good when the answer is factually consistent with available evidence or Bad when it contains an error relevant to the evaluation criteria. Annotators can inspect complete MCP traces. An initial pool of 20 annotators performs labeling; we then select the 10 highest-performing annotators and conduct 100% double annotation, so every instance receives two independent judgments. All disagreements receive third-stage adjudication by a senior reviewer. The final benchmark contains 368 Good and 103 Bad Cases.

## 4.2 BASELINES AND METRICS

**Vanilla LLM-as-a-Judge** uses GLM-5. **Calculator Judge** is an earlier system using Qwen3-235B-A22B-Instruct-2507: the LLM extracts operands, constructs arithmetic expressions, and invokes a calculator. **Code-Executable Judge** uses GLM-5 and the complete framework. Because Calculator uses a different backbone, its comparison with the final system is system-level rather than a controlled attribution of the gain to code execution. The primary controlled comparison is Vanilla versus Code-Executable under GLM-5.

Bad Case is the positive class. We report precision, recall, F1, and accuracy, using F1 as primary: the production-preserving benchmark contains 78.1% Good Cases, so an always-Good classifier would obtain 78.1% accuracy while detecting no errors.

## 4.3 MAIN RESULTS

Code-Executable Judge reaches **93.8% accuracy and 86.0% Bad Case F1**. Under the same GLM-5 backbone, the complete executable framework raises F1 from 46.6% to 86.0%, an absolute gain of **39.4 points**. The confusion matrix contains 89 TP, 14 FN, 15 FP, and 353 TN.

The earlier calculator system reaches 63.0% F1. Although its different backbone prevents a component-level causal comparison, it captures the motivation for moving beyond arithmetic-only tooling: a calculator evaluates an expression after the LLM selects operands, whereas executable verification derives operands through programmatic retrieval, semantic filtering, and aggregation.

## 4.4 AUTONOMOUS EXECUTION BEHAVIOR

Early executable variants frequently required more than ten rounds per instance. After structure-aware summaries, consolidated probing, intermediate-state reuse, duplicate prevention, and bounded reflection, the final pipeline requires approximately **three rounds per instance on average**. In development traces, probing accounted for approximately **58–65%** of execution calls before structure-aware summaries and approximately **12%** afterward. These are operational observations from iterative development, not controlled ablations.

## 5 Discussion and Failure Analysis

**Arithmetic correctness is not sufficient.** A calculator can guarantee arithmetic correctness only after the correct operands have been identified. Wallet evaluation additionally requires deciding which records belong to the quantity being verified. Code-Executable Judge therefore externalizes not only arithmetic but evidence retrieval, semantic filtering, aggregation, and comparison. The LLM shifts from directly answering "is this correct?" toward constructing a procedure that can establish whether the claim is correct.

**Executability changes the error surface.** Execution does not eliminate errors; it changes their form. Implicit numerical mistakes become explicit artifacts such as an incorrect filter, field reference, or evidence scope. These artifacts can be inspected and revised through execution feedback, although bounded self-correction cannot guarantee a correct verifier in every corner case.

**Ambiguous evaluation semantics.** The largest residual failure category is semantic ambiguity. For "How much did I spend on kitchen supplies last month?", aggregation is straightforward once relevant records are known, but "kitchen supplies" may lack a deterministic product definition. This separates three components of reliable evaluation:

$$\boxed{\text{evaluation specification}} + \boxed{\text{verification procedure}} + \boxed{\text{evidence}}.$$

Execution can make a well-specified computation deterministic, but cannot make an underspecified evaluation criterion deterministic. The second major residual category is verification-program error, including wrong schema assumptions, filtering conditions, or incomplete handling of unusual records.

## 6 Limitations

Our validation is limited to wallet agents; cross-domain generalization remains untested. Verification assumes required evidence exists in the trace. Generated programs remain fallible, and ambiguous domain semantics require an external evaluation specification. Finally, Calculator and the final system use different backbones, so the same-backbone GLM-5 comparison between Vanilla and Code-Executable Judge is our primary controlled result.

## 7 Conclusion

Code-Executable Judge mitigates arithmetic hallucinations by replacing implicit numerical reasoning with verification programs executed against structured evidence. On 471 real-world interactions, it reaches 86.0% Bad Case F1 versus 46.6% for vanilla LLM-as-a-Judge under the same GLM-5 backbone. More broadly, data-intensive factual evaluation can benefit from asking an LLM to construct an executable verification procedure rather than internally reproduce the computation over the evidence.

## 8 AI Use Statement

Generative AI tools were used to assist with literature retrieval and discovery, manuscript organization, English-language drafting, editing, and polishing. The research idea, methodology, system design and implementation, dataset construction, human annotation protocol, experimental design and execution, and all reported experimental results were developed independently by the authors without the use of generative AI. The authors reviewed and verified all AI-assisted manuscript text and citations and take full responsibility for the final content of the submission.

## Reproducibility Statement

Section 3 specifies the executable-verification formulation and the probe–verify–reflect procedure. The appendix documents the restricted execution interface, evidence-condensation mechanism, verification policy, backbone-selection record, and a distilled research prompt. Section 4 describes

dataset sampling, human annotation, model configurations, metrics, and the primary evaluation results. Product-specific internal policies and user data are not released; the paper separates these domain semantics from the general executable-verification procedure.

## REFERENCES

Lianmin Zheng et al. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. *arXiv preprint arXiv:2306.05685*, 2023.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment. In *EMNLP*, 2023.

Luyu Gao et al. PAL: Program-aided Language Models. In *ICML*, 2023.

Timo Schick et al. Toolformer: Language Models Can Teach Themselves to Use Tools. In *NeurIPS*, 2023.

Mingchen Zhuge et al. Agent-as-a-Judge: Evaluate Agents with Agents. *arXiv preprint arXiv:2410.10934*, 2024.

## A IMPLEMENTATION DETAILS

The restricted Python environment exposes `get_node(node_id)` for complete trace-node retrieval and persistent intermediate storage for reuse across calls. File-system access, network access, process creation, and multi-process/threaded execution are disabled. Our production configuration uses a 5,000-character code limit and a 10-second timeout. Exact duplicate programs are normalized and rejected.

## B EVIDENCE CONDENSATION

For large billing nodes, the reasoning view contains frequency summaries over transaction type, status, category, merchant, payment method, and time, plus representative records. The summary guides synthesis only; the executor recomputes results from complete records.

## C VERIFICATION POLICY

The research-level policy is: no mental arithmetic for supported numerical claims; reconstruct evidence scope independently; never hard-code operands copied from textual logs; and do not alter evidence scope merely to reproduce the claimed answer. Wallet-specific semantic policies are supplied separately as evaluation specification.

## D JUDGE BACKBONE SELECTION

In an intermediate configuration, GLM-5 achieved 69.9% precision, 68.4% recall, and 69.2% F1; Qwen3-235B-A22B-Instruct-2507 achieved 62.2% precision, 69.4% recall, and 65.7% F1. These results were used only for backbone selection and are not directly comparable with Table 1.

## E DISTILLED RESEARCH PROMPT

The production prompt contains general executable-verification instructions and wallet-specific policies. The general prompt requires code execution for supported numerical claims, independent evidence construction, complete evidence access through trace node identifiers, consolidated probing of unfamiliar schemas, explicit executed comparisons, bounded reflection, duplicate-execution avoidance, and output grounded in execution results. Product-specific transaction semantics are omitted because they define domain correctness rather than the executable judging algorithm.