TYPED RETRY TRAJECTORIES: STRUCTURING AGENT CONTEXT FOR TOOL-USE RECOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

Agents often need to recover from failed or uncertain actions, using information from earlier steps to decide their next action. However, these histories often combine task instructions, external results, errors, constraints, feedback and earlier actions in a single text representation, leaving the model to infer the role and reliability of each part. We introduce **typed retry trajectories**, a structured representation that separates an agent's decision context into four parts: *instruction, data, requirement,* and *execution*. The *instruction* part states the task goal. The *data* part contains observations, tool results, earlier actions, and other external information. The *requirement* part contains rules, limits, and verifier feedback. The *execution* part describes the next proposed action. We implement this schema in LLMON (Hind et al., 2026), a general-purpose markup language that allows the definition of arbitrary tags or concepts. We leverage a fine-tuned LLMON model pretrained on tags such as `instr`, `data`, and `exec`, and extend it by defining retry-specific tags (e.g., `goal`, `stopping_criteria`) and mapping them to LLMON's existing tags.

We evaluate typed action trajectories on the repair of failed tool-use cases from tau-bench Yao et al. (2024). We compare the same model and the same trajectory information in three formats: typed channels, plain JSON, and prose logs. Typed channels improve F1 for predicting the next tool-call name by 0.088 (JSON) and 0.060 (prose logs). The structured format also improves tool selection and tool-argument accuracy. These results show that separating instructions, observations, rules, and actions can help agents recover from failed tool calls. More broadly, typed action trajectories provide a practical way to distinguish trusted task instructions and constraints from external information that may be incomplete, incorrect, or unsafe. This distinction may also support other agent actions, including memory writes, record updates, message sending, and delegation to other agents.

1 INTRODUCTION

Agents make mistakes and must often retry. An agent might call the wrong endpoint, write incorrect information to memory, update a record with incomplete data, or pass unreliable information to another agent (Yao et al., 2023). To recover, the agent can use the task, prior actions, tool outputs, errors, constraints, and verifier feedback to identify the failure and select a corrective next action. Often this retry history is stored as a single prose or JSON log. This approach mixes task instructions, external observations, constraints, verifier feedback, and prior actions, forcing the model to infer which content states the goal, describes the current state, restricts the next action, or may be unreliable.

Existing frameworks structure parts of the LLM interface. DSPy defines typed module inputs and outputs fields for a module (Khattab et al., 2023), prompt-orchestration markup templates a prompt's components (Zhang et al., 2025; POML), and constrained decoding shapes the output (Willard & Louf, 2023; Beurer-Kellner et al., 2024; Guidance). Although these approaches structure the call, the retry trajectory, which is the part that grows and carries the errors, is captured as accumulated text between those typed slots and stays unstructured.

We propose a schema for retry histories that has tags for the goal, stopping criteria, attempt count, numbered calls, results, failures, and verifier feedback (§3). Each tag belongs to one of four channels: instruction, data, requirement, or execution.

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.

While we evaluate this schema in the context of tool-use recovery, the separation of instructions, observations, rules, and actions is a general principle that could apply to other agent tasks, such as memory management, planning, or multi-agent coordination.

On τ-bench repair (Yao et al., 2024), focusing on the problem of given a failed trajectory, produce the corrected list of tool calls, a retry agent given the channel-structured trajectory beats the same model given the same trajectory as plain JSON by 0.088 F1, and beats it as prose logs by 0.060 F1, under 5-fold cross-validation over 180 held-out failed trajectories with paired bootstrap intervals (§5.1). We then examine which parts of the representation account for this effect.

- **Channel assignment and syntax:** Channel assignment accounts for more of the gain than tags alone. Combining both does not improve on either alone. Assigning tool outputs to data rather than instruction is key (§5.2, §5.3).
- **Tool selection vs. arguments:** The gain is in tool-call-name prediction; argument-exact measures do not show consistent improvement (§5.6).
- **Markup and pretraining:** Two architecture-matched checkpoint pairs show no consistent benefit from markup pretraining. However, the markup's single-token delimiters are a practical prerequisite for reliable use §5.4).
- **Prompt injection:** Structured renderings reduce attack success relative to plain JSON in our tests, but none prevents all attacks. Authority framing is more effective than forged delimiters in these experiments (§5.5).

Our own earlier single-split experiments on this task produced a larger, cleaner, and partly false version of this result. §6 documents five pipeline artifacts that contributed to this discrepancy, including a small test set, inconsistent reference action lists, a decoding penalty, a missing role header, and training data overwritten after checkpoint creation. These findings show that evaluation artifacts can be comparable in magnitude to the representation effects under study.

2 Related Work

Structure at the interface. A growing family of systems gives LLM programs explicit structure. DSPy declares typed input and output fields per module and optimises the prompt around them (Khattab et al., 2023); prompt-orchestration markup templates a prompt's components (Zhang et al., 2025; POML); programming abstractions over prompts add control flow and typed values (Beurer-Kellner et al., 2023a;b; Zheng et al., 2024); compact serialisations target token economy (Schopplich, 2025). These methods structure the call's inputs or outputs, while the growing retry history, which includes calls, errors and verifier feedback remains unstructured. Our schema addresses this history and can be carried in a field of these frameworks.

LLMON (Hind et al., 2026) provides single-token delimiters (e.g., `<|open|>`, `<|close|>`) for marking channel boundaries, and includes model checkpoints pretrained on this vocabulary. However, the tags themselves (e.g., `instr`, `data`, `exec`) are not guaranteed to be single tokens-their tokenization depends on the tokenizer.

This allows us to examine whether the observed effect depends on the prompt representation or on prior exposure to the markup during pretraining.

Constrained and grammar-guided generation. A parallel line enforces structure on the *output* rather than the input (Willard & Louf, 2023; Beurer-Kellner et al., 2024; Guidance). This motivates a contol in our experiments: all arms use the same output format, since constrained output formats can themselves affect accuracy Mizrahi et al. (2024) , Sclar et al. (2024).

Prompt format sensitivity. Sensitivity to surface form is well documented (Chatterjee et al., 2024; Liu et al., 2025). In our earlier experiments, changing the test-set size reversed a statistically significant result (§6.1). We therefore use cross-validation and paired intervals rather than relying on a single split.

Agent retry loops. Propose, observe, retry is standard (Yao et al., 2023) and orchestration frameworks build on it (Wu et al., 2024; LangChain, 2023; LlamaIndex). Degradation over long accumulated context is a known failure mode (Laban et al., 2025), and it is the pressure our schema is meant to relieve.

Tool agent benchmarks. We use the retail domain of τ-bench, whose per-task reference action lists permit deterministic scoring without an LLM judge (Yao et al., 2024); τ^2-bench extends it (Barres et al., 2025). We document a scoring pitfall we have not seen reported elsewhere: the references disagree with *themselves* about whether read-only calls belong in the answer (§6.2).

Trust boundaries. Instructions smuggled through retrieved or tool-returned content are a live threat class (Greshake et al., 2023; Rawat et al., 2024), and channel separation is the standard argument for typed context. We test this argument directly by injecting an instruction into tool output at three attack strengths and measuring how often it hijacks the agent (§5.5). Our results show that structuring reduces attack success in the tested settings, but does not eliminate prompt injection.

3 A Schema for Retry Trajectories

The tag vocabulary. A retry prompt carries seven kinds of content. We give each a tag, and we number every element that can recur so that a failure is addressable rather than merely described.

Tag	Channel	Justification
`goal`	*instruction*	it is the task

`stopping_criteria(`
`  max_attempts,`
`  success_condition)`
	requirement	a constraint on the next attempt, not a new task; the harness can evaluate every pass without parsing prose

`attempt_count`	*data*	observation about the loop's own state
`tool_call:N` (history)	*data*	an observation of a past action
`tool_result:N`	*data*	**untrusted**, the model did not author it
`failure:N`	*data*	an observation, numbered to the step that produced it
`verifier_feedback`	*requirement*	constrains the retry, not a new task
`tool_call` (current)	*execution*	the only action under consideration

The tags vocabulary is one part of the proposal, and the channel assignment is the main factor we evaluate. §5.2 shows the assignment carries more of the effect than the syntax does, so a practitioner adopting this should port the assignment even if they keep their own syntax. Two assignments are worth defending explicitly. Assigning tool_result to data rather than instruction explicitly marks tool output as untrusted content (Greshake et al., 2023). Assigning verifier_feedback to requirement distinguishes feedback about the current attempt from the task itself, reducing the risk that the agent treats the critique as a replacement goal.

4 Experimental Design

Task and data. Trajectories come from the retail domain of τ-bench (Yao et al., 2024): 460 recorded runs over 115 tasks, of which 182 fail. The repair task is to take a failed run and output the complete correct list of tool calls that would have accomplished it. This task isolates the effect of history representation on the predicted repair, while per-task reference action lists allow deterministic scoring without an LLM judge.

LLMON Realisation. LLMON (Hind et al., 2026) is a general-purpose markup language designed to improve LLM interpretability by allowing the definition of arbitrary tags or concepts. We leverage a fine-tuned LLMON model pretrained on a set of tags, including `instr`, `data`, and `exec`. In this work, we extend this framework by defining a set of retry-specific tags (e.g., `goal`, `stopping_criteria`, `tool_call`, etc.) and map them to LLMON's existing tags, as shown in Table 1. We render the schema in LLMON, whose delimiters (e.g., $< |open| >$, $< |close| >$) are single tokens in the model vocabulary, while the tags (e.g., `instr`, `data`, `exec`) may be tokenized into multiple tokens depending on the tokenizer. Thus, a boundary costs one token rather than five. Figure 1 shows an example of mapping a free-form prose trajectory into our typed LLMON representation.

The schema is independent of LLMON and could be expressed in XML or chat-template role sections. In §5.4, expressing the same assignment through role labels alone does not reproduce the gain, so transfer to other syntaxes remains unverified. We have not yet evaluated an XML representation.

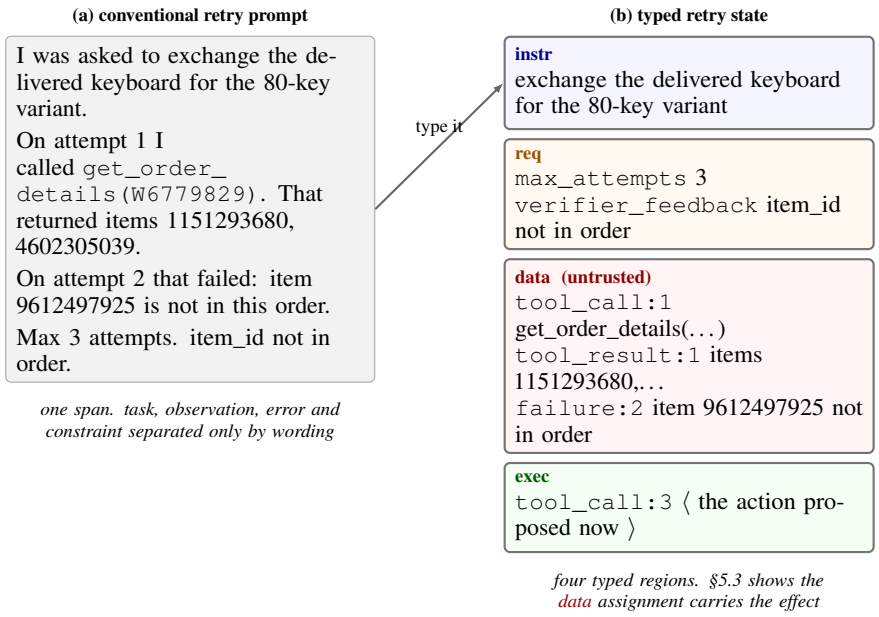


Figure 1: What a retry agent reads. (a) The conventional prompt: one span in which task, observation, error and constraint are distinguished only by wording. (b) The same content as typed retry state. Tool output lands in `data` and is therefore marked as something the model did not author; constraints land in `req`; only the action under consideration is in `exec`. §5.3 shows the `data` assignment is where the effect lives.

Table 1: Mapping of retry-specific tags to LLMON's pretrained tags.

Retry-Specific Tag	LLMON Tag
`goal`	`instr`
`stopping_criteria`	`requirement`
`tool_call`	`exec`
`tool_result`	`data`
`failure`	`data`
`verifier_feedback`	`requirement`

What we vary. To separate the channel assignment from the syntax that expresses it, we render each failed trajectory four ways through one renderer with a `--dialect` switch, so truncation, deduplication, and parse tolerance cannot drift between arms. The first arm is unstructured: the trajectory as a JSON object in a plain chat prompt. This is the conventional way a retry agent receives its history, and our main baseline. The second arm is prose: the same trajectory rewritten as narrative log text, which is the format §1 motivates against most directly. The third and fourth arms are the decomposition that matters. Channels applies the instruction, data, requirement, and execution assignment to the plain JSON trajectory with no new syntax. Flat tags re-renders the trajectory in LLMON tags with no channel assignment at all. Tagged channels does both together. This four-arm comparison separates the effect of channel assignment from that of tag syntax §5.2

Experimental factors. The design separates two things a single "does markup help" comparison conflates: the channel assignment (§5.1) and the tag syntax that expresses it (§5.2).

Factor	Levels	Isolated by
Pretraining	{markup, none}	identical adapter, content, and wrapper, base swapped within a matched pair; two pairs
Input rendering	4 (§4.1)	§5.2
Wrapper	{channel envelope, chat}	§5.1: same base, same content

Every arm is LoRA-tuned (Hu et al., 2021) with an identical recipe on identical splits, so within-campaign contrasts vary exactly one factor: rank 32, $\alpha = 64$, dropout 0.05, learning rate 2×10^{-4}, cosine schedule over 5 epochs with best-checkpoint selection by validation loss, batch 4 with 2-step accumulation, completion-only loss, bf16. We fix 5 epochs for every arm because a 2-epoch run is not simply a 5-epoch run stopped early: in our own probe the two differed in the direction of validation loss between epochs 1 and 2.

The matched pairs. Prior comparisons of this kind swap a markup-pretrained checkpoint for an unrelated baseline, which confounds pretraining with model class and wrapper. We instead use two pairs, each matched within an architecture class on depth, width, attention heads, key-value heads, feed-forward width, and vocabulary size, and differing from each other in architecture class, giving an independent replication. Both members of a pair train and evaluate on the same neutral chat wrapper over identical content, and we verified that both tokenizers treat that wrapper's role markers as native single tokens. The markup-pretrained member tokenises the markup delimiters as single tokens; the control does not, which is the treatment itself rather than a confound. We have not verified that the two checkpoints in a pair also share a pretraining data recipe and token budget apart from the markup. A null found independently on two pairs is more robust to that gap than a positive claim would have been, but it remains an open item.

Metrics. Let $P = (p_1, \ldots, p_m)$ be the predicted call list and $G = (g_1, \ldots, g_k)$ the reference list for one case, each call a pair $(\textsc{name}, \textsc{args})$. Two calls match under an equivalence $\sim$ that is either name-only or name-and-arguments:

$$p \sim_{\text{name}} g \iff \textsc{name}(p) = \textsc{name}(g) \tag{1}$$

$$p \sim_{\text{arg}} g \iff \textsc{name}(p) = \textsc{name}(g) \wedge \textsc{args}(p) = \textsc{args}(g). \tag{2}$$

A correct answer may legitimately repeat a call, so we match as multisets: $\tau_\sim(P, G)$ is the size of a maximum matching between P and G under $\sim$, and

$$\mathrm{F1}_\sim(P, G) = \frac{2\,\tau_\sim(P, G)}{|P| + |G|}. \tag{3}$$

Crossing $\sim \in \{\text{name}, \text{arg}\}$ with a scope restriction $W(\cdot)$ that keeps only state-changing calls gives the four measures we report: $\mathrm{F1}^{\text{all}}_{\text{name}}$, $\mathrm{F1}^{\text{all}}_{\text{arg}}$, $\mathrm{F1}^{\text{write}}_{\text{name}}$, and $\mathrm{F1}^{\text{write}}_{\text{arg}}$. Write actions are those whose effect a user would notice, identified by tool-name prefix. We treat write-action F1 as an important secondary measure, for a reason we document in §6.2: the references omit read-only calls on some cases and include them on others, so all-action F1 partly measures label composition rather than model behaviour. Against plain JSON, the effect we report clears zero only on the all-action measures; against prose logs it clears zero on all four. We say this plainly here rather than only in a caption, since a headline number that holds up against one baseline and not the other should not be smoothed over.

Splits and statistics. We partition all 69 failure-bearing tasks into 5 task-disjoint folds, giving 180 held-out stems per arm, and retrain each arm per fold on that fold's complement. We verified that no stem appears in two folds and that all renderings of a fold contain the same rows. This avoid relying on a single split: in our earlier experiments, a significant effect on 46 items reversed sign when evaluated on 180 (§6.1).

Every arm scores the same cases, so for arms A, B we take the per-case differences $d_i = \mathrm{F1}(P_i^A, G_i) - \mathrm{F1}(P_i^B, G_i)$ and report the mean with a bias-corrected and accelerated bootstrap interval over $10{,}000$ resamples. Alongside it for each contrast, we report the paired difference and bootstrap interval, win–loss–tie counts, a sign test, and a Wilcoxon signed-rank test, because ties dominate at this scale and the two families of test can disagree (§6.6). For a null claim we also report equivalence tests against a small margin and the minimum detectable effect at power 0.8,

$$\mathrm{MDE} = \left(z_{1-\alpha/2} + z_{1-\beta}\right) \frac{s_d}{\sqrt{n}}, \tag{4}$$

which ranges from 0.033 to 0.074 across our four measures, so a reported null should be read as no effect larger than roughly 0.05 F1, not as evidence of exactly zero.

All arms decode greedily with no repetition penalty, on one inference stack in bf16: a repetition penalty of 1.2, harmless for prose, punishes a repeating JSON action list and cost 0.39 F1 when we first measured these arms (§6.3). Every trained checkpoint records the SHA-256 of its training and validation files, added after we discovered that earlier checkpoints had been trained on a dataset build overwritten hours later (§6.5).

Table 2: Structuring the retry trajectory, against both baselines. Paired per-case differences with BCa 95% intervals, $n = 180$, 5-fold cross-validation; * marks an interval excluding zero. Against prose logs, structure wins on three of four measures including write-action naming. Against JSON, already partly structured, it wins on tool naming only.

	$F1^{all}_{name}$	$F1^{all}_{arg}$	$F1^{write}_{name}$	$F1^{write}_{arg}$
prose logs (absolute)	0.524	0.319	0.656	0.291
+ channel assignment	$+\mathbf{0.060}^{*}$ [+0.02,+0.11]	$+\mathbf{0.046}^{*}$ [+0.00,+0.09]	$+\mathbf{0.048}^{*}$ [+0.02,+0.09]	-0.016 [-0.05,+0.02]
plain JSON (absolute)	0.497	0.327	0.697	0.313
+ channel assignment	$+\mathbf{0.088}^{*}$ [+0.04,+0.14]	$+0.039$ [-0.00,+0.08]	$+0.007$ [-0.03,+0.05]	$-\mathbf{0.037}^{*}$ [-0.07,-0.00]

Table 3: Top: two layers of structure against the same plain-JSON baseline, and both together. Bottom: removing one assignment at a time from the full schema; a negative number means the removed assignment was doing work.

	$F1^{all}_{name}$	$F1^{all}_{arg}$	$F1^{write}_{name}$	$F1^{write}_{arg}$
plain JSON (absolute)	0.497	0.327	0.697	0.313
tags only, no channel assignment	$+\mathbf{0.049}^{*}$ [+0.00,+0.09]	$+\mathbf{0.041}^{*}$ [+0.00,+0.08]	$+0.004$ [-0.04,+0.05]	-0.021 [-0.05,+0.00]
channel assignment	$+\mathbf{0.088}^{*}$ [+0.04,+0.14]	$+0.039$ [-0.00,+0.08]	$+0.007$ [-0.03,+0.05]	$-\mathbf{0.037}^{*}$ [-0.07,-0.00]
both	$+0.043$ [-0.00,+0.09]	$+0.014$ [-0.03,+0.06]	-0.000 [-0.04,+0.04]	$-\mathbf{0.029}^{*}$ [-0.06,-0.00]
removing one assignment from the full schema				
trajectory in `instr`, not `data`	$-\mathbf{0.091}^{*}$ [-0.14,-0.04]	$-\mathbf{0.063}^{*}$ [-0.11,-0.02]	-0.012 [-0.05,+0.02]	$+0.015$ [-0.02,+0.05]
requirement folded into `instr`	-0.038 [-0.08,+0.00]	-0.013 [-0.06,+0.03]	$+0.015$ [-0.03,+0.06]	$+0.027$ [-0.01,+0.06]
instance numbers stripped	$+0.019$ [-0.03,+0.07]	$+0.004$ [-0.04,+0.04]	$+0.006$ [-0.03,+0.05]	-0.023 [-0.05,+0.00]

5 RESULTS

In Table 2 , we provide our main results where we show that structuring the trajectory helps, against both baselines. Here all arms share the same base model, LoRA recipe, folds, and failed trajectories, so only the trajectory's presentation differs. Against plain JSON, channel assignment gains 0.088 F1 on tool naming, and $F1^{all}_{arg}$ moves the same way without clearing zero. Against prose logs the gain is 0.060 on tool naming, and it also clears zero on write-action naming, a secondary check (§4.3). Neither contrast improves argument-exact matching; see §5.6.

Table 3 shows that channel assignment and tag syntax do not provide additive gains. It isolates the tag syntax from the channel assignment: plain JSON, tags with no channel assignment, channel assignment with no new syntax, and both together. Tags alone recover 0.049 on tool naming, significant on both all-action measures. Channel assignment alone recovers more, 0.088. Both together score 0.043, no better than either layer alone and on $F1^{write}_{arg}$ slightly worse than plain JSON. The effects do not add, and channel assignment accounts for the larger gain.

The bottom half of Table 3 removes one assignment at a time. Assigning tool output to data accounts for the largest ablation effect. Moving trajectory history back into the instruction channel costs the most, 0.091 on tool naming and 0.063 on argument matching, both significant. Folding requirement feedback back into instruction costs less, and does not clear zero. Stripping instance numbers off tags costs nothing measurable. Among the tested assignments, separating tool output from instructions has the largest measurable effect

Two facts bear on whether this schema is portable. First, the LLMON envelope has a hard vocabulary prerequisite: on a base whose tokenizer sees a channel delimiter as roughly eleven sub-word tokens rather than one, the envelope fails on most cases, parsing only 127 of 180 outputs, 29 more ending in a bare end-of-text token, and 22 more looping on syntax fragments, with validation loss giving no warning of any of it.

Table 4: The same channel assignment expressed in role labels native to every checkpoint, against an identical prompt with the labels removed. Neither base shows a reliable gain, and the interaction between the two bases does not clear zero on any measure.

Δ (channel labels − none)	$F1^{all}_{name}$	$F1^{all}_{arg}$	$F1^{write}_{name}$	$F1^{write}_{arg}$
markup-pretrained base	$+0.034$ [-0.01,+0.08]	$+0.007$ [-0.02,+0.04]	$+0.024$ [-0.02,+0.08]	-0.005 [-0.03,+0.02]
				sign-test p: 0.688, 0.581, 1.000, 1.000
vanilla base	-0.005 [-0.05,+0.04]	-0.009 [-0.04,+0.03]	-0.035 [-0.08,+0.01]	-0.020 [-0.05,+0.01]
				sign-test p: 0.555, 0.069, 0.009, 0.268
interaction (pretrained − vanilla)	$+0.040$ [-0.02,+0.10]	$+0.017$ [-0.03,+0.07]	$+0.059$ [-0.01,+0.13]	$+0.014$ [-0.02,+0.05]
				sign-test p: 0.353, 0.773, 0.187, 0.788

Table 5: Attack success rate, pooled over $n = 180$, at three injection strengths. Every structured rendering resists better than plain JSON at every strength. The forged-boundary attack (spoof) is not the strongest one on any rendering; the generic authority framing is.

rendering	bare	authority	spoof
plain JSON	36%	44%	37%
prose logs	19%	25%	24%
channel envelope	22%	36%	19%
role-labelled	14%	31%	10%

Second, expressing the identical assignment as role labels, native single tokens on every checkpoint, against an identical prompt with the labels removed, does not reproduce the gain. Table 4 shows nothing clearing zero on the markup-pretrained base, and on the base that never saw the markup the comparison is negative on all four measures, significantly so on write-action naming. The interaction between the two bases also does not clear zero on any measure, so the apparent difference in sign is not reliable at this sample size. These results suggest that the observed effect depends on the combination of the model and markup representation; semantic labels alone do not reproduce it.

The data channel provides a partial defense against prompt injection. §5.3 shows the data assignment carries the accuracy effect, and §3.1 justifies it as a trust boundary, a security claim accuracy cannot test. We inject an instruction into one tool result at three strengths, bare, an authority framing that claims to come from the system, and a spoof that forges the rendering's own boundary syntax. The injected instruction names a tool absent from the benchmark, so any occurrence in the output is the attack landing unambiguously; we pair this with goal-retention F1 so outright refusal to act does not read as a good defence.

Table 5 gives the pooled attack success rate. Every structured rendering, prose channel envelope, and role labels alike, resists better than plain JSON at every strength, but none resists fully. Authority framing produced higher attack success than boundary spoofing in every rendering, including JSON and prose. Thus, the reduction in attack success is not specific to resistance against forged markup delimiters. Channel separation reduces the frequency with which injected tool output is followed, but does not specifically prevent delimiter spoofing. Goal-retention F1 also decreases under attack across all renderings.

Where it does not help. The effect in Table 2 is confined to tool selection against plain JSON: it does not clear zero on write-action naming and is significantly negative on write-action argument matching against that baseline. The results suggest that structuring helps tool selection, but not the accurate prediction of order and item identifiers once a tool is selected. This limits the scope of the result: the schema may not help when failures primarily involve incorrect tool arguments. The comparison is paired on identical reference lists, so it is valid, but absolute levels depend on how those lists were built (§6.2), and a live environment with task reward would confirm this more strongly than a replay comparison.

Iterative retry does not consistently improve overall performance. We also close the loop: the model proposes one call, an environment answers, the result is appended as a new data block,

Table 6: Markup pretraining, within architecture-matched pairs, both arms reading an identical wrapper over identical content. Eight contrasts, none clearing zero.

Δ (markup-pretrained − control)	$\mathrm{F1}^{\text{all}}_{\text{name}}$	$\mathrm{F1}^{\text{all}}_{\text{arg}}$	$\mathrm{F1}^{\text{write}}_{\text{name}}$	$\mathrm{F1}^{\text{write}}_{\text{arg}}$
micro (MoE-hybrid)	$+0.012$ [-0.04,+0.07]	-0.019 [-0.06,+0.02]	-0.028 [-0.08,+0.02]	-0.002 [-0.03,+0.02]
3B (dense)	-0.021 [-0.07,+0.03]	$+0.021$ [-0.02,+0.07]	-0.003 [-0.04,+0.03]	$+0.008$ [-0.02,+0.04]

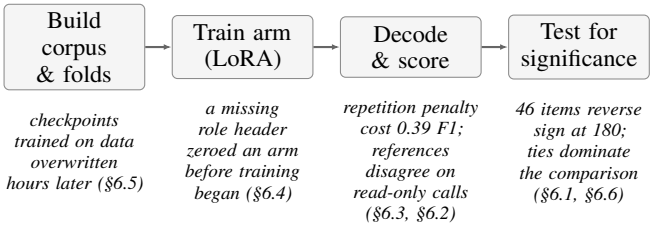


Figure 2: Where each finding below struck in the campaign that produced our earlier, partly false result. The same four stages, run correctly, are what let us report the result in §5 instead.

the attempt count increments, and the harness checks the typed stopping criteria. Against a replay environment built from all observed call-result pairs (72% coverage), the loop is worse than submitting the failed run unchanged on three of four measures, -0.055 on $\mathrm{F1}^{\text{all}}_{\text{name}}$, while write-action argument exactness improves, $+0.068$: seeing real results helps the agent copy identifiers correctly, but it also adds calls that should not have been made, and F1 charges for precision.

Although the retry loop does not improve overall performance, typed stopping criteria provide explicit fields that the harness can evaluate directly. The repeated-call rule fires in 43 of 46 episodes after a mean of 5.7 iterations, indicating that the agent often repeats calls without reaching a correction. Training a policy on failure-conditioned pairs, using the numbered tags to make "the corrected next action" expressible, raises write-action naming on the loop's own contributions from 0.344 to 0.515, yet still does not beat doing nothing, because the policy cannot reliably terminate; an oracle that replays the reference actions scores a perfect 1.000, confirming the harness itself is not the bottleneck. Two limitations remain open: our training data omitted the prefixes where every reference action was already satisfied, exactly the cases that teach termination, and we have no unstructured control for this setting.

6 FINDINGS

Several conclusions from our earlier single-split experiments did not hold under the corrected evaluation. Those experiments suggested a monotonic improvement across representations, a pretraining benefit on all four measures, and a consistent mechanism. We trace these discrepancies to the pipeline artifacts described below. Each turned out to rest on one of the pipeline artifacts below. We report them because the setup that produced them, reference-list scoring of a tool-using agent, a few dozen test items, one training run per arm, is close to standard practice in this literature, and in every case the wrong reading was the natural one.

Test-set size can reverse a significant effect. On 46 items, single-token tags beat punctuation tags on $\mathrm{F1}^{\text{all}}_{\text{name}}$ by $+0.124$, significant. At 180 items the same contrast is -0.077, significant in the opposite direction. We also incorrectly expected a second contrast to replicate: although it fell short of significance at $n = 46$, its estimated effect exceeded the minimum detectable effect later calculated for $n = 180$, cross-validation would confirm it. It did not; the point estimate moved toward zero instead. The minimum detectable effect describes the size of an effect the study could detect; it does not establish that the effect exists.

Reference lists disagree with themselves about read-only calls. In one 46-case split, 29 references contain only write actions, while 17 also contain reads. Because prompts request complete action lists, identical lookup predictions are penalized in the first group but rewarded in the second. This

label inconsistency fully explains an apparent gap between write-action and all-action F1, so we report write-action F1 as a secondary check.

A repetition penalty penalized repeated JSON keys. A repetition penalty of 1.2, harmless for prose, penalized valid repeated JSON keys. Models then stopped early, reducing write-action F1 by 0.39, a larger effect than any representation result and one that differed systematically by format.

Two artifacts of format recognition. Two setup errors also affected results. A missing role header made a stop token the most likely first output, invalidating one training arm despite apparently normal completion loss. Unseeded instance identifiers made rebuilt datasets irreproducible, preventing reliable cross-validation.

Training data overwritten beneath saved checkpoints. Finally, earlier checkpoints referenced training data that were later overwritten when the dataset builder was rerun. Old checkpoints reproduced their recorded scores, but newly trained same-setting checkpoints behaved differently on the current data. This explains earlier performance trends and termination failures; after correcting the dataset build, all conditions terminate normally. We now save dataset SHA-256 hashes with checkpoints and prevent overwriting referenced data.

High tie rates motivate multiple statistical tests. Between 54 and 149 of 180 paired items are exact ties, where two arms return call lists that score identically, which is structural to reference-list scoring at this scale. When many cases are tied, the mean difference can be sensitive to a small number of high-variance cases, while the sign test focuses on non-tied cases. The two tests can therefore reach different conclusions. The interval captures effect magnitude and uncertainty, while the sign test summarizes the direction of non-tied outcomes. Reporting both helps distinguish small, consistent effects from larger differences driven by a few cases. We report the paired difference with its interval, the win, lose, tie split, a sign test, and a Wilcoxon test for every contrast in this paper.

7 Discussion

Typed retry trajectories improve tool-call-name F1 over plain JSON and prose logs by 0.088 and 0.060, respectively. The gain is concentrated in tool selection, and channel assignment accounts for more of it than tag syntax alone. Two architecture-matched checkpoint pairs show no consistent benefit from markup pretraining. In prompt-injection tests, structured representations reduce attack success relative to plain JSON, but do not eliminate it. Although we focus on tool-use recovery, the schema's core idea—separating trusted instructions and constraints from untrusted external information—is broadly applicable. For example, it could support memory writes (distinguishing user-provided facts from retrieved information), record updates (separating constraints from data), or delegation to other agents (clarifying task boundaries).

For practitioners, this points to a specific and checkable recommendation: adopt the channel assignment first, since that is where the accuracy effect lives, and treat the tag syntax as the secondary choice. For researchers running a similar comparison, cross-validate rather than relying on a few dozen items, use at least two matched base pairs before attributing anything to pretraining, give the baseline the same wrapper as the treatment, and report both a bootstrap interval and a sign test, since the two can disagree at this scale. §6 shows why each of these checks earns its place: skipping any one of them, in our own earlier campaign, turned this same comparison into a larger and cleaner-looking result than the one we report here.

8 Conclusion

We introduce a schema for retry trajectories that assigns seven types of information to four channels: instruction, data, requirement, and execution. On failed tool-use trajectories from tau-bench, this representation improves tool-call-name F1 over plain JSON and prose logs, with gains concentrated in tool selection rather than argument accuracy. Channel assignment accounts for more of the gain than tag syntax alone, while markup pretraining shows no consistent benefit across two matched checkpoint pairs. Structured representations also reduce, but do not eliminate, the success of injected instructions in tool outputs. These findings support explicit role separation in retry histories while highlighting the need for careful evaluation and further testing beyond the studied setting.

REFERENCES

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023a. doi: 10.1145/3591300. URL https://doi.org/10.1145/3591300.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. In *Proceedings of the ACM on Programming Languages*, 2023b.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding llms the right way: fast, non-invasive constrained generation. In *ICML'24: 41st International Conference on Machine Learning*, 2024.

Anwoy Chatterjee, H S V N S Kowndinya Renduchintala, Sumit Bhatia, and Tanmoy Chakraborty. POSIX: A prompt sensitivity index for large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 14550–14565, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.852. URL https://aclanthology.org/2024.findings-emnlp.852/.

Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. More than you've asked for: A comprehensive analysis of novel prompt injection threats to application-integrated large language models. *CoRR*, abs/2302.12173, 2023. doi: 10.48550/ARXIV.2302.12173. URL https://doi.org/10.48550/arXiv.2302.12173.

Guidance. A guidance language for controlling large language models., 2025. https://github.com/guidance-ai/guidance.

Michael Hind, Basel Shbita, Bo Wu, Farhan Ahmed, Chad DeLuca, Nathan Fulton, David Cox, and Dan Gutfreund. Llmon: An llm-native markup language to leverage structure and semantics at the llm interface, 2026. URL https://arxiv.org/abs/2603.22519.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685. ICLR.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. Llms get lost in multi-turn conversation, 2025. URL https://arxiv.org/abs/2505.06120.

LangChain. langchain: The agent engineering platform., 2023. https://github.com/langchain-ai/langchain. Accessed: 2026-03-17.

Yuanye Liu, Jiahang Xu, Li Lyna Zhang, Qi Chen, Xuan Feng, Yang Chen, Zhongxin Guo, Yuqing Yang, and Peng Cheng. Beyond prompt content: Enhancing llm performance via content-format integrated prompt optimization, 2025. URL https://arxiv.org/abs/2502.04295.

LlamaIndex. Llamaindex, 2024. https://github.com/run-llama/llama_index. Accessed: 2026-03-17.

Moran Mizrahi, Guy Kaplan, Dan Malkin, Rotem Dror, Dafna Shahaf, and Gabriel Stanovsky. State of what art? a call for multi-prompt llm evaluation. *Transactions of the Association for Computational Linguistics*, 12:933–949, 08 2024. ISSN 2307-387X.

POML, 2025. URL https://github.com/microsoft/poml. Accessed: 2026-03-17.

Ambrish Rawat, Stefan Schoepf, Giulio Zizzo, Giandomenico Cornacchia, Muhammad Zaid Hameed, Kieran Fraser, Erik Miehling, Beat Buesser, Elizabeth M. Daly, Mark Purcell, Prasanna Sattigeri, Pin-Yu Chen, and Kush R. Varshney. Attack atlas: A practitioner's perspective on challenges and pitfalls in red teaming genai, 2024. URL https://arxiv.org/abs/2409.15398.

Johann Schopplich. Token-oriented object notation (toon), 2025. https://github.com/toon-format/toon.

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. 2024. URL https://par.nsf.gov/biblio/10520219.

Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models, 2023. URL https://arxiv.org/abs/2307.09702.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed H Awadallah, Ryen White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. *Workshop on Large Language Models for Agents*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629. ICLR.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Yuge Zhang, Nan Chen, Jiahang Xu, and Yuqing Yang. Prompt orchestration markup language, 2025. URL https://arxiv.org/abs/2508.13948.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

A Reproducibility Statement

We will release the renderer, fold splits, evaluation harness, oracle policy, per-item predictions for every arm, and the analysis script. Each of these is a single deterministic code path (§4), so every reported number can be recomputed from what we release. Two defects should be fixed first: one list-separator delimiter is spelled three inconsistent ways in the codebase, and 24 of 460 trajectories fail conversion because of an unescaped slash in free text.

B Ethics Statement

This work uses a public benchmark of synthetic customer-service dialogues; no human subjects, personal data, or sensitive attributes are involved. We also test whether putting tool output in a `data` channel resists an instruction injected through it (§5.5); it lowers the attack success rate on every strength we tried, but should not be read as a security guarantee.