EditAgent: State Edit Programs for Revision-Aware Tool-Using Language Agents

Anonymous ACL submission

Abstract

Tool-using language agents often fail in multi-turn interactions not because they cannot call tools, but because they do not reliably update the current user intent before calling tools. When a user replaces a date, deletes a preference, relaxes a constraint, or adds a conditional fallback, a history-based agent must infer which earlier constraints remain active and which have become obsolete. We propose EDITAGENT, a revision-aware agent architecture that represents each user revision as an executable *state edit program*. Instead of regenerating the entire task state or relying on a free-form conversation summary, EDITAGENT predicts local edit operations such as `add`, `replace`, `delete`, `relax`, and `add_conditional`; applies them with a deterministic reducer; verifies the resulting state and tool call for stale constraints; and then plans tool use from the updated state. We construct 98,793 state-editing candidates from MultiWOZ 2.4 and SGD and 10,000 tool-revision candidates from τ-bench reservations. On a stratified API-evaluated subset of 600 state-editing instances and 300 tool-revision instances, EDITAGENT improves slot-level state F1 over a full-history baseline from 87.3 to 93.2 and improves tool argument accuracy from 64.7 to 99.7 while keeping obsolete-constraint leakage near zero. On sampled MultiWOZ 2.4 test turns, EDITAGENT reaches 82.0 JGA, suggesting that executable state edits provide a method-level bridge between dialogue state tracking and modern tool-using agents. Code: https://anonymous.4open.science/r/EditAgent-18F7/.

1 Introduction

Large language model (LLM) agents increasingly operate through external tools: they query databases, search the web, schedule meetings, modify orders, and interact with APIs. This has motivated a large body of work on tool use and function calling, from reasoning-and-acting agents (Yao et al., 2022) to large-scale tool-use benchmarks and API-call evaluation (Qin et al., 2024; Li et al., 2023; Patil et al., 2025; Chen et al., 2025). A common assumption in many such evaluations is that the user's goal is available in the dialogue history and can be recovered by giving the model enough context.

Real interactions violate this assumption. Users revise goals incrementally: "book it for Friday" becomes "actually Saturday morning"; "only direct trains" becomes "a transfer is fine if direct trains are sold out"; "choose the cheapest option" becomes "arrival time matters more now." Such turns do not merely add information. They mutate the active task state: some constraints are preserved, some are overwritten, some are deleted, and some become conditional. If an agent fails to perform this mutation before acting, it may call a syntactically correct tool with semantically stale arguments.

Current agents usually handle this problem implicitly. ReAct-style agents keep a textual trajectory (Yao et al., 2022); memory agents store summaries or retrieved facts (Shinn et al., 2023; Packer et al., 2023; Kang et al., 2025); function-calling systems generate tool arguments from the full conversation (Patil et al., 2025). These approaches can work when goals accumulate monotonically. They are brittle when later turns invalidate earlier ones. Remembering more history can even be harmful, because the history contains obsolete constraints that should no longer control the next action.

Dialogue state tracking (DST) has long studied evolving user goals (Ma, 2013; Han et al., 2021; Ye et al., 2022), and selective state-update models such as SOM-DST show that updating only changed slots can be more effective than regenerating a full state (Kim et al., 2020). However, modern tool agents require more than belief-state tracking. Their states must be executable: they must be converted into tool calls, checked against tool schemas, updated using tool observations, and protected from obsolete constraints leaking into

actions.

We propose EDITAGENT, an agent architecture for revision-aware tool use. Its central idea is to treat user revisions as **state edit programs**. At each turn, the LLM does not directly call a tool and does not rewrite the entire state. Instead, it predicts a small program of typed edits over the previous state. A deterministic reducer executes these edits, a verifier checks edit validity and stale-constraint leakage, and a planner calls tools from the reduced state. This gives the agent an auditable intermediate representation: when it fails, we can localize whether the error came from revision parsing, state mutation, verification, or tool planning.

Our contributions are:

- We formulate revision-aware tool use as an executable state-editing problem for language agents, distinguishing it from both broad goal-shift evaluation and classical DST.
- We introduce EDITAGENT, which decouples revision understanding from state mutation via typed edit programs, deterministic reduction, stale-constraint verification, and state-grounded tool planning.
- We define an edit operation space for user revisions, covering addition, replacement, deletion, relaxation, tightening, conditional fallback, and preference reordering.
- We evaluate on state-editing and tool-revision instances, showing that EDITAGENT improves over full-history prediction and sharply reduces obsolete-constraint leakage while remaining competitive with full-state tracking.

2 Related Work

Tool-use and function-calling agents. ReAct interleaves reasoning and actions in a textual trajectory (Yao et al., 2022), while Toolformer, Gorilla, ToolLLM, API-Bank, ToolBench, BFCL, ACEBench, and ToolSandbox study how models select tools, generate arguments, and execute multi-step tool calls (Schick et al., 2023; Patil et al., 2024; Qin et al., 2024; Li et al., 2023; Patil et al., 2025; Chen et al., 2025; Lu et al., 2025). These works establish tool grounding as a core capability. EDITAGENT addresses a complementary failure: before selecting a tool, the agent must update which user constraints are currently valid.

Stateful and goal-shift agent evaluation. Recent benchmarks move beyond static function calling toward interactive, stateful environments, including τ-bench, τ^2-bench, DialogTool, AppWorld, WebArena, WorkArena, and AgentChangeBench (Yao et al., 2024; Wang et al., 2025; Rana et al., 2025). These benchmarks show that agents struggle when goals, environments, or user behavior change. Our work differs in focus: rather than proposing another evaluation suite for goal shifts, we propose a concrete control mechanism for handling such shifts as executable state edits.

Memory, reflection, and search agents. Reflexion, Self-Refine, Tree-of-Thoughts, LATS, Voyager, Generative Agents, MemGPT, MemoryBank, LoCoMo, and MemoryOS explore reflection, iterative refinement, search, skill libraries, and long-term memory (Shinn et al., 2023; Madaan et al., 2023; Yao et al., 2023; Zhou et al., 2023; Wang et al., 2023; Park et al., 2023; Packer et al., 2023; Zhong et al., 2024; Maharana et al., 2024; Kang et al., 2025). These methods improve planning or memory, but they usually do not represent current task-state changes as typed, verifiable mutations. EDITAGENT is narrower: it manages the active task state during a single evolving tool-use dialogue.

Dialogue state tracking and structured updates. DST maintains user goals over dialogue turns. MultiWOZ, SGD, TRADE, TripPy, SOM-DST, ParsingDST, and function-calling DST provide datasets and methods for structured state tracking (Han et al., 2021; Ye et al., 2022; Rastogi et al., 2020; Wu et al., 2019; Heck et al., 2020; Kim et al., 2020; Wu et al., 2023; Li et al., 2024). SOM-DST is especially relevant because it selectively overwrites state memory. EDITAGENT generalizes this intuition to tool agents: edits apply not only to slots, but also to preferences, conditional rules, tool-grounded facts, and unresolved fields, and the resulting state is used to control executable tool calls.

Table 1 is qualitative because prior systems use different tasks and metrics; Section 5 compares reimplemented baselines under a shared protocol.

The closest same-data comparisons are non-supervised DST methods on MultiWOZ, especially IC-DST, ParsingDST, and FnCTOD. Table 2 reports EDITAGENT on MultiWOZ 2.4 with the gold previous state, which isolates the current-turn update problem that edit programs are designed to

Work	Tool	State/shift	Edit+verify
ReAct (Yao et al., 2022)	✓	–	–
ToolSandbox/DialogTool (Lu et al., 2025; Wang et al., 2025)	✓	✓	–
AgentChangeBench (Rana et al., 2025)	✓	✓	–
SOM-DST/ParsingDST (Kim et al., 2020; Wu et al., 2023)	–	✓	partial
FnCTOD (Li et al., 2024)	✓	✓	–
EDITAGENT	✓	✓	✓

Table 1: Qualitative comparison with closely related work. EDITAGENT combines tool use, revision-aware state tracking, executable edits, and stale-constraint verification.

Method	Setting	JGA
IC-DST	zero-shot ICL	53.60†
FnCTOD GPT-3.5	zero-shot func.	61.31†
FnCTOD GPT-4	zero-shot func.	62.59†
ParsingDST	zero-shot parsing	64.65†
EDITAGENT	zero-shot, oracle prev.	**82.00**

Table 2: MultiWOZ 2.4 comparison with non-supervised DST methods. † denotes average per-domain JGA reported by prior work; EDITAGENT uses sampled test-turn JGA in an oracle-previous-state current-turn update setting.

Revision	Example edit
Replace date	`replace(/constraints/date, Fri, Sat)`
Delete preference	`delete(/preferences/window)`
Relax constraint	`relax(/constraints/direct, allow transfer)`
Add conditional	`add_conditional(if price>300, economy)`
Reorder preference	`reorder(/preferences, price > time)`

Table 3: Examples of state edit operations.

solve. EDITAGENT reaches 82.00 JGA on sampled test turns. This oracle-previous result is not a strict leaderboard comparison, but it shows that the edit formulation is strong when previous-state error accumulation is controlled. No prior result directly evaluates our derived edit programs, obsolete-value leakage, or state-action consistency metrics.

3 Task: Revision-Aware Tool Use

We consider a multi-turn tool-use dialogue consisting of user utterances, agent messages, tool calls, and tool observations. At turn t, the agent has a previous task state s_{t-1}, a new user utterance u_t, and possibly a tool observation o_t. The goal is to produce a valid next state s_t and either ask a clarification question or call a tool with arguments grounded in s_t.

State schema. We represent s_t as a structured record:

- **constraints**: hard requirements such as date, location, budget, or service class;
- **preferences**: soft priorities such as cheapest, fastest, or most convenient;
- **conditionals**: trigger-action rules such as "if no direct trains, allow transfer";
- **tool facts**: observations returned by tools, such as sold-out options or unavailable slots;
- **obsolete**: constraints or preferences that have been explicitly invalidated;
- **unresolved**: required fields that must be clarified before execution.

Revision types. User turns can perform multiple state operations. We use the following revision categories: add a new constraint, replace an old value, delete a prior requirement, relax a hard constraint into a fallback, tighten a preference into a hard constraint, add a conditional rule, reorder preferences, or confirm an unresolved value. These categories are not intended as a linguistic taxonomy; they are an operation space for controlling a tool-using agent.

Why edits rather than states? The most direct approach is to prompt the model to output the full current state after every turn. This is simple, but it creates two problems. First, the model can accidentally drop fields that were not mentioned in the current turn. Second, it is hard to tell whether a wrong state comes from misunderstanding the latest revision or from regenerating unrelated parts of the state. Edit programs make the default behavior explicit: unchanged fields persist unless an edit touches them. This mirrors the intuition behind selective overwriting in DST (Kim et al., 2020), but extends it to tool-agent state and action control.

Additional edit examples are provided in Appendix B.

4 EDITAGENT

EDITAGENT decomposes each turn into four stages: edit parsing, deterministic reduction, verification, and state-grounded planning. Figure 1 shows the architecture.

4.1 State Edit Programs

An edit program $e_t = [e_t^1, \ldots, e_t^k]$ is a list of typed operations over paths in the task state. The LLM edit parser receives (s_{t-1}, u_t, o_t) and outputs JSON-like operations. The operation inventory is intentionally small:

- `add(path, value)`
- `replace(path, old, new)`
- `delete(path, value)`
- `relax(path, fallback)`
- `tighten(path, value)`
- `add_conditional(trigger, update)`
- `reorder(path, priority)`
- `confirm(path, value)`

This edit-first design has two advantages over full-state regeneration. First, most user revisions are local: if the user changes the date, the destination and budget should remain unchanged by default. Second, edits are auditable: a stale tool call can be traced to a missing `replace`, an invalid `delete`, or a planner that ignored the reduced state.

Edit parser prompt. The edit parser is an LLM prompted with the previous state, latest user turn, latest tool observation, allowed operations, and a schema. It must output only a JSON list of operations. We include few-shot examples for ambiguous cases: addition versus replacement, deletion versus relaxation, and conditional versus immediate update. The parser is instructed not to infer unstated preferences and not to rewrite untouched fields.

Compositional edits. A single user turn can produce multiple edits. For example, "Actually Saturday morning, and if it is expensive choose second class" requires a date replacement, a time replacement, and a conditional fare-class rule. We preserve edit order because later edits may refer to values created by earlier edits. The reducer rejects cyclic or contradictory edit sequences, such as replacing a field and deleting the same value in the same turn without an explicit reason.

4.2 Deterministic Reducer

The reducer applies edit programs to s_{t-1} and returns s_t. Unlike a free-form summary, the reducer is deterministic and schema-aware. A valid `replace` must identify an existing value or an equivalent canonical value. A `delete` moves a value into `obsolete` unless the field is purely transient. A `relax` preserves the original preference as primary while adding a fallback rule. An `add_conditional` creates a trigger-action pair rather than immediately changing active constraints.

The reducer also canonicalizes values. Dates are normalized to ISO-style values when a reference date is known; times are converted into intervals; locations and categorical fields are mapped to schema values. This prevents surface paraphrases from being treated as different states.

Obsolete-state bookkeeping. The `obsolete` field is not simply a trash bin. It supports verification. When a value is replaced or deleted, the reducer records the path, old value, turn id, and reason. The planner is not allowed to use obsolete values unless a later `confirm` or `replace` operation reactivates them. This explicit bookkeeping is what lets us measure stale constraint leakage and provide targeted feedback.

4.3 Edit and Action Verification

The verifier checks both the edit program and the planned tool call. It flags four error classes:

- **invalid edit**: the path or old value does not exist;
- **missing update**: the user utterance entails a revision that is absent from the edit program;
- **stale leakage**: an obsolete value appears in active constraints or tool arguments;
- **premature execution**: required fields remain unresolved, but the planner calls a tool.

When verification fails, the model receives the flagged issue and regenerates the edit program or tool call once. This is similar in spirit to self-refinement (Madaan et al., 2023), but the feedback target is a structured edit/action pair rather than an unconstrained natural-language answer.

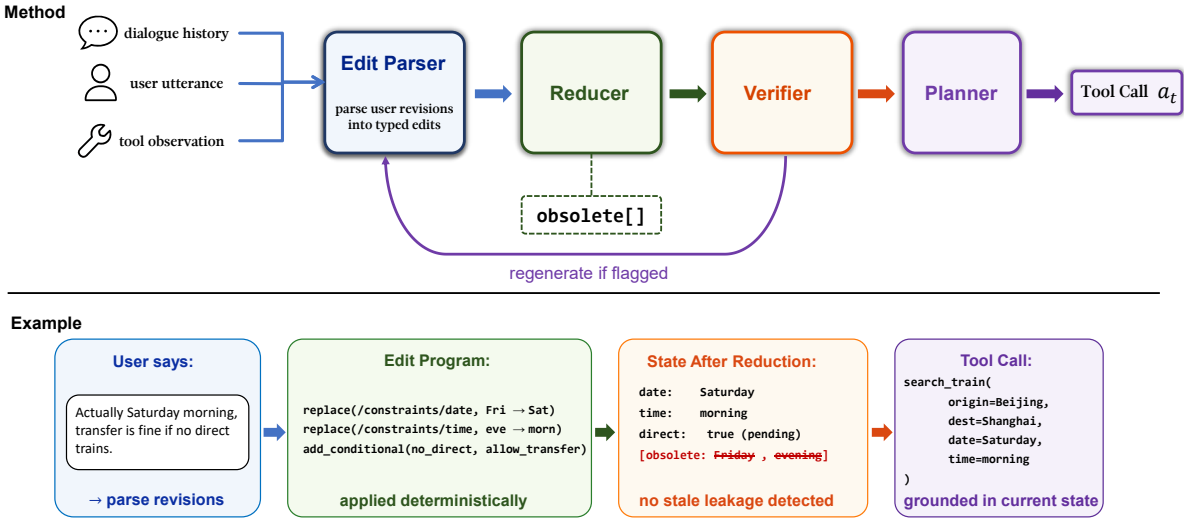

Figure 1: EDITAGENT method overview and running example. The edit parser converts the latest user revision, dialogue history, and tool observation into typed edits; the reducer updates the task state and records obsolete values; the verifier can request regeneration when stale values are detected; and the planner emits tool calls grounded only in the current state.

Verifier implementation. The verifier has two layers. The rule-based layer checks schema paths, old-value existence, required fields, obsolete leakage, and conditional triggers. The semantic layer is used only when string equivalence is insufficient, such as recognizing that "tomorrow morning" and a normalized date-time interval refer to the same value. Keeping most checks deterministic is important: if the verifier were just another unconstrained LLM judge, the architecture would be harder to debug and reproduce.

4.4 State-Grounded Tool Planner

After reduction and verification, the planner sees only the current state and the tool schema, not the full dialogue history. This forces old information to affect action only if it survived as active state. The planner either asks a targeted clarification question if `unresolved` is non-empty, or emits a tool call whose arguments are grounded in active constraints, preferences, conditionals, and tool facts.

Planner variants. The planner can be a simple function-calling prompt, a ReAct-style planner, or a search-based planner such as LATS. Our main experiments use a simple planner to isolate the effect of state editing. Testing whether stronger planners further benefit from explicit state edits is left to future work.

4.5 Running Example

Consider a travel assistant. The initial state is:

```
origin=Beijing,   destination=Shanghai,
date=Friday, time=evening, direct=true,
preference=cheapest.
```

The tool returns no direct evening trains. The user then says: "Actually Saturday morning is fine, and a transfer is okay if direct trains are unavailable." A history-based planner may call the search tool with Friday or keep `direct=true`. EDITAGENT predicts:

```
replace(/constraints/date,
Friday,                    Saturday);
replace(/constraints/time,     evening,
morning);    add_conditional(no_direct,
direct=false).
```

The reducer updates only the date and time, preserves origin, destination, and price preference, and records Friday/evening as obsolete. Since the latest observation already says no direct trains are available for the previous query but not necessarily for Saturday morning, the conditional is not automatically triggered unless a new Saturday direct search fails. This example illustrates why conditionals must be represented separately from immediate updates.

5 Experiments

Evaluation data. We evaluate EDITAGENT on two complementary test sets sampled from larger automatically constructed candidate pools. The first test set measures state editing. We derive edit candidates from MultiWOZ 2.4 and SGD by

Component	Input / Output	Purpose
Edit parser	$(s_{t-1}, u_t, o_t) \rightarrow e_t$	Convert language revisions into typed edit operations.
Reducer	$(s_{t-1}, e_t) \rightarrow s_t$	Mutate state deterministically and preserve untouched fields.
Verifier	(e_t, s_t, a_t)	Detect invalid edits, stale leakage, and premature actions.
Planner	$(s_t, \text{schema}) \rightarrow a_t$	Ask clarification or call tools from current state.

Table 4: EDITAGENT decomposes revision-aware tool use into inspectable modules.

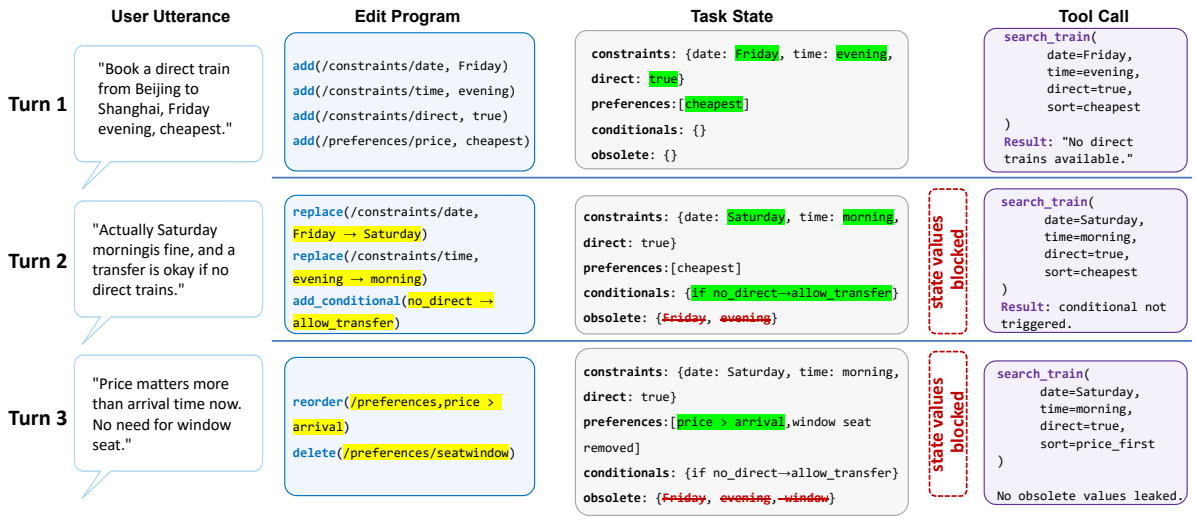

Figure 2: Multi-turn trace of executable state edits. Additions initialize active constraints, replacements move old values into the obsolete set, conditional edits remain inactive until triggered, and tool calls are generated from the reduced current state rather than from stale dialogue history.

taking adjacent gold dialogue states, computing typed state differences, and retaining turns whose edited values are grounded in the user utterance. Each instance contains a dialogue history, previous state, schema paths, user revision, gold edit program, and gold reduced state. The resulting pool contains additions, replacements, and deletions, with replacement-heavy sampling to stress non-monotonic updates.

The second test set measures stale-constraint leakage in tool calls. We construct revision-aware airline update tasks from τ-bench reservations. Each instance begins with a structured reservation state and a dialogue history that makes an old value active, then introduces a user revision such as explicit replacement, terse replacement, contrastive replacement, multi-field replacement, or a conditional fallback whose trigger observation is supplied. Gold tool arguments are produced by deterministic reduction over the revised state.

Baselines. We compare five methods. *Full-history direct* receives the dialogue history and latest user turn and directly predicts the current state or tool arguments. *Full-state tracking* rewrites the complete current state at each turn. EDITAGENT *without verifier* predicts edits and applies the deterministic reducer, but does not retry on verifier feedback. EDITAGENT adds rule-based edit checking and stale-value verification before state-grounded planning. *Stale-state lower bound* directly calls tools from the previous state and measures the worst case where obsolete constraints are never removed.

Source	Candidates	Rev. cand.
MultiWOZ 2.4 edits	37,820	4,286
SGD dev edits	22,302	11,817
SGD test edits	38,671	20,417
Total edit pool	98,793	36,520
τ-bench tool revisions	10,000	10,000

Table 5: Automatically constructed evaluation candidate pools. Rev. cand. denotes instances containing at least one `replace` or `delete` edit. The main API-evaluated table uses a stratified subset because each method requires model calls; larger sampled validation runs use the same pools.

Model setup. All LLM-based methods use `deepseek-v4-flash` through an OpenAI-compatible chat-completions interface. We use deterministic decoding, cache all model calls, and count reported total tokens as cost. For state-based tool-use methods, the planner is deterministic: it extracts the required tool-schema fields from the predicted current state. This isolates the effect of state updating from a second tool-planning prompt.

Metrics. For state editing, we report edit exact match, state exact match (State EM), and slot-level F1. For tool use, we report obsolete constraint leakage, argument accuracy, and state-action consistency. Leakage is the fraction of tool calls containing a value that the user revision made obsolete. API or parsing failures are counted as incorrect predictions and as leakage in tool-use evaluation.

Statistical testing. We compute bootstrap confidence intervals over instances. EDITAGENT obtains 66.3 edit accuracy with a 95% interval of [62.5, 70.0], 68.7 State EM with [65.0, 72.5], and the highest slot F1 among the compared methods (93.2). On tool revisions, EDITAGENT reaches 99.7 argument accuracy with [99.0, 100.0], compared with 64.7 [59.0, 70.3] for full-history direct prediction. Appendix 10 reports leakage and stale-state diagnostics.

6 Analysis

Main effect. The full-history baseline struggles because the history contains both active and obsolete constraints. On the stratified natural edit set, full-history direct prediction reaches 59.2 State EM, while EDITAGENT reaches 68.7 and obtains the strongest slot-level state quality (93.2 slot F1). Full-state tracking remains a strong baseline, slightly leading strict edit accuracy at 68.3, but EDITAGENT is comparable while exposing an auditable intermediate edit program. On tool revisions, state-grounded methods reach 99.7 argument accuracy, compared with 64.7 for full-history direct tool prediction.

Revision type. EDITAGENT performs best on replacement edits, reaching 73.3 edit accuracy and 85.6 edit F1 under strict exact-match scoring. This is the central case for stale-constraint prevention: an old value must be explicitly deactivated and replaced by a new one. Additions remain moderately difficult because the model sometimes normalizes values differently from the dataset. Deletions are the hardest category in the derived MultiWOZ/SGD set. Many deletion labels correspond to implicit entity or domain shifts rather than explicit user statements such as "remove this constraint," so the model often preserves a previous field rather than producing a `delete` operation.

Where errors occur. EDITAGENT's intermediate edit program makes errors local. In the current run, most state errors come from edit parsing rather than reduction: the deterministic reducer passes sanity checks when applied to gold edits. Common parser errors include preserving an obsolete field under implicit deletion, normalizing values differently from the dataset (e.g., relative dates or copied surface forms), and adding a confirm operation for a value already present in the state. Tool-call errors are rare after state reduction because the planner sees only the current state and the required tool schema.

Verifier effect. The verifier provides a modest but consistent gain on state quality. Removing it lowers strict edit accuracy from 66.3 to 65.7 and State EM from 68.7 to 68.2, while tool argument accuracy remains 99.7 in this controlled tool-revision set. This suggests that the main benefit comes from the edit-reducer decomposition, while verifier feedback catches a smaller number of invalid or stale updates before tool use.

Error taxonomy. We analyze five failure types, illustrated in Figure 3. *Missed edit* means the parser fails to produce an operation entailed by the user turn. *Wrong scope* means the parser edits the wrong field, such as changing destination instead of departure date. *Over-editing* means the parser changes fields not mentioned by the user. *Invalid reduction* means the edit is syntactically plausible but cannot be applied to the previous state. *Planner leakage* means the state is correct but the tool call includes an obsolete value.

7 Discussion

EDITAGENT is deliberately not a large agent framework. Its claim is narrower: revision-aware tool use benefits from representing state changes as executable, verifiable edits. This makes the method compatible with existing planners, memories, and tool environments. A ReAct or LATS planner can operate after the reducer; a MemGPT-style memory can store long-term user facts; a ToolSandbox or τ-bench environment can provide executable tools.

Method	Edit Acc.	State EM	Slot F1	Arg. Acc.	Consistency
Full-history direct	59.2	59.2	87.3	64.7	64.7
Full-state tracking	**68.3**	68.3	92.6	**99.7**	**99.7**
EDITAGENT w/o verifier	65.7	68.2	93.0	**99.7**	**99.7**
EDITAGENT	66.3	**68.7**	**93.2**	**99.7**	**99.7**

Table 6: Main results on 600 edit-tracking instances and 300 tool-revision instances. All scores are percentages. Exact edit match is strict; slot-level behavior and action consistency capture whether the reduced state supports reliable tool arguments. Diagnostic leakage and stale-state lower-bound results are reported in Appendix F.

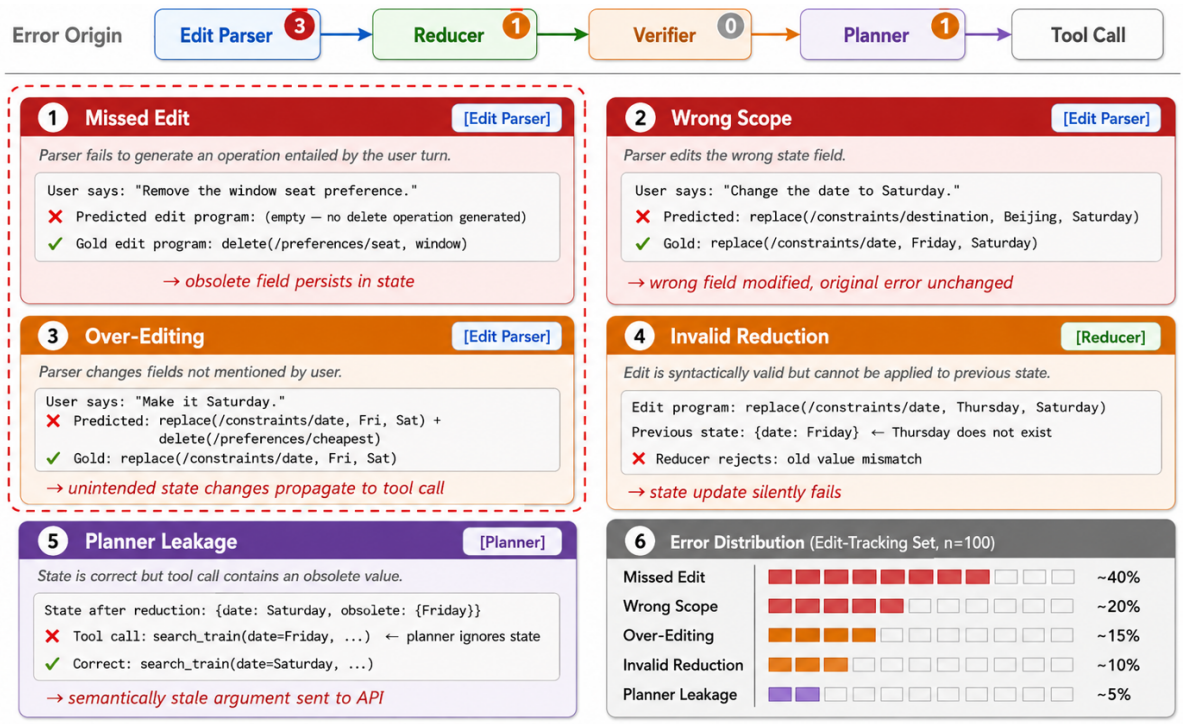

Figure 3: Error taxonomy and localization. Most failures originate in edit parsing, including missed edits, wrong-scope edits, and over-editing. Reducer and planner failures are easier to isolate because edit programs and reduced states are explicit intermediate artifacts.

Edit type	Edit Acc.	Edit F1	State EM
Add	58.8	80.0	60.8
Replace	**73.3**	**85.6**	**76.1**
Delete	4.7	53.0	7.0

Table 7: EDITAGENT performance by edit type on the edit-tracking set. Scores are percentages.

Revision type	Leakage ↓	Arg. Acc.
Terse replace	**0.0**	**100.0**
Explicit replace	**0.0**	**100.0**
Contrastive replace	**0.0**	**100.0**
Multi-field replace	**0.0**	**100.0**
Conditional fallback	1.9	98.1

Table 8: EDITAGENT tool-use results by revision type. Scores are percentages. Leakage is lower-is-better.

The edit layer controls which current constraints reach those components.

The closest prior work falls into two categories. AgentChangeBench and DialogTool evaluate dynamic or stateful tool-use behavior, but they do not propose an edit-program control mechanism. SOM-DST, ParsingDST, and FnCTOD model structured dialogue states, but they do not ground state edits into verified tool actions. EDITAGENT sits between them: it borrows selective state update from DST and makes it operational for tool agents.

8 Conclusion

We presented EDITAGENT, a revision-aware tool-agent architecture that converts user changes into executable state edit programs. The results show that explicit edits help agents keep task state current after user revisions.

9 Limitations

The first version of EDITAGENT targets structured API-style tools. GUI and web agents may require richer state paths and perceptual grounding. The edit parser is still an LLM component, so it can miss implicit revisions or hallucinate edits. The verifier can catch schema-level and stale-constraint errors, but cannot perfectly judge all semantic equivalence. Finally, benchmark construction with synthetic revisions must be validated carefully to avoid template artifacts.

References

Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Yuefeng Huang, Xiangcheng Liu, Wang Xinzhi, and 1 others. 2025. Acebench: A comprehensive evaluation of llm tool usage. *Findings of the Association for Computational Linguistics: EMNLP*, 2025:12970–12998.

Ting Han, Ximing Liu, Ryuichi Takanabu, Yixin Lian, Chongxuan Huang, Dazhen Wan, Wei Peng, and Minlie Huang. 2021. Multiwoz 2.3: A multi-domain task-oriented dialogue dataset enhanced with annotation corrections and co-reference annotation. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 206–218. Springer.

Michael Heck, Carel van Niekerk, Nurul Lubis, Christian Geishauser, Hsien-Chin Lin, Marco Moresi, and Milica Gasic. 2020. Trippy: A triple copy strategy for value independent neural dialog state tracking. In *Proceedings of the 21th annual meeting of the special interest group on discourse and dialogue*, pages 35–44.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. 2025. Memory os of ai agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 25972–25981.

Sungdong Kim, Sohee Yang, Gyuwan Kim, and Sang-Woo Lee. 2020. Efficient dialogue state tracking by selectively overwriting memory. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 567–582.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 3102–3116.

Zekun Li, Zhiyu Chen, Mike Ross, Patrick Huber, Seungwhan Moon, Zhaojiang Lin, Xin Luna Dong, Adithya Sagar, Xifeng Yan, and Paul A Crook. 2024. Large language models as zero-shot dialogue state tracker through function calling. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8688–8704.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, and 1 others. 2025. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1160–1183.

Yi Ma. 2013. User goal change model for spoken dialog state tracking. In *Proceedings of the 2013 NAACL HLT Student Research Workshop*, pages 91–97.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. 2024. Evaluating very long-term conversational memory of llm agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13851–13870.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. 2023. Memgpt: towards llms as operating systems.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. 2025. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2024. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, volume 2024, pages 9695–9717.

Manik Rana, Calissa Man, Anotida Expected Msiiwa, Jeffrey Paine, Kevin Zhu, Sunishchal Dev, Vasu Sharma, and 1 others. 2025. Agentchangebench: A multi-dimensional evaluation framework for goal-shift robustness in conversational ai. *arXiv preprint arXiv:2510.18170*.

Abhinav Rastogi, Xiaoxue Zang, Srinivas Sunkara, Raghav Gupta, and Pranav Khaitan. 2020. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8689–8696.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Hongru Wang, Wenyu Huang, Yufei Wang, Yuanhao Xi, Jianqiao Lu, Huan Zhang, Nan Hu, Zeming Liu, Jeff Z Pan, and Kam-Fai Wong. 2025. Rethinking stateful tool use in multi-turn dialogues: Benchmarks and challenges. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 5433–5453.

Chien-Sheng Wu, Andrea Madotto, Ehsan Hosseini-Asl, Caiming Xiong, Richard Socher, and Pascale Fung. 2019. Transferable multi-domain state generator for task-oriented dialogue systems. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pages 808–819.

Yuxiang Wu, Guanting Dong, and Weiran Xu. 2023. Semantic parsing by large language models for intricate updating strategies of zero-shot dialogue state tracking. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 11093–11099.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Fanghua Ye, Jarana Manotumruksa, and Emine Yilmaz. 2022. Multiwoz 2.4: A multi-domain task-oriented dialogue dataset with essential annotation corrections to improve state tracking evaluation. In *Proceedings of the 23rd Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 351–360.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 19724–19731.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

A Dataset Construction Details

Edit-tracking instances. For MultiWOZ 2.4 and SGD, we construct edit-tracking examples from adjacent dialogue states. At each user turn, we take the previous gold state s_{t-1} and the next gold state s_t, canonicalize both into nested key-value records, and compute a structural difference. Newly observed paths become `add` edits; paths whose values change become `replace`; and paths present in s_{t-1} but absent in s_t become `delete`. We retain the full dialogue history available before the user turn, the previous state, the user utterance, the gold edit program, and the resulting gold state.

Filtering. The raw state difference can include artifacts that are not recoverable from the local user utterance, such as ontology normalization, implicit service transitions, or system-side slot filling. We therefore apply lightweight filters before sampling. We remove turns with empty state differences, discard examples where neither the old nor new value is grounded in the local context, and prioritize examples containing replacements or deletions because these are the most relevant to stale-constraint leakage. We do not train on these examples; they are used only for zero-shot and few-shot evaluation.

Tool-revision instances. For tool-use evaluation, we construct controlled revision tasks from τ-bench-style airline reservations. Each instance contains a previous reservation state, a short history that makes an old value active, a revision utterance, obsolete values, a tool schema, and gold tool arguments. The revision templates cover terse replacement, explicit replacement, contrastive replacement, multi-field replacement, and conditional fallback. Gold tool arguments are produced by applying the gold edit program to the previous state and extracting the required schema fields.

Data fields. Each edit-tracking record contains:

`dialogue_id`, `turn_id`, `history`, `previous_state`, `user_utterance`, `gold_edits`, `gold_state`.

Each tool-revision record contains:

`previous_state`, `revision_utterance`, `obsolete_values`, `tool_schema`, `required_tool_arg_keys`, `gold_edits`, `gold_tool_args`.

B Additional Edit Examples

C Method Prompts

All LLM calls use deterministic decoding. We instruct the model to produce machine-readable JSON and count unparsable outputs as incorrect. The following prompt sketches omit few-shot demonstrations for space but preserve the task interface.

Full-history direct state.

> You are tracking the current task state. Given the full dialogue history and the latest user turn, output the current state as JSON. Include only currently active constraints, preferences, conditionals, and unresolved fields. Do not include obsolete constraints that the user has replaced, deleted, or relaxed.

Full-state tracking.

> You are given the previous state and the latest user turn. Rewrite the complete current state as JSON. Preserve unchanged fields, update changed fields, delete fields that are no longer active, and include conditionals only when the user states a fallback rule.

Edit parser.

> You are an edit parser. Given the previous state, the latest user utterance, the latest tool observation, and the allowed schema paths, output only a JSON list of edit operations. Allowed operations are `add`, `replace`, `delete`, `relax`, `tighten`, `add_conditional`, `reorder`, and `confirm`. Do not rewrite untouched fields. If the user changes a value, use `replace` and include both the old and new values. If the user cancels a requirement, use `delete`. If the user gives a fallback, use `add_conditional`.

Tool argument prediction.

> Given the verified current state and the tool schema, output JSON tool arguments. Use only active values in the current state. Never include values listed in `obsolete`. If required fields are missing, output a clarification request instead of a tool call.

D Reducer and Verifier

Reducer. The reducer is deterministic and schema-aware. It applies edits in order and either returns an updated state or a structured error. It is intentionally simple: unchanged fields persist, edited fields are updated locally, and obsolete values are tracked separately from active constraints.

```
reduce(state, edits):
  next = copy(state)
  for edit in edits:
    check path is allowed by schema
    if edit.op == add:
      insert value at path
    if edit.op == replace:
     require old value exists or is equivalent
      move old value to obsolete
      write new value at path
    if edit.op == delete:
      require value or path exists
      move value to obsolete
      remove value from active state
    if edit.op == add_conditional:
      append trigger-action rule
    if edit.op == reorder:
      update preference order only
  canonicalize values
  return next
```

Verifier. The verifier checks both edit validity and stale leakage. The first layer rejects schema errors, old-value mismatches, duplicate contradictory edits, and missing required fields. The second layer checks whether any obsolete value appears in active state fields or generated tool arguments. If a violation is detected, EDITAGENT sends a concise error message to the edit parser and allows one retry.

```
verify(previous, edits, reduced, tool_args):
  errors = []
  errors += schema_path_errors(edits)
 errors += old_value_mismatch(previous, edits)
  errors += contradictory_edits(edits)
  for value in reduced.obsolete:
   if value appears in active_fields(reduced):
     errors += ["obsolete value still active"]
     if value appears in tool_args:
     errors += ["obsolete value leaked to tool"]
  return errors
```

E Metric Definitions

Edit exact match. An example receives credit if the predicted edit set exactly matches the gold edit set after operation normalization, path normalization, and value canonicalization. Because edit order is usually not semantically meaningful, we compare normalized edit multisets except when two edits touch the same path.

User turn	Edit program	Effect on state
"Actually make it Saturday morning."	`replace(/constraints/date, Fri, Sat); replace(/constraints/time, evening, morning)`	Old date/time moved to obsolete; unrelated route and budget persist.
"I no longer need a window seat."	`delete(/preferences/seat, window)`	Window-seat preference removed; seat field no longer affects tool calls.
"If direct trains are sold out, a transfer is fine."	`add_conditional(no_direct, allow_transfer)`	Direct remains primary until the tool observation triggers fallback.
"Price matters more than arrival time now."	`reorder(/preferences, price > arrival)`	Preference priority changes without changing hard constraints.

Table 9: Edit programs preserve local revision semantics. This is the main method-level distinction from full-state regeneration.

Edit operation F1. We compute micro F1 over operation-path pairs. A predicted edit receives operation credit when both its operation type and state path match a gold edit, regardless of minor value normalization differences. This metric separates structural update recognition from exact value rendering.

State exact match and slot F1. State EM requires the reduced predicted state to match the gold state after canonicalization. Slot F1 treats flattened state paths as key-value items and computes precision, recall, and F1 over active slots.

Over-edit and missed-edit rates. Over-edit rate is the fraction of predicted edits that do not correspond to any gold edit. Missed-edit rate is the fraction of gold edits not recovered by the model. These two metrics distinguish hallucinated state changes from failures to update stale fields.

Tool-use metrics. Argument accuracy requires all required tool arguments to match the gold arguments. Obsolete-constraint leakage is the fraction of tool calls that include at least one obsolete value. State-action consistency requires the tool arguments to be entailed by the predicted current state and not contradicted by the obsolete set. Clarification-or-action accuracy checks whether the method makes the correct high-level decision: ask for missing information or call the tool.

F Additional Results

G Qualitative Error Cases

Table 14 gives representative failure modes observed during manual inspection of logged error cases. These examples are intended to clarify what the automatic metrics count as missed edits, wrong-scope edits, over-editing, invalid reduction, and planner leakage.

H Benchmark Construction Examples

Table 15 summarizes the controlled tool-revision templates. The templates are not used to train the model; they are used to create a small, interpretable stress test for stale-constraint leakage. Each template contains an old value made active in the history, a user revision that makes that value obsolete, and gold tool arguments generated from the reduced state.

Why controlled revisions? Existing tool-use benchmarks contain many realistic tasks but do not always isolate whether an error comes from tool selection, schema grounding, environment dynamics, or stale task state. Our revision templates intentionally control the tool schema and required arguments so that the evaluation focuses on state mutation before action. This does not replace full interactive benchmarks; instead, it provides a diagnostic complement that can be run cheaply and inspected manually.

Relationship to real dialogue. The edit-tracking set covers naturally occurring MultiWOZ/SGD turns, while the tool-revision set provides more controlled obsolete-value tests. The two sets therefore stress different aspects of the method: naturally occurring state changes for edit parsing, and targeted stale-value suppression for tool arguments.

I Reproducibility Notes

Model and API. All reported LLM calls use `deepseek-v4-flash` through an OpenAI-compatible chat-completions endpoint. API keys are read only from environment variables or local configuration and are not committed. The default environment variables are `DEEPSEEK_API_KEY`, `DEEPSEEK_BASE_URL`, and `EDITAGENT_MODEL`.

Method	Edit Acc.	State EM	Leakage ↓	Arg. Acc.	Consistency
Full-history direct	59.2	59.2	**0.0**	64.7	64.7
Full-state tracking	**68.3**	68.3	0.3	**99.7**	**99.7**
EDITAGENT w/o verifier	65.7	68.2	0.3	**99.7**	**99.7**
EDITAGENT	66.3	**68.7**	0.3	**99.7**	**99.7**
Stale-state lower bound	0.0	0.0	100.0	0.0	0.0

Table 10: Full diagnostic result table including obsolete-constraint leakage and the stale-state lower bound. Scores are percentages. The lower bound is not a deployable method; it verifies that the tool-revision set requires updating obsolete constraints before action.

Method	Edit type	Edit Acc.	Edit F1	Missed	Over-edit
Full-history direct	Add	52.0	76.2	18.3	22.0
Full-history direct	Replace	65.6	84.6	8.6	16.3
Full-history direct	Delete	38.1	73.4	24.1	17.7
Full-state tracking	Add	60.8	81.0	20.8	10.3
Full-state tracking	Replace	75.2	87.3	13.3	6.4
Full-state tracking	Delete	19.0	63.5	44.1	7.7
EDITAGENT	Add	58.8	80.0	21.2	13.5
EDITAGENT	Replace	73.3	85.6	14.6	8.5
EDITAGENT	Delete	4.7	53.0	58.1	11.0

Table 11: Additional edit-tracking breakdown by edit type. Scores are percentages.

Caching. Every model request is cached on disk using a hash of the model name, prompt payload, method, and sample identifier. Re-running an experiment with the same configuration reads from cache and does not call the API again. This is important because the evaluation uses multiple prompting methods over the same examples.

Sanity checks. Before running model evaluation, we apply the deterministic reducer to every gold edit program and require the resulting state to equal the gold state. We also inspect random generated examples to verify that replacements, deletions, and additions are plausible. During model evaluation, unparsable JSON outputs are logged and counted as incorrect. Error cases are written to JSONL with the input, gold labels, prediction, reduced state, and metric failures.

Cost accounting. The reported cost column is based on total token usage returned by the API. Because cached calls are not reissued, repeated summarization and metric scripts do not increase the measured model-call cost.

Revision type	Arg. Acc.	Leakage ↓	Consistency	Action Acc.
Terse replace	100.0	0.0	100.0	100.0
Explicit replace	100.0	0.0	100.0	100.0
Contrastive replace	100.0	0.0	100.0	100.0
Multi-field replace	100.0	0.0	100.0	100.0
Conditional fallback	98.1	1.9	98.1	98.1

Table 12: EDITAGENT tool-use breakdown by revision type. Scores are percentages. Leakage is lower-is-better.

Method	Setting	MultiWOZ 2.4 JGA	Slot F1 / Acc.
TRADE	supervised DST	55.05	–
SimpleTOD	supervised DST	57.18	–
TripPy	supervised DST	64.75	–
SOM-DST	supervised DST	66.78	–
STAR	supervised DST	73.62	–
EDITAGENT	zero-shot sampled turns, oracle previous	82.00	96.04

Table 13: Supplementary MultiWOZ 2.4 context. Prior supervised numbers are reported in MultiWOZ 2.4 references; EDITAGENT results are sampled test-turn evaluations and should not be read as a strict leaderboard comparison.

Failure type	User revision	Typical wrong output	Why it fails
Missed edit	"I no longer need a window seat."	Empty edit list, previous state unchanged.	The obsolete preference remains active and may later be copied into a tool call.
Wrong scope	"Change the date to Saturday."	`replace(/destination, Beijing, Saturday)`	The parser detects a replacement but edits the wrong state path.
Over-editing	"Make it Saturday."	Replaces date and also deletes price preference.	The model changes an untouched field, reducing state precision.
Invalid reduction	"Use Saturday instead." with previous date Friday	`replace(/date, Thursday, Saturday)`	The old value does not exist in the previous state, so the reducer rejects the edit.
Planner leakage	Correct reduced state has Saturday and obsolete Friday.	Tool call contains `date=Friday`.	The planner ignores the reduced state or copies a stale value from history.

Table 14: Representative qualitative error cases. The explicit edit representation makes each failure attributable to a parser, reducer, verifier, or planner stage.

Revision type	Revision utterance pattern	Obsolete value	Expected behavior
Terse replace	"Actually make it economy."	Previous cabin class.	Infer replacement from context and update the active cabin.
Explicit replace	"Switch the cabin to economy; do not keep business."	Mentioned old cabin.	Replace the old value and block it from tool arguments.
Contrastive replace	"Economy, not business."	Contrastive old value.	Resolve the contrast and avoid copying the negated value.
Multi-field replace	"Move it to May 28 and make the cabin economy."	Previous date and cabin.	Apply multiple replacements before tool planning.
Conditional fallback	"Keep business only if economy is unavailable; otherwise economy."	Conditional old value.	Represent the fallback condition and activate the correct value under the supplied observation.

Table 15: Tool-revision construction templates. These templates isolate non-monotonic updates that full-history prompting often handles implicitly and unreliably.