

A TOOL-ASSISTED AGENT FOR PREPROCESSING DIAGNOSIS AND REPAIR IN TABULAR ML WORKFLOWS

Jonghu Jeong, Hsien-Hao Li, Sean Kraemer, Shreya Rao

University of Illinois Urbana-Champaign

{jonghuj2, hl128, seanmk2, shreya29}@illinois.edu

1 INTRODUCTION

Data preprocessing is a central but error-prone part of practical tabular machine learning. Before training, practitioners must decide how to handle missing values, encode categorical features, aggregate relational tables, align temporal signals, and avoid leakage-prone transformations. These decisions are highly context-dependent, which makes preprocessing a natural target for LLM agents but also a difficult setting for controlled evaluation.

In this work, we study preprocessing diagnosis and repair on KaggleBench, which formulates the task as constrained editing of a testcase-specific pipeline state rather than open-ended pipeline generation. Given task context, a current set of active preprocessing actions, and a fixed candidate action bank, the system must decide which beneficial inactive actions to add and which harmful active actions to remove. This setting exposes a limitation of existing approaches: one-shot prompting cannot iteratively inspect missing evidence, while general-purpose coding agents are not naturally aligned with preprocessing-specific repair under a constrained action space.

We propose a benchmark-aware agent that combines iterative tool use with problem decomposition tailored to preprocessing repair. Our design separates add-side recovery from remove-side rollback, uses preprocessing-specific inspection tools, and returns structured benchmark predictions grounded in the benchmark’s input and output contract. We evaluate the agent on all 20 tasks against a deterministic rule baseline, a one-shot LLM baseline, a generic agent, Claude Code, and a human `tbl_from_scratch` reference. Empirically, the proposed agent does not outperform the strongest one-shot LLM baseline overall. The main contribution is therefore diagnostic: benchmark-specific tools and add/remove decomposition reveal rollback failures that remain unsolved by agent orchestration alone.

The rest of the paper reviews related work, presents the agent and setup, reports results, discusses limitations, and concludes (Sections 2–7).

2 RELATED WORK

Recent work relevant to our setting falls into two broad groups: generic coding agents and data-centric agents. Generic agents such as SWE-agent couple LLM reasoning with broad interfaces for reading files, editing code, and running commands, while commercial systems such as Claude Code extend the same pattern to polished developer workflows with multi-step codebase interaction and execution support (Yang et al., 2024; Anthropic, 2026). These systems are relevant baselines, but they target open-ended software tasks rather than constrained preprocessing repair over a fixed action bank.

A second line of work brings agents closer to data science and tabular ML. AutoKaggle and AutoML-Agent decompose end-to-end Kaggle or AutoML workflows across multiple roles or sub-tasks, suggesting that full-pipeline data problems benefit from explicit task decomposition (Li et al., 2024; Trirat et al., 2024). CleanAgent and DeepPrep further show that data cleaning and preparation benefit from domain-specific operations, execution-grounded iteration, and structured revision over intermediate table states (Qi et al., 2024; Fan et al., 2026). Our agent follows this broader pattern, but specializes it to preprocessing diagnosis under constrained evaluation: we repair a testcase-specific

preprocessing state under a fixed action bank by separating add-side recovery from remove-side rollback and using benchmark-specific tools rather than only generic coding interfaces. Our contribution is therefore a domain-specialized instantiation of existing agent patterns for preprocessing diagnosis and repair.

3 AGENT DESIGN

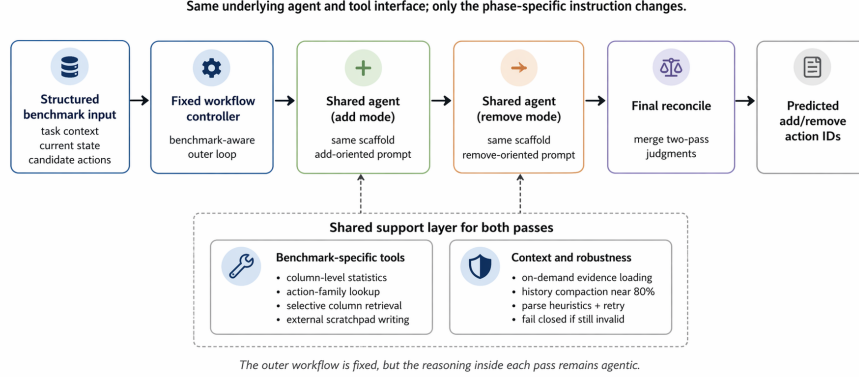


Figure 1: Architecture of our benchmark-aware agent.

KaggleBench frames the task as structured preprocessing repair over a fixed action bank. Our system is a benchmark-aware workflow rather than a free-form planner: a lightweight controller orchestrates two diagnostic repair phases, first in add mode and then in remove mode. The agent itself is the model–tool–scratchpad loop invoked by the controller in each phase; the orchestration is our own implementation rather than an existing agent framework. Figure 1 summarizes the information flow.

The agent receives structured task context, testcase context, and the visible candidate action bank. It performs diagnostic decision making rather than data transformation: each pass gathers evidence to judge candidate actions and returns phase-specific predictions. The add phase targets inactive beneficial actions, while the remove phase targets active harmful actions and receives the add-phase output to reason about compatibility with the repair set. This separation narrows the decision space without asking the model to solve the full add/remove problem in one pass.

Evidence collection uses benchmark-specific tools rather than open-ended coding or filesystem exploration. The tool surface supports action lookup, context summaries, dataset inspection, column profiling, row previews, join-key profiling, target profiling, and scratchpad read/write. Tool outputs are truncated, and a run-local scratchpad is shared across phases so observations can persist without reloading the full context. Each phase runs under fixed turn, tool-call, timeout, and cost budgets, keeping iteration focused on preprocessing diagnosis.

After both phases complete, the controller applies a deterministic reconcile step that merges phase-local judgments into a benchmark-compatible prediction. Reconcile is not an additional agentic planning pass: it deduplicates action IDs, enforces inactive-add and active-remove constraints, resolves direct add/remove conflicts, applies a narrow raw-column blocker check, and validates the final payload against the visible action bank.

We use Claude Sonnet 4.6 as the underlying model and a shared prompt scaffold containing task description, dataset summary, testcase context, candidate actions, tool constraints, and output format. Outputs are parsed as JSON and validated with bounded retries; invalid runs are marked failed rather than silently repaired. Overall, the design trades open-ended coding flexibility for benchmark alignment. One-shot prompting can compare the whole bank globally but cannot inspect missing evidence, while generic agents may waste context on irrelevant workspace exploration. Our method constrains iteration around the benchmark’s add/remove repair decision while retaining enough tool use to resolve missing evidence. Detailed settings, prompts, and tool surfaces are in Appendices A, B, and C.

Table 1: Overall primary-scope results across all 20 tasks.

Method	Add F1	Remove R	TC $\mu \pm \sigma$	Success	Cost/run	Tokens/run	Time/run (s)	n
Human	0.772	1.000	0.886 ± 0.107	1.000	n/a	n/a	n/a	20
Rule-based	0.223	0.633	0.428 ± 0.240	0.562	\$0.000	n/a	0.0	80
Single LLM	0.676	0.878	0.777 ± 0.176	0.963	\$0.057	2034.0	36.4	80
Generic agent	0.642	0.889	0.766 ± 0.178	0.950	\$0.143	15855.2	81.1	80
Claude Code	0.614	0.899	0.756 ± 0.194	0.925	\$0.310	5718.9	127.6	80
Proposed agent	0.676	0.822	0.749 ± 0.205	0.912	\$0.196	20389.0	109.2	80

For LLM-based methods, $n = 80$ denotes grouped task-testcase units, each averaged over five runs. Task-completion $\pm$ values are standard deviations across grouped units. The human baseline covers only `tc1_from_scratch` and is included as a reference, not as a full four-scenario method comparison.

Table 2: Task completion by testcase scenario.

Method	TC1	TC2	TC3	TC4	Overall
Human	0.886	not run	not run	not run	0.886
Rule-based	0.639	0.583	0.269	0.221	0.428
Single LLM	0.876	0.822	0.671	0.738	0.777
Generic agent	0.872	0.799	0.663	0.728	0.766
Claude Code	0.883	0.810	0.635	0.697	0.756
Proposed agent	0.870	0.864	0.605	0.655	0.749

4 EXPERIMENTAL SETUP

Benchmark and Metrics We evaluate on KaggleBench, where each testcase provides a candidate action bank and current pipeline state, and the system predicts actions to add and remove. We report Action Precision/Recall, Add F1, Remove Recall, Task Completion Score, and Task Success Rate. The main metric is $0.5 \times \text{Add F1} + 0.5 \times \text{Remove Recall}$. A run is counted as successful when its Task Completion Score is at least threshold $\tau = 0.5$, matching the benchmark validation setting. We also report runtime, token usage, tool usage, and cost when available.

Baselines We compare against a deterministic rule baseline, a one-shot LLM with the same provider family and prompt scaffold, a generic agent using broad workspace tools but no benchmark-specific profilers, Claude Code (Anthropic, 2026), and a human `tc1_from_scratch` reference. Single LLM and Generic Agent are partial ablations, not controlled ones: multiple factors differ, and we do not yet isolate the two-pass decomposition, benchmark-specific tools, scratchpad, and reconciliation stage independently.

Experimental Settings The final evaluation runs every automated method on all four testcase scenarios for all 20 tasks, including Zillow; Claude Code is therefore an $n = 80$ reference rather than a `tc1`-only comparison. Each non-heuristic agent-testcase configuration is evaluated over 5 runs. The main tables report the grouped means; the $\pm$ values report the standard deviation in the grouped task test case means after repeated runs are averaged within each unit. We make descriptive rather than formal significance claims because methods differ in runtime interface. Claude Code is treated as an external commercial coding-agent reference rather than a controlled ablation because its runtime, tool policy, hidden orchestration, and cost/token reporting differ from the repo-run methods.

5 RESULTS

Table 1 reports the final primary-scope results over all 20 tasks, using threshold $\tau = 0.5$ to compute task success. Each automated LLM-based method has 400 single runs, corresponding to five attempts for each task-testcase pair; the deterministic rule-based baseline has one run per pair, and the human reference covers the 20 `tc1_from_scratch` cases. The strongest automated method overall is the Single LLM baseline, with task completion 0.777 and success rate 0.963. The proposed agent is competitive on additions, matching the Single LLM’s rounded Add F1 at 0.676, but its lower remove recall of 0.822 reduces overall task completion to 0.749. Average within-group repeated-run variability is smaller than cross-task heterogeneity: within-group SD ranges from 0.045 for Single LLM to 0.060 for Claude Code, while grouped task-testcase SDs range from 0.176 to 0.205 for automated LLM methods.

Table 2 shows that method separation is clearest in repair-heavy settings. All strong automated methods are close on TC1 and TC2, but TC3 and TC4 expose rollback and mixed-history weakness. The proposed agent is strongest on TC2 (0.864), suggesting that its decomposition helps when the task is mostly add-side continuation. However, it drops to 0.605 on TC3 and 0.655 on TC4, where missing harmful active actions has a larger effect. Claude Code has the strongest automated remove recall at 0.899, but it is also the most expensive method at \$0.310 per run on average.

Add/remove and efficiency breakdowns are reported in Appendix Tables 7 and 8. These confirm that the proposed agent matches Single LLM on rounded Add F1 but lags on remove recall, while Claude Code has the strongest remove-side metrics and the highest recorded cost.

A representative success is Allstate TC2, where the proposed agent selected CA-000027, matching the target-transform gold unit, with no removals required. A representative failure is Rossmann TC4: the agent proposed plausible additions but predicted no removals, missing three harmful active actions and scoring 0.125. This failure mirrors the TC3/TC4 aggregate drop, where the agent often proposes plausible additions but under-removes active harmful actions. Zillow shows the same rollback pattern: the proposed agent remains competitive on TC1/TC2 but falls to 0.321 on TC3 and 0.334 on TC4, suggesting that fault-injected and mixed-history states require more explicit harmful-action verification.

Appendix Tables 4, 5, and 6 report robustness, per-task means, and the four-scenario Zillow comparison; Appendix Tables 9 and 10 provide additional breakdowns by problem type, table structure, and recorded preprocessing-complexity bucket.

6 DISCUSSION

The full benchmark results suggest three takeaways. First, structured benchmark inputs support nontrivial preprocessing repair: the rule-based baseline remains far below LLM-based methods, while human `tc1` references solve the easiest scenario reliably. Second, one-shot prompting is a strong automated baseline in this constrained action-bank setting: the fixed action bank makes global comparison surprisingly effective, and direct prompting avoids spending budget on evidence gathering that may not improve final ID selection. Third, TC3 and TC4 separate methods more clearly than cleaner recovery settings because they require recognizing harmful active actions, not just recommending useful inactive actions.

One interpretation is that KaggleBench’s fixed action bank reduces the advantage of tool use: the one-shot model can compare all visible candidates in one global context, while tool-using agents must decide what evidence to inspect and may lose global coherence across turns. The proposed two-pass split helps when the problem is mostly add-side continuation, as in TC2, but it can also separate evidence for additions from evidence for removals. This makes final reconciliation fragile in TC3/TC4, where success depends on distrusting prior context and removing plausible but harmful active actions.

The main limitation of the proposed agent is selective editing of prior context. Two-pass decomposition helps add-side continuation on TC2, but it creates reconciliation and rollback failure modes: weak runs overpredict broad additions, miss harmful active actions, or drift toward a generic good pipeline instead of testcase-specific repair. Claude Code is strongest on remove-side metrics, but this does not translate into best repair-scenario completion and it is the most expensive method. A better next version should keep benchmark-specific tools while adding explicit rollback checks, stricter final-state validation, and targeted re-inspection of active bad candidates before reconciliation.

7 CONCLUSION

We presented a benchmark-aware agent design for preprocessing diagnosis and repair in tabular machine learning workflows and evaluated it across the full 20-task KaggleBench suite. The results show that task-aware tools and decomposition produce competitive add-side behavior, but that rollback and final repair consistency remain the limiting factors. Future work should focus on explicit harmful-action verification, ablations for tool access and two-pass decomposition, and semantic matching that can recognize valid preprocessing alternatives beyond exact action-bank IDs.

REFERENCES

- Anthropic. Claude code overview. Claude Code documentation, 2026. Accessed 2026-04-15.
- Meihao Fan, Ju Fan, Yuxin Zhang, Shaolei Zhang, Xiaoyong Du, Jie Song, Peng Li, Fuxin Jiang, Tieying Zhang, and Jianjun Chen. Deepprep: An llm-powered agentic system for autonomous data preparation. *arXiv preprint arXiv:2602.07371*, 2026.
- Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, Wanjun Zhong, Wangchunshu Zhou, Wenhao Huang, and Ge Zhang. Autokaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424*, 2024.
- Danrui Qi, Zhengjie Miao, and Jiannan Wang. Cleanagent: Automating data standardization with llm-based agents. *arXiv preprint arXiv:2403.08291*, 2024.
- Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. Automl-agent: A multi-agent llm framework for full-pipeline automl. *arXiv preprint arXiv:2410.02958*, 2024.
- John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024. NeurIPS 2024 Poster.

A IMPLEMENTATION SETTINGS

Table 3: Verified implementation settings from the repo runners and provenance metadata.

Setting	Value
Model and provider	Repo-run LLM methods use <code>claude-sonnet-4-6</code> through the Anthropic API; Claude Code uses the Claude Code CLI with the same model name.
Generation controls	Default max output tokens = 8192; reasoning is enabled with adaptive summarized thinking and medium effort; reasoning-enabled requests force temperature to 1.0, otherwise the default temperature is 0.0.
Prompt caching and cost	Prompt cache enabled with ephemeral 5-minute TTL. Cost estimates use in-repo rates: input \$3/M, output \$15/M, cache read \$0.30/M, cache write 5m \$3.75/M, cache write 1h \$6/M tokens.
Common limits	Request timeout = 120 seconds; per-run cost cap = \$1.00; scratchpad limit = 20,000 characters; tool-output truncation = 4,000 characters.
Single LLM	One provider call per attempt, no tools, max attempts = 1.
Generic agent	Max turns = 10, max tool calls = 30, max validation retries = 1, tool timeout = 30 seconds; tools are <code>bash</code> , <code>python</code> , <code>scratchpad_read</code> , and <code>scratchpad_write</code> .
Proposed agent	Add and remove phases each have max turns = 8, max tool calls = 20, and max validation retries = 1; action lookup limit = 50; row preview limit = 20; tool allowlist is shown in Appendix C.
Claude Code	CLI command uses <code>print/verbose stream-JSON</code> mode, partial messages, model <code>claude-sonnet-4-6</code> , <code>bypassPermissions</code> , and added task/dataset directories; default auth mode is subscription and <code>CLAUDE_CODE_EFFORT_LEVEL=auto</code> .
Evaluation repetitions	LLM-based automated methods are run five times for each of 80 task-testcase units; <code>rule_based</code> is deterministic and run once per unit; human reference covers 20 <code>tcl_from_scratch</code> units.

B PROMPT TEMPLATES

Shared prompt scaffold. All LLM-based baselines use the same six-block scaffold: [TASK], [Method-specific instruction], [Dataset summary], [Current context (testcase)], [Candidate actions], and [Output format]. The [TASK] block contains not only the dataset-specific ML objective but also shared repair guidance: the model is told that this is a constrained action-selection repair task rather than an open-ended pipeline design task, that it must choose only from the provided candidate action bank, and that `predicted_add_action_ids` and `predicted_remove_action_ids` correspond to adding currently inactive actions and removing currently active harmful actions, respectively. The shared guidance also instructs the model to avoid proposing generally useful but instance-irrelevant actions, prefer a small high-confidence edit set, and leave `predicted_remove_action_ids` empty when `context_action_ids` is empty. The testcase block provides the testcase identifier, scenario, and current `context_action_ids`, while the candidate-action block exposes the full visible action bank as JSON with only `action_id`, `action_type`, and `canonical_params`. Finally, the output-format block requires benchmark predictions with `predicted_add_action_ids` and `predicted_remove_action_ids`, with optional notes or rationales depending on the method.

```
[TASK]
<dataset-specific ML objective>

Shared repair guidance:
- constrained action-selection repair task
- choose only from the provided candidate action bank
- predicted_add_action_ids = inactive actions to add
- predicted_remove_action_ids = active harmful actions to remove
- prefer a small, high-confidence edit set
- if context_action_ids is empty, predicted_remove_action_ids should
  normally be empty

[Method-specific instruction]
<method-dependent execution instruction>

[Dataset summary]
<precomputed dataset profile, or a method-specific instruction to
  inspect raw files>

[Current context (testcase)]
Testcase: <testcase_id>
```

```

Scenario: <scenario>
Current context_action_ids: [...]

[Candidate actions]
[
  {
    "action_id": "...",
    "action_type": "...",
    "canonical_params": {...}
  },
  ...
]

[Output format]
Return benchmark JSON with predicted_add_action_ids and
predicted_remove_action_ids
(+ optional notes/rationales when enabled).

```

One-shot LLM baseline. The one-shot LLM baseline uses the shared scaffold but executes it as a direct single-pass model call without tool use. In the implementation, the prompt is split into a static part and a dynamic part: the shared task framing, dataset summary, candidate actions, and output-format instructions are placed in the static prompt, while the testcase-specific context block is provided separately as dynamic input. The method-specific instruction is:

```

[Method-specific instruction]
Produce add/remove predictions in one pass without tool use.
Do not ask follow-up questions.
Use only the provided benchmark instance materials and return a
single structured JSON prediction.

```

Proposed agent add pass. The proposed agent uses the shared scaffold twice. In the first pass, it focuses only on inactive beneficial actions that should be added:

```

[Method-specific instruction]
You are in ADD mode. Identify inactive candidate actions that should be
added to repair the current preprocessing state. Use benchmark-specific
inspection tools when evidence is missing. Do not remove actions in this
pass. Prefer a compact, high-confidence set of expert-supported
additions.

```

Proposed agent remove pass. The second pass inspects only active context actions and decides which harmful actions should be removed:

```

[Method-specific instruction]
You are in REMOVE mode. Inspect the active context actions and identify
harmful, leakage-prone, unsupported, or context-incoherent actions that
should be rolled back. Do not add actions in this pass. If no active
context action is harmful, return an empty removal set.

```

Generic agent baseline. The generic agent keeps the same repair contract but removes benchmark-specific inspection tools:

```

[Method-specific instruction]
Use general workspace tools only to inspect the benchmark instance and
produce a final structured add/remove prediction. You may read files,
run lightweight commands, and write scratchpad notes, but you do not have
access to benchmark-specific action lookup or profiling tools.

```

Claude Code baseline. The Claude Code baseline keeps the same benchmark framing and output contract, but runs as a workspace-based agent rather than a direct one-shot completion. Unlike the one-shot baseline, it receives the full prompt as a single workspace prompt and can additionally inspect testcase and dataset files in the workspace before writing the final prediction. The method-specific instruction is:

```
[Method-specific instruction]
Use the provided workspace files to analyze the benchmark instance and
write the final JSON prediction to 'prediction.json'. You may inspect
files in the current working directory, including the actual raw dataset
files exposed at 'dataset'. Keep the final answer limited to the required
benchmark prediction structure.
```

C BENCHMARK-SPECIFIC TOOL SURFACE

We distinguish between two meanings of tool in our current implementation. First, the shared repo tool layer provides benchmark-specific callable backends. Second, Claude Code uses external runtime tools exposed by the Claude Code CLI.

C.1 SHARED BENCHMARK-SPECIFIC TOOLS

The current shared registry exposes tools for task loading, visible action-bank construction, dataset path resolution, raw row loading, joined-view loading, and profile computation. Concretely, the currently registered tools are:

- `load_task_bundle`
- `build_agent_visible_action_bank`
- `resolve_dataset_paths`
- `load_csv_rows`
- `load_joined_training_view`
- `profile_table_schema`
- `profile_missingness`
- `profile_numeric_columns`
- `profile_boolean_like_columns`
- `profile_join_key`
- `profile_target_distribution`

C.2 METHOD-SPECIFIC TOOL USAGE

The `rule_based` and `single_llm` baselines do not use interactive runtime tools. Instead, they consume structured benchmark context and profile outputs produced by the shared backend. The `generic_agent` baseline receives only broad workspace tools: `bash`, `python`, `scratchpad.read`, and `scratchpad.write`. The `proposed_agent` receives benchmark-specific tools: `lookup_actions`, `summarize_context_actions`, `inspect_dataset_tables`, `inspect_columns`, `profile_join_key`, `profile_target_distribution`, `preview_rows`, `scratchpad.read`, and `scratchpad.write`. By contrast, `claude_code` uses external runtime tools such as `Task`, `Bash`, `Read`, `Write`, and `TodoWrite` through the Claude Code CLI.

C.3 CURRENT LIMITATIONS

The current repo provides a lightweight shared registry, but not yet a full common tool-calling runtime. Generic tools such as shared `bash`, `python`, or web-search tools are not currently implemented in the shared registry.

D ADDITIONAL RESULT BREAKDOWNS

Table 4: Task-completion robustness. Group SD is the standard deviation across grouped task-testcase means after repeated runs are averaged within each unit. Within-group SD is the average standard deviation across five repeated runs for each task-testcase; deterministic or single-run rows report n/a.

Method	Groups	Raw runs	Group TC $\mu \pm \sigma$	Avg. within-group SD
Human	20	20	0.886 ± 0.107	n/a
Rule-based	80	80	0.428 ± 0.240	n/a
Single LLM	80	400	0.777 ± 0.176	0.045
Generic agent	80	400	0.766 ± 0.178	0.054
Claude Code	80	400	0.756 ± 0.194	0.060
Proposed agent	80	400	0.749 ± 0.205	0.055

Table 5: Task-level Task Completion means over TC1–TC4 after averaging repeated runs within each task-testcase unit. Gap is Proposed agent minus the best value among the four automated LLM-based methods in that row.

Task	Single LLM	Generic agent	Claude Code	Proposed agent	Prop. gap vs best
Allstate Claims Severity	0.796	0.801	0.703	0.871	0.000
AMEX Default Prediction	0.590	0.582	0.552	0.585	-0.006
ASHRAE Energy Prediction	0.923	0.956	0.923	0.787	-0.169
Predict Future Sales	0.839	0.864	0.853	0.792	-0.072
Elo Merchant Category Recommendation	0.787	0.730	0.771	0.782	-0.006
Home Credit Default Risk	0.768	0.685	0.690	0.758	-0.011
ICR Identify Age Related Conditions	0.918	0.928	0.897	0.778	-0.150
IEEE Fraud Detection	0.682	0.717	0.719	0.684	-0.034
LISH-MOA	0.739	0.690	0.736	0.691	-0.048
Mercedes Benz Greener Manufacturing	1.000	0.890	0.932	0.988	-0.012
Optiver Trading at the Close	0.524	0.491	0.450	0.588	0.000
Porto Seguro Safe Driver	0.768	0.799	0.872	0.724	-0.148
Recruit Restaurant Visitor Forecasting	0.887	0.837	0.807	0.771	-0.115
Restaurant Revenue Prediction	0.934	0.995	0.982	0.942	-0.053
Rossmann Store Sales	0.830	0.718	0.749	0.641	-0.189
Santander Customer Transaction	0.675	0.895	0.792	0.800	-0.095
Sberbank Russian Housing Market	0.616	0.583	0.557	0.580	-0.036
Spaceship Titanic	0.861	0.828	0.796	0.854	-0.007
Store Sales Time Series	0.829	0.755	0.743	0.828	-0.001
Zillow Prize 1	0.569	0.564	0.599	0.530	-0.069

Table 6: Zillow Prize 1 Task Completion by testcase scenario. LLM-based rows are means over five repeated runs per scenario; the rule-based row is deterministic.

Method	TC1	TC2	TC3	TC4	Overall
Rule-based	0.650	0.625	0.275	0.676	0.557
Single LLM	0.750	0.708	0.399	0.419	0.569
Generic agent	0.744	0.729	0.386	0.398	0.564
Claude Code	0.788	0.749	0.450	0.410	0.599
Proposed agent	0.720	0.746	0.321	0.334	0.530

Table 7: Add-side and remove-side metric breakdown.

Method	Add P	Add R	Add F1	Remove P	Remove R	Remove F1	Task Completion
Human	0.844	0.753	0.772	1.000	1.000	1.000	0.886
Rule-based	0.289	0.229	0.223	0.539	0.633	0.466	0.428
Single LLM	0.805	0.630	0.676	0.931	0.878	0.856	0.777
Generic agent	0.792	0.597	0.642	0.925	0.889	0.858	0.766
Claude Code	0.799	0.560	0.614	0.923	0.899	0.867	0.756
Proposed agent	0.816	0.615	0.676	0.930	0.822	0.820	0.749

Table 8: Efficiency and cost signals from recorded run metadata.

Method	Tokens/run	Time/run (s)	Tools/run	Cost/run	Total Cost	n runs
Human	not recorded	not recorded	not recorded	not recorded	not recorded	20
Rule-based	not recorded	0.0	0.0	\$0.000	\$0.000	80
Single LLM	2034.0	36.4	0.0	\$0.057	\$22.838	400
Generic agent	15855.2	81.1	5.3	\$0.143	\$57.379	400
Claude Code	5718.9	127.6	11.8	\$0.310	\$123.941	400
Proposed agent	20389.0	109.2	5.0	\$0.196	\$78.496	400

Table 9: Task completion grouped by problem type and table structure.

Dimension	Value	Method	n	Task Completion
Problem type	binary_classification	Human	7	0.855
Problem type	binary_classification	Rule-based	28	0.402
Problem type	binary_classification	Single LLM	28	0.752
Problem type	binary_classification	Generic agent	28	0.776
Problem type	binary_classification	Claude Code	28	0.760
Problem type	binary_classification	Proposed agent	28	0.740
Problem type	forecasting	Human	5	0.927
Problem type	forecasting	Rule-based	20	0.387
Problem type	forecasting	Single LLM	20	0.861
Problem type	forecasting	Generic agent	20	0.826
Problem type	forecasting	Claude Code	20	0.815
Problem type	forecasting	Proposed agent	20	0.764
Problem type	other	Human	1	0.921
Problem type	other	Rule-based	4	0.365
Problem type	other	Single LLM	4	0.739
Problem type	other	Generic agent	4	0.690
Problem type	other	Claude Code	4	0.736
Problem type	other	Proposed agent	4	0.691
Problem type	regression	Human	7	0.882
Problem type	regression	Rule-based	28	0.492
Problem type	regression	Single LLM	28	0.747
Problem type	regression	Generic agent	28	0.722
Problem type	regression	Claude Code	28	0.713
Problem type	regression	Proposed agent	28	0.754
Table structure	multi_table_lookup	Human	6	0.888
Table structure	multi_table_lookup	Rule-based	24	0.436
Table structure	multi_table_lookup	Single LLM	24	0.781
Table structure	multi_table_lookup	Generic agent	24	0.751
Table structure	multi_table_lookup	Claude Code	24	0.745
Table structure	multi_table_lookup	Proposed agent	24	0.691
Table structure	multi_table_relational	Human	5	0.816
Table structure	multi_table_relational	Rule-based	20	0.384
Table structure	multi_table_relational	Single LLM	20	0.793
Table structure	multi_table_relational	Generic agent	20	0.767
Table structure	multi_table_relational	Claude Code	20	0.768
Table structure	multi_table_relational	Proposed agent	20	0.757
Table structure	single_table	Human	9	0.924
Table structure	single_table	Rule-based	36	0.447
Table structure	single_table	Single LLM	36	0.765
Table structure	single_table	Generic agent	36	0.775
Table structure	single_table	Claude Code	36	0.757
Table structure	single_table	Proposed agent	36	0.782

Table 10: Task completion grouped by recorded preprocessing-complexity bucket.

Difficulty Proxy	Bucket	Method	n	Task Completion
Prep complexity	high	Human	2	0.744
Prep complexity	high	Rule-based	8	0.364
Prep complexity	high	Single LLM	8	0.692
Prep complexity	high	Generic agent	8	0.634
Prep complexity	high	Claude Code	8	0.623
Prep complexity	high	Proposed agent	8	0.669
Prep complexity	low	Human	3	0.888
Prep complexity	low	Rule-based	12	0.424
Prep complexity	low	Single LLM	12	0.746
Prep complexity	low	Generic agent	12	0.831
Prep complexity	low	Claude Code	12	0.789
Prep complexity	low	Proposed agent	12	0.798
Prep complexity	medium	Human	15	0.904
Prep complexity	medium	Rule-based	60	0.437
Prep complexity	medium	Single LLM	60	0.794
Prep complexity	medium	Generic agent	60	0.770
Prep complexity	medium	Claude Code	60	0.767
Prep complexity	medium	Proposed agent	60	0.749

E RULE-BASED HEURISTIC DETAILS

E.1 EXECUTION MODEL

The `rule_based` baseline is a single-pass, action-local policy over the visible action bank. For actions not in `context_action_ids`, it evaluates add-side heuristics; for actions already in context, it evaluates remove-side heuristics.

E.2 SHARED INPUTS AND THRESHOLDS

The policy uses structured dataset insights including schema, missingness, numeric, boolean-like, join, and target profiles, together with the task target column and primary key. Key global thresholds include join coverage (0.8), numeric missingness (0.2), missing-indicator minimum rate (0.05), high-null drop threshold (0.95), low/medium cardinality thresholds (20/100), outlier IQR factor (3.0), and identifier-ratio threshold (0.95).

E.3 COVERED ADD-SIDE HEURISTICS

The current generic add-side policy covers:

- JOIN_LOOKUP
- PARSE_DATETIME
- IMPUTE_MISSING (median / supported constant cases)
- CREATE_MISSING_INDICATOR
- DROP_HIGH_NULL_COLUMNS
- DATE_PART_FEATURE, CYCLICAL_ENCODE, TIME_SINCE_REFERENCE
- ENCODE_CATEGORICAL

- CLIP_OUTLIERS
- DROP_COLUMNS

E.4 COVERED REMOVE-SIDE HEURISTICS

The current generic remove-side policy covers:

- JOIN_LOOKUP
- PARSE_DATETIME
- **date-feature families**
- IMPUTE_MISSING
- CREATE_MISSING_INDICATOR
- DROP_HIGH_NULL_COLUMNS
- ENCODE_CATEGORICAL
- CLIP_OUTLIERS
- DROP_COLUMNS
- FILTER_ROWS

E.5 KNOWN CAVEATS

The policy is deterministic and interpretable, but it is mostly action-local. It does not reason deeply about action interactions, sequencing, or compactness of the final repair set, and some action families still have no generic heuristic coverage.

CHANGELOG FROM DRAFT

Since the draft, we completed evaluation across all four testcase scenarios for all 20 benchmark tasks and all automated methods, including Claude Code. We added mean $\pm$ standard-deviation robustness reporting over repeated runs, clarified that Claude Code is an external reference rather than a controlled ablation, and expanded the discussion of negative results: the proposed agent is competitive on add-side recovery but does not outperform the strongest one-shot LLM baseline overall because rollback consistency remains weak. We also clarified implementation settings, added task-level and Zillow-specific results in the appendix, and expanded prompt, tool-surface, and rule-based heuristic documentation.