ODFUSE: OBSERVATION-DEPENDENCY-GUIDED STEP FUSION FOR EFFICIENT TOOL-USE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

A multi-step tool-use LLM agent must end a step before any operation with an *observation dependency*: it cannot determine how it uses an intermediate result until it observes that result. Existing agents such as vanilla CodeAct satisfy this dependency only implicitly: they stop after every short block of code and pay for a new round of model reasoning at every stop. Fusing regardless removes those stops, but the agent then acts on evidence it has not seen. We introduce ODFuse, a prompt-only extension of CodeAct agents that constructs each execution step according to observation dependency: it fuses observation-resolved operations and defers unresolved ones until their required evidence has been observed. On two popular tool-use benchmarks (AppWorld and MCP-Universe) with four backbones, ODFuse takes 27–78% fewer steps than vanilla CodeAct, which cuts per-task cost by up to 50% in most settings at a limited loss of accuracy. It also commits 95% fewer over-fusions than an agent that fuses regardless, on the benchmark where that agent errs most.

1 INTRODUCTION

Multi-step tool-use agents (Schick et al., 2023; Li et al., 2023; Qin et al., 2024; Zhou et al., 2024) operate in everyday applications (scheduling, emails, payments) under *partial observability*: the agent sees application state only through what its own calls return. It alternates between acting and observing (Yao et al., 2023), and where each step ends is its own choice. A step can run several *operations*, each a tool call or a computation on a call's result, such as parsing or filtering. An operation carries an *observation dependency* when the agent must observe an intermediate result before it can determine how the operation uses that result. Operations without this dependency may join the current step; the others must wait. Asked to *update RSVPs as per their latest replies* for a reunion party, the agent can fetch the replies but cannot write the update in the same step. Only after the model has read a reply such as Tracy's 'Okay, my plans got canceled, I'll be at the reunion!' can the next step mark her as coming (Figure 1c).

Existing agents satisfy this dependency only implicitly, at the price of more steps and higher cost. They keep each step to a small chunk of operations, and so stop far more often than the dependency requires. Each stop costs a new round of reasoning tokens. When the next operation does not need to observe the previous result, those tokens are wasted. Fusing regardless removes the stops the dependency requires: the agent then acts on evidence it has not seen, and a wrong guess either raises an error or silently leaves the task in the wrong state. Section 3 measures both costs.

We introduce ODFuse, a step-construction rule for CodeAct agents (Wang et al., 2024) that fuses as many operations into one step as observation dependency allows, and so cuts per-task cost sharply while keeping accuracy close to vanilla CodeAct's; it is given to the model as a prompt instruction and needs no additional component. To realize this rule, we develop *eager-ready fusion*: each step takes every operation that is ready, meaning the observations it needs are already in hand, including the lookups that deferred operations wait on, so the evidence those operations need is observed by the next step.

We evaluate ODFuse on two tool-use agent benchmarks (AppWorld and MCP-Universe) with four reasoning-augmented LLM backbones (Gemini-2.5-Pro, GPT-5, Gemma-4-31B, and Qwen3.6-27B). Compared with vanilla CodeAct, ODFuse takes 27–78% fewer steps, which cuts per-task cost by up to 50% in most settings at a limited loss of accuracy. It also commits 95% fewer over-fusions than

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.

Pink is what a step wrote without observing; ~~grey~~ is the observing step it skipped; blue is what ODFuse observed before writing the next step.

	Fuses across the observation dependency (AggFuse)	**Stops at the observation dependency** (ODFuse)

(a) **Tool choice**	~~`api_docs.show_api_descriptions('simple_note')`~~
	`notes = simple_note.show_notes(...)`
	not observed the API list → `No API named 'show_notes' found in the simple_note app.`
		`api_docs.show_api_descriptions('simple_note')`
		observed `[..., search_notes, show_note, ...]`
		`notes = simple_note.search_notes(query=...)`

(b) **Parsing**	`note = simple_note.show_note(...)['content']`
	`for b in note.split('\n\n'):`
	`  name = b.split('\n')[0].split(': ')[1]`
	`  # line 1 = a date, line 2 = 'a, b'`
	not observed the note's layout → `IndexError: list index out of range`
		`note = simple_note.show_note(...)['content']`
		observed `Meeting Name: Project Kickoff Meeting Attendees: - Denise - Matthew ...`
		`for b in note.split('Meeting Name: ')[1:]:`
		`  name = b.split('\n')[0]`
		`  who = [l[2:] for l in b.split('\n') if l.startswith('- ')]`

(c) **Semantic analysis**	`msgs = phone.search_text_messages(...)`
	`if re.search(r"yes|coming", m):`
	`  rsvp[name] = 'yes'`
	not observed Tracy's reply, `Okay, my plans got cancelled. I'll be at the reunion!` → matches neither word, so her row stays unknown
		`msgs = phone.search_text_messages(...)`
		observed `Tracy: "Okay, my plans got cancelled. I'll be at the reunion!"`
		`rsvp['Tracy'] = 'yes'  # no pattern needed`

Figure 1: Representative examples of observation dependency on AppWorld's tools: a step that fuses across the dependency (left; AggFuse, Section 3) and ODFuse's steps, which stop at it and observe (right). Pink marks code the fused step wrote without observing, grey strikethrough marks the observing step it skipped, and blue marks what ODFuse observed. Where each step stops and what it observes come from the runs; the code is shortened, and the code after each stop in (b) and (c), like the error in (b), is illustrative.

an agent instructed to fuse as far as it can, on the benchmark where that agent errs most. A fusion detector, run on the emitted code, confirms that the criterion is what governs where its steps end.

2 Related Work

Fusing operations by data dependency. Prior work fuses several tool operations into one step by *data dependency*: a later operation consumes a value an earlier one returns, and a runtime variable can carry it without a further model decision. CodeAct exploits this within a block, since with executable code as the action space a value one call returns reaches the next as a variable (Wang et al., 2024); later work treats code as a concise, verifiable medium for tool invocation (Lu et al., 2025). Planners make the same dependency explicit across a task. Graph planners run each round's independent calls in parallel and write a dependent call only after observing its inputs (Zhu et al., 2025; Wu et al., 2025b). Compiler-style planners batch calls before execution (Kim et al., 2024; Singh et al., 2024), and ReWOO and Chain-of-Abstraction fix the plan or the reasoning chain first and substitute observations into it afterward (Xu et al., 2023; Gao et al., 2025). LLMCompiler also replans, ending a plan once its planner can plan no further without executing it. These methods account for observation dependency implicitly by waiting for results when a round or plan ends. ODFuse makes it the explicit criterion for where a step ends: it adds a dependent operation to the step once the history fixes how the operation uses the result, and stops before one that must first observe it (Section 3 shows this criterion governs both cost and correctness).

Complementary efficiency levers. Gating methods weigh the model's own knowledge against external calls to decide when invocation is warranted (Qian et al., 2025; Xu et al., 2025a), work on tool retrieval and representation examines which tool is selected or exposed (Su et al., 2025; Xu et al., 2024; Shi et al., 2025), and further methods train for tool-use generalization or personalization (He et al., 2025; Xu et al., 2025b) or refine agent prompts and tool descriptions (Wu et al., 2025a; Yuan et al., 2025). A further family shrinks what each call consumes or produces: compressing

	Under-fusion		Over-fusion		Score		Cost (m$)	
Task set	CA	AF	CA	AF	CA	AF	CA	AF
AW NORMAL	8.23	0.64	0.04	5.78	79.8	19.6	103.5	121.8
AW CHALLENGE	11.38	0.99	0.16	10.67	52.0	10.8	126.0	169.0
FINANCIAL	0.53	0.07	0.00	0.80	72.7	55.2	58.0	72.1
LOCATION	5.83	0.49	0.03	1.74	81.7	75.7	97.9	74.9
REPOSITORY	9.18	0.32	0.00	1.14	65.1	61.9	123.4	87.6
3D DESIGN	6.89	0.16	0.00	1.53	68.4	55.3	74.6	76.8

Table 1: Under-fusions and over-fusions per task, beside the score and cost of the same runs, for vanilla CodeAct (CA) and AggFuse (AF) under Gemini-2.5-Pro on two AppWorld (AW) splits and four MCP-Universe domains (Section 5.1, which also defines Score and Cost; m$ is milli-USD per task).

input prompts (Jiang et al., 2023; 2024), pruning or compressing agent history (Xiao et al., 2025; Kang et al., 2025), or shortening per-call reasoning (Han et al., 2025; Qiao et al., 2025; Xu et al., 2025c). They decide whether a call is made, which tool it uses, or how many tokens it consumes and produces; ODFuse decides how many observation-resolved operations run inside one step, and is complementary to all of them.

3 OBSERVATION DEPENDENCY GOVERNS BOTH COST AND CORRECTNESS

An operation has an *observation dependency* on an earlier one when how it uses the earlier result cannot be determined from the current instruction and the execution history, so the model must read the result before it can decide what to write (Figure 2, right). Figure 1 gives representative examples, in which the model must first observe which call exists, how a free-form result is laid out, or what a message means. Every other dependency can be resolved without an observation: either the later operation needs only the state the earlier one leaves behind, such as logging in before using an app or creating a folder before a file inside it, or code written in advance decides how the later operation uses the earlier result (reading a field the response schema names, selecting by a criterion the task states, parsing a known format, or branching on a condition the task gives). Either way, statement order carries the state and runtime variables carry the results, so both operations can share one step (left).

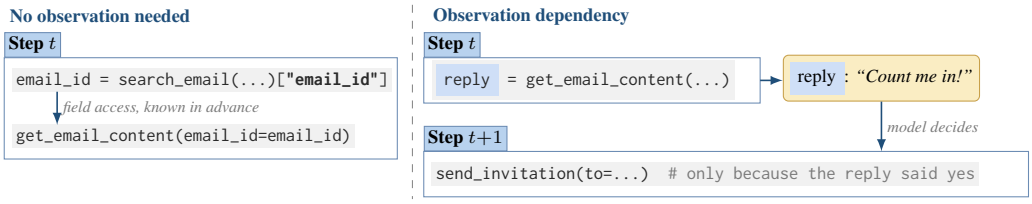


Figure 2: The two forms of a dependency. *Left*: no observation is needed; the input is a field of the result, so both operations share one step. *Right*: observation dependency; whether to invite depends on what the reply says.

When an agent fuses two operations that an observation dependency separates, it writes the later one without the observation it needs, which can lead to incorrect behavior. We call this an *over-fusion*. In Figure 1, the fused steps in the left column presume which call exists, how a free-form result is laid out, or what a message means; the right column shows the same operations written after the observation. To see what over-fusion costs, we run an aggressive-fusion probe (AggFuse): the same vanilla CodeAct agent simply instructed to pack as many operations as possible into each step. We count its over-fusions with a *fusion detector* that extracts candidate over-fusions from the emitted code. Static analysis settles the candidates its rules cover, and an LLM judge settles the rest (Appendix B). AggFuse over-fuses often and is also less accurate (Table 1): across the six task sets it commits up to 10.7 over-fusions per task and loses between 3.2 and 60.2 percentage points (pp) of accuracy.

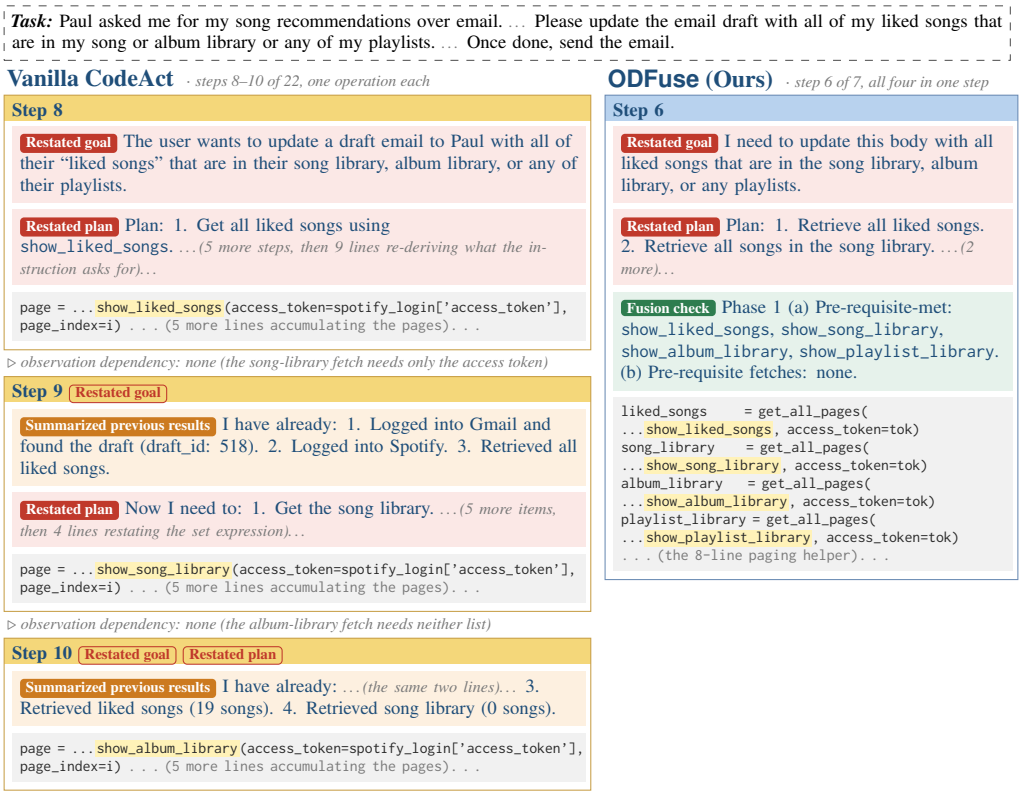


Figure 3: Reasoning repeated at under-fusions. In steps 8–10, Vanilla CodeAct (left) fetches one list per step, and each step restates the goal and the plan (steps 9 and 10 also re-summarize earlier results); ODFuse (right) fetches all four lists in one step and restates the goal and the plan once.

The naive way to avoid over-fusion, stopping after each short group of operations as vanilla CodeAct does, leads to high cost. Each stop costs a fresh round of hidden deliberation that restates the goal and the plan and re-summarizes earlier results (Section 5; Figure 3). When the next step uses nothing from the observation a stop paid for, we call that stop an *under-fusion*. We count them at the stops between two tool-calling steps and, on AppWorld, at the stops next to a documentation lookup, leaving out stops after a failed step; a stop is under-fused when nothing the next step does with the previous step's output needs that output to have been observed (Appendix B). By this count, most such stops in a vanilla CodeAct trajectory are under-fused on both benchmarks (Table 1; Appendix B).

4 ODFUSE

ODFuse is a CodeAct agent that instructs the model to fuse operations into one step as far as observation dependency allows, so that a task takes fewer steps while the agent still acts on evidence it has observed. It realizes this by *eager-ready fusion*: every ready operation, meaning the observations it needs are in hand, enters the current block, including the lookup each deferred operation is waiting on whenever that lookup is itself ready. The right column of Figure 1 shows where ODFuse's steps end on real AppWorld tasks.

The instruction builds each step's code block in two phases (Figure 4); Figure 5 shows how the two phases handle four schematic operations, with vanilla CodeAct alongside. The first phase decides what enters the block: every operation whose observation dependency is already satisfied and, for each operation it defers, a fetch of the observation that operation is missing, provided the fetch can itself be written from observations already in hand. Here it admits `A` and `B`, defers `C` and `D`, and adds a lookup of `X`, because `D` needs `X` and the lookup itself needs nothing unobserved; `C` needs only `A`'s result, which `A`'s run provides. Without that lookup, `D` would reach the next step with its dependency

[1] https://anonymous.4open.science/r/odfuse-anon-B865/

```
[ODFuse - Observation Dependency Instruction (condensed)]

Plan each step's code block in two phases. The step bundles exactly what Phase 1 selects -
    neither smaller (which skips fusible operations) nor larger (which includes
    operations Phase 1 rejects).

* Phase 1 (select operations). Include (a) every operation whose observation dependency is
    already satisfied, and (b) for every operation deferred to a later step, a fetch of
    the observation it is missing, if that fetch can be written from observations already
    in hand so that the next step can run it.

* Phase 2 (observation policy). For each selected operation, print what the next step
    needs to read; a value used only as an argument passes on as a variable without
    printing.
```

Figure 4: ODFuse's instruction, condensed to the parts that construct a step: which operations enter the block, including a fetch for each deferred one (Phase 1), and which results it prints for the next step (Phase 2). The full prompt is in our released code.[1]

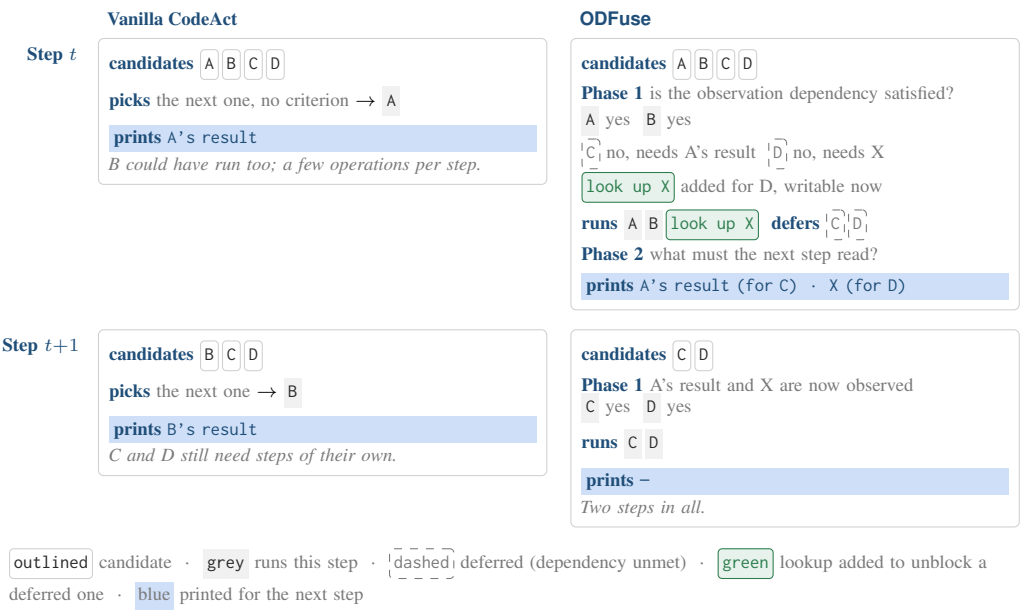


Figure 5: Step construction in vanilla CodeAct and ODFuse on four schematic candidate operations; time runs downward. ODFuse selects the operations to run and the results to print according to the two phases of Figure 4.

still unmet, and the agent would have to defer it again or over-fuse it. The second phase decides what the block prints: what the next step must read, here A's result and X, while a value used only as an argument passes on as a variable. Step $t+1$ then runs C and D together, and the four operations take two steps. Vanilla CodeAct runs A alone at step t and B at step $t+1$, so C and D still need steps of their own.

We implement the procedure as a prompt instruction inside a standard CodeAct loop (Wang et al., 2024): the model writes a single block, a stateful runtime executes it against the available apps, and whatever the block prints becomes its observation and is appended to the history for the next pass. ODFuse needs no additional model call, verifier, or planning module, so it applies unchanged to any CodeAct agent.

	Vanilla CodeAct				AggFuse				ODFuse			
	AppWorld		MCP-U		AppWorld		MCP-U		AppWorld		MCP-U	
Backbone	TGC	m$	AE	m$	TGC	m$	AE	m$	TGC	m$	AE	m$
Gemini-2.5-Pro	65.9	114.7	72.0	88.5	15.2	145.4	62.0	77.9	64.7	93.8	70.6	79.7
GPT-5	75.5	240.7	69.0	115.1	13.5	413.5	71.2	65.3	76.4	129.4	72.8	89.0
Gemma-4-31B	73.1	11.3	74.6	8.9	37.9	7.6	66.7	2.8	76.7	9.8	75.5	5.8
Qwen3.6-27B	61.8	52.9	70.4	41.3	24.8	72.7	61.4	29.7	79.9	42.8	70.4	37.0

Table 2: Task Goal Completion (TGC, %) on AppWorld (mean of TEST_NORMAL/TEST_CHALLENGE) and Average Evaluator Score (AE, %) on MCP-Universe (MCP-U; macro-averaged over its four domains), with per-task cost (m$) in milli-USD under ideal prompt-cache pricing. Appendix A gives the values for each of the 24 configurations.

5 Evaluation

5.1 Experimental Setup

Benchmarks. We evaluate ODFuse on **AppWorld** (Trivedi et al., 2024) (TEST_NORMAL, 168 tasks; TEST_CHALLENGE, 417 tasks) and **MCP-Universe** (Luo et al., 2025) (Financial Analysis, 40; Location Navigation, 35; Repository Management, 28; 3D Design, 19). In MCP-Universe, we restrict to domains where vanilla CodeAct on Gemini-2.5-Pro scores above 50%, excluding Web Search (24%) and Browser Automation (30%); below that, the baseline completes too few tasks for a cost–accuracy trade-off to be read off the domain at all. For MCP-Universe, we complete the tool descriptions with the output schemas that the released wrappers omit, identically for all methods (Appendix C.1). We refer to each AppWorld split and each MCP-Universe domain as a *task set*, and to a (backbone, task set) pair as a *configuration*.

Backbone models. We use four reasoning-augmented backbones from three developers, spanning both access regimes: the proprietary Gemini-2.5-Pro (Google DeepMind, 2025) and GPT-5 (OpenAI, 2025), and the open-weight Gemma-4-31B (Google DeepMind, 2026) and Qwen3.6-27B (Qwen Team, 2026), both served locally. Decoding is fixed per backbone and identical across the compared methods; Appendix C.2 gives the setting for every backbone.

Compared methods. We compare three prompt-only variants that bracket the fusion axis, all sharing the same backbone, decoding settings, and tool interfaces to isolate the effect of the fusion scaffolding. **Vanilla CodeAct** (Wang et al., 2024), the most accurate on average of the training-free baselines in Table 3, is the CodeAct loop under the benchmark's released prompt. Its step policy is the instruction *write small chunks of code and only one chunk of code in every step*, which restates the original CodeAct prompt's *attempt fewer things at a time*; AggFuse and ODFuse modify this system prompt. **AggFuse** is a vanilla CodeAct variant whose prompt instructs the agent to look as far ahead as possible and fuse as many operations as it can into a single step (prompt in our released code). **ODFuse (ours)** applies selective fusion under the explicit observation-dependency check of Section 4. Table 3 further compares against six training-free efficiency methods from prior work, on Gemma-4-31B.

Evaluation metrics. For AppWorld, we use *Task Goal Completion* (TGC) (Trivedi et al., 2024): binary pass/fail based on state-based unit tests. For MCP-Universe, we use *Average Evaluator Score* (AE) (Luo et al., 2025): fraction of evaluators passed per task (partial credit). Cost is per-task spend, computed from token-type accounting under an ideal prompt cache that bills every repeated prefix at the cached-input rate (Table 7); since whether a provider serves a prefix from cache depends on server load and provider policy, we assume the theoretically attainable 100% prefix hit rate. Appendix A.1 re-runs the same runs at the providers' actual charges and with the cache disabled; the trend is unchanged, with ODFuse's savings equal to or larger in most settings.

5.2 Does ODFuse Reduce Steps and Cost?

ODFuse takes fewer steps than vanilla CodeAct in every one of the 24 configurations, by 27–78%, and costs less per task in 22 of them, by up to 50% (Table 6); averaged over each benchmark's

	AW NORMAL		AW CHALLENGE		FINANCIAL		LOCATION		REPOSITORY		3D DESIGN	
Method	TGC	m$	TGC	m$	AE	m$	AE	m$	AE	m$	AE	m$
Vanilla CodeAct	83.3	8.91	62.8	13.70	79.9	4.02	87.4	6.46	67.8	19.86	63.2	5.42
ODFuse (ours)	84.5	8.18	68.8	11.39	79.9	2.89	85.3	6.10	63.2	10.02	73.7	4.25
AggFuse (ablation)	47.0	5.69	28.8	9.47	72.7	1.37	83.5	1.64	50.0	5.39	60.5	2.97
One-line rule (ablation)	76.2	6.62	57.3	9.63	70.8	2.66	87.1	5.70	64.7	9.16	68.4	2.55
ReAct	67.9	21.04	48.0	20.80	61.7	5.95	86.3	11.33	54.9	39.24	65.8	5.96
ReWOO	19.0	7.02	12.5	7.56	48.3	4.59	70.3	6.49	50.5	7.69	50.0	2.87
LLMCompiler	68.5	18.73	54.0	18.72	70.0	13.45	82.9	20.71	51.0	34.91	68.4	3.78
TALE-EP	79.2	9.29	54.4	13.01	55.7	5.63	78.9	20.79	55.3	35.71	68.4	3.84
Chain-of-Draft	82.7	11.22	57.6	15.17	59.2	4.06	86.1	14.92	53.7	34.94	63.2	3.40
ACON	81.5	12.25	56.1	15.57	51.7	8.79	84.5	10.59	58.1	32.58	60.5	6.56

Table 3: Efficiency baselines on Gemma-4-31B; metrics, pricing, and task sets as in Table 2. AW = AppWorld. Methods: vanilla CodeAct (Wang et al., 2024), ReAct (Yao et al., 2023), ReWOO (Xu et al., 2023), LLMCompiler (Kim et al., 2024), TALE-EP (Han et al., 2025), Chain-of-Draft (Xu et al., 2025c), ACON (Kang et al., 2025). The One-line rule (ablation) is vanilla CodeAct with its step policy replaced by a one-sentence statement of the criterion (Section 5.7).

task sets, it is cheaper for every backbone on both benchmarks (Table 2). The two configurations where cost rises are both Qwen3.6-27B: on FINANCIAL ANALYSIS ($+5.7\%$) the higher cost buys $+4.9$ pp accuracy, while on 3D DESIGN ($+27.5\%$) accuracy is unchanged. No prior efficiency method matches the saving: under one protocol on Gemma-4-31B, ODFuse is cheaper than every prior method in Table 3 except ReWOO on five of the six task sets, and on the sixth, 3D DESIGN, the three further methods that undercut it give up 5.3 to 10.5 AE for the saving. ReWOO is cheaper only because it rarely finishes a task: it fixes its plan before observing a result, scores 19.0 and 12.5 TGC on the two AppWorld splits, and trails vanilla CodeAct by 13 to 32 AE on MCP-Universe (Appendix A.3).

The steps ODFuse saves differ by benchmark, but on both, many of them are fetches of what later calls need (Appendix A.4). On AppWorld they make up 73–81% of the savings: documentation reads 32–47%, and logging in and lookups, which obtain the access token and the records that later calls use, another 30–42%. A call has an observation dependency on its API's documentation, but the read itself needs only the list of APIs, so ODFuse reads the documentation of several calls in one step, where vanilla CodeAct spends a step on each. On MCP-Universe, which has neither documentation reads nor logins, lookups make up 28–46% of the saving and writes most of the rest, with lookups dominating on LOCATION NAVIGATION and writes on 3D DESIGN. In each of the 24 configurations, ODFuse logs in, looks up, and writes through about as many distinct APIs as vanilla CodeAct, so the saving comes from fitting the same calls into fewer steps.

5.3 DOES IT PRESERVE TASK PERFORMANCE?

Accuracy drops no more than 5 pp below vanilla CodeAct in every configuration, and improves in 12 of the 24, by up to $+24.2$ pp (Qwen3.6-27B on AppWorld TEST_CHALLENGE; Table 6). Nine repeated runs on Gemma-4-31B confirm this: accuracy falls significantly on only one of the six task sets, by 1.29 pp (Section 5.5). Fusing without the dependency check does not preserve accuracy: across the four backbones, AggFuse's AppWorld TGC collapses to 13.5–37.9 against ODFuse's 64.7–79.9, and on MCP-Universe its AE is 1.6–9.0 pp lower (Table 2). We attribute much of this difference to where the steps end (Appendix B.2). The fusion detector of Section 3 (Appendix B) shows that ODFuse, on Gemini-2.5-Pro and TEST_NORMAL, stops needlessly far less often than vanilla CodeAct (2.06 against 8.23 under-fusions per task) and fuses too early more often but still rarely (0.26 against 0.04 over-fusions). Against AggFuse, which fuses without checking, it commits 95% fewer over-fusions on AppWorld. ODFuse is also more accurate than each of the six efficiency methods in Table 3 on five of the six task sets. On LOCATION NAVIGATION, ReAct (86.3) and Chain-of-Draft (86.1) score up to one point higher than ODFuse (85.3), but at $1.9\times$ and $2.4\times$ its cost (11.33 and 14.92 against 6.10 m$). More task-specific analysis is in Appendix A.2.

Task set	runs	Vanilla CodeAct	ODFuse	Δ (pp)	p	$\delta^\star$ (pp)
AW NORMAL	9	82.8 ± 1.3	83.5 ± 1.2	$+0.73$	0.703	0.8
AW CHALLENGE	9	61.2 ± 0.9	69.1 ± 0.5	$+7.99$	0.004	0.0
FINANCIAL	9	78.8 ± 1.6	76.7 ± 3.6	-2.04	0.207	4.4
LOCATION	9	86.1 ± 0.7	84.8 ± 0.8	-1.29	0.039	2.1
REPOSITORY	9	67.6 ± 1.0	66.1 ± 2.1	-1.46	0.250	3.2
3D DESIGN	9	58.2 ± 3.4	61.4 ± 2.7	$+3.22$	0.125	0.0

Table 4: Repeated runs on Gemma-4-31B. Accuracy (TGC on AppWorld (AW), AE on MCP-Universe) is the mean over runs with its 95% confidence interval; p is the p-value from a paired Wilcoxon signed-rank test over seed-matched runs; $\delta^\star$ is the accuracy loss ruled out by a one-sided 95% bootstrap lower bound on the mean of the nine per-seed differences.

5.4 ARE THE SAVINGS EXPLAINED BY FEWER REASONING ROUNDS?

Most of the savings come from generating fewer rounds of hidden reasoning. On the proprietary backbones, averaged over the six task sets, hidden reasoning is 5–17% of vanilla CodeAct's token volume but 52–76% of its bill (Figure 7), and the model reasons once per step, so every step removed is one round of reasoning not generated. Across the twelve configurations of Figure 7, ODFuse cuts 32–78% of the steps; reasoning tokens fall on ten of them, by as much as 55%, and within each backbone, apart from 3D DESIGN and the two configurations where reasoning rises, the task sets where reasoning falls more are the ones where cost falls more. The cost saving is smaller than the step saving in every configuration (Figure 6) because a fused step reasons over more of the task than any step it replaced. The two configurations where reasoning rises instead, Gemini-2.5-Pro on FINANCIAL ANALYSIS and REPOSITORY MANAGEMENT, are the two where reasoning takes the smallest share of the bill: input tokens are 38% and 49% of it, against at most 17% on GPT-5, and there the saving comes from the history that each removed step no longer feeds back to the model.

The efficiency baselines support the same explanation (Table 3). Methods that shorten what the model writes per step (TALE-EP, Chain-of-Draft) or compress the history it re-reads (ACON) do not target the number of steps, so the model still reasons once per step: they take 13.5–19.4 steps on AppWorld TEST_NORMAL, close to or above the 13.9 of vanilla CodeAct and far from the 5.8 of ODFuse, and none of the three is cheaper than ODFuse outside 3D DESIGN. ReWOO removes the stops by planning every call before execution, but its plan is a fixed list of calls with no loop, so it fails on AppWorld tasks whose number of calls depends on what earlier calls return, such as paging through a listing until it runs out (Appendix A.3). LLMCompiler instead replans: its planner ends a plan once it can plan no further without executing it, which respects observation dependency only implicitly, since nothing tells the planner which calls must wait for a result, the question ODFuse answers for each operation; each plan also costs a planning call and a joining call. It takes 16.4 steps on AppWorld TEST_NORMAL, more than vanilla CodeAct's 13.9 and ODFuse's 5.8, and costs more and scores lower than vanilla CodeAct on five of the six task sets, spending 3.4× what ODFuse does on LOCATION NAVIGATION (20.7 against 6.1 m$).

5.5 RUN-TO-RUN VARIATION

Repeated nine times, the single-run findings hold: on all six task sets ODFuse takes 26–65% fewer steps, a reduction significant on every one, and costs 7.6–42.0% less, while accuracy falls significantly on only one, LOCATION, by 1.29 pp (Table 4). We repeat the comparison on Gemma-4-31B, changing only the sampling seed, nine runs per method on each AppWorld split and each MCP-Universe domain, and report beside each difference $\delta^\star$, the tightest cap on accuracy loss the nine runs support: with 95% confidence ODFuse loses less than $\delta^\star$. TEST_CHALLENGE gains on both axes: +7.99 pp, ahead in all nine seeds ($p = 0.004$), for 15.3% less cost. The mean accuracy is lower on three domains: LOCATION by 1.29 pp, REPOSITORY by 1.46 pp, and FINANCIAL by 2.04 pp, with $\delta^\star$ of 2.1, 3.2, and 4.4 pp.

	Accuracy (%)					Steps					Cost (m$)				
Task set	CA	OD	OD_1	OD_2	OD_3	CA	OD	OD_1	OD_2	OD_3	CA	OD	OD_1	OD_2	OD_3
AW NORMAL	83.3	84.5	83.9	82.7	85.7	13.9	5.8	5.8	5.9	5.9	8.91	8.18	8.15	8.58	8.73
AW CHALLENGE	62.8	68.8	70.3	71.0	70.3	18.1	7.3	7.4	7.5	7.5	13.70	11.39	11.59	12.11	12.62
FINANCIAL	79.9	79.9	77.4	75.2	67.4	3.5	2.3	2.8	2.2	2.8	4.02	2.89	3.64	2.99	3.48
LOCATION	87.4	85.3	85.6	85.9	85.7	5.6	4.1	4.4	3.8	4.3	6.46	6.10	6.31	5.98	7.13
REPOSITORY	67.8	63.2	69.4	62.5	69.3	10.9	4.0	3.8	4.8	3.6	19.86	10.02	10.37	14.18	10.18
3D DESIGN	63.2	73.7	68.4	68.4	63.2	9.7	2.9	3.0	3.8	3.0	5.42	4.25	4.49	5.53	4.66

Table 5: Rewrites of the fusion instruction, on Gemma-4-31B, one run per configuration, over the AppWorld (AW) splits and MCP-Universe domains of Section 5.1, abbreviated in the row labels by their first word. CA = vanilla CodeAct and OD = ODFuse under the wording used throughout the paper, both the runs of Table 3; OD_1, OD_2, OD_3 = the three rewrites, in increasing order of departure from that wording. Metrics and pricing are as in Table 2; *Steps* is the mean model calls per task.

5.6 ROBUSTNESS TO PROMPT WORDING

Rewriting ODFuse's instruction three ways keeps its step saving and most of its cost saving, and costs at most 5.3 pp of accuracy in all but one of the eighteen (task set, rewrite) configurations (Table 5). We wrote three variations of ODFuse's fusion instruction in Figure 4 and ran each on Gemma-4-31B: OD_1 keeps the same sentences with different words, OD_2 phrases the same rules as commands to the model, and OD_3 rewrites the instruction from scratch. All eighteen configurations stay 20 to 69% below vanilla CodeAct in steps, and the cost saving holds in sixteen of the eighteen. The accuracy gain is robust on AppWorld too, where five of the six configurations beat vanilla CodeAct and the sixth is 0.6 pp below it, and on TEST_CHALLENGE the three rewrites finish within three tasks of one another (293–296 of 417), above both ODFuse's 287 and vanilla CodeAct's 262. On MCP-Universe, four of the twelve configurations beat vanilla CodeAct and one ties it; the other seven fall no more than 5.3 pp below it, except OD_3 on FINANCIAL ANALYSIS at 12.5 pp. On LOCATION NAVIGATION, where the nine runs of Section 5.5 already place ODFuse 1.29 pp below vanilla CodeAct, the three rewrites sit 1.5 to 1.8 pp below it, close to the original wording's 2.0 pp. On 3D DESIGN they score 5.3 to 10.5 pp below OD, whose single run of 73.7 lies above its nine-run mean of 61.4 (Section 5.5).

5.7 A ONE-LINE VERSION OF THE INSTRUCTION

To see the effect of eager-ready fusion's procedure, we give vanilla CodeAct only the criterion it follows, as a single instruction in place of its step policy: *batch as many operations as possible into one code block; stop when deciding what to do next requires inspecting an intermediate result*. On AppWorld and FINANCIAL ANALYSIS this One-line rule costs less than ODFuse on Gemma-4-31B but is less accurate: 76.2 and 57.3 against 84.5 and 68.8 TGC on the two AppWorld splits, and 70.8 against 79.9 AE, below even vanilla CodeAct (83.3, 62.8, and 79.9); on the other three domains the two are within 5.3 AE (Table 3). The criterion alone keeps the saving, but without the procedure that carries it out, it loses the accuracy that vanilla CodeAct keeps.

6 CONCLUSION

Which operations a tool-use agent may commit together is decided by *observation dependency*: an operation is ready only once the model can determine how it uses the intermediate results it depends on. ODFuse applies that criterion as a prompt-only step-construction rule, fusing what past observations already resolve and deferring the rest. It takes fewer steps than vanilla CodeAct, which cuts per-task cost in most settings at a limited loss of accuracy, and commits fewer over-fusions than an agent that fuses regardless (Section 5).

Ethics Statement

Benchmarks, licenses, and data. We evaluate on AppWorld (Trivedi et al., 2024) and MCP-Universe (Luo et al., 2025), both released under the Apache License 2.0, and use them consistently with their intended research use; the code we release builds on both and is released under the same license. All API calls were issued by the authors; no end-user data was collected or processed.

Potential risks. Reducing inference cost for tool-use agents lowers the barrier to deploying autonomous agents that act on user accounts (e.g., email, calendars, payments). ODFuse does not change the safety properties of the underlying model or scaffolding, and failure modes such as silent mis-execution after fusing past an unobserved result (Section 3) can cause side effects in stateful environments if deployed without appropriate guardrails such as dry-run previews, per-action confirmation for sensitive operations, and human-in-the-loop review for high-stakes tasks.

Reproducibility Statement

All code, prompts, decoding settings, and analysis scripts used to produce the reported results, including both parts of the fusion detector and its recorded verdicts, are in our anonymized repository at `https://anonymous.4open.science/r/odfuse-anon-B865/`; the de-anonymized link will be added to the camera-ready version. Appendix C gives the model versions, decoding settings, and step limits of every run, and every way our harness differs from the released benchmarks.

Two parts of MCP-Universe need care to reproduce. Location Navigation queries live mapping and routing services, both through the agent's tools and for the ground truth of its elevation checks, so we score every method against the same recorded tool responses. Each domain also reaches its service through a separate tool server that the harness launches by name, without a version; our release pins a version for every such server, though the runs reported here predate that pinning.

AI Use Statement

We used generative AI tools for the following tasks for which ICLR requires disclosure: developing conceptual frameworks, proposing and refining hypotheses, designing and giving feedback on research methodology, implementing methods, cleaning and reformatting experimental data, supporting qualitative analysis of agent trajectories, and interpreting results. An LLM judge is also part of the fusion detector, deciding the over-fusions and under-fusions its rules do not settle (Appendix B). We did not use them to generate synthetic data, to formulate or prove mathematical claims, or for translation.

We reviewed the AI-assisted work before relying on it: generated code was run against the benchmarks' own evaluators, and every reported number is regenerated from raw per-call logs by the released analysis scripts. The fusion detector uses the judge's verdicts as recorded, and they are released with the code. We take full responsibility for all content of this paper, including text, code, and figures produced with the aid of generative AI.

References

Silin Gao, Jane Dwivedi-Yu, Ping Yu, Xiaoqing Ellen Tan, Ramakanth Pasunuru, Olga Golovneva, Koustuv Sinha, Asli Celikyilmaz, Antoine Bosselut, and Tianlu Wang. Efficient tool use with chain-of-abstraction reasoning. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 2727–2743, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL `https://aclanthology.org/2025.coling-main.185/`.

Google DeepMind. Gemini 2.5: Our most intelligent ai model, 2025. URL `https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/`. Accessed: 2025-04-10.

Google DeepMind. Gemma 4 model card. `https://ai.google.dev/gemma/docs/core/model_card_4`, April 2026. Accessed: 2026-05-22.

Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. Token-budget-aware LLM reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24842–24855, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.1274. URL https://aclanthology.org/2025.findings-acl.1274/.

Jie He, Jennifer Neville, Mengting Wan, Longqi Yang, Hui Liu, Xiaofeng Xu, Xia Song, Jeff Z. Pan, and Pei Zhou. GenTool: Enhancing tool generalization in language models through zero-to-one and weak-to-strong simulation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 1097–1122, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.61. URL https://aclanthology.org/2025.findings-acl.61/.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 13358–13376, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.825. URL https://aclanthology.org/2023.emnlp-main.825/.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1658–1677, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.91. URL https://aclanthology.org/2024.acl-long.91/.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A. Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon LLM agents. *arXiv preprint arXiv:2510.00615*, 2025. doi: 10.48550/arXiv.2510.00615. URL https://arxiv.org/abs/2510.00615.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2312.04511.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.187. URL https://aclanthology.org/2023.emnlp-main.187/.

Yifei Lu, Fanghua Ye, Jian Li, Qiang Gao, Cheng Liu, Haibo Luo, Nan Du, Xiaolong Li, and Feiliang Ren. CodeTool: Enhancing programmatic tool invocation of LLMs via process supervision. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 18287–18304, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.895. URL https://aclanthology.org/2025.acl-long.895/.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. MCP-Universe: Benchmarking large language models with real-world model context protocol servers. *arXiv preprint arXiv:2508.14704*, 2025. doi: 10.48550/arXiv.2508.14704. URL https://arxiv.org/abs/2508.14704.

OpenAI. GPT-5 System Card. https://openai.com/index/gpt-5-system-card/, August 2025. Accessed: 2026-05-26.

Cheng Qian, Emre Can Acikgoz, Hongru Wang, Xiusi Chen, Avirup Sil, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. SMART: Self-aware agent for tool overuse mitigation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 4604–4621, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.239. URL https://aclanthology.org/2025.findings-acl.239/.

Ziqing Qiao, Yongheng Deng, Jiali Zeng, Dong Wang, Lai Wei, Guanbo Wang, Fandong Meng, Jie Zhou, Ju Ren, and Yaoxue Zhang. ConCISE: Confidence-guided compression in step-by-step efficient reasoning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Suzhou, China, November 2025. Association for Computational Linguistics. URL https://arxiv.org/abs/2505.04881.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model. https://qwen.ai/blog?id=qwen3.6-27b, April 2026. Accessed: 2026-09-26.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://arxiv.org/abs/2302.04761.

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24497–24524, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.1258. URL https://aclanthology.org/2025.findings-acl.1258/.

Simranjit Singh, Andreas Karatzas, Michael Fore, Iraklis Anagnostopoulos, and Dimitrios Stamoulis. An LLM-tool compiler for fused parallel function calling. *arXiv preprint arXiv:2405.17438*, 2024. doi: 10.48550/arXiv.2405.17438. URL https://arxiv.org/abs/2405.17438.

Yunyue Su, Zhang Jinshuai, Bowen Fang, Wen Ye, Jinghao Zhang, Bowen Song, Weiqiang Wang, Qiang Liu, and Liang Wang. Toolscaler: Scalable generative tool calling via structure-aware semantic tokenization. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 556–578, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-emnlp.30. URL https://aclanthology.org/2025.findings-emnlp.30/.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/wang24h.html.

Bin Wu, Edgar Meij, and Emine Yilmaz. A joint optimization framework for enhancing efficiency of tool utilization in LLM agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 22361–22373, Vienna, Austria, July 2025a. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.1149. URL https://aclanthology.org/2025.findings-acl.1149/.

Jiaqi Wu, Qinlao Zhao, Zefeng Chen, Kai Qin, Yifei Zhao, Xueqian Wang, and Yuhang Yao. GAP: Graph-based agent planning with parallel tool use and reinforcement learning. *arXiv preprint arXiv:2510.25320*, 2025b. doi: 10.48550/arXiv.2510.25320. URL https://arxiv.org/abs/2510.25320.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of LLM agents with trajectory reduction. *arXiv preprint arXiv:2509.23586*, 2025. doi: 10.48550/arXiv.2509.23586. URL https://arxiv.org/abs/2509.23586.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. Rewoo: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323*, 2023. doi: 10.48550/arXiv.2305.18323.

Hongshen Xu, Zihan Wang, Zichen Zhu, Lei Pan, Xingyu Chen, Shuai Fan, Lu Chen, and Kai Yu. Alignment for efficient tool calling of large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17776–17792, Suzhou, China, November 2025a. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.898. URL https://aclanthology.org/2025.emnlp-main.898/.

Qiancheng Xu, Yongqi Li, Heming Xia, and Wenjie Li. Enhancing tool retrieval with iterative feedback from large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 9609–9619, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.561. URL https://aclanthology.org/2024.findings-emnlp.561/.

Qiancheng Xu, Yongqi Li, Heming Xia, Fan Liu, Min Yang, and Wenjie Li. PEToolLLM: Towards personalized tool learning in large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 21488–21503, Vienna, Austria, July 2025b. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.1107. URL https://aclanthology.org/2025.findings-acl.1107/.

Silei Xu, Wenhao Xie, Lingxiao Zhao, and Pengcheng He. Chain of draft: Thinking faster by writing less. *arXiv preprint arXiv:2502.18600*, 2025c. doi: 10.48550/arXiv.2502.18600. URL https://arxiv.org/abs/2502.18600.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. EASYTOOL: Enhancing LLM-based agents with concise tool instruction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 951–972, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.naacl-long.44. URL https://aclanthology.org/2025.naacl-long.44/.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=oKn9c6ytLx.

Dongsheng Zhu, Weixian Shi, Zhengliang Shi, Zhaochun Ren, Shuaiqiang Wang, Lingyong Yan, and Dawei Yin. Divide-then-aggregate: An efficient tool learning method via parallel tool invocation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 28859–28875, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1401. URL https://aclanthology.org/2025.acl-long.1401/.

A Additional Evaluation Results

Table 6 gives the score, cost, and step count of vanilla CodeAct and ODFuse in each of the 24 configurations, which Sections 5.2 and 5.3 summarize. Figure 6 plots each configuration's change in steps against its change in cost, and Figure 7 breaks the cost of the two proprietary backbones down by token category; both support Section 5.4.

			Score (%)			Cost (m$/task)			Steps	
Backbone	Task set	N	Vanilla CodeAct	ODFuse	Δpp	Vanilla CodeAct	ODFuse	Δ%	Vanilla CodeAct	ODFuse
Gemini-2.5-Pro	AW NORMAL	168	79.8	75.6	−4.2	103.47	79.48	−23.2	15.79	6.39
	AW CHALLENGE	417	52.0	53.7	+1.7	125.98	108.11	−14.2	19.85	9.00
	FINANCIAL	40	72.7	72.1	−0.6	58.03	52.07	−10.3	4.60	2.65
	LOCATION	35	81.7	78.3	−3.4	97.91	91.97	−6.1	9.46	6.40
	REPOSITORY	28	65.1	60.8	−4.3	123.36	104.09	−15.6	11.18	4.50
	3D DESIGN	19	68.4	71.1	+2.6	74.61	70.52	−5.5	8.89	4.21
GPT-5	AW NORMAL	168	82.1	81.0	−1.2	203.76	109.67	−46.2	19.96	4.89
	AW CHALLENGE	417	68.8	71.9	+3.1	277.58	149.20	−46.2	26.15	6.34
	FINANCIAL	40	64.7	67.2	+2.5	47.89	42.42	−11.4	2.85	1.35
	LOCATION	35	84.6	82.1	−2.6	78.46	51.99	−33.7	4.51	1.26
	REPOSITORY	28	52.8	65.4	+12.6	164.93	104.42	−36.7	9.11	2.04
	3D DESIGN	19	73.7	76.3	+2.6	169.08	157.29	−7.0	7.37	4.16
Gemma-4-31B	AW NORMAL	168	83.3	84.5	+1.2	8.91	8.18	−8.2	13.88	5.82
	AW CHALLENGE	417	62.8	68.8	+6.0	13.70	11.39	−16.9	18.06	7.30
	FINANCIAL	40	79.9	79.9	+0.0	4.02	2.89	−28.1	3.45	2.30
	LOCATION	35	87.4	85.3	−2.0	6.46	6.10	−5.5	5.57	4.06
	REPOSITORY	28	67.8	63.2	−4.6	19.86	10.02	−49.6	10.86	3.96
	3D DESIGN	19	63.2	73.7	+10.5	5.42	4.25	−21.4	9.68	2.95
Qwen3.6-27B	AW NORMAL	168	72.6	84.5	+11.9	43.89	33.42	−23.9	15.49	6.10
	AW CHALLENGE	417	51.1	75.3	+24.2	61.87	52.09	−15.8	20.22	7.96
	FINANCIAL	40	64.2	69.1	+4.9	15.13	15.99	+5.7	3.58	2.62
	LOCATION	35	85.2	83.6	−1.6	31.11	26.44	−15.0	7.66	4.69
	REPOSITORY	28	71.6	68.2	−3.3	75.53	50.18	−33.6	14.21	8.43
	3D DESIGN	19	60.5	60.5	+0.0	43.31	55.24	+27.5	10.68	5.68

Table 6: All 24 configurations. Score is TGC on AppWorld and AE on MCP-Universe; cost is milli-USD per task on the ideal prompt-cache basis of Table 7. A step is one model call.

A.1 Cost Under Three Billing Regimes

Cost throughout this paper assumes an ideal prompt cache, billing every repeated prefix at the cached-input rate of Table 7. Table 8 re-bills the same trajectories at what the providers charged and with the cache disabled: on the two API backbones, ODFuse's savings under the charged price equal or exceed the reported one in 11 of 12 configurations (mean -21.3% to -28.1%), and with the cache off ODFuse is cheaper in 23 of 24 (mean -29.6%).

A.2 Task-Specific Analysis: Where Each Trade-off Comes From

Beyond the aggregate trade-off, the cost–accuracy pattern varies substantially across configurations (Table 6). We examine four representative cases that illustrate where each method's trade-off comes from.

Financial Analysis: vs. Vanilla CodeAct. For FINANCIAL ANALYSIS, cost reduction is among the smallest of the six task sets under both proprietary backbones: second only to 3D DESIGN under GPT-5, and third after 3D DESIGN and LOCATION NAVIGATION under Gemini-2.5-Pro. These tasks already run few steps (Table 6), so little reasoning goes to re-establishing the objective, and there is little redundancy left to eliminate. Accuracy splits the two backbones: $+2.5$ pp on GPT-5, -0.6 pp on Gemini-2.5-Pro. The gain is worth explaining because it is not a cost effect; we examined representative vanilla CodeAct failures. Many errors trace to a value the agent had not observed: it infers or formats a date from prior assumptions, or refers in code to an argument no observation has defined yet. Neither shape reaches the over-fusion column of Table 1, which counts only a producer and a consumer placed in one step and so reads 0.00 for vanilla CodeAct on this task set; an agent that rarely fuses presumes across steps instead. These failures demonstrate that vanilla CodeAct can

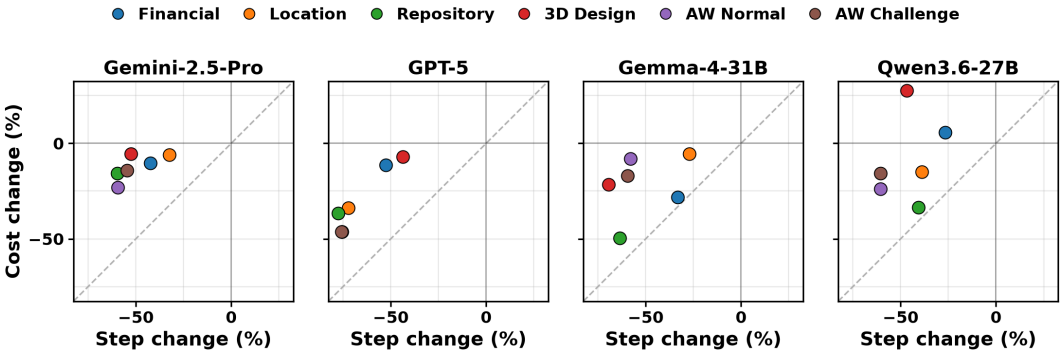


Figure 6: Change in step count (%) versus change in cost (%) of ODFuse relative to vanilla CodeAct, one point per task set, one panel per backbone, all on the same axes; negative values are savings. Above the dashed $y = x$, cost fell by less than the step count did, so the per-step cost rose; above $y = 0$, cost rose. On GPT-5, the two AppWorld points overlap.

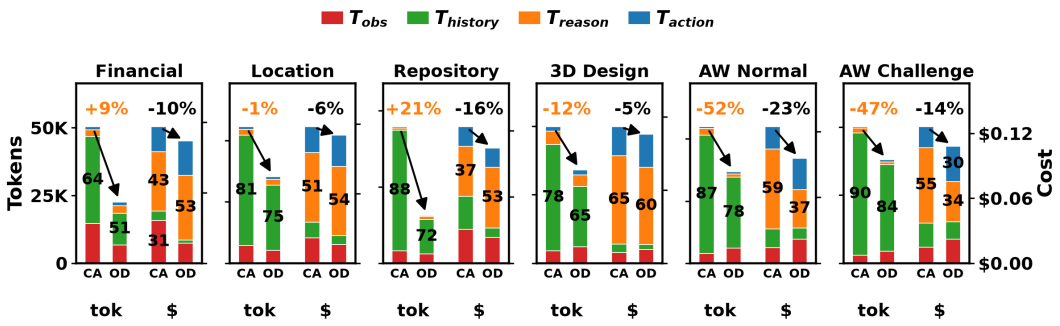
	T_{obs}	T_{history}	T_{reason} & T_{action}
GPT-5	1.25	0.125	10
GPT-5.2	1.75	0.175	14
Gemini-2.5-Pro	1.25	0.125	10
Gemini-3-Pro-Preview	2	0.2	12
Claude-Sonnet-4.5	3	0.3	15
Claude-Opus-4.5	5	0.5	25
Gemma-4-31B	0.130	0.065	0.40
Qwen3.6-27B	0.31	0.15	2.55

Table 7: Unit pricing (USD per 1M tokens) for recent frontier LLMs, across four token categories: new input (T_{obs}), cached input (T_{history}), visible output (T_{action}), and hidden output (T_{reason}). The first six rows are vendor rate cards, as of 2026-05-22. Uniform proportional changes to all token-category rates for a backbone preserve relative cost comparisons. The last two rows are open-weight backbones served by us, so they have no vendor card: they are priced at the OpenRouter median among the providers that support prompt caching, sampled 2026-05-21 for Gemma-4-31B and 2026-07-12 for Qwen3.6-27B. Those two rows are what every open-weight cost cell in this paper is computed from.

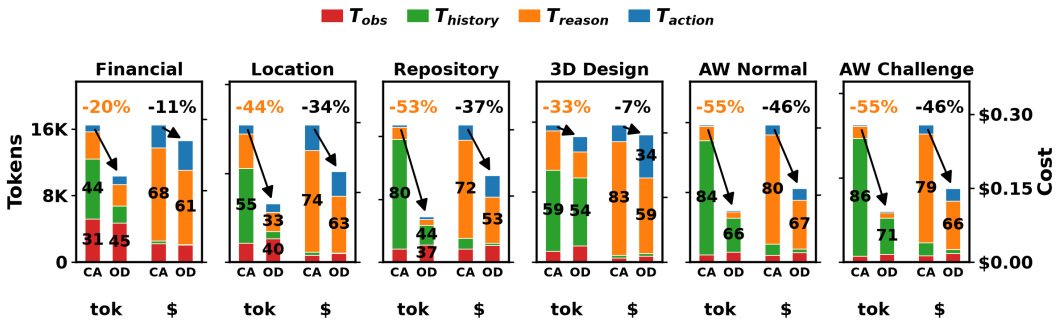
presume past an observation even when the step count is already low, so the check has something to catch here even though there is little cost to save, which is what the GPT-5 gain reflects.

LOCATION NAVIGATION: vs. AggFuse. On LOCATION NAVIGATION, AggFuse reaches accuracy comparable to ODFuse at a lower cost. The difference comes from how the two methods handle route discovery. ODFuse tends to construct the route sequentially, resolving each intermediate location only after considering the previous one. AggFuse often searches for all condition-satisfying intermediate destinations in a single fused step. This is effective because the current MCP-Universe LOCATION NAVIGATION tasks specify the constraints for each intermediate destination independently of the preceding location on the route, making these searches parallelizable. ODFuse under-fuses here: it issues these searches one per step, although none depends on another's result.

AppWorld TEST_NORMAL: vs. Vanilla CodeAct. AppWorld TEST_NORMAL on Gemini-2.5-Pro is the AppWorld configuration in which ODFuse loses the most accuracy relative to vanilla CodeAct (-4.2 pp; Table 6). Of the 134 tasks vanilla CodeAct passes on this split, ODFuse fails 16. The fusion detector of Section 3 finds an over-fusion in only 3 of these 16 tasks, a rate indistinguishable from that on the 118 tasks both methods pass (16 of them). The 16 tasks are the long ones: ODFuse stayed heavily fused on them, ran longer than on the tasks it passes, and still failed. We attribute the loss to how fusion changes the trajectory the agent plans, which the observation-dependency criterion does not cover.



(a) Gemini-2.5-Pro



(b) GPT-5

Figure 7: Token volume (**tok**, left axis) and monetary cost (**$**, right axis) per task set for vanilla CodeAct (CA) versus ODFuse (OD), for the two proprietary backbones: (a) Gemini-2.5-Pro and (b) GPT-5. Each task set has its own vertical scale, so compare bar heights only within a task set; the axis values shown belong to the leftmost (tokens) and rightmost (cost) task sets. Each bar stacks the four token categories in the legend (T_{obs}, T_{history}, T_{reason}, T_{action}); inline numbers give a category's share of that bar (only $\geq$30% labeled). Black arrows trace the CA$\rightarrow$OD change for the token bars (left pair) and the cost bars (right pair) within each task set. Above each pair, the orange number is the change in T_{reason} alone, the largest share of each cost bar, and the black number is the change in total cost. The two open-weight backbones are not broken down here; their per-configuration scores, costs, and step counts are in Table 6. Averaged over the six task sets with equal weight, T_{reason} is 5–17% of vanilla CodeAct's token volume but 52–76% of its cost.

3D DESIGN: vs. Vanilla CodeAct. The domain favors ODFuse on three of the four backbones, by +2.6 pp on Gemini-2.5-Pro and GPT-5 and +10.5 pp on Gemma-4-31B, and ties on Qwen3.6-27B, while step count falls by 44 to 70%. Repeated nine times (Section 5.5), the domain gives +3.22 pp, ahead in seven of nine runs ($p = 0.125$), at 12.0% lower cost; the difference is not significant, but the repetition rules out an accuracy loss at the 5% level ($\delta^\star = 0.0$). On the two proprietary backbones, its cost falls by less than its reasoning tokens (5.5% and 7.0% against 12% and 33%; Figure 7), while the share of the bill spent on visible output (T_{action}) rises from 21% to 25% on Gemini-2.5-Pro and from 12% to 34% on GPT-5.

A.3 WHY REWOO FAILS ON APPWORLD

ReWOO fails on AppWorld (19.0 and 12.5 TGC in Table 3) because its planner writes one plan from the task alone and never sees a tool result, a limitation its authors note for environments the planner knows little about (Xu et al., 2023). It cannot plan by observing: it cannot page through a listing until the results run out, or open each item it finds. None of its 585 AppWorld plans call an API inside a loop, and the two that read past the first page request five pages fixed in advance. AppWorld tasks are mostly of this kind, since 523 of the 585 reference solutions call an API they already used, and ReWOO's pass rate falls from 88.9% on tasks solved in at most six calls to 4.4% on those that need

	Gemini-2.5-Pro				GPT-5				Gemma-4-31B		Qwen3.6-27B	
Task set	ideal	billed	off	hit	ideal	billed	off	hit	ideal	off	ideal	off
AW-N	−23.2	−25.1	−32.5	58.0	−46.2	−52.3	−56.4	66.8	−8.2	−18.1	−23.9	−30.2
AW-C	−14.2	−15.5	−23.5	64.1	−46.2	−54.3	−56.8	69.1	−16.9	−25.4	−15.8	−24.2
Fin	−10.3	−23.5	−30.8	54.5	−11.4	−16.3	−20.2	77.2	−28.1	−35.3	+5.7	−3.7
Loc	−6.1	−14.5	−23.8	67.5	−33.7	−37.1	−42.4	81.1	−5.5	−9.8	−15.0	−22.4
Repo	−15.6	−33.9	−53.9	79.0	−36.7	−42.9	−55.4	92.9	−49.6	−58.0	−33.6	−37.3
3D	−5.5	−15.4	−19.4	66.7	−7.0	−6.0	−8.4	82.7	−21.4	−38.0	+27.5	+14.8

Table 8: Per-task cost change from vanilla CodeAct to ODFuse under three billing regimes, for every configuration; negative means ODFuse is cheaper. Every entry is a percentage. Task sets are AppWorld TEST_NORMAL (AW-N) and TEST_CHALLENGE (AW-C), and the MCP-Universe FINANCIAL (Fin), LOCATION (Loc), REPOSITORY (Repo) and 3D DESIGN (3D) domains. The three regimes are *ideal* (the paper's basis), *billed* (from each response's cached-token count), and *off* (cache disabled). *hit* is the share of input tokens the provider reported serving from cache. Gemma-4-31B and Qwen3.6-27B were served on our own hardware, so no provider bill exists for them, and they have only the *ideal* and *off* columns, priced as in Table 7.

more than thirty (vanilla CodeAct: 94.4% and 60.4%). MCP-Universe questions mostly name what to fetch, so a fixed plan loses less there.

A.4 WHICH STEPS ODFUSE SAVES

Table 9 splits the steps ODFuse saves in each configuration by what those steps did. The saved steps do not appear in ODFuse's runs, so the table compares counts: each step of both methods is labeled by the tool calls in the code it ran, and the entry for a label is vanilla CodeAct's steps per task with that label minus ODFuse's. A failed step is labeled Failed whatever it called, and a step with calls of several kinds counts equally toward each, so one that logs in and then looks up adds half a step to Login and half to Lookup. Splitting such a step in proportion to its distinct APIs of each kind, or crediting it wholly to its first or last call, instead moves at most 0.57 steps per task among Docs, Login, Lookup and Write, and no share cited in Section 5.2 by more than 2.5 pp. For logging in, lookups, and writes, the two methods call about as many distinct APIs, at most 0.5 apart per task.

			Split by what the step did						
Backbone	Task set	Saved	Docs	Login	Lookup	Write	Submit	No call	Failed
Gemini-2.5-Pro	AW NORMAL	9.40	3.36	2.08	1.49	0.80	0.73	0.67	0.27
	AW CHALLENGE	10.85	3.50	2.32	2.24	1.16	0.84	0.51	0.27
	FINANCIAL	1.95	–	–	0.57	–	0.23	1.27	−0.12
	LOCATION	3.06	–	–	4.09	–	0.31	−0.97	−0.37
	REPOSITORY	6.68	–	–	2.86	3.43	0.93	−0.11	−0.43
	3D DESIGN	4.68	–	–	0.08	4.18	0.95	−0.32	−0.21
GPT-5	AW NORMAL	15.08	6.80	2.10	2.37	1.17	0.85	1.74	0.05
	AW CHALLENGE	19.81	9.41	2.42	4.14	1.39	0.80	1.78	−0.12
	FINANCIAL	1.50	–	–	0.72	–	0.60	0.23	−0.05
	LOCATION	3.26	–	–	2.23	–	0.69	0.43	−0.09
	REPOSITORY	7.07	–	–	3.16	3.73	0.50	0.36	−0.68
	3D DESIGN	3.21	–	–	0.42	4.68	0.63	0.00	−2.53
Gemma-4-31B	AW NORMAL	8.06	3.05	1.84	1.29	0.32	0.83	0.29	0.44
	AW CHALLENGE	10.76	3.95	2.25	2.13	0.69	0.85	0.13	0.75
	FINANCIAL	1.15	–	–	0.70	–	0.15	0.35	−0.05
	LOCATION	1.51	–	–	1.23	–	0.26	0.23	−0.20
	REPOSITORY	6.89	–	–	3.02	3.02	0.71	0.00	0.14
	3D DESIGN	6.74	–	–	0.32	5.68	0.79	−0.11	0.05
Qwen3.6-27B	AW NORMAL	9.39	3.35	1.82	1.71	1.01	0.74	0.43	0.33
	AW CHALLENGE	12.26	4.80	2.16	2.63	1.12	0.75	0.25	0.55
	FINANCIAL	0.95	–	–	0.42	–	0.12	0.42	−0.03
	LOCATION	2.97	–	–	2.54	–	0.06	0.29	0.09
	REPOSITORY	5.79	–	–	1.27	3.27	0.14	0.14	0.96
	3D DESIGN	5.00	–	–	−0.05	4.00	0.37	−0.11	0.79

Table 9: Steps per task that ODFuse saves over vanilla CodeAct (*Saved*, the difference between the two step columns of Table 6 before rounding), split by what each step did. *Docs*: reading AppWorld's API documentation. *Login*: fetching an account password or logging in to an app. *Lookup*: a call that only reads (on AppWorld, an API documented as a GET request; on MCP-Universe, a tool that returns data without changing anything). *Write*: any other call. *Submit*: a step whose only call submits the answer. *No call*: a step that calls no tool. *Failed*: a step whose code raised or whose tool returned an error. Entries sum to Saved up to rounding; a negative entry means ODFuse spends more such steps, and a dash marks a kind of call neither method made.

B Detecting Over-fusion and Under-fusion

B.1 How the Detector Works

(a) What the rules use

Each result, as the tool documentation describes it

`search_songs(query) → [{"song_id": 1, ...}, ...]`
A **list**, ranked by relevance to the query; `song_id` is an identifier.

`show_song(song_id) → {"album_id": 1, ...}`
One song; `album_id` is an identifier.

`show_venmo_balance() → {"venmo_balance": 12.5}`
One number.

`show_note(note_id) → {"content": "...", ...}`
`content` is free text with no documented format.

Operations on a path from a result to a later call

Operation	Needs an observation
a field of one result, `r["album_id"]`	no
`for x in results`	no
arithmetic, `len`	no
`results[0]`	yes; if the documentation fixes an order, the judge decides
first match, `next(...)`	yes, unless it matches on an identifier
`max`, a filter, `if x < c`	no if its constants were known; else the judge
parsing free text	yes
anything else	an LLM judge decides

A constant is known when it appears in the task, the documentation, or an earlier output. A criterion carried over from the task's meaning, such as "the latest" becoming `max(key=created_at)`, goes to the judge.

(b) Over-fusion: inside one step

Task: "Like the Beatles' song Yesterday."

```
songs = spotify.search_songs(
  query='Yesterday')
spotify.like_song(song_id=
  songs[0]["song_id"])
```

Path: the unobserved list → `[0]` → `like_song`'s argument. The list is ranked by relevance, but the task asks for the Beatles' version: **over-fusion**.

Task: "If my Venmo balance is below 50, add 50."

```
bal = venmo.show_venmo_balance()[
  "venmo_balance"]
if bal < 50:
  venmo.add_to_venmo_balance(
    amount=50, payment_card_id=card_id)
```

Path: the unobserved balance → `< 50` → whether `add_to_venmo_balance` runs. The `50` is in the task: **no over-fusion**.

(c) Under-fusion: between two steps

```
# step 1
song = spotify.show_song(song_id=sid)
print(song)
# step 2
spotify.show_album(album_id=17)
```

Step 2 copies the `17` out of step 1's output. It is a field of the one song printed, which code could have read as `song["album_id"]`, so step 2 could have joined step 1: **under-fusion**.

```
# step 1
songs = spotify.search_songs(query='Yesterday')
print(songs)
# step 2
spotify.like_song(song_id=16)
```

The `16` is one of several songs printed; picking the Beatles' one needs the observation, as in (b): **no under-fusion**.

Figure 8: Fusion-detector rules and examples. (a) Representative rules for deciding whether an operation requires observing a result; Appendix B.1 gives the full rules. (b) Over-fusion within a step. (c) Under-fusion across a step boundary. All steps are illustrative.

Overview. The fusion detector asks whether later API calls could have been written without observing preceding results, both within a step and across step boundaries. It follows each path along which an API result reaches a later API call, either as an argument of that call or through the condition that decides whether, or how often, the call runs; across steps, it also tracks values copied from an earlier output as literals. The operations along the path, or the operation needed to select a copied value, determine whether the result needs to be observed (Figure 8). Over-fusion and under-fusion are the same analysis started from different results: inside a step, from the results the model has not yet observed; at a stop between two steps, from the results the earlier step printed. We run it on the Gemini-2.5-Pro trajectories of vanilla CodeAct, AggFuse, and ODFuse on the six task sets of Table 1.

Counting over-fusions. Inside a step, the sources are the results the step itself receives. A result counts as already observed when an earlier step printed the result of the same call, meaning the same API with the same literal arguments apart from the access token; a call with computed arguments

cannot be matched and counts as unobserved. A call that a path from an unobserved result reaches through an operation needing the observation is an over-fusion, and so is a call to an undocumented API made without a lookup. Each over-fused call counts once per task: an agent that rewrites its program after an error writes the same call again, and the rewrite repeats a fusion decision already counted.

Counting under-fusions. At each stop between two tool-calling steps, the sources are the results the earlier step printed. The under-fusion pass marks the stop under-fused when no path from those results into the later step needs the observation. A value that the later step copies out of the earlier step's output, written as a literal after the agent read it, is judged the same way, by the operation code would have needed to pick it out of that output (the rules are in our released code). The pass also marks the stop under-fused when the earlier step printed nothing to observe, an empty result included, since code that handles both an empty and a non-empty result can be written in advance. We also count stops next to AppWorld documentation lookups when both adjacent steps call a tool. Such a stop needs the observation if the later step uses an API name, documentation, or value first revealed by the lookup. Stops after a failed step are not counted, because the next step repairs a failure that no fusion could have anticipated. A step fails when its code raises an error or, in MCP-Universe, when a tool returns an error message where an AppWorld API would raise; an error the agent's own code catches and prints is not a failure.

What the documentation fixes. Before analyzing a trajectory, we reduce each API's documented response to whether it returns one object or a list, each field's kind (identifier, documented-format value, or free text), and any documented list ordering. These properties determine what the model can assume without observing the result.

Paths. The analysis walks each step's code in execution order and follows values through assignments, field reads, indexing, function calls, comparisons, and loops, into functions the step defines and into containers the code appends to. Variables persist across steps, as they do in the agent's runtime. A value that passes through another API call counts as that call's result, and a path that reaches no API call is dropped.

Which operations need the observation. An operation needs the observation when it presumes something only the result reveals. Taking a list item by position needs it when the documentation gives no order or ranks by relevance; taking the first item satisfying a condition needs it unless the condition is equality on an identifier field; and parsing free text needs it. In contrast, reading a field of a single result, iterating over a list, checking whether a result is empty, arithmetic, counting, and treating one page of a paginated list as the whole list need no observation. A selection, filter, or comparison also needs none when every constant it uses was available before the result arrived—from the task, user profile, documentation, an earlier output, or the name of an app called in the step. Otherwise, and for operations these rules do not cover, the LLM judge decides; this includes criteria carried over from the task's meaning, such as "the latest" implemented as a maximum over creation time. A path needs the observation when any operation on it does. Using an undocumented API or field likewise presumes an unobserved result unless an earlier step looked up its documentation. Because MCP-Universe provides tool documentation in the first prompt, only fields that documentation leaves open count there.

LLM judge. The judge is Claude Opus 5.5, run with no tools. Each question concerns one operation on one path and shows the task, the documentation of the APIs involved, the code with comments and the agent's reasoning removed, the path, and the outputs observed so far; it does not name the agent. The judge answers whether the operation could have been written before the result was observed. Only questions whose answer can change a count are asked, since the others lie on a call or stop a rule has already decided. The verdicts are recorded with the model version and released with the code.

B.2 Where the Steps End

Most vanilla CodeAct stops are unnecessary. Under the detector above, 1,383 of the 2,130 eligible vanilla CodeAct stops on AppWorld TEST_NORMAL (64.9%) and 4,745 of 6,855 on TEST_CHALLENGE (69.2%) are under-fused; the fraction ranges from 65.6–98.5% on the four MCP-Universe domains. A lower step count alone, however, does not show whether a method

removes these unnecessary stops or stops that carry observations needed by the next step. Table 10 therefore sorts the stops in the Gemini-2.5-Pro runs of Table 1 by whether the next step needs the observation, and also reports over-fused calls. Figure 9 marks them on one task.

	All stops			Needs the observation			Under-fused			Over-fused calls		
Task set	CA	AF	OD	CA	AF	OD	CA	AF	OD	CA	AF	OD
AW NORMAL	14.79	6.46	5.39	4.45	2.11	2.76	8.23	0.64	2.06	0.04	5.78	0.26
AW CHALLENGE	18.85	9.56	8.00	5.06	3.18	3.54	11.38	0.99	3.35	0.16	10.67	0.55
FINANCIAL	3.60	2.62	1.65	0.28	0.00	0.28	0.53	0.07	0.93	0.00	0.80	0.10
LOCATION	8.46	1.63	5.40	1.20	0.06	0.74	5.83	0.49	2.31	0.03	1.74	0.09
REPOSITORY	10.18	1.50	3.50	0.14	0.04	0.29	9.18	0.32	1.96	0.00	1.14	0.50
3D DESIGN	7.89	4.26	3.21	0.37	0.05	0.32	6.89	0.16	1.84	0.00	1.53	0.00

Table 10: Stops and over-fused calls per task in the Gemini-2.5-Pro runs of Table 1, for vanilla CodeAct (CA), AggFuse (AF), and ODFuse (OD). A stop *needs the observation* when the next step needs what the previous step showed, and is *under-fused* otherwise. The detector of Appendix B judges the stops between two tool-calling steps and those next to an AppWorld documentation read, and counts the over-fused calls. Stops after a failed step count neither way; with the stops next to a step that calls no tool, they make up the difference to *All stops*. Several observations can share one stop, so ODFuse can need fewer stops than vanilla CodeAct while observing as much; an observation it skips appears as an over-fused call.

Task: I have received the weekly work meeting schedule from my manager via email. Add an alarm to my phone for every meeting 5 minutes before the scheduled time. The alarms should repeat weekly on that day, labeled as per meeting name, and have 5 minutes of snooze.

Vanilla CodeAct — *20 steps, passes*

Steps 1–5: read API documentation

6 `supervisor.show_account_passwords`

↓ no observation dependency: picks the phone password by the app the task names

7 `phone.login(phone_number=...)` → `error`

↓ after a failed call: counted neither way

8 `phone.login`

↓ no observation dependency: passes the token on

9 `phone.search_contacts( relationship='manager')`

↓ no observation dependency: step 10 logs in to Gmail, which needs nothing from the contact

10 `gmail.login`

↓ no observation dependency: passes the token on

11 `gmail.show_inbox_threads(subject=...)` → `error`

Step 12: read API documentation

13 `gmail.show_inbox_threads`

↓ observation dependency: which of the five threads found holds the schedule: 47824

14 `gmail.show_thread`

Step 15: read API documentation

16–19 `phone.create_alarm, one meeting per step`

↓ no observation dependency: four stops through step 20, each after a step that prints nothing to observe

20 `supervisor.complete_task`

AggFuse — *9 steps, fails*

1–8 five attempts at the whole task in one block, each raising an error (APIs that do not exist, a missing access token, a wrong parameter name); steps 4, 6 and 8 read the documentation

9 `passwords, logins, show_inbox_threads, show_thread, line.split(","), create_alarm, complete_task`

× over-fusion: assumes each line of the email it has not seen reads "name, day, time"

→ **no line parses, so no alarm is created**

ODFuse (Ours) — *7 steps, passes*

Steps 1–3: read API documentation

4 `show_account_passwords, gmail.login, phone.login, gmail.show_inbox_threads`

↓ observation dependency: which of the 46 threads found holds the schedule: 47824, "Schedule for Weekly Meetings"

5 `gmail.show_thread`

↓ observation dependency: what the email lists, such as "Quality Assurance and Testing Roundtable => Monday @ 13:00"

6 `phone.create_alarm for each meeting, supervisor.complete_task` → `error`

↓ after a failed call (the weekday must be lower-case): counted neither way

7 `phone.create_alarm for each meeting, supervisor.complete_task`

Figure 9: Where each agent stops on one AppWorld TEST_CHALLENGE task (Gemini-2.5-Pro), read by hand. Arrows mark stops between tool-calling steps by whether the next step has an observation dependency (Section 3), grey ones follow a failed call, and a cross marks an over-fusion. Stops next to documentation reads are unmarked. The task is one of 16 in this split that vanilla CodeAct and ODFuse both pass while vanilla CodeAct under-fuses at least twice between tool-calling steps and ODFuse never does, stopping there at least once on an observation dependency; AggFuse fails 11 of them. The detector of Appendix B agrees with each mark, and ODFuse costs 21.4% less than vanilla CodeAct.

C Benchmark Implementation Details

C.1 Tool Output Schemas Missing from the MCP-Universe Wrappers

The MCP-Universe tool descriptions specify input arguments only, and several wrappers reshape what the provider returns (renaming fields, or wrapping the payload in a JSON string), so an agent relying on the provider's native schema indexes into fields that never arrive. We add to each affected description a `Returns:` block giving the structure the wrapper actually returns, taken from the provider's own specification; the change is documentation-only and applied identically to every compared method.

C.2 Backbone Configuration

The two proprietary backbones are `gemini-2.5-pro` (Google) and `gpt-5-2025-08-07` with `reasoning_effort=medium` (OpenAI), both accessed through their providers' paid APIs; the two open-weight backbones are the checkpoints `google/gemma-4-31B-it` and `Qwen/Qwen3.6-27B`. Decoding is fixed per backbone and identical across the compared methods and both benchmarks: Gemini-2.5-Pro is greedy (`temperature=0`), the two open-weight backbones use `temperature=0.6` in thinking mode, and GPT-5 uses `temperature=1`, the only value its API accepts. The one exception is the Web Search score quoted in Section 5.1, which comes from an earlier Gemini-2.5-Pro run at `temperature=1`.

C.3 Step Limits and Context Window

The compared methods share one per-task step limit: 40 steps on AppWorld, and on MCP-Universe 20 on Financial Analysis and 3D Design, 50 on Location Navigation, and 40 on Repository Management. These match the benchmarks' current defaults except on Location Navigation and Repository Management, where MCP-Universe allows 20 steps. The higher limits favor the methods that take more steps, not ODFuse: truncating the recorded runs at 20 steps would lower ACON, Chain-of-Draft, ReAct, and TALE-EP by 2.1–10.7 points on these two domains and vanilla CodeAct by 11.0 points on Qwen3.6-27B Repository Management, against at most 3.6 for ODFuse. The open-weight backbones are served with a context window of 131,072 tokens, and the API backbones use the full window their providers offer. A task that does not finish within its step limit, or whose conversation outgrows the context window, is scored zero.

C.4 Other Harness Settings

Beyond the tool output schemas of Appendix C.1 and the step limits of Appendix C.3, the harness used for every reported run, including our changes to the released benchmarks, is documented in our released code so that the results can be reproduced; it is the same for every compared method.

C.5 API Expenditure

Aggregating the per-call usage logs emitted by the benchmark harness for the API-backbone runs of this study (AppWorld TEST_NORMAL and TEST_CHALLENGE and the four MCP-Universe domain suites under Gemini-2.5-Pro and GPT-5, covering vanilla CodeAct, AggFuse, and ODFuse, plus per-step budget variants that this paper does not report), provider-billed API expenditure was approximately USD 1,240 across $\sim$83k LLM calls and $\sim$49M hidden reasoning tokens. The training-free baselines of Section 5 ran on the locally served open-weight backbone and incurred no provider billing. Including the full research process (exploratory prompt ablations, superseded agent variants, and re-runs after pipeline fixes), cumulative API expenditure was approximately USD 3,950 across $\sim$284k LLM calls and $\sim$145M hidden reasoning tokens. These figures are raw per-call provider billing, which is larger than the cached-history token accounting used for the cost comparisons in Section 5. We did not train or fine-tune any models; API backbones are inference-only.