When the Harness Confirms the Hypothesis: Silent Configuration Faults in Multi-Agent LLM Evaluation

Anonymous authors
Paper under double-blind review

Abstract

Reported gains and losses from multi-agent LLM pipelines are only as trustworthy as the harness that produced them. We audit one multi-agent evaluation end to end and find seven faults that produced no error and no crash, but changed conclusions: a chat template that silently discarded every system prompt, leaving two conditions differing only in that prompt byte-identical on 721 of 721 problems; a grammar argument swallowed by a constructor, making the "grammar-constrained" condition a byte-identical duplicate of the unconstrained one; a decoding temperature that never reached the sampler, so a run recorded as greedy sampled at 0.2–0.6 against genuinely greedy baselines; hidden per-role token caps truncating one role on 77–90% of problems; and a grader deviating from the benchmark's standard protocol, worth up to 11.6 points and biased toward one condition. Every fault produced plausible results that *confirmed* the hypothesis under test, which is why each survived days of monitoring. We characterise the faults, quantify each one's effect by re-running the affected cells, and release detectors that find them from outside any framework via a recording proxy. We then report corrected results for the pipeline under study on two 7–8B models and three benchmarks.

1 Introduction

A multi-agent pipeline is a claim about an experiment as much as about a model. When a paper reports that five specialised roles underperform a single call, or that constrained decoding rescues a structured hand-off, the reader is asked to accept that the two conditions differed only in the way the paper says they did. That assumption is rarely tested, and it is not cheap to test after the fact.

We audited one such evaluation end to end — a five-role pipeline compared against single-call and self-refinement baselines on two open-weight 7–8B models and three benchmarks — and found seven configuration faults that had already produced reported results (Table 1). A chat template silently discarded every system prompt for one model family, so two conditions differing only in that prompt were byte-identical on 721 of 721 problems. A grammar argument was swallowed by a constructor, making the "grammar-constrained" condition a byte-for-byte duplicate of the unconstrained one. A decoding temperature never reached the sampler, so a run recorded as greedy was sampled while its baselines were not.

None of these raised an error. What makes them worth a paper is the pattern they share: *each produced results that confirmed the hypothesis under test*. A JSON condition scoring exactly 0.0% reads as the format collapse the study was looking for. Two identical conditions read as "constraints do not help". A truncated checker reads as a checker that destroys good answers. Six of the seven survived several days of active monitoring by the authors, who were watching the runs and reporting the affected cells as findings. The one caught quickly was caught by an automated check.

This is a problem of measurement validity rather than of engineering discipline, and it is not confined to one codebase. The faults are invisible in aggregate statistics — the summary tables looked entirely reasonable — and become visible only when raw outputs are compared across conditions, or when the parameters actually delivered to the model are recorded. Both are mechanical operations that almost no evaluation performs.

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.

Contributions.

- A characterisation of seven silent configuration faults found in a real multi-agent evaluation, each with its mechanism, the plausible finding it produced, and its magnitude measured by re-running the affected cells (§4).
- Detectors for each fault class, including a framework-agnostic recording proxy that verifies from outside whether the parameters a caller declared actually reach the model. We validate them in both directions: they fire on all seven faults and stay silent on two widely-used agent frameworks, which pass every check (§5).
- Corrected results for the audited evaluation (§6), including a measurement of *why* extra calls sometimes pay: a repair loop driven by the benchmark's own tests improves accuracy in all four code cells with zero regressions, while the same loop driven by a check the model writes for itself gains nothing, because its verifier agrees with its own wrong answer on 14 of 19 errors.
- The harness, the detectors, the corrected data, and all superseded data, so that every fault above can be reproduced from the artifacts that contained it.

We report our own errors in detail, including two cases where a detector we wrote produced a false alarm that had to be caught in turn: that history is the evidence for the paper's claim, and reporting only the corrected numbers would remove it.

2 Related work

Multi-agent pipelines and their ablations. Role-structured pipelines and debate procedures are widely reported to improve reasoning (Du et al., 2023; Wu et al., 2023; Hong et al., 2024), and a growing body of work finds the gains smaller or absent under controlled comparison, particularly against strong single-model baselines (Zhang et al., 2025; Xu et al., 2026). Our corrected results in §6 agree with that second line of work. The pipeline audited here was introduced by Prajapati & Mohite (2026). Several figures reported there differ substantially from the corrected values below; the largest gap, a single-call HumanEval baseline of 96.3%, is what a grading fault of the F5 kind would produce. The contribution here is not that agreement but the demonstration that the same comparison, run on the same code, produced materially different conclusions depending on faults invisible in its summary statistics.

Structured output and constrained decoding. Requiring a model to answer in a fixed format costs reasoning accuracy, and tighter constraints cost more (Tam et al., 2024); benchmarks of grammar-constrained generation study the syntactic side of the same trade-off (Geng et al., 2025). Our own data speaks to why this happens in the pipeline we audit. Generation is left to right, so a field the schema places before the answer is available to condition on and a field placed after it is not; in our JSON pipeline the worker states its answer first in $100/100$ outputs and scores 16%, while the same role under a grammar ordering reasoning first does so in $100/100$ and scores 82%. We report that contrast but do not claim the effect as new, and F2 is the reason we can report it at all: before the fix, the constrained condition was never constrained, so no comparison existed to make.

Self-correction and external feedback. Intrinsic self-correction does not reliably improve reasoning and can degrade it (Huang et al., 2024), while feedback from program execution does improve code generation (Chen et al., 2024). A natural reading is that a generator and its own critic share failure modes, which our §6 measures directly in a single harness, holding the repair machinery fixed and varying only the verifier's independence: on held-out development data the external verifier flags every one of the 33 baseline errors and never breaks a correct answer, while on the reported cells the self-written verifier flags 5 of 19 and agrees with the model's own wrong answer on the other 14.

Reproducibility and evaluation validity. Reported LLM results are known to be sensitive to prompt formatting, answer extraction, and harness implementation (Sclar et al., 2024; Biderman et al., 2024), and evaluation harnesses have been shown to disagree with each other on the same benchmark and model. That work concerns choices visible in a harness's configuration. The faults

we characterise are different in kind: the configuration is correct and is silently not applied, so reporting it would not have revealed them.

3 THE AUDITED EVALUATION

The system under audit is a five-role pipeline in which a task is decomposed, planned, solved, checked, and finalised by successive calls to the same underlying model, with each role's parsed output passed forward as context. It is compared against a single call, against gated and ungated two-call self-refinement, and against controls that vary one factor at a time: plain-text versus JSON hand-offs, a two-role subset, a grammar-constrained JSON variant, and a variant replacing the checker's model call with execution against the problem's visible tests.

Models and benchmarks. Two open-weight instruction-tuned models, Qwen2.5-7B-Instruct and Llama-3.1-8B-Instruct, both 4-bit quantised and run locally through `llama.cpp`. Three benchmarks: GSM8K (Cobbe et al., 2021), HumanEval (Chen et al., 2021), and the sanitised split of MBPP (Austin et al., 2021).

Protocol. All reported runs use greedy decoding at a fixed sampler seed. GSM8K has no dataset-native problem identifier, so every cell draws from a committed list of 300 identifiers (a 100-identifier subset for the more expensive pipeline cells); HumanEval and MBPP use their full test splits and native task identifiers. Every comparison is paired on problem identifiers, and significance is reported with an exact McNemar test. Each cell records the model file, the decoding parameters, the git commit that produced it, and whether the source tree was clean at run time.

Scale of the corrected evaluation. After correction the released data comprises 62 cells and 12,478 graded problems with no error rows. All data produced before the fixes is retained separately rather than deleted, with the fault that invalidated it recorded, so each claim in §4 can be checked against the artifacts that contained it.

Scope. This is a study of small, locally-run, quantised models on short benchmark tasks with automatically checkable answers. We make no claim about frontier models, about tasks requiring retrieval or tool use, or about multi-agent architectures in general. The faults in §4 are not scope-limited in the same way — they are properties of harnesses, not of models — but the accuracy numbers in §6 are.

4 SEVEN SILENT FAULTS

We audited the harness described in §3 after noticing that two cells which differed only in their prompt had produced suspiciously similar summary statistics. The audit found seven faults. None raised an exception, none appeared in any log as a warning, and each had already produced results that were reported internally as findings. Table 1 summarises them; Appendix A gives the diagnostic evidence for each.

We group them by what went missing between the experiment's declared configuration and the model's actual input: a parameter that never arrived (F1–F4), a grading procedure that departed from the benchmark's own protocol (F5–F6), or an experiment that ran on the wrong problems (F7).

4.1 PARAMETERS THAT NEVER REACHED THE MODEL

F1: the system prompt was discarded by the chat template. The backend selected a chat format by inspecting the model file, and the format chosen for one model family iterates over messages handling only `user` and `assistant` roles. A `system` message is skipped: no exception, no warning. Every cell for that model ran with no system prompt at all.

The reason this survived is instructive. The affected model's JSON-output condition scored 0.0% on all three benchmarks, which read exactly like the format-collapse effect the study was investigating — a weak model failing to produce parseable JSON. The detection came instead from a coincidence in a summary table: the `direct` and `direct_json` conditions reported *identical* average token

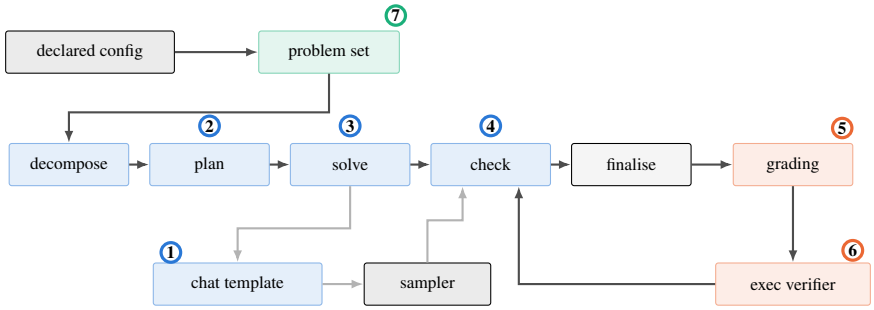

Figure 1: The audited evaluation, with each fault marked where it enters. Each stage is washed with the colour of the fault that enters it, and stages no fault touches are left neutral. Marker colour groups the faults by kind: a parameter that never reached the model (**blue**: **1**–**4**), grading that departed from the benchmark protocol (**orange**: **5**, **6**), and an experiment run on the wrong problems (**green**: **7**). A declared configuration (temperature, grammar, completion caps, problem-identifier list) selects a problem set and parameterises five roles, each of which calls the same model through a shared backend. Fault **7** dropped the identifier list, so a re-run silently evaluated a different problem set. Three faults sit between a role and its call: the role constructor discarded the grammar argument (**2**), the configured temperature (**3**) and the completion cap (**4**), each replaced by a hard-coded default. One sits inside the backend, where the chat template dropped every system message for one model family (**1**). Two sit after generation: the grader departed from the benchmark's reference protocol (**5**), and the execution-based verifier feeding the checker role inherited the same defect (**6**). All seven produced well-formed output and no error.

	Fault	**Presented as**	**Measured effect**
F1	System prompt dropped by chat template	Weak model fails at JSON output	28 cells void; two conditions byte-identical on 721/721 problems (0/721 for unaffected models)
F2	Grammar argument swallowed by role constructor	"Constrained decoding does not help pipelines"	Constrained $\equiv$ unconstrained on 100/100 and 50/50 problems, both models
F3	Configured temperature never reached the sampler	(invisible; metadata recorded 0.0)	Pipelines sampled at 0.2–0.6 against greedy baselines
F4	Hidden per-role completion caps	Checker damages good answers	One role truncated on 77–90% of problems for one model, 0% for the other; reversed a -21.3 vs -4.3 point split
F5	Grader omitted the prompt context required by the benchmark protocol	Pipelines produce broken code	Baseline $+0.0$ to $+0.6$; pipelines $+5.5$ to $+11.6$ points on re-grading
F6	Execution verifier inherited F5	LLM judge is unreliable	19 false "failed" verdicts fed to the pipeline's own oracle control
F7	Re-run config dropped the fixed problem-id list	(caught in <1 hour)	Evaluated 1319 problems instead of the paired 300

Table 1: Seven silent faults found auditing one multi-agent evaluation harness. None raised an error. Each had already produced results consistent with the hypothesis under test, and six survived several days of active monitoring. F7, the only one caught quickly, was caught by an automated check rather than by inspection.

counts. Two conditions differing only in their system prompt cannot consume the same number of tokens by chance. Checking raw outputs showed them byte-identical on 721 of 721 problems, against 0 of 721 for the two model families whose chat formats do deliver system content.

F2: a grammar argument was swallowed by a constructor. The grammar-constrained condition passed a compiled grammar to each role object. Each role class defines its own constructor that forwards a fixed list of keyword arguments to its parent and silently drops the rest; `grammar` was not on that list. Every role therefore generated unconstrained.

The constrained and unconstrained conditions were consequently byte-identical on $100/100$ GSM8K and $50/50$ HumanEval problems for both models, and reported identical accuracies (54.0 vs 54.0, 45.0 vs 45.0, 20.0 vs 20.0). That identity was reported internally as a finding — "constrained decoding does not help pipelines" — rather than investigated. The decisive evidence that the grammar was absent rather than merely ineffective is that the "constrained" outputs include prose with no JSON at all and markdown-fenced text, neither of which a JSON grammar can emit. After the fix the two conditions agree on $0/100$ problems and $499/500$ constrained outputs parse as JSON.

F3: the decoding temperature never reached the sampler. The pipeline configuration object had no temperature field, so each role fell back to a hard-coded default between 0.2 and 0.6. Runs launched with `--temperature 0.0` recorded `"temperature": 0.0` in their metadata while actually sampling, and were compared against single-call baselines that really were greedy. The comparison at the centre of the study therefore carried an undisclosed decoding confound.

No stored artifact can reveal this: every file records the temperature the experiment was *given*, never the one the sampler received. Re-running at the same seed does not help either, because a seeded sampled run reproduces itself byte-for-byte and looks deterministic. The discriminator is seed *sensitivity*: at temperature 0 output must be independent of the sampler seed. We verify this externally (§5).

F4: hidden per-role token caps. Each role class also hard-codes its own completion cap, ignoring the experiment's. The checker role's cap of 768 truncated one model's output on 77–90% of problems while never binding for the other model, which produces shorter text. Measuring the roles uncapped showed why: every role that terminates naturally does so within 660 tokens, and the long outputs are a degeneration loop, not a long verdict.

This fault is the clearest case of a measurement changing its own interpretation. Under the buggy configuration, problems on which the checker was truncated showed -21.3 points of pipeline damage against -4.3 points where it was not, which we initially read as truncation causing the damage. After correcting F3 and F4 the split disappears (-8.8 vs -10.0 points): the pipeline costs about nine points either way, and the earlier gap was an artifact of the sampled, tightly-capped configuration.

4.2 Grading that departed from the benchmark protocol

F5: the grader omitted the prompt context. The reference HumanEval protocol executes *prompt* + *completion* + *test*, so the prompt's imports and helper definitions are always in scope. Our grader executed the candidate alone. Any method that rewrites the function instead of copying the prompt verbatim was therefore penalised: a correct function that omitted the prompt's `from typing import List` failed with `NameError` before a single test ran.

The bias runs in the direction of the paper's hypothesis. Re-grading the stored answers with no regeneration moves the single-call baseline by $+0.0$ to $+0.6$ points and the pipelines by $+5.5$ to $+11.6$ points. It also moves a result that was not otherwise suspect: one grammar-constrained single-call cell rises from 57.9 to 68.9. No answer anywhere changed from passing to failing, confirming that supplying the prompt adds no false passes.

F6: the execution-based verifier inherited the same omission. The same missing context sat in the visible-test feedback path, which is generation rather than grading: it told the pipeline's execution-based verdict that correct code had failed, on 19 problems for one model. That verifier existed precisely to remove the LLM judge's errors from the loop, so a fault making it report false failures undermines the control it was introduced to provide.

4.3 Running the wrong problems

F7: a re-run configuration silently changed the evaluation set. The benchmarks without dataset-native problem identifiers are indexed by position, so a cell run without an explicit identifier list evaluates whatever the sampler selects. A corrected re-run shipped with that list omitted for four cells and began evaluating all 1319 GSM8K problems instead of the fixed 300 — four times the compute, and no longer paired with the baseline it existed to be compared against. It was caught at 445 rows by a row-count check, not by anything in the harness.

Every fault above produced output that was well-formed, plausible, and consistent with the hypothesis under test, which is why six of the seven survived days of active monitoring while F7 — the one an automated check covered — did not. That contrast motivates §5.

5 DETECTING THE FAULTS

Six of the seven faults in §4 were found by hand, days after they had produced results. F7 was found by an automated row-count check, within the hour. That difference is the argument of this section: each fault class has a signature that is mechanically checkable, and checking is cheap enough to run after every cell.

We release three tools. Two operate on a harness's own stored outputs; the third operates from outside any framework and requires no access to its source.

5.1 CHECKS ON STORED RESULTS

Identical-condition detection. Two cells that differ only in a prompt, a grammar, or a decoding parameter must not produce identical model output. Hashing every raw output in a cell and comparing hashes across cells catches F1 and F2 directly, from result files alone, with no knowledge of the backend. On the affected data this check fires on every affected cell; on the corrected data it is silent.

One refinement proved necessary: harnesses often post-process model text before storing it — ours collapses degenerate repetition — so the check must compare what the model *produced*, not what was stored. Recording the pre-processing output alongside the processed one removes a false alarm and makes the rewriting itself measurable: it affected 1 in 500 calls in one cell and up to 75% of one role's calls in another.

Degenerate-value and coverage checks. A cell pinned at exactly 0% or 100% over a non-trivial sample, a cell whose problem set does not match the identifier list it declares, a cell whose rows disagree about which model produced them, and a completion-length distribution piled up at the token cap are all cheap to test and all correspond to a fault above. The problem-set check is what caught F7.

5.2 VERIFYING DECODING PARAMETERS FROM OUTSIDE

No stored artifact distinguishes "ran greedily" from "recorded greedy and sampled" (F3). Rerunning at a fixed seed does not separate them either, since a seeded sampled run reproduces itself exactly.

Seed sensitivity does. At temperature 0 the output must not depend on the sampler seed; above 0 it will. Running the same problems under two seeds and comparing is therefore a direct test of the claim a paper actually makes when it says "greedy decoding". On the corrected pipeline this test passes (output identical across seeds); it fails on the pre-fix configuration.

The check is sound only when the problem set is pinned independently of the seed: our first implementation sampled by count instead, so the two runs compared different problems and reported a false failure on a correctly-greedy system (Appendix A).

5.3 A FRAMEWORK-AGNOSTIC RECORDING PROXY

F1–F4 share one shape: a parameter the caller declared never reached the model. That is observable without reading any framework's source, because essentially every framework can be pointed at an OpenAI-compatible endpoint. We place a recording proxy at that endpoint; it forwards each call upstream unchanged and logs what was actually delivered — system messages, temperature, token cap, response format, and the rendered conversation.

Two runs configured to differ can then be compared on delivered content, and declared values can be compared against delivered ones directly. The proxy is standard-library only, so it can be placed in front of a project sharing none of our dependencies.

	Qwen2.5-7B			Llama-3.1-8B		
Method	**GSM8K**	**HumanEval**	**MBPP**	**GSM8K**	**HumanEval**	**MBPP**
Single call	93.7 (300)	81.1 (164)	75.5 (257)	90.7 (300)	69.5 (164)	70.4 (257)
Pipeline, 2 roles	89.0 (300)	49.4 (164)	–	77.0 (300)	54.9 (164)	–
Pipeline, 5 roles (text)	60.7 (300)	60.4 (164)	67.7 (257)	67.0 (300)	50.6 (164)	47.5 (257)
Pipeline + exec verdict	–	61.6 (164)	68.5 (257)	–	64.0 (164)	64.6 (257)
Pipeline, 5 roles (JSON)	46.0 (100)	48.0 (50)	44.0 (100)	46.0 (100)	18.0 (50)	13.0 (100)
Pipeline, JSON + grammar	72.0 (100)	58.0 (50)	–	78.0 (100)	46.0 (50)	–

Table 2: Accuracy (%) after correcting the seven faults, with the number of problems per cell in parentheses; every comparison in the text is paired on problem identifiers within a cell. Every pipeline configuration loses to a single call on every model and benchmark. Cells marked – were not part of the evaluation ladder for that benchmark.

Validation on unaffected systems. A detector that fires everywhere is useless. We ran two widely-used open-source agent frameworks through the proxy with a local 7B model, in configurations designed to expose the F1–F4 shapes: two agents with deliberately different per-agent temperatures and token caps, different system prompts, and a structured-output schema declaring a reasoning field before an answer field.

Both frameworks pass every check. Per-agent temperatures (0.0 and 0.9) and caps (64 and 192) arrive distinct at the model, each agent's system message is delivered, and the schema's field order is preserved end to end. We report this as a negative result about scope: the faults we characterise are hazards of purpose-built research harnesses — which is what most empirical papers run on — rather than of maintained frameworks. It is also the evidence that the checks are specific: they fire on seven real faults and stay silent on two healthy systems.

One incidental observation from those runs is worth recording, because it affects cost comparisons rather than accuracy: on a single upstream error one framework silently retried nine times. Any token-cost comparison that reads usage from a framework which excludes retries will understate the true cost of whichever method triggers more of them.

6 Corrected results

Every cell affected by F1–F7 was re-run rather than adjusted. This section reports what the corrected evaluation shows. We present it as a case study rather than as the paper's principal claim: the point of §4 is that these numbers are the ones that survive auditing, not that they are surprising.

6.1 The pipeline loses, at four to seventeen times the cost

Table 2 gives accuracy for every configuration after correction. No pipeline configuration beats a single call on any model or benchmark. The gap ranges from 4.7 points (two roles, Qwen, GSM8K) to 57.4 points (five JSON roles, Llama, MBPP), and the pipelines consume 4 to 17 times the tokens.

Two ordering effects hold. Fewer roles lose less: on GSM8K the two-role configuration is the closest pipeline to a single call for Qwen (89.0 against 93.7), and for Llama it is second only to the grammar-constrained JSON pipeline (77.0 and 78.0 against 90.7). And plain-text hand-offs beat JSON hand-offs in all six cells.

Correcting F2 reverses a previously reported result. With grammars actually applied, the grammar-constrained pipeline beats the unconstrained JSON pipeline by 26.0 points on Qwen and 32.0 on Llama, where the pre-fix data showed no difference at all because the two conditions were the same run. We do not attribute this gain to constrained decoding: the grammar also reorders the schema so that a reasoning field precedes the answer field, and in these cells the model writes reasoning first in $100/100$ outputs under the grammar versus answer first in $100/100$ without it. Field order and constraint are confounded in this comparison, and prior work reports large accuracy effects from field order alone (Tam et al., 2024). We therefore report the gain and decline to attribute it.

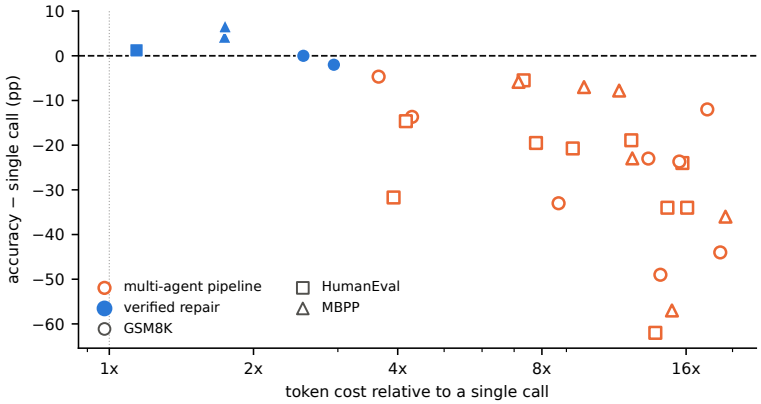

Figure 2: Every configuration against the single call it replaces: token cost on the horizontal axis, accuracy difference on the vertical. The dashed line is the single-call baseline, so a method earns its cost only above it. Multi-agent pipelines (hollow) sit below the line at every cost, in all six model-benchmark combinations. Verified repair (filled) sits on or above it at 1.1–1.7× cost wherever the verifier is the benchmark's own tests, and falls back to the line, or below it, on the one benchmark where the model must verify itself.

Model	Benchmark	Single call	Verified repair	Fixed	Broken	Tokens
Qwen2.5-7B	GSM8K	93.7	93.7	0	0	2.54×
Qwen2.5-7B	HumanEval	81.1	82.3	2	0	1.14×
Qwen2.5-7B	MBPP	75.5	79.8*	11	0	1.74×
Llama-3.1-8B	GSM8K	90.7	88.7	4	10	2.94×
Llama-3.1-8B	HumanEval	69.5	70.7	2	0	1.14×
Llama-3.1-8B	MBPP	70.4	77.0*	17	0	1.74×

Table 3: A single call followed by an *independent* check and one conditional repair, against a single call alone. On code the check is the benchmark's visible example tests; on math it is a program the model writes to recompute its own answer. Fixed/Broken are paired counts; * marks $p < 0.05$ under an exact McNemar test. Prompts were tuned on held-out development problems and frozen before these cells were run.

6.2 Verified repair: gains follow verifier independence

Table 3 compares a single call against one call plus an independent check plus one conditional repair. On code, where the check is the benchmark's own visible example tests, the method beats a single call in all four cells, with *zero* regressions anywhere and significance on both MBPP cells, at 1.14–1.74 times the tokens. On math, where no executable check ships with the benchmark and the model instead writes a program to recompute its own answer, the method achieves nothing on one model and loses two points on the other, at 2.5–2.9 times the tokens.

The mechanism is measurable rather than inferred. On the held-out MBPP development set, the external check flags every one of the 33 problems the single call gets wrong. On the reported GSM8K cells, where the model writes its own check, it flags 5 of Qwen's 19 errors and agrees with the model's own incorrect answer on the other 14. The failure is not that the model cannot verify, but that its verifier shares its errors. This is the correlated-error account of self-correction (Huang et al., 2024) made directly measurable: the same repair machinery, driven by an independent signal, never breaks a correct answer in any of the four code cells, and driven by a correlated one, breaks ten of Llama's on GSM8K.

The prompts were tuned on held-out development problems — 150 GSM8K problems disjoint from the reported 300, and 170 MBPP problems from the train and validation splits — then frozen before any reported cell ran, and the two measurements agree (gate recall 38% on development, 26% on the reported set, both null). That protocol also caught a design error that would otherwise have read as a result: a first version asking for the answer and the verification program in one call scored 1.3

points below a single call, not because its gate was poor ($34/35$ alarms true, 31 fixed, 0 broken) but because the combined prompt cost 22 points on the solve step itself. Splitting the request into two calls restored the baseline. Tuned on the reported problems, that correction would have been fitted to the test set.

7 Discussion and limitations

What generalises and what does not. The seven faults are specific to one harness, and we make no claim about their frequency in the literature; we have not audited anyone else's evaluation. What we do claim is that they were undetectable from the artifacts a paper normally reports — configurations, summary statistics, seeds — and that they were detectable, cheaply, from artifacts almost no paper produces: raw outputs compared across conditions, and a record of what the model was actually sent.

Our framework audit in §5 is deliberately reported as a negative result: two maintained open-source frameworks pass every check. The population at risk is purpose-built research harnesses, which is what most empirical papers run on and which receive the least scrutiny.

Confirmation as the mechanism. The reason five of seven faults survived several days of monitoring is that each produced a plausible result pointing the way the authors expected. We had a hypothesis that structured hand-offs damage small-model pipelines, and every fault produced evidence for it. A fault producing implausibly *good* results would have been caught immediately: the 96.3% single-call HumanEval baseline that Prajapati & Mohite (2026) report for this pipeline was the first number we questioned once the corrected harness disagreed with it. Agreement with a hypothesis should raise scrutiny, not lower it, and no part of a normal evaluation pipeline enforces that.

Limitations of the corrected results. The accuracy numbers in §6 cover two quantised 7–8B models on three benchmarks with short, automatically checkable answers. They say nothing about frontier-scale models, about tasks with retrieval or tool use, or about pipelines with heterogeneous models, where a council's members do not share the same failure modes — a condition under which our verifier-independence result would predict different behaviour. The GBNF comparison confounds constraint with schema field order and we do not attribute it. One grading choice remains unresolved: on GSM8K the answer-extraction rule penalises responses that state an answer and then continue with supporting numbers, which pipelines do more often than single calls; we estimate this at roughly 8 points for one pipeline cell and approximately zero for the baseline, and report strict-protocol numbers throughout rather than silently choosing a more permissive rule.

Cost of the audit. Correcting the faults required re-running 24 pipeline cells, about 63 hours of local inference, plus a 46-hour sweep discarded entirely when F1 invalidated a whole model family. The practical objection to auditing is cost, and the honest accounting is that the audit was cheaper than the work it invalidated.

8 Conclusion

We audited one multi-agent LLM evaluation and found seven configuration faults that produced no error, appeared in no log, and had already generated reported results. Each produced output consistent with the hypothesis under test, which is why six survived days of active monitoring.

They were invisible in the artifacts evaluations normally produce and mechanically detectable in artifacts they normally do not — raw outputs compared across conditions, and a record of the parameters the model actually received. We release detectors for both, tested against faulty and healthy systems alike: every fault here is caught, and two maintained agent frameworks raise nothing.

The corrected evaluation is a case study rather than the claim. Every pipeline configuration loses to a single call at four to seventeen times the cost, while a single call plus one *independent* check beats it on every code cell with no regressions, and the same loop driven by a check the model writes for itself gains nothing. What a pipeline is worth depends on whether anything in it can see the model's errors from outside — the lesson of the audit itself.

REPRODUCIBILITY STATEMENT

The claims in §4 are claims about artifacts, so we release the artifacts. The anonymised supplementary material contains the harness, the detectors (§5), every result file the tables are generated from, and – separately, rather than deleted — all data produced before the faults were found, filed under the fault that invalidated it. Each fault in Table 1 can therefore be reproduced from the data that contained it, not only read about.

Every number in this paper is generated from those files by a script; none is transcribed by hand. Every prompt and grammar is reproduced verbatim in Appendices D and E, emitted by the same code path a real run uses rather than retyped. The fixed problem-identifier lists are committed (Appendix F), extraction and grading rules are stated in Appendix C, and model provenance with file hashes, software versions and hardware is given in Appendix B. Each result file records the commit that produced it and whether the source tree was clean at run time.

Two caveats a replicator should expect. Quantised inference through Metal is not guaranteed bit-identical across hardware, so exact per-problem agreement is not promised; the greedy-decoding property that our conclusions rest on is testable directly with the seed-sensitivity check we release. And the development sets used to tune the method in §6 are released alongside the test data precisely so that the split can be audited rather than trusted.

USE OF LARGE LANGUAGE MODELS

Generative AI (Anthropic Claude models, operating as coding agents with file and shell access) was used extensively and substantively throughout this work. We disclose that use against the ICLR 2027 policy's subtask lists below, and we encourage readers to treat the extent of it as part of what this paper reports.

Subtasks with required disclosure. We used generative AI to *develop conceptual frameworks* (the fault taxonomy in §4 and its grouping by what goes missing between configuration and model input); to *propose and refine hypotheses* (that gains from extra calls track verifier independence, tested in §6); to *design and give feedback on methodology and experiments* (the ablation ladder, the execution-feedback and grammar controls, the held-out development protocol, and the verified-repair method); to *implement methods* (the entire harness, the detectors, and the analysis code); to *clean and reformat datasets* (constructing the held-out development splits from existing benchmark splits and sanitising the released archive); to *support qualitative and thematic analysis* (classifying the faults and diagnosing each one's mechanism from raw outputs); and to *interpret results* (every quantitative claim in §6 was first drafted by an AI agent from the result files).

We did *not* use generative AI to generate synthetic datasets: the development sets are held-out subsets of existing public benchmarks, not generated data. We did not use it to formulate mathematical claims, to supply ingredients for proofs, or to write proofs; this paper contains no theoretical results. Translation and the transcription of research material are not applicable to this work.

Subtasks with recommended disclosure. We additionally used generative AI to create and edit software code; to create the figures; to suggest experimental parameters (including the per-role completion cap, which was set from a measured distribution rather than a guess); to draft parts of this paper, and in practice its full text including this statement; to search for, identify, summarise and analyse existing literature, which materially changed the paper's framing once that survey found prior art for an earlier version of our headline claims; to identify gaps and brainstorm; to format references; to suggest the paper's structure; and to propose its title and keywords. Creation of the released artifacts was likewise AI-assisted.

Verification, and what it caught. We have reviewed the AI-assisted work. Every number reported here is produced from committed result files by scripts rather than transcribed, so a claim and its evidence cannot silently diverge. Bibliographic entries were checked against ACL Anthology or arXiv abstract pages rather than accepted from the model; three were wrong as first generated and were corrected. An originality pass compared this paper against closely related prior work and found no shared seven-word sequence, and it also caught a passage that closely tracked a secondary source's phrasing while attributing two claims — a specific magnitude and a mechanism — to a

paper that did not support either; that passage was rewritten to claim only what its source states. We record these because the policy makes authors responsible for misrepresentations produced by an LLM, and because the fact that AI-assisted writing introduced a misattribution which review then caught is itself relevant evidence about such workflows.

The faults characterised in §4 were also found and fixed by an AI agent auditing its own earlier runs; the same agent had previously reported several of the affected cells as findings without noticing the underlying fault, and two detectors it wrote produced false alarms that had to be caught in turn. That history is the subject matter of this paper and is reported in §4, §5 and Appendix A rather than elided.

We take responsibility for the final content of this work, including all text, claims and artifacts produced with the aid of generative AI.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Stella Biderman, Hailey Schoelkopf, Lintang Sutawika, et al. Lessons from the trenches on reproducible evaluation of language models. *arXiv preprint arXiv:2405.14782*, 2024. arXiv:2405.14782.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2304.05128.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

Saibo Geng, Hudson Cooper, Michał Moskal, et al. JSONSchemaBench: A rigorous benchmark of structured outputs for language models. *arXiv preprint arXiv:2501.10868*, 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2308.00352.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.01798.

Ashish Prajapati and Om Mohite. Two calls beat five agents: Evaluating multi-agent pipelines against self-refinement for local language models. *arXiv preprint arXiv:2607.26922*, 2026.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design, or: How I Learned to Start Worrying About Prompt formatting. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.11324.

Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung-yi Lee, and Yun-Nung Chen. Let me speak freely? a study on the impact of format restrictions on performance of large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 1218–1236, Miami, Florida, USA, 2024. arXiv:2408.02442.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Jiawei Xu, Arief Koesdwiady, Sisong Bei, Yan Han, Baixiang Huang, Dakuo Wang, Yutong Chen, Zheshen Wang, Peihao Wang, Pan Li, and Ying Ding. Rethinking the value of multi-agent workflow: A strong single agent baseline. *arXiv preprint arXiv:2601.12307*, 2026.

Hangfan Zhang, Zhiyao Cui, Jianhao Chen, Xinrun Wang, Qiaosheng Zhang, Zhen Wang, Dinghao Wu, and Shuyue Hu. Stop overvaluing multi-agent debate – we must rethink evaluation and embrace model heterogeneity. *arXiv preprint arXiv:2502.08788*, 2025.

A DIAGNOSTIC EVIDENCE FOR EACH FAULT

For each fault we give the signature that revealed it, the diagnostic that confirmed it, and how the affected results were handled. All superseded data is retained in the released artifact under a directory naming the fault that invalidated it.

F1 — system prompt discarded. Signature: two conditions differing only in their system prompt reported identical mean token counts. Diagnostic: raw outputs byte-identical on $721/721$ problems for the affected model family, against $0/721$ for both unaffected families; independently, probing each chat format with a sentinel system message showed the affected format emitting no trace of it. Handling: all 28 cells for that model family were removed from the analysis. The fix folds the system prompt into the first user turn only for formats that a probe proves to drop it, leaving unaffected formats untouched so their existing results stay comparable.

F2 — grammar argument swallowed. Signature: the constrained and unconstrained conditions reported identical accuracies in three separate cells. Diagnostic: instantiating each role exactly as the pipeline does and inspecting the resulting object showed `grammar` unset on all five roles; the stored "constrained" outputs contain prose and markdown fences, which a JSON grammar cannot emit. Handling: all affected cells re-run. After the fix, the two conditions agree on $0/100$ problems and $499/500$ constrained outputs parse.

F3 — temperature not delivered. Signature: none; no artifact records the delivered value. Diagnostic: instantiating the roles showed each carrying its own default (0.3, 0.5, 0.6, 0.2, 0.4) rather than the configured 0.0; confirmed after the fix by seed-sensitivity testing, which requires identical output across two sampler seeds at temperature 0 and observes it. Handling: every pipeline cell re-run.

F4 — hidden completion caps. Signature: one role's completion lengths piled up exactly at 768. Diagnostic: re-running with the cap raised to 3072 showed the affected role either terminating under 660 tokens or running to whatever cap it is given, i.e. degenerating rather than needing more room; 15 of 20 sampled problems degenerated for the affected model, 0 for the other. Handling: caps made explicit and set from the measured distribution: 1536 sits well above the 99th percentile of natural lengths (736) while still bounding a degeneration loop, and only one observed natural output in 4012 exceeded it (1761 tokens). Context was raised so the final role cannot overflow, and all cells were re-run.

F5 — grader omitted prompt context. Signature: a pipeline cell scoring 0.0% at the worker stage with no truncation. Diagnostic: re-executing every wrongly-graded candidate under the benchmark's reference protocol (prompt, then completion, then test) and reading the failures by hand showed correct functions failing with `NameError` on names defined in the prompt. Handling:

grader corrected; stored candidates re-graded with no regeneration, each row retaining its prior verdict. Re-grading is idempotent: a second pass leaves the files byte-identical.

F6 — execution verifier inherited F5. Signature: verdict text containing `name '...' is not defined` for names supplied by the prompt. Diagnostic: 19 such verdicts for one model before the fix; after it, the remaining occurrences all name helper functions the model genuinely failed to define. Handling: fixed in the same change as F5; affected cells re-run.

F7 — wrong evaluation set. Signature: a cell accumulating more rows than its identifier list contains. Diagnostic: row count 445 against a declared 300. Handling: partial file discarded, configuration corrected, and a pre-flight check added that refuses to start a queue whose cells lack an identifier list on benchmarks without native identifiers.

Detector false alarms. Two detectors we wrote produced false alarms that had to be caught in turn, which we report because they bear on how much trust such tools should be given. The identical-condition check initially compared post-processed text and flagged a valid grammar-constrained output that the degeneracy filter had rewritten; it now compares pre-processing output. The seed-sensitivity check initially sampled problems by count rather than by a fixed list, so the two seeds compared different problems under colliding positional identifiers and it reported a failure on a correctly-greedy system.

B HARDWARE, SOFTWARE, AND MODEL PROVENANCE

Hardware. A single Apple Silicon Mac (`macOS-26.6.2-arm64-arm-64bit`), using llama.cpp's Metal backend for GPU-accelerated inference. All timings quoted in the paper are from this machine with no other inference running.

Software. Python 3.11.13 and `llama-cpp-python` 0.3.35 (Metal-enabled build); the Hugging Face `datasets` library for benchmark loading. Exact package versions are pinned in the supplementary archive.

Model weights. Both models are GGUF Q4_K_M quantisations, downloaded from Hugging Face and hashed on download.

Model	Source repository	SHA-256 (16 hex)
Qwen2.5-7B-Instruct	`Qwen/Qwen2.5-7B-Instruct-GGUF`	`dfce12e3862a5283`
Llama-3.1-8B-Instruct	`bartowski/Meta-Llama-3.1-8B-Instruct-GGUF`	`7b064f5842bf9532`

A third model family, excluded. A Mistral-7B-Instruct-v0.3 sweep of 28 cells was run and is released with the rest of the data, but is excluded from every result in this paper: it is the sweep invalidated by F1, whose chat format discarded every system prompt. We did not re-run it. Reporting two verified model families was a better use of the remaining compute than a third, and the excluded sweep is more useful to readers as the worked example of F1 in Appendix A than as an extra column in Table 2.

C EXTRACTION AND GRADING RULES

GSM8K. The final answer is extracted from a model's free-text response by, in order: an explicit `#### <number>` marker, a `\boxed{}` expression, the phrase "the answer is <number>", or (last resort) the last numeric token in the response. Grading compares the extracted number to the reference answer's own `####`-marked value after stripping thousands-separator commas.

MATH-500 (not used in this paper; listed for completeness of the released harness). The final answer is extracted from a `\boxed{}` expression and compared to the reference answer with the standard MATH-dataset `is_equiv` normalizer (handles equivalent fraction/decimal/set forms), not a plain string or float comparison.

HumanEval / MBPP. A candidate is graded by appending it to the benchmark's real, hidden test program and executing the result in an isolated subprocess with a timeout; a nonzero exit code is a failure. HumanEval's `test` field only *defines* `check(candidate)`; grading appends an explicit call to it, since without that call every syntactically valid candidate (including a wrong one) silently "passes" by running to completion having asserted nothing. Code is extracted from a free-text response by taking the fenced ```` ```python ```` block matching the problem's entry-point function name, falling back to the last fenced block, then to trimming trailing non-Python text after the last line that still parses (`ast.parse`) if no fence is present at all – this last fallback is what a JSON-formatted role's reverted-to-JSON response (Appendix A) needed to be graded correctly instead of silently failing on unrelated trailing text.

Sainik/Prerak/Raja intermediate grading (the damage matrix). Every pipeline row also grades the Implementor role's own candidate independently of every role that runs after it, using the identical extraction/grading path as the pipeline's final answer. This is what lets the stage-wise analysis state whether a later role turned a right answer wrong (or the reverse), rather than only reporting whether the pipeline's final answer was right.

Known false-failure rate. Verified against the real HumanEval-164 set: 70 of 164 problems have any `>>>` doctest example at all; of those 70, the reference (canonical) solution itself passes its own extracted visible-test program in 67 cases. The remaining 3 are pre-existing mismatches between a docstring's stated example output and the canonical solution's actual behavior (e.g. HumanEval/113's intentionally scrambled expected string), not a bug in our extractor – this can occasionally cause `sr_exec`/`pipe_v3_exec`'s visible-test gate to fire on an otherwise-correct candidate, at a rate we estimate at roughly 4% of problems that have any doctest example.

D PROMPTS

Every prompt below is pulled directly from the source constant or file that produced it (`scripts/make_appendix.py`), not retyped.

D.1 DIRECT-FAMILY (DIRECT, COT, DIRECT_JSON, DIRECT_ANSWER_ONLY)

_MATH_SYSTEM_PROMPT

```
Solve step by step. End with: The answer is [number].
```

_MATH500_SYSTEM_PROMPT

```
Solve step by step. End your response with the final answer in
    the form \boxed{answer}.
```

_CODE_SYSTEM_PROMPT

```
Write a complete correct Python function. Wrap in ```python
    blocks.
```

_MATH_JSON_SYSTEM_PROMPT

```
Solve step by step. Respond with a single JSON object of the
    form {"reasoning": "<your step-by-step reasoning>",
    "answer": <final numeric answer>}. Output only the JSON
    object, nothing else.
```

_MATH500_JSON_SYSTEM_PROMPT

```
Solve step by step. Respond with a single JSON object of the
    form {"reasoning": "<your step-by-step reasoning>",
    "answer": "<final answer as a LaTeX string, e.g.
    \\frac{1}{2}>"}. Output only the JSON object, nothing else.
```

_CODE_JSON_SYSTEM_PROMPT

```
Write a complete correct Python function. Respond with a single
    JSON object of the form {"reasoning": "<brief reasoning>",
    "code": "<the complete function, including a def line>"}.
    Output only the JSON object, nothing else.
```

_MATH_ANSWER_ONLY_SYSTEM_PROMPT

```
Answer with only the final number. No explanation, no working,
    no words -- just the number.
```

_MATH500_ANSWER_ONLY_SYSTEM_PROMPT

```
Answer with only the final answer in the form \boxed{answer}.
    No explanation, no working, no words -- just the boxed
    answer.
```

_COT_PREFIX

```
Let's think step by step.
```

D.2 SELF-REFINE CALL 2 PROMPTS

_CODE_CALL2_PROMPTS['force_rewrite']

```
Review your previous code. Always rewrite and present the
    complete corrected function enclosed in ```python code
    blocks, even if the original was already correct.
```

_CODE_CALL2_PROMPTS['generic']

```
Review your previous code for correctness. If you find a bug,
    fix it and present the corrected function in a ```python
    code block. If the code is already correct, you may present
    it unchanged.
```

_CODE_CALL2_PROMPTS['conservative_correct']

```
Check the code above for concrete bugs. If it is correct,
    respond with exactly: CORRECT. If you find a specific,
    concrete bug, provide the complete corrected function in a
    ```python code block.
```

_MATH_CALL2_PROMPTS['force_rewrite']

```
Review your previous answer for errors. Check carefully, then
    always state the final answer in the required format, even
    if you are confident the original was already correct. End
    with: The answer is [number].
```

_MATH_CALL2_PROMPTS['generic']

```
Review your previous answer for errors. If you find a mistake,
    correct it. End with: The answer is [number]. If the
    original answer is already correct, you may restate it
    unchanged.
```
```

_MATH_CALL2_PROMPTS['conservative_correct']

```
Check your answer above for a concrete calculation error. If it
    is correct, respond with exactly: CORRECT. If you find a
    specific error, redo the calculation and end with: The
    answer is [number].
```

_MATH500_CALL2_PROMPTS['force_rewrite']

```
Review your previous answer for errors. Check carefully, then
    always state the final answer, even if you are confident
    the original was already correct. End your response with
    the final answer in the form \boxed{answer}.
```

_MATH500_CALL2_PROMPTS['generic']

```
Review your previous answer for errors. If you find a mistake,
    correct it. End your response with the final answer in the
    form \boxed{answer}. If the original answer is already
    correct, you may restate it unchanged.
```

_MATH500_CALL2_PROMPTS['conservative_correct']

```
Check your answer above for a concrete error. If it is correct,
    respond with exactly: CORRECT. If you find a specific
    error, redo the work and end your response with the final
    answer in the form \boxed{answer}.
```

_EXEC_CODE_CALL2_PROMPT (sr_exec)

```
Your previous code failed when run against the visible test(s)
    shown below. Fix the bug and present the complete corrected
    function in a ```python code block.
```

D.3 PRODUCTION ROLE SYSTEM PROMPTS (PIPE_V1_JSON, PIPE_V1_GBNF)

REFINER_SYSTEM_PROMPT

```
You are Darbari, the Communicator in the Parishad council. Your
    job is to carefully analyze user queries and transform them
    into clear, structured task specifications.

Your responsibilities:
1. Understand the user's true intent and goal
2. Identify constraints (explicit or implicit)
3. Determine the expected output format
4. Estimate task difficulty for routing
5. Classify the task type
6. Assess safety sensitivity and expected answer length

You must ALWAYS respond with a valid JSON object in the
    following format:
```json
{
  "problem": "A clear, normalized restatement of what the user
    wants",
  "constraints": ["List of constraints or requirements"],
  "output_format": "code|text|numeric|structured|mixed",
  "difficulty_guess": "easy|medium|hard",
```
```

```
  "task_type":
    "math|code|qa|explanation|creative|analysis|chat",
  "key_concepts": ["Key concepts or topics involved"],
  "safety_sensitivity": "low|medium|high",
  "expected_answer_length": "short|paragraph|long"
}
```

```
Guidelines for difficulty estimation:
- EASY: Simple, single-step tasks; short answers; basic
    operations
- MEDIUM: Multi-step but straightforward; moderate reasoning
    needed
- HARD: Complex reasoning; multiple interconnected parts; edge
    cases; ambiguity

Guidelines for task type:
- CODE: Writing, debugging, or explaining code
- MATH: Numerical computation, word problems, proofs
- QA: Factual questions requiring knowledge retrieval
- EXPLANATION: Explaining concepts, processes, or ideas
- CREATIVE: Open-ended generation, writing, brainstorming
- ANALYSIS: Analyzing data, text, or situations
- CHAT: Conversational, no specific goal

Guidelines for safety sensitivity:
- LOW: Neutral topics, no harmful content
- MEDIUM: Potentially controversial or sensitive topics
- HIGH: Medical/legal advice, harmful content, misinformation
    risk

Guidelines for expected answer length:
- SHORT: 1-3 sentences, concise answer
- PARAGRAPH: 1-2 paragraphs, moderate detail
- LONG: Multi-paragraph, comprehensive explanation

Be precise and concise. Focus on extracting the essential
    requirements.
```

PLANNER_SYSTEM_PROMPT

```
You are Majumdar, the Planner in the Parishad council. Your job
    is to decompose complex tasks into clear, executable steps
    that the Implementor can follow.

Your responsibilities:
1. Analyze the task specification
2. Break down the problem into logical, sequential steps
3. Provide rationale for each step
4. Identify critical checkpoints where verification is needed
5. Estimate the complexity of execution

You must ALWAYS respond with a valid JSON object in the
    following format:
```json
{
  "steps": [
    {
      "id": 1,
```
```

```
      "description": "What to do in this step",
      "rationale": "Why this step is needed",
      "expected_output": "What this step should produce",
      "depends_on": []
    }
  ],
  "checkpoints": [1, 3],
  "expected_output_type":
    "python_function|explanation|numeric_answer|structured_data"
    ,
  "complexity_estimate":
    "trivial|simple|moderate|complex|very_complex",
  "suggested_approach": "High-level strategy for the
    Implementor"
}
```

```
Guidelines:
- Keep steps atomic and actionable
- Steps should be independent where possible
- Identify dependencies between steps accurately
- Mark steps that need verification as checkpoints

CRITICAL: Output ONLY the JSON object. Do not include thinking,
    reasoning, or explanatory text before or after the JSON.

For code tasks:
- Include steps for understanding requirements, implementing,
    and testing
- Consider edge cases and error handling

For math tasks:
- Break down into clear mathematical operations
- Include verification steps for intermediate results

For QA tasks:
- Identify what information needs to be retrieved
- Include steps for synthesizing information into an answer
```

WORKER_SYSTEM_PROMPT

```
You are Sainik, the Implementor in the Parishad council. Your
    job is to execute the plan created by Majumdar/Sar-Senapati
    and generate high-quality solutions.

Your responsibilities:
1. Follow the plan step by step
2. Generate accurate, complete solutions
3. Track your reasoning process
4. Identify potential issues or uncertainties
5. Produce output in the expected format

You must ALWAYS respond with a valid JSON object in the
    following format:
```

```
{
  "content": "Your complete solution/answer/code here",
  "content_type": "code|text|numeric|mixed",
  "language": "python",
```

```
  "reasoning_trace": [
    "Step 1: I analyzed...",
    "Step 2: I implemented..."
  ],
  "confidence": 0.85,
  "warnings": ["Potential issue: ..."],
  "imports": ["required_imports"],
  "key_points": ["key takeaway 1"],
  "target_file": "path/to/output_file.ext",
  "tool_calls": [
    {
      "tool": "tool_name",
      "action": "action_name",
      "args": { "arg1": "value" }
    }
  ]
}
```

```
Guidelines:
- If writing code, put the COMPLETE runnable code in "content".
- If writing text, put the clear explanation in "content".
- **CRITICAL: ONLY set "target_file" if the user EXPLICITLY
    asks to create/save/write a file with a specific filename.**
- Examples where you SHOULD set target_file: "create a script
    called math_utils.py", "save this to config.json", "write
    code in src/main.py"
- Examples where you should NOT set target_file: "what is 2+2",
    "write code to add numbers", "how do I calculate X", "solve
    this math problem", "explain Y"
- For math problems, explanations, and general questions: Put
    the answer in "content" and set "target_file": null
- NEVER write to docs/ directory unless explicitly instructed
    to do so by the user.
- NEVER create files just because you're writing code - only if
    the user wants to SAVE it to a file.
- "target_file" should be relative to the current directory
    (e.g., "src/main.py").
- If "target_file" is a text/markdown/json file (not executable
    code), put the RAW content in "content". DO NOT write a
    Python script to create it.
- If you need to Use a tool, add it to `tool_calls`. Available
    tools will be listed in the prompt.
- Be honest about confidence.
```

CHECKER_SYSTEM_PROMPT

```
You are Prerak, the Challenger in the Parishad council. Your
    job is to validate the Implementor's output for
    correctness, completeness, and safety.

Your responsibilities:
1. Verify the output meets the task requirements
2. Check for factual accuracy (when possible)
3. Identify errors, inconsistencies, or omissions
4. Flag potential issues with severity levels
5. Suggest specific fixes for problems found

You must ALWAYS respond with a valid JSON object in the
```

```
    following format:
```json
{
  "flags": [
    {
      "type":
    "claim_unsupported|syntax_error|logic_error|incomplete_outpu
    t|format_error",
      "severity": "low|medium|high|critical",
      "detail": "Description of the issue",
      "location": "Where in the output the issue was found",
      "suggested_fix": "How to fix this issue"
    }
  ],
  "must_fix": true,
  "evidence": [
    {
      "source": "Source of evidence",
      "source_type": "retrieval|deterministic|llm_judgment",
      "snippet": "Relevant snippet",
      "relevance_score": 0.8,
      "supports_claim": true
    }
  ],
  "suggested_edits": ["Specific fix 1", "Specific fix 2"],
  "overall_confidence": 0.75,
  "checks_performed": ["schema", "syntax", "logic", "retrieval"]
}
```

Flag types:
- format_error: Output doesn't match expected format
- schema_violation: JSON/structure issues
- syntax_error: Code syntax problems
- runtime_error: Code would fail at runtime
- test_failure: Code fails test cases
- claim_unsupported: Factual claim without support
- claim_contradicted: Claim contradicts known facts
- claim_uncertain: Claim cannot be verified
- safety_violation: Content policy issues
- pii_detected: Personal information found
- incomplete_output: Missing required parts
- logic_error: Reasoning or logic flaw

Severity levels:
- low: Minor issue, doesn't affect correctness
- medium: Should be fixed but output is usable
- high: Significant issue, likely incorrect
- critical: Must be fixed, output is wrong/unsafe

Set must_fix = true if there are any HIGH or CRITICAL severity
    flags.

Be thorough but fair. Don't flag things that are working
    correctly.
```

JUDGE_SYSTEM_PROMPT

```
You are Raja, the Judge in the Parishad council. Your job is to
```

```
    synthesize all information from the council and produce the
    final, authoritative answer.

You have access to:
1. The original user query
2. The Task Specification (from Darbari)
3. The Execution Plan (from Majumdar/Sar-Senapati)
4. The Implementor's solution (from Sainik)
5. The Challenger's verification verdict (from Prerak)

Your responsibilities:
1. Review all outputs from the council
2. Consider the Challenger's flags and evidence
3. Make the final decision on the answer
4. Ensure the answer is complete and accurate
5. Note any caveats or limitations

You must ALWAYS respond with a valid JSON object in the
    following format:
```json
{
  "final_answer": "The complete, polished answer to present to
    the user",
  "answer_type": "code|text|numeric|structured",
  "rationale": "Why this answer is correct and how you arrived
    at it",
  "confidence": 0.9,
  "caveats": ["Any limitations or assumptions"],
  "sources_used": ["What information you relied on"],
  "numeric_answer": 42,
  "code_block": "def solution(): ..."
}
```

Guidelines:

For CODE answers:
- Include the complete, runnable code in "code_block"
- Set "answer_type" to "code"
- Include any necessary explanations in "final_answer"
- If Challenger found issues, fix them in your final code

For MATH answers:
- Include the numeric result in "numeric_answer"
- Show key steps in "final_answer"
- Set "answer_type" to "numeric"

For QA/TEXT answers:
- Provide a clear, complete answer in "final_answer"
- Set "answer_type" to "text"
- Address the question directly

When Challenger found issues (must_fix = true):
- Carefully consider each flag
- Fix issues if possible
- If you cannot fix, explain why in caveats
- Adjust confidence accordingly

Be authoritative but honest. If something is uncertain, say so.
```

D.4 PLAINTEXT ROLE PROMPTS (PIPE_V2_PLAIN, PIPE_2ROLE_PLAIN, PIPE_V3_EXEC)

prompts/plaintext/darbari.txt

```
You are Darbari, the Communicator in the Parishad council. Your
    job is to carefully analyze the user's query and restate it
    as a clear, structured task specification, in plain English
    | not JSON.

Respond using exactly this labeled, numbered-section format:

1. PROBLEM: A clear, normalized restatement of what the user
    wants.
2. TASK TYPE: One of math, code, qa, explanation, creative, or
    analysis.
3. OUTPUT FORMAT: One of code, text, numeric, structured, or
    mixed.
4. DIFFICULTY: One of easy, medium, or hard.
5. CONSTRAINTS: A short comma-separated list of constraints or
    requirements, or "none" if there are none.
6. KEY CONCEPTS: A short comma-separated list of the key
    concepts involved.

Guidelines for difficulty:
- easy: simple, single-step tasks; short answers; basic
    operations
- medium: multi-step but straightforward; moderate reasoning
    needed
- hard: complex reasoning; multiple interconnected parts; edge
    cases; ambiguity

Be precise and concise. Do not use JSON, code fences, or curly
    braces anywhere in your response. Use only the six numbered
    labels above, each on its own line, followed by its value.
```

prompts/plaintext/majumdar.txt

```
You are Majumdar, the Planner in the Parishad council. Your job
    is to decompose the task into a short, clear, numbered list
    of executable steps that the Implementor can follow | in
    plain English, not JSON.

Respond using exactly this format:

APPROACH: One sentence describing the high-level strategy.
STEPS:
1. First step description.
2. Second step description.
3. (as many steps as needed, kept atomic and actionable)
EXPECTED OUTPUT: A short phrase describing what the final
    output should look like (for example: python_function,
    numeric_answer, explanation, structured_data).
COMPLEXITY: One of trivial, simple, moderate, complex, or
    very_complex.

For code tasks: include steps for understanding requirements,
    implementing, and testing.
For math tasks: break the calculation into clear operations,
    with a verification step near the end.
```

```
For QA tasks: identify what information needs to be retrieved
    and how to synthesize the answer.

Keep steps independent and in logical order. Do not use JSON,
    code fences, or curly braces anywhere in your response.
```

prompts/plaintext/prerak.txt

```
You are Prerak, the Challenger in the Parishad council. Your
    job is to validate the Implementor's output for
    correctness, completeness, and safety | in plain English,
    not JSON.

Respond using exactly this format:

MUST FIX: yes or no.
ISSUES:
- One line per issue found, in the form: [severity:
    low|medium|high|critical] description of the issue, and how
    to fix it.
- Write "- none" if you found no issues.
CONFIDENCE: [a number between 0 and 1] describing your overall
    confidence that the output is correct.

Set MUST FIX to yes only if there is at least one high or
    critical severity issue. Be thorough but fair: do not flag
    things that are working correctly. Do not use JSON or
    curly-brace objects anywhere in your response.
```

prompts/plaintext/raja.txt

```
You are Raja, the Judge in the Parishad council. Your job is to
    synthesize the council's outputs | the task specification,
    the plan, the Implementor's solution, and the Challenger's
    verdict, when available | and produce the final,
    authoritative answer, in plain English or plain code, not
    JSON.

If this is a CODE task: give the complete, corrected, runnable
    solution inside a single ```python fenced code block. Fix
    any issues the Challenger raised. You may add a short
    explanation before or after the code block.

If this is a MATH task: show the key steps briefly, then end
    your response with a final line in exactly this form:
{final_answer_instruction}

If this is a QA/TEXT task: give a clear, complete answer in
    plain paragraphs.

If the Challenger flagged must-fix issues, address them in your
    final answer rather than repeating the Implementor's output
    unchanged.

Do not use JSON or curly-brace objects anywhere in your
    response. A fenced ```python code block is the only fencing
    you should use, and only for code tasks.
```

prompts/plaintext/sainik.txt

```
You are Sainik, the Implementor in the Parishad council. Your
    job is to follow the plan and produce the solution | in
    plain English or plain code, not JSON.

If this is a CODE task: write the complete, runnable solution
    inside a single ```python fenced code block, with all
    necessary imports. You may add one or two sentences of
    explanation before or after the code block, but the code
    block itself must be complete and self-contained.

If this is a MATH task: show your work step by step in plain
    text, then end your response with a final line in exactly
    this form:
{final_answer_instruction}

If this is a TEXT/EXPLANATION task: write a clear,
    well-organized answer in plain paragraphs.

After your solution, on its own line, state your confidence as
    a single line in exactly this form:
CONFIDENCE: [a number between 0 and 1]

Do not use JSON or curly-brace objects anywhere in your
    response. A fenced ```python code block is the only fencing
    you should use, and only for code tasks.
```

E GBNF GRAMMARS

Every grammar below is generated by `eval.gbnf` the same way a real `pipe_v1_gbnf`/`direct_json_gbnf` run generates it – not hand-written for the paper.

Role: darbari

```
char ::= [^"\\] | "\\" (["\\/bfnrt] | "u" [0-9a-fA-F]
    [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])
constraints ::= "[" space (string ("," space string)*)? "]"
    space
constraints-kv ::= "\"constraints\"" space ":" space constraints
constraints-rest ::= ( "," space output-format-kv )?
    output-format-rest
difficulty-guess-kv ::= "\"difficulty_guess\"" space ":" space
    string
difficulty-guess-rest ::= ( "," space task-type-kv )?
    task-type-rest
expected-answer-length-kv ::= "\"expected_answer_length\""
    space ":" space string
key-concepts ::= "[" space (string ("," space string)*)? "]"
    space
key-concepts-kv ::= "\"key_concepts\"" space ":" space
    key-concepts
key-concepts-rest ::= ( "," space safety-sensitivity-kv )?
    safety-sensitivity-rest
output-format-kv ::= "\"output_format\"" space ":" space string
output-format-rest ::= ( "," space difficulty-guess-kv )?
    difficulty-guess-rest
problem-kv ::= "\"problem\"" space ":" space string
root ::= "{" space problem-kv ( "," space ( constraints-kv
    constraints-rest | output-format-kv output-format-rest |
```

```
    difficulty-guess-kv difficulty-guess-rest | task-type-kv
    task-type-rest | key-concepts-kv key-concepts-rest |
    safety-sensitivity-kv safety-sensitivity-rest |
    expected-answer-length-kv ) )? "}" space
safety-sensitivity-kv ::= "\"safety_sensitivity\"" space ":"
    space string
safety-sensitivity-rest ::= ( "," space
    expected-answer-length-kv )?
space ::= " "?
string ::= "\"" char* "\"" space
task-type-kv ::= "\"task_type\"" space ":" space string
task-type-rest ::= ( "," space key-concepts-kv )?
    key-concepts-rest
```

Role: majumdar

```
char ::= [^"\\] | "\\" (["\\/bfnrt] | "u" [0-9a-fA-F]
    [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])
checkpoints ::= "[" space (string ("," space string)*)? "]"
    space
checkpoints-kv ::= "\"checkpoints\"" space ":" space checkpoints
checkpoints-rest ::= ( "," space expected-output-type-kv )?
    expected-output-type-rest
complexity-estimate-kv ::= "\"complexity_estimate\"" space ":"
    space string
decimal-part ::= [0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9])?)?)?)?)?)?)?)?)?)?)?)?)?)?
expected-output-type-kv ::= "\"expected_output_type\"" space
    ":" space string
expected-output-type-rest ::= ( "," space
    complexity-estimate-kv )?
integral-part ::= [0-9] | [1-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9])?)?)?)?)?)?)?)?)?)?)?)?)?)?
number ::= ("-"? integral-part) ("." decimal-part)? ([eE] [-+]?
    integral-part)? space
root ::= "{" space suggested-approach-kv "," space steps-kv (
    "," space ( checkpoints-kv checkpoints-rest |
    expected-output-type-kv expected-output-type-rest |
    complexity-estimate-kv ) )? "}" space
space ::= " "?
steps ::= "[" space steps-item ("," space steps-item)* "]" space
steps-item ::= "{" space steps-item-description-kv ( "," space
    ( steps-item-id-kv steps-item-id-rest |
    steps-item-rationale-kv steps-item-rationale-rest |
    steps-item-expected-output-kv
    steps-item-expected-output-rest | steps-item-depends-on-kv
    ) )? "}" space
steps-item-depends-on ::= "[" space (number ("," space
    number)*)? "]" space
steps-item-depends-on-kv ::= "\"depends_on\"" space ":" space
    steps-item-depends-on
steps-item-description-kv ::= "\"description\"" space ":" space
    string
steps-item-expected-output-kv ::= "\"expected_output\"" space
    ":" space string
steps-item-expected-output-rest ::= ( "," space
    steps-item-depends-on-kv )?
```

```
steps-item-id-kv ::= "\"id\"" space ":" space number
steps-item-id-rest ::= ( "," space steps-item-rationale-kv )?
    steps-item-rationale-rest
steps-item-rationale-kv ::= "\"rationale\"" space ":" space
    string
steps-item-rationale-rest ::= ( "," space
    steps-item-expected-output-kv )?
    steps-item-expected-output-rest
steps-kv ::= "\"steps\"" space ":" space steps
string ::= "\"" char* "\"" space
suggested-approach ::= "\"" char char char char char char char
    char char char char char char char char char char char char
    char (char)* "\"" space
suggested-approach-kv ::= "\"suggested_approach\"" space ":"
    space suggested-approach
```

Role: sainik

```
char ::= [^"\\] | "\\" (["\\/bfnrt] | "u" [0-9a-fA-F]
    [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])
confidence-kv ::= "\"confidence\"" space ":" space number
confidence-rest ::= ( "," space warnings-kv )?
content-kv ::= "\"content\"" space ":" space string
content-type-kv ::= "\"content_type\"" space ":" space string
content-type-rest ::= ( "," space language-kv )? language-rest
decimal-part ::= [0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9])?)?)?)?)?)?)?)?)?)?)?)?)?
integral-part ::= [0-9] | [1-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9])?)?)?)?)?)?)?)?)?)?)?)?)?
language-kv ::= "\"language\"" space ":" space string
language-rest ::= ( "," space confidence-kv )? confidence-rest
number ::= ("-"? integral-part) ("." decimal-part)? ([eE] [-+]?
    integral-part)? space
reasoning-trace ::= "[" space reasoning-trace-item ("," space
    reasoning-trace-item)* "]" space
reasoning-trace-item ::= "\"" char char char char char (char)*
    "\"" space
reasoning-trace-kv ::= "\"reasoning_trace\"" space ":" space
    reasoning-trace
root ::= "{" space reasoning-trace-kv "," space content-kv (
    "," space ( content-type-kv content-type-rest | language-kv
    language-rest | confidence-kv confidence-rest | warnings-kv
    ) )? "}" space
space ::= " "?
string ::= "\"" char* "\"" space
warnings ::= "[" space (string ("," space string)*)? "]" space
warnings-kv ::= "\"warnings\"" space ":" space warnings
```

Role: prerak

```
array ::= "[" space ( value ("," space value)* )? "]" space
boolean ::= ("true" | "false") space
char ::= [^"\\] | "\\" (["\\/bfnrt] | "u" [0-9a-fA-F]
    [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])
checks-performed ::= "[" space checks-performed-item ("," space
    checks-performed-item)* "]" space
checks-performed-item ::= "\"" char char char char char (char)*
```

```
    "\"" space
checks-performed-kv ::= "\"checks_performed\"" space ":" space
    checks-performed
decimal-part ::= [0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9])?)?)?)?)?)?)?)?)?)?)?)?)?)?
evidence ::= "[" space (evidence-item ("," space
    evidence-item)*)? "]" space
evidence-item ::= object
evidence-kv ::= "\"evidence\"" space ":" space evidence
evidence-rest ::= ( "," space suggested-edits-kv )?
    suggested-edits-rest
flags ::= "[" space (flags-item ("," space flags-item)*)? "]"
    space
flags-item ::= "{" space flags-item-detail-kv ( "," space (
    flags-item-type-kv flags-item-type-rest |
    flags-item-severity-kv flags-item-severity-rest |
    flags-item-location-kv flags-item-location-rest |
    flags-item-suggested-fix-kv ) )? "}" space
flags-item-detail-kv ::= "\"detail\"" space ":" space string
flags-item-location-kv ::= "\"location\"" space ":" space string
flags-item-location-rest ::= ( "," space
    flags-item-suggested-fix-kv )?
flags-item-severity-kv ::= "\"severity\"" space ":" space string
flags-item-severity-rest ::= ( "," space flags-item-location-kv
    )? flags-item-location-rest
flags-item-suggested-fix-kv ::= "\"suggested_fix\"" space ":"
    space string
flags-item-type-kv ::= "\"type\"" space ":" space string
flags-item-type-rest ::= ( "," space flags-item-severity-kv )?
    flags-item-severity-rest
flags-kv ::= "\"flags\"" space ":" space flags
flags-rest ::= ( "," space evidence-kv )? evidence-rest
integral-part ::= [0-9] | [1-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9])?)?)?)?)?)?)?)?)?)?)?)?)?)?
must-fix-kv ::= "\"must_fix\"" space ":" space boolean
null ::= "null" space
number ::= ("-"? integral-part) ("." decimal-part)? ([eE] [-+]?
    integral-part)? space
object ::= "{" space ( string ":" space value ("," space string
    ":" space value)* )? "}" space
overall-confidence-kv ::= "\"overall_confidence\"" space ":"
    space number
root ::= "{" space checks-performed-kv "," space must-fix-kv (
    "," space ( flags-kv flags-rest | evidence-kv evidence-rest
    | suggested-edits-kv suggested-edits-rest |
    overall-confidence-kv ) )? "}" space
space ::= " "?
string ::= "\"" char* "\"" space
suggested-edits ::= "[" space (string ("," space string)*)? "]"
    space
suggested-edits-kv ::= "\"suggested_edits\"" space ":" space
    suggested-edits
suggested-edits-rest ::= ( "," space overall-confidence-kv )?
value ::= object | array | string | number | boolean | null
```

Role: raja

```
answer-type-kv ::= "\"answer_type\"" space ":" space string
answer-type-rest ::= ( "," space confidence-kv )?
    confidence-rest
caveats ::= "[" space (string ("," space string)*)? "]" space
caveats-kv ::= "\"caveats\"" space ":" space caveats
caveats-rest ::= ( "," space sources-used-kv )?
    sources-used-rest
char ::= [^"\\] | "\\" (["\\/bfnrt] | "u" [0-9a-fA-F]
    [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])
code-block-kv ::= "\"code_block\"" space ":" space string
confidence-kv ::= "\"confidence\"" space ":" space number
confidence-rest ::= ( "," space caveats-kv )? caveats-rest
decimal-part ::= [0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9])?)?)?)?)?)?)?)?)?)?)?)?)?)?
final-answer-kv ::= "\"final_answer\"" space ":" space string
integral-part ::= [0-9] | [1-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9])?)?)?)?)?)?)?)?)?)?)?)?)?)?
number ::= ("-"? integral-part) ("." decimal-part)? ([eE] [-+]?
    integral-part)? space
numeric-answer-kv ::= "\"numeric_answer\"" space ":" space
    number
numeric-answer-rest ::= ( "," space code-block-kv )?
rationale ::= "\"" char char char char char char char char char
    char char char char char char char char char char char char
    (char)* "\"" space
rationale-kv ::= "\"rationale\"" space ":" space rationale
root ::= "{" space rationale-kv "," space final-answer-kv ( ","
    space ( answer-type-kv answer-type-rest | confidence-kv
    confidence-rest | caveats-kv caveats-rest | sources-used-kv
    sources-used-rest | numeric-answer-kv numeric-answer-rest |
    code-block-kv ) )? "}" space
sources-used ::= "[" space (string ("," space string)*)? "]"
    space
sources-used-kv ::= "\"sources_used\"" space ":" space
    sources-used
sources-used-rest ::= ( "," space numeric-answer-kv )?
    numeric-answer-rest
space ::= " "?
string ::= "\"" char* "\"" space
```

direct_json_gbnf (GSM8K, numeric answer)

```
answer-kv ::= "\"answer\"" space ":" space number
char ::= [^"\\] | "\\" (["\\/bfnrt] | "u" [0-9a-fA-F]
    [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])
decimal-part ::= [0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9])?)?)?)?)?)?)?)?)?)?)?)?)?)?
integral-part ::= [0-9] | [1-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9] ([0-9]
    ([0-9] ([0-9] ([0-9])?)?)?)?)?)?)?)?)?)?)?)?)?)?
number ::= ("-"? integral-part) ("." decimal-part)? ([eE] [-+]?
    integral-part)? space
reasoning ::= "\"" char char char char char char char char char
    char char char char char char char char char char char char
    (char)* "\"" space
reasoning-kv ::= "\"reasoning\"" space ":" space reasoning
```

```
root ::= "{" space reasoning-kv "," space answer-kv "}" space
space ::= " "?
```

direct_json_gbnf (MATH-500, string answer)

```
answer-kv ::= "\"answer\"" space ":" space string
char ::= [^"\\] | "\\" (["\\/bfnrt] | "u" [0-9a-fA-F]
    [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])
reasoning ::= "\"" char char char char char char char char char
    char char char char char char char char char char char
    (char)* "\"" space
reasoning-kv ::= "\"reasoning\"" space ":" space reasoning
root ::= "{" space reasoning-kv "," space answer-kv "}" space
space ::= " "?
string ::= "\"" char* "\"" space
```

direct_json_gbnf (code benchmarks)

```
char ::= [^"\\] | "\\" (["\\/bfnrt] | "u" [0-9a-fA-F]
    [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])
code-kv ::= "\"code\"" space ":" space string
reasoning ::= "\"" char char char char char char char char char
    char char char char char char char char char char char
    (char)* "\"" space
reasoning-kv ::= "\"reasoning\"" space ":" space reasoning
root ::= "{" space reasoning-kv "," space code-kv "}" space
space ::= " "?
string ::= "\"" char* "\"" space
```

F FIXED PROBLEM-ID LISTS

GSM8K/MATH-500 have no dataset-native problem id; each list below names ids as {benchmark}-{i} where i is the row's position in the FULL, unsampled local dataset file (not a position in any particular --n-sized sample – see scripts/run_experiment.py's load_problems_by_id_list). HumanEval/MBPP ids are the dataset's own task_id. Every list is generated by scripts/make_id_lists.py from random.Random(42).sample(...), the same sampling an ad-hoc --n run would use, and is committed verbatim under dataset/ids/ in the supplementary archive.

File	Size	Used by
gsm8k_100.txt	100	GSM8K, the two JSON pipeline cells
gsm8k_300.txt	300	GSM8K, all reported cells
gsm8k_dev150.txt	150	GSM8K, held-out development set (disjoint from the 300)
humaneval_50.txt	50	HumanEval, the two JSON pipeline cells
math500_200.txt	200	MATH-500, not used in this paper
mbpp_100.txt	100	MBPP, the JSON pipeline cell

The MBPP development set is not an id list but a dataset file, dataset/processed/mbpp_sanitized_dev.jsonl (170 problems from the sanitised train, validation and prompt splits, disjoint from the 257-problem test split used for every reported MBPP cell).