TABCLEAN: EVIDENCE-GUIDED PROGRAM SYNTHESIS FOR SCALABLE TABULAR DATA CLEANING

Anonymous authors
Paper under double-blind review

ABSTRACT

Reliable analytics and machine-learning pipelines depend on clean tabular data, yet production tables contain heterogeneous errors that require different repair mechanisms. Constraint-based systems require explicit rules, while learning-based systems require labels or training. LLMs can help infer repairs, but repeatedly reasoning over cells or chunks incurs inference costs as tables grow and new batches arrive. Synthesizing executable cleaning logic offers a way to amortize this reasoning, provided that the evidence supports both the transformation and the conditions under which it should apply. We present TABCLEAN, a model-training-free system for scalable tabular data cleaning through evidence-guided program synthesis. TABCLEAN uses table evidence and execution feedback to synthesize guarded cleaning programs, amortizing synthesis-time LLM cost through deterministic execution over large tables and compatible batches. Table profiles and annotated examples guide the choice of repair mechanisms, target representations, and guard predicates. Cell-level feedback from executed candidates then guides program revision, and a deterministic controller retains the best development-set program. Across six benchmarks, our approach achieves a mean F1 of 0.842, improving on two LLM baselines by 0.456–0.764 with identical development examples and test rows. On a 200,000-row table, it achieves 0.983 F1 in 4.42 minutes, an approximately 1,200× speedup over a semi-supervised repair baseline. Its mean LLM API cost across the six benchmarks is 98% lower than that of an iterative LLM repair baseline.

1 INTRODUCTION

Tabular data are central to scientific studies, business operations, and public services, and are widely used to train and evaluate machine learning models (Shwartz-Ziv & Armon, 2022; Grinsztajn et al., 2022; Xing et al., 2026). Yet tables collected in practice frequently contain missing values, typographical errors, inconsistent formats, unit mismatches, and violations of relationships between attributes (Chu et al., 2016; Ilyas & Chu, 2019). Finding and correcting these errors requires manual effort to inspect suspicious values, specify repairs, and validate the results (Abedjan et al., 2016). This work is often iterative, as analysts alternate between cleaning and analysis to identify further problems and revise their repairs (Krishnan et al., 2016). If errors remain, they can distort analytical conclusions and bias model training even when the computational workflow completes successfully (Krishnan et al., 2016). These costs and downstream risks motivate cleaning methods that improve data quality with less manual intervention.

Existing cleaning methods differ in how they obtain repair knowledge and in the effort and computation required to apply it. Rule-based systems encode domain knowledge as constraints and repair rules, but specifying and maintaining these rules requires expertise (Fan et al., 2008; Chu et al., 2013; Yakout et al., 2013; Rekatsinas et al., 2017). Learning-based systems reduce manual rule authoring through supervision or transfer learning, but require labeled data, model training, or dataset-specific adaptation (Krishnan et al., 2016; Mahdavi et al., 2019; Heidari et al., 2019; Mahdavi & Abedjan, 2020). Large language models (LLMs) can infer repairs from table context and domain cues without task-specific training (Chen et al., 2026). When applied directly to cells, rows, or chunks, however, they keep inference in the cleaning path, incurring repeated reasoning costs as tables grow and new batches arrive (Narayan et al., 2022; Zhang et al., 2023; Naeem et al., 2023; Qi et al., 2024; Ni et al., 2024). Code and workflow generators reduce recurring inference by producing

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.

executable cleaning operations (Zhang et al., 2025; Li et al., 2025). Restricting these operations to single columns excludes cross-column dependency repairs, while mappings over observed values may miss new instances of an error pattern. Allowing broader transformations can instead introduce incorrect changes to valid values. The remaining challenge is to derive reusable repair logic from limited table evidence, with enough coverage to address recurring errors and clear conditions that restrict unsupported changes.

Our key insight is to infer recurring repair mechanisms together with evidence-guided conditions for applying them. Table profiles expose candidate patterns and dependencies, while annotated dirty–clean examples establish target representations and help distinguish errors from valid variation. Neither source alone resolves every ambiguity, so candidate transformations and their applicability guards must be tested through execution. Cell-level feedback identifies missed repairs, incorrect changes, and regressions, providing concrete evidence for revising a rule or narrowing its scope. This view turns cleaning into synthesis and refinement of guarded programs whose behavior can be checked on annotated examples.

We present TABCLEAN[1], a model-training-free system that implements this approach through multi-agent synthesis of Python repair programs. Specialized agents diagnose errors, plan ordered repairs, generate code, and review execution feedback using cleaning skills and role-specific context. A deterministic controller evaluates candidates on a small annotated development set and retains the best program. The resulting program combines value-pattern and cross-column dependency repairs and executes without further LLM inference when it runs successfully. This *synthesize once, execute repeatedly* design supports scalable cleaning through deterministic program execution and reuse on later batches when column semantics, target representations, and repair assumptions remain stable.

Across six standard benchmarks, TABCLEAN combines high repair quality with low LLM inference cost. It achieves a mean F1 of 0.842, compared with 0.078 for Cocoon and 0.386 for AutoDCWorkflow given identical development examples and test rows. On the 200,000-row Tax table, it achieves 0.983 F1 in 4.42 minutes. Its average LLM API cost across the six benchmarks is 98% lower than IterClean's. Ablations and an additional address benchmark also identify limitations, including variation in the benefits of refinement and difficulty repairing values that require evidence beyond the table and annotated sample.

Our contributions are threefold: (1) We formulate tabular data cleaning as evidence-guided synthesis of guarded programs combining value-pattern and cross-column repairs (Section 4.2). (2) We develop TABCLEAN, a multi-agent workflow that refines transformations and guards using cell-level execution feedback and selects the best development-set program for execution without further LLM inference (Section 4). (3) We evaluate repair quality and inference cost across six standard benchmarks, including comparisons with identical development examples and test rows, and analyze benefits and limitations through ablations and sensitivity studies (Section 5 and Appendix E).

2 RELATED WORK

Data cleaning. Classical systems rely on integrity constraints, statistical inference, or learned repair models (Chu et al., 2016; Ilyas & Chu, 2015; Abedjan et al., 2016). Learning-based systems use labels, weak supervision, transfer learning, or active feedback (Krishnan et al., 2016; Mahdavi et al., 2019; Heidari et al., 2019; Mahdavi & Abedjan, 2020; Mayfield et al., 2010). These approaches typically require dataset-specific constraints, features, labels, or retraining. TABCLEAN uses profiles and a small development set to synthesize executable repair logic, rather than requiring users to fully specify constraints or train a dataset-specific model.

LLM-based cleaning and executable logic. Recent work applies LLMs to wrangling, preprocessing, retrieval-assisted repair, standardization, workflow generation, and iterative cleaning (Narayan et al., 2022; Zhang et al., 2023; Naeem et al., 2023; Qi et al., 2024; Li et al., 2025; Ni et al., 2024; Yan et al., 2024; Mehra et al., 2024). Direct prompting keeps inference in the recurring repair path, while task-adaptive systems train models for particular cleaning tasks (Yan et al., 2024; Mehra et al., 2024). Code and workflow generation are the closest approaches to TABCLEAN. LLMClean infers ontological functional dependencies (OFDs) but delegates correction to Baran or HoloClean (Biester et al., 2024). Cocoon synthesizes regular expressions and SQL `CASE WHEN` transformations, but

[1]Code is available at `https://anonymous.4open.science/r/TabClean-830E/`.

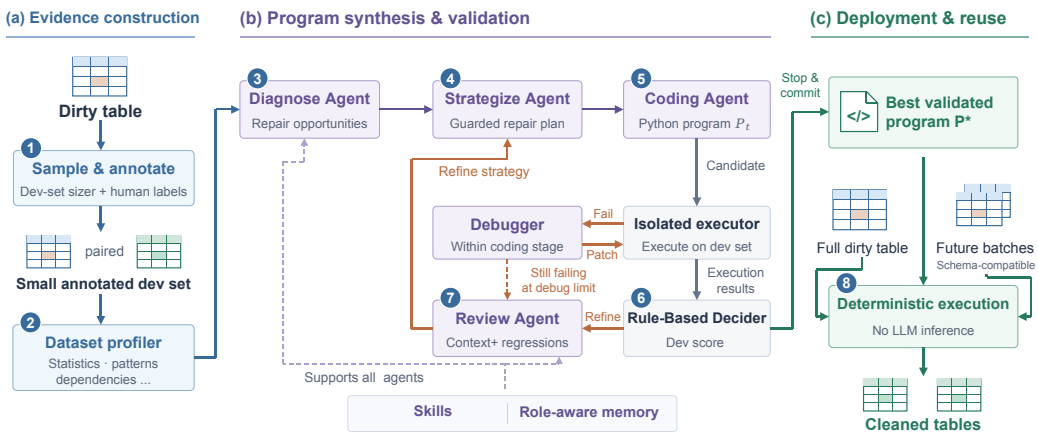


Figure 1: Evidence-guided cleaning program synthesis in TABCLEAN.

its observed-value mappings can leave unseen values with the same error pattern unrepaired (Zhang et al., 2025). AutoDCWorkflow's six fixed single-column operations exclude cross-column and functional-dependency (FD) repair and do not support guard refinement (Li et al., 2025). Beyond repair, Castle compiles intended changes into causally consistent SQL `UPDATE` statements across dependent columns (Su et al., 2025). TABCLEAN shares the use of executable artifacts and develops an evidence-guided process for synthesizing guarded Python repair logic.

3 PROBLEM SETTING

Let $\mathcal{D}$ be a dirty table with schema $\mathcal{S} = \langle A_1, \ldots, A_m \rangle$ and n rows. Let $\mathcal{C}$ be the corresponding clean table with the same schema and row alignment. A cell (i, j) is dirty when $\mathcal{D}_{ij} \neq \mathcal{C}_{ij}$. TABCLEAN receives the dirty table and a small annotated development set $\mathcal{A}$. The development set can be produced by human annotation, by a sampled historical dirty-clean pair, or by any workflow that supplies trusted clean values for a small subset of rows. TABCLEAN uses $\mathcal{A}$ only for synthesis and validation. Final quality is measured on held-out rows using cell-level correction precision, recall, and F1 (Section 5.1).

We formulate cleaning as program synthesis from examples (Devlin et al., 2017), seeking a program $\mathcal{P}$ that preserves schema and row order while repairing supported errors:

$$\max_{\mathcal{P}} \; F_1(\mathcal{P}(\mathcal{D}_{test}), \mathcal{C}_{test}). \tag{1}$$

The controller selects candidates by development-set F1, without using held-out labels. LLM calls occur during synthesis and refinement. Successful execution of the committed program requires no LLM inference. Later batches are compatible when they preserve the schema, column meanings, target conventions, and evidence assumptions used by the repair logic.

4 EVIDENCE-GUIDED PROGRAM SYNTHESIS

Figure 1 shows how TABCLEAN converts table evidence into a repair program. Profiles and annotated examples guide initial synthesis, and execution results provide evidence for revision. LLM agents analyze and propose mechanisms, guards, and code, while deterministic components execute candidates, compute development-set feedback, and select the best program.

4.1 EVIDENCE CONSTRUCTION

Development sampling determines how much labeled evidence is available for synthesis. The adaptive sizer visits rows in a fixed random order, checking minimum row and dirty-cell counts and error coverage in each observed error-bearing column. It also requires a window with no newly discovered error-bearing columns. Sampling stops when these conditions hold or a hard cap is reached.

Benchmark runs simulate annotation using the supplied clean references; stopping depends on errors observed in sampled rows. Columns with few labeled errors remain undercovered and provide limited support for diagnosis and validation.

The profiler separately summarizes the dirty input table without clean labels. It computes each column's cardinality, missingness, frequent values, and sampled values, and identifies candidate entity-group keys and columns that are approximately constant within a group. These summaries suggest repair mechanisms, while annotated examples establish target values and distinguish errors from valid variation. For example, a group majority suggests a dependency, but does not by itself establish that a minority value is erroneous. Execution feedback then tests whether the proposed transformation and guard respect the annotated examples.

The Diagnose Agent classifies repair opportunities into *deterministic evidence*, where the dirty pattern and clean target are mechanically inferable; *table-backed evidence*, such as group majorities or near-functional dependencies; and *weak or external evidence*, where the correct value cannot be inferred from the development set and profile alone. Synthesis instructions prioritize the first two levels and direct agents to skip weakly supported repairs. This discipline is implemented through prompts and skills, with resulting behavior checked on development examples. It does not formally guarantee that every generated guard or repair is correct. Appendix B details the evidence construction and error taxonomy.

4.2 Guarded Repair Programs

TabClean represents an inferred repair mechanism as a deterministic transformation paired with an evidence-guided applicability guard. A program orders these clauses:

$$\mathcal{P} = \langle c_1, \ldots, c_k \rangle, \qquad c_\ell = (A_\ell, s_\ell, \mathrm{g}_\ell, f_\ell, e_\ell, \pi_\ell). \tag{2}$$

Here A_ℓ is the target column, s_ℓ is a repair skill from library $\mathcal{K}$, g_ℓ is an applicability predicate, f_ℓ is a deterministic transformation, e_ℓ records supporting evidence, and π_ℓ specifies priority or application order. The Strategize Agent produces this plan, and the Coding Agent compiles it into Python. For a row x_i, a clause has the semantics

$$c_\ell(x_i)_{A_\ell} = \begin{cases} f_\ell(x_i, A_\ell), & \text{if } \mathrm{g}_\ell(x_i, A_\ell, \Phi, e_\ell) = 1, \\ x_{iA_\ell}, & \text{otherwise,} \end{cases} \tag{3}$$

where Φ is the dirty-table profile. All non-target columns are left unchanged by the clause. Guards may depend on cell values, other columns in the same row, profile statistics, and development-set evidence, but not on row identifiers, held-out labels, or manually enumerated test repairs. Clauses apply in strategy order and preserve schema, row order, and table shape. For example, different string representations of the same time can split its support within a group, distorting majority counts. First normalizing these representations to the format confirmed by annotated examples lets subsequent dependency repairs compute majority support over a common representation.

The natural-language skills specify required evidence, guard conditions, and candidate transformations (Table 1). The Strategize Agent instantiates them for the observed columns and target conventions, and the Coding Agent translates the plan into Python. A guarded repair usually combines

$$\mathcal{G} = \mathrm{g}_{\mathrm{dirty}} \wedge \mathrm{g}_{\mathrm{target}} \wedge \mathrm{g}_{\mathrm{evidence}} \wedge \mathrm{g}_{\mathrm{scope}}. \tag{4}$$

These guards check the dirty pattern, skip already-canonical values, require supporting evidence, and restrict the repair scope. Some evidence fixes constants or mappings during synthesis; table-dependent conditions such as group support are checked during execution. Anti-regression guards can be added after feedback. A narrow guard can limit over-correction, while an overly broad or incorrect guard can still make false repairs.

For example, a generated Tax program converts `7.0` to `7` in `rate`. Its guard requires digits before the decimal point and only zeros after it. Integer strings and decimals with a nonzero fractional part remain unchanged. Appendix F provides the original program excerpt. Generated programs read values as strings, protect cell updates with explicit guards, and are instructed to leave unsupported or ambiguous values unchanged. Value-level mappings require evidence for a canonicalization rule.

Cross-column repairs use the same evidence–guard–transformation structure. For MajorityFDRepair, group regularity suggests a candidate dependency, while labeled examples help distinguish errors from valid variation. The generated rule replaces a disagreeing value only when the

Table 1: Three representative cleaning skills. Each connects evidence to a guarded transformation. Table 8 gives additional examples.

Skill	Required evidence	Applicability guard	Transformation
FORMATNORMALIZE	Examples establish the target representation.	Value is parseable in a supported format and is not already canonical.	Parse and emit the canonical string.
MAJORITYFDREPAIR	Group regularity and labels support a dependency repair.	A majority exists, its support exceeds a threshold, and the value disagrees.	Replace with the supported group majority.
MISSINGFILL	A column or group provides a deterministic source.	Target is missing and the source value is available and unambiguous.	Fill from the supported source.

group majority has sufficient support (Table 1). Undercovered columns require caution: agents may propose deterministic format repairs but avoid broad dependency or semantic repairs without concrete evidence.

4.3 SYNTHESIS AND EXECUTION-GUIDED REFINEMENT

Within a finite-state workflow, the Diagnose Agent identifies error types, affected columns, dependencies, and candidate repairs. The Strategize Agent specifies skills, guards, target columns, and repair order; the Coding Agent translates this plan into a program. Syntax or runtime failures invoke a bounded debugging loop. The debugger receives the script and execution error. Subsequent cell-level evaluation checks the patched program. Semantic failures instead pass to the Review Agent so that the next strategy can change clauses or guards. Appendix A gives the full FSM and Algorithm 1.

The initial strategy prioritizes deterministic repairs, whose targets are unambiguous and whose guards can be checked with few examples. Later strategies expand coverage when feedback identifies missed repairs supported by the available evidence. False positives instead prompt narrower guards or removal of the offending clause. This progression limits simultaneous changes so that each revision can be checked before adding further repairs.

For execution-based candidate selection (Chen et al., 2022), executable outputs are compared with annotated development rows. The evaluator returns

$$F_t = (TP_t, FP_t, FN_t, \Delta_t, M_t), \qquad \mathcal{P}^* = \arg\max_{\mathcal{P}_t} F1_t, \tag{5}$$

where TP_t, FP_t, and FN_t count cell-level true positives, false positives, and false negatives; Δ_t summarizes regressions relative to the best program; and M_t stores per-column failure messages. The controller retains the best development-set program because iterative refinement need not improve quality monotonically (Madaan et al., 2023). A later candidate can improve recall on one column while introducing false positives or regressions on another. The stopping rule fires when current development F1 reaches the target threshold, the iteration budget is exhausted, or no improvement has been observed for a patience window.

The Review Agent produces a structured memo that connects execution feedback to program edits, an instance of feedback-guided refinement (Gou et al., 2024; Shinn et al., 2023). For false negatives, it distinguishes missed mechanisms from implementation bugs and cases with insufficient evidence. For false positives, it identifies over-broad transformations and missing guards. For regressions, it points to clauses that should be restored from $\mathcal{P}^*$. The memo marks logic as KEEP, ADD, MODIFY, REMOVE, or ACCEPT. When a best program already exists, the Coding Agent is instructed to copy KEEP logic unchanged and revise MODIFY logic according to FP/FN feedback. This focuses edits on observed failures while directing the agent to preserve validated repairs.

Role-aware memory retains the current profile, diagnosis, strategy, generated program, feedback, evaluation history, and best validated program (Qi et al., 2024; Zhang et al., 2025; Song et al., 2025; Yang et al., 2024; Chen et al., 2024; Shinn et al., 2023; Shi et al., 2023; Kuratov et al., 2024). Each agent receives context selected for its role, consistent with selective information sharing in MetaGPT (Hong et al., 2024). This selection limits exposure to stale code and verbose logs while

preserving evidence needed for the next decision. The Strategize Agent receives the diagnosis, recent feedback, evaluation summary, and best prior strategy. The Coding Agent receives the current strategy, relevant skills, profile, best code, and recent implementation feedback. The Review Agent receives the executed code, cell-level FP/FN feedback, diff summary, and iteration dashboard. Appendix G.1 provides agent inputs, output formats, prompt excerpts, and Decider rules.

Once committed, $\mathcal{P}^*$ applies its guarded repair logic without further LLM inference on inputs for which it runs successfully. Later-batch reuse assumes its guards and repair targets remain appropriate.

5 Experimental Evaluation

We evaluate cleaning effectiveness on standard benchmarks, with shared development examples, and on newly released data, followed by runtime, LLM cost, workflow ablations, and sensitivity analyses.

5.1 Experimental Setup

Datasets and supervision. We use Hospital (Chu et al., 2013; Centers for Medicare & Medicaid Services, 2012), Flight (Li et al., 2015), Beers (Hould, n.d.), Tax (Fan et al., 2008; Arocena et al., 2015), Rayyan (Ouzzani et al., 2016), and Movies (Das et al., n.d.). These benchmarks contain typos, format inconsistencies, dependency violations, and missing values. Table 2 reports table sizes, erroneous cells, and adaptive development budgets. We compare HoloClean (Rekatsinas et al., 2017), Baran (Mahdavi & Abedjan, 2020), Cocoon (Zhang et al., 2025), IterClean (Ni et al., 2024), and AutoDCWorkflow (Li et al., 2025). HoloClean receives oracle error locations and manually specified constraints; Baran uses labels on 20 tuples; IterClean uses five labeled seed tuples; Cocoon and AutoDCWorkflow require no labels in the standard comparison. TabClean uses development labels to guide synthesis and evaluate programs, with reported annotation time of 1–2 hours per dataset. Table 10 details human input.

Table 2: Standard benchmark statistics. Development rows are excluded from TabClean's test set.

Dataset	Rows	Columns	Erroneous cells	Dev rows
Hospital	1,000	19	509	150
Flight	2,376	7	9,504	110
Beers	2,410	10	4,765	240
Tax	200,000	15	121,219	150
Rayyan	1,000	11	960	150
Movies	7,390	17	7,675	369

Metrics and implementation. For original, final, and ground-truth values d, o, and g, TP counts $d \neq g$ and $o = g$; FP counts $o \neq d$ and $o \neq g$; and FN counts $d \neq g$ and $o \neq g$. A wrong repair counts as both FP and FN. We compute $P = TP/(TP + FP)$, $R = TP/(TP + FN)$, and $F1 = 2TP/(2TP + FP + FN)$, assigning zero for a zero denominator. Corrected values must match ground-truth strings exactly. TabClean is a Python prototype; its generated programs preserve string values, schema, row order, and table shape. TabClean and the LLM baselines use GPT-5-mini, with one model shared across all agents. Runtime includes profiling, synthesis, LLM calls, execution, debugging, and evaluation. Appendix C collects evaluation details.

5.2 Cleaning Effectiveness

Standard benchmarks. TabClean has the highest reported F1 on five datasets (Table 3). Precision and recall reveal different sources of these gaps (Table 11). On Tax, TabClean corrects more errors than AutoDCWorkflow at similarly high precision. On Movies, the two systems achieve similar recall, but their precision is 0.92 and 0.10, respectively. AutoDCWorkflow's incorrect date conversions and category merges lower its precision. Cocoon also introduces false positives through capitalization and missing-value conversions on Hospital. These cases illustrate why a normalization must match the target representation and preserve valid values.

Table 3: Standard-benchmark F1. Scores are rounded to two decimals, so small nonzero values can appear as 0.00. Precision, recall, and F1 are in Table 11.

System	Hospital	Flight	Beers	Tax	Rayyan	Movies
HoloClean	0.74	0.00	0.00	0.00	0.00	0.00
Baran	0.68	0.25	0.79	0.84	0.24	0.79
Cocoon	0.15	0.00	0.04	0.00	0.00	0.00
IterClean	**0.77**	0.00	0.64	0.02	0.00	0.07
AutoDCWorkflow	0.00	0.00	0.00	0.84	0.00	0.17
TABCLEAN	0.69	**0.72**	**0.89**	**0.98**	**0.89**	**0.88**

Table 4: Held-out F1 with the same development examples and test rows as TABCLEAN. TABCLEAN scores are repeated from Table 3. Average F1 is the mean across the six standard datasets.

Dataset	Cocoon	AutoDCWorkflow	TABCLEAN
Hospital	0.267	0.046	**0.692**
Flight	0.094	0.460	**0.723**
Beers	0.103	0.501	**0.887**
Tax	0.003	**0.986**	0.983
Rayyan	0.002	0.073	**0.889**
Movies	0.001	0.251	**0.879**
Average F1	0.078	0.386	**0.842**
AddressTable	0.264	0.261	**0.334**

Hospital also exposes a coverage limitation: TABCLEAN has precision of 0.88 but recall of 0.57, leaving many errors unrepaired. HoloClean's candidates, after its own value normalization, include the correct value for 98.8% of Hospital errors but at most 19.6% on the other datasets. This limits which errors it can repair even with oracle error locations. The standard comparison also uses different forms and amounts of supervision, so its F1 gaps cannot be attributed to workflow design alone.

Same development examples. To control the labeled evidence, we give Cocoon and AutoDCWorkflow TABCLEAN's development examples in their prompts and evaluate them on the same test rows (Table 4). Average F1 is 0.078, 0.386, and 0.842, respectively, for Cocoon, AutoDCWorkflow, and TABCLEAN. AutoDCWorkflow reaches 0.986 on Tax but scores poorly on Hospital and Rayyan. Supplying the same examples does not by itself give these baselines consistently high F1. IterClean is omitted because of its inference cost. Appendix G.2 lists instructions for both label settings.

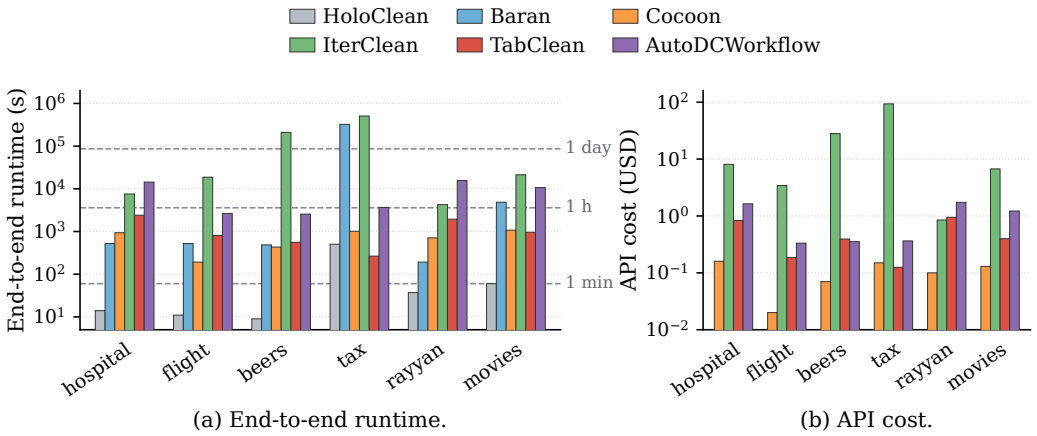
Newly released benchmark. AddressTable (Ahmadi et al., 2026) was released after the model version used by all LLM methods, `gpt-5-mini-2025-08-07`. We randomly sample 1,000 rows from the official `full-named` subset to limit IterClean's inference cost. TABCLEAN has the highest precision, with P/R/F1 of 0.815/0.210/0.334, whereas IterClean achieves 0.727/0.438/0.546. IterClean performs better using contextual LLM inference to repair address variations, but uses 5,915 LLM calls compared with 18 for TABCLEAN. Table 5 gives all systems' results.

5.3 RUNTIME EFFICIENCY AND LLM COST

Figure 2 reports runtime and API cost. On the 200,000-row Tax table, TABCLEAN takes 4.42 minutes compared with 89.8 hours for Baran. Its LLM calls generate and refine a program that subsequently cleans the table in Python without further LLM calls. These results illustrate the scalability benefit of synthesizing reusable programs: once synthesis is complete, cleaning additional rows relies on deterministic execution rather than repeated LLM inference. Tax takes less time than the smaller Hospital table, so row count alone does not explain runtime. TABCLEAN costs an average of $0.48 per dataset, with its lowest cost on Tax. Cocoon often costs less but has poor F1 scores. IterClean's average API cost is $48.8\times$ that of TABCLEAN. On Hospital, IterClean achieves higher F1 (0.77 vs. 0.69), but costs $9.7\times$ as much ($8.07 vs. $0.83). Appendix D provides token counts and

Table 5: AddressTable repair quality. HoloClean receives oracle error locations.

Method	Precision	Recall	F1
HoloClean	0.010	0.015	0.012
Baran	0.165	0.096	0.122
Cocoon	0.092	0.055	0.069
IterClean	0.727	**0.438**	**0.546**
AutoDCWorkflow	0.244	0.149	0.185
TABCLEAN	**0.815**	0.210	0.334



(a) End-to-end runtime. (b) API cost.

Figure 2: Efficiency across six datasets. (a) Wall-clock runtime in seconds, on a log scale; dashed lines mark 1 min, 1 h, and 1 day. (b) API cost in US dollars, on a log scale. Lower is better in both panels.

cost per correct repair. Program reuse requires compatible column meanings, target representations, and repair assumptions.

We also report the API cost per 100 correctly repaired cells, $\mathrm{Cost}_{100} = 100C_{\mathrm{run}}/TP$, where C_{run} is the run's API cost and TP is its number of correct repairs. Each method's cost and repair count come from the same run and test rows. On Tax, TABCLEAN costs $\$1.07 \times 10^{-4}$ per 100 correct repairs, compared with $\$4.17 \times 10^{-4}$ for AutoDCWorkflow and $\$6.68$ for IterClean. This difference combines lower spending with more correct repairs.

5.4 ABLATIONS AND SENSITIVITY

Table 6 compares Full with five variants. *Flat iterative agent* combines diagnosis, strategy, and coding in one agent, retaining development feedback and best-program selection (Appendix G.3). *w/o execution feedback* hides feedback from agents while retaining controller evaluation and best-program selection. *w/o skills* removes the cleaning-skills library. *w/o cross-iteration context* removes earlier history but retains the current program, immediate feedback, and best-program selection. *One-shot* generates code once without feedback, debugging, or refinement.

Full achieves the highest average F1, with gains that vary substantially by dataset. Flat's average is 3.7 points below Full's, showing that a single agent retains much of the overall quality. Without skills or retained context, Hospital F1 falls from 0.692 to 0.281 or 0.228, respectively. On Rayyan, removing execution feedback lowers F1 from 0.889 to 0.384, while One-shot reaches only 0.302.

The benefits are not uniform. On Hospital, removing agent feedback raises F1 from 0.692 to 0.824, even though the controller still evaluates and selects candidates. On Beers, Tax, and Movies, Flat and One-shot match or slightly exceed Full. Thus, additional refinement does not reliably improve every table. Best-program selection preserves the highest development score among evaluated candidates,

Table 6: Ablation F1, reported to three decimals. Columns abbreviate the variants defined in the text; Average F1 is across the six datasets.

Dataset	Full	Flat	w/o exec. feedback	w/o skills	w/o cross-iter. context	One-shot
Hospital	0.692	0.628	0.824	0.281	0.228	0.540
Flight	0.723	0.591	0.518	0.495	0.517	0.688
Beers	0.887	0.890	0.819	0.886	0.888	0.892
Tax	0.983	0.986	0.986	0.986	0.838	0.986
Rayyan	0.889	0.841	0.384	0.713	0.768	0.302
Movies	0.879	0.895	0.866	0.891	0.895	0.892
Average F1	0.842	0.805	0.733	0.709	0.689	0.717

but does not guarantee the best held-out score. The results support the complete workflow on average without implying that every component benefits every dataset.

Appendix E reports development-budget and model sensitivity. Each budget setting is run once and excludes its own labeled rows from testing. Doubling the adaptive budget raises average F1 by 3.4 points, but more labels do not consistently improve results.

6 Discussion and Conclusion

TabClean may miss repairs whose evidence is sparse, external to the table, or semantically ambiguous. The Hospital results illustrate this limitation: some identifier-like and address-like fields require domain knowledge or external evidence beyond a small annotated sample. A program synthesized only from table-local examples may be unable to distinguish a valid but rare value from an erroneous one. Reuse is also conditional: column meanings can drift, formatting conventions may change, and new error types may appear. Stronger reuse-time checks and lightweight program adaptation are needed before deploying cached cleaning programs in long-running pipelines. Future work includes retrieval-backed evidence, interactive validation and uncertainty estimates, and stronger drift checks.

We presented TabClean, a model-training-free system for evidence-guided program synthesis in tabular data cleaning. Table profiles and annotated examples guide repair mechanisms and applicability guards, while cell-level execution feedback guides their revision. Across six standard benchmarks, TabClean achieves higher reported F1 than the compared baselines on five datasets under method-specific supervision. These results support evidence-guided synthesis as a practical way to derive executable cleaning logic. Repeated deterministic execution supports scalable cleaning by amortizing synthesis-time LLM cost across large tables and compatible batches, provided that repair assumptions remain valid.

AI Use Statement

Generative AI assisted with coding, figures and tables, language editing, experimental design discussions, and literature search.

Reproducibility Statement

Our code is available in an anonymous repository: `https://anonymous.4open.science/r/TabClean-830E/`. Section 4 describes the repair representation and synthesis procedure. Appendix A gives the workflow, algorithm, and agent specifications; Appendix B documents evidence construction and cleaning skills. Section 5.1 and Appendix C describe the evaluation protocol, datasets, and supervision. Appendices D and E give complete results and sensitivity analyses, including the model-scoring limitation, and Appendix F provides a generated-program excerpt. Appendix G documents agent interfaces, prompt excerpts, Decider rules, and our AutoDCWorkflow adaptations.

REFERENCES

Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. Detecting data errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment*, 9(12):993–1004, 2016.

Fatemeh Ahmadi, Tobias Bernhard, Mohamed Abdelmaksoud, Luca Zecchini, Tilmann Rabl, and Ziawasch Abedjan. Clean me if you can: A large collection of real-world addresses for data cleaning benchmarking. *arXiv preprint arXiv:2606.31983*, 2026.

Patricia C Arocena, Boris Glavic, Giansalvatore Mecca, Renée J Miller, Paolo Papotti, and Donatello Santoro. Messing up with bart: error generation for evaluating data-cleaning algorithms. *Proceedings of the VLDB Endowment*, 9(2):36–47, 2015.

Fabian Biester, Mohamed Abdelaal, and Daniel Del Gaudio. Llmclean: Context-aware tabular data cleaning via llm-generated ofds. In *European Conference on Advances in Databases and Information Systems*, pp. 68–78. Springer, 2024.

Centers for Medicare & Medicaid Services. Hospital compare. Provider Data Catalog, 2012. URL `https://data.cms.gov/provider-data/topics/hospitals`. Accessed: 2026-09-09.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

Mengshi Chen, Yuxiang Sun, Tengchao Li, Jianwei Wang, Kai Wang, Xuemin Lin, Ying Zhang, and Wenjie Zhang. Empowering tabular data preparation with language models: Why and how? In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 228–246, 2026.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024.

Xu Chu, Ihab F Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 458–469. IEEE, 2013.

Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 international conference on management of data*, pp. 2201–2206, 2016.

Sanjib Das, AnHai Doan, Paul Suganthan G. C., Chaitanya Gokhale, Pradap Konda, Yash Govind, and Derek Paulsen. The magellan data repository. `https://sites.google.com/site/anhaidgroup/useful-stuff/the-magellan-data-repository`, n.d.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2):1–48, 2008.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, et al. Critic: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, volume 2024, pp. 57734–57811, 2024.

Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? *Advances in neural information processing systems*, 35: 507–520, 2022.

Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 international conference on management of data*, pp. 829–846, 2019.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

J. N. Hould. Craft beers dataset. Kaggle dataset, n.d. URL `https://www.kaggle.com/datasets/nickhould/craft-cans`. Accessed: 2026-09-22.

Ihab F Ilyas and Xu Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.

Ihab F Ilyas and Xu Chu. *Data cleaning*. Morgan & Claypool, 2019.

Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proc. VLDB Endow.*, 9(12):948–959, 2016.

Yuri Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. Babilong: Testing the limits of llms with long context reasoning-in-a-haystack. *Advances in Neural Information Processing Systems*, 37:106519–106554, 2024.

Lan Li, Liri Fang, Bertram Ludäscher, and Vetle I Torvik. Autodcworkflow: Llm-based data cleaning workflow auto-generation and benchmark. In *EMNLP (Findings)*, pp. 7766–7780, 2025.

Xian Li, Xin Luna Dong, Kenneth Lyons, Weiyi Meng, and Divesh Srivastava. Truth finding on the deep web: Is the problem solved? *arXiv preprint arXiv:1503.00303*, 2015.

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

Mohammad Mahdavi and Ziawasch Abedjan. Baran: Effective error correction via a unified context representation and transfer learning. *Proceedings of the VLDB Endowment*, 13(12):1948–1961, 2020.

Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. Raha: A configuration-free error detection system. In *Proceedings of the 2019 International Conference on Management of Data*, pp. 865–882, 2019.

Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. Eracer: a database approach for statistical inference and data cleaning. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 75–86, 2010.

Pavitra Mehra et al. Leveraging structured and unstructured data for tabular data cleaning. In *2024 IEEE International Conference on Big Data (BigData)*, pp. 5765–5768. IEEE, 2024.

Zan Ahmad Naeem, Mohammad Shahmeer Ahmad, Mohamed Eltabakh, Mourad Ouzzani, and Nan Tang. Retclean: Retrieval-based data cleaning using foundation models and data lakes. *arXiv preprint arXiv:2303.16909*, 2023.

Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911*, 2022.

Wei Ni, Kaihang Zhang, Xiaoye Miao, Xiangyu Zhao, Yangyang Wu, and Jianwei Yin. Iterclean: An iterative data cleaning framework with large language models. In *Proceedings of the ACM turing award celebration conference-China 2024*, pp. 100–105, 2024.

Mourad Ouzzani, Hossam Hammady, Zbys Fedorowicz, and Ahmed Elmagarmid. Rayyan—a web and mobile app for systematic reviews. *Systematic reviews*, 5(1):210, 2016.

Danrui Qi, Zhengjie Miao, and Jiannan Wang. Cleanagent: Automating data standardization with llm-based agents. *arXiv preprint arXiv:2403.08291*, 2024.

Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *arXiv preprint arXiv:1702.00820*, 2017.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International conference on machine learning*, pp. 31210–31227. PMLR, 2023.

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

Ravid Shwartz-Ziv and Amitai Armon. Tabular data: Deep learning is not all you need. *Information fusion*, 81:84–90, 2022.

Yuyang Song, Hanxu Yan, Jiale Lao, Yibo Wang, Yufei Li, Yuanchun Zhou, Jianguo Wang, and Mingjie Tang. Quite: A query rewrite system beyond rules with llm agents. *arXiv preprint arXiv:2506.07675*, 2025.

Yongye Su, Yucheng Zhang, Zeru Shi, Bruno Ribeiro, and Elisa Bertino. Castle: Causal cascade updates in relational databases with large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33501–33513, 2025.

Junjie Xing, Yeye He, Mengyu Zhou, Haoyu Dong, Shi Han, Lingjiao Chen, Dongmei Zhang, Surajit Chaudhuri, and HV Jagadish. Mmtu: A massive multi-task table understanding and reasoning benchmark. *Advances in Neural Information Processing Systems*, 38, 2026.

Mohamed Yakout, Laure Berti-Équille, and Ahmed K Elmagarmid. Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 553–564, 2013.

Mengyi Yan, Yaoshu Wang, Yue Wang, Xiaoye Miao, and Jianxin Li. Gidcl: A graph-enhanced interpretable data cleaning framework with large language models. *Proceedings of the ACM on Management of Data*, 2(6):1–29, 2024.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Haochen Zhang, Yuyang Dong, Chuan Xiao, and Masafumi Oyamada. Large language models as data preprocessors. *arXiv preprint arXiv:2308.16361*, 2023.

Shuo Zhang, Zezhou Huang, and Eugene Wu. Data cleaning using large language models. In *2025 IEEE 41st International Conference on Data Engineering Workshops (ICDEW)*, pp. 28–32. IEEE, 2025.

A WORKFLOW AND AGENT SPECIFICATIONS

A.1 SYNTHESIS AND REFINEMENT WORKFLOW

TABCLEAN organizes evidence-guided synthesis as a finite-state machine (FSM). Profiles and labeled examples inform mechanism selection, skills guide the guards and transformations, and execution feedback guides subsequent revisions. LLM agents produce the intermediate diagnosis, strategy, code, and review artifacts. Deterministic tools execute and score programs, retain the best candidate, and enforce stopping limits. The FSM specifies this control flow, while repair correctness is assessed empirically on development and held-out data.

Let Φ be the profile of the dirty input table, $\mathcal{K}$ be the cleaning-skills library, $\mathcal{B}$ be the role-aware memory buffer, and $\mathcal{X}$ be the isolated code execution environment. The FSM is defined as

$$\mathcal{M}_{\text{TABCLEAN}} = (\mathcal{S}, \mathcal{U}, \mathcal{O}, \delta, S_1, \mathsf{Acc}), \tag{6}$$

where $\mathcal{S} = \{S_1, \ldots, S_6\}$ is the set of workflow states, $\mathcal{U}$ is the set of agents and deterministic tools, $\mathcal{O}$ is the set of transition actions, $\delta : \mathcal{S} \times \mathcal{O} \to \mathcal{S}$ is the transition function, S_1 is the initial state, and $\mathsf{Acc} = \{S_6\}$ is the accepting state. The FSM operates in the environment

$$\mathcal{E}_{\text{TABCLEAN}} = (\mathcal{D}, \mathcal{A}, \Phi, \mathcal{K}, \mathcal{B}, \mathcal{X}), \tag{7}$$

which exposes table evidence, execution logs, and validation metrics to the agents. At state S_i, the responsible agent emits an action $O_k^{S_i}$, and the FSM transitions according to

$$\delta(S_i, O_k^{S_i}) \to S_j. \tag{8}$$

The default forward transition of each state is denoted by $O_0^{S_i}$. Non-default actions encode debugging, rejection, and feedback-driven replanning.

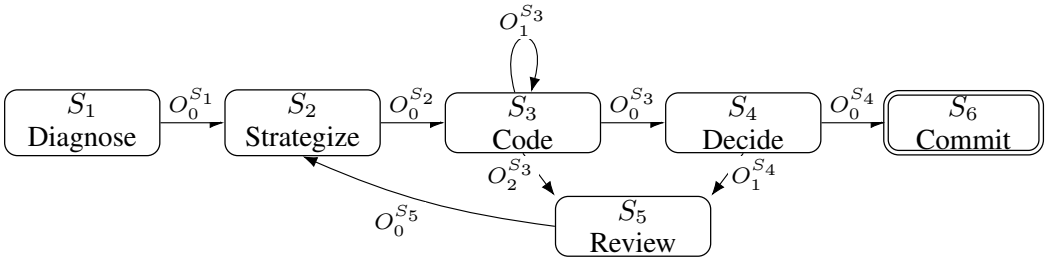


Figure 3: FSM workflow of TABCLEAN. Edge label $O_k^{S_i}$ denotes the k-th action of state S_i. $O_0^{S_i}$ is the default forward action. S_3 has a bounded debugging self-loop. The double ring marks S_6 as the accepting state.

Table 7: Key non-default transition conditions in the TABCLEAN FSM.

Action	Condition
$O_1^{S_3}$	The candidate program fails syntax or runtime checks and the debugging budget is not exhausted.
$O_2^{S_3}$	The candidate program still fails after the debugging budget is exhausted.
$O_1^{S_4}$	The stopping criteria have not been met, so cell-level feedback is passed to review.
$O_0^{S_4}$	The best validated program satisfies the target score, or the iteration/patience budget is exhausted.

Figure 3 shows the FSM workflow, and Table 7 summarizes the conditions for each non-default FSM transition. The important distinction is that syntax and runtime failures are handled locally in S_3, while semantic failures are routed through S_5 so that the next strategy can change the repair clauses or their guards. When the FSM reaches S_6, the controller commits the best validated program $\mathcal{P}^*$ and applies it to the raw table.

$$\widehat{\mathcal{C}} = \mathcal{P}^*(\mathcal{D}). \tag{9}$$

After synthesis, $\mathcal{P}^*$ executes the inferred repair logic without LLM inference on inputs for which it runs successfully. Applying it to later batches assumes that its guard and repair targets remain appropriate.

Algorithm 1 summarizes the operational loop. Executed candidates are evaluated against the annotated development rows, and the controller retains the program with the best development F1. This tracking matters because refinement is not monotonic. A later candidate can improve recall on one column while introducing false positives or regressions on another.

Algorithm 1: TABCLEAN Evidence-Guided Program Synthesis

Input: Dirty table $\mathcal{D}$, development set $\mathcal{A}$, skill library $\mathcal{K}$, iteration budget T, debugging budget B

Output: Guarded cleaning program $\mathcal{P}^*$ and cleaned table $\widehat{\mathcal{C}}$

$\Phi \leftarrow \text{Profile}(\mathcal{D})$;
$d \leftarrow \text{Diagnose}(\mathcal{A}, \Phi, \mathcal{K})$;
$r \leftarrow \text{Strategize}(d, \mathcal{K})$;
$\mathcal{P}^* \leftarrow \bot$; $s^* \leftarrow 0$;
for $t \leftarrow 1$ **to** T **do**
  $\mathcal{P}_t \leftarrow \text{Code}(r, \mathcal{P}^*)$;
  $ok \leftarrow \text{false}$;
  for $b \leftarrow 1$ **to** B **do**
    $(ok, log, out) \leftarrow \text{Execute}(\mathcal{P}_t, \mathcal{A})$;
    if ok **then**
      break;
    $\mathcal{P}_t \leftarrow \text{Debug}(\mathcal{P}_t, log)$;
  if $\neg ok$ **then**
    $q \leftarrow \text{ReviewFailure}(\mathcal{P}_t, log)$;
    $r \leftarrow \text{Strategize}(d, q, \mathcal{K})$;
    continue;
  $(s_t, F_t) \leftarrow \text{Evaluate}(out, \mathcal{A})$;
  if $s_t > s^*$ **then**
    $\mathcal{P}^* \leftarrow \mathcal{P}_t$;
    $s^* \leftarrow s_t$;
  if $\text{Stop}(s^*, t, F_t)$ **then**
    break;
  $q \leftarrow \text{Review}(\mathcal{P}_t, F_t, \mathcal{P}^*)$;
  $r \leftarrow \text{Strategize}(d, q, \mathcal{K})$;
$\widehat{\mathcal{C}} \leftarrow \mathcal{P}^*(\mathcal{D})$;
return $\mathcal{P}^*, \widehat{\mathcal{C}}$;

A.2 CONSTRAINED CODE SYNTHESIS

The Coding Agent translates the evidence-guided strategy into a complete Python program. Its prompt specifies the following program contract.

- The program must read the input table as strings and avoid unintended type coercion.
- The output must preserve the original schema, row order, and table shape.
- Each cell update must be protected by an explicit guard.
- Transformations must be expressed through column values, patterns, or structural relationships.
- The program must not memorize repairs by row identifier or use held-out labels. Value-level mappings require evidence for a canonicalization rule.
- Unsupported or ambiguous values must be left unchanged.

When a best program $\mathcal{P}^*$ already exists, the Coding Agent edits incrementally rather than generating from scratch. The review memo marks clauses as KEEP, ADD, MODIFY, REMOVE, or ACCEPT.

Logic marked KEEP should be copied without modification because it has already passed validation. Logic marked MODIFY should be narrowed or corrected according to FP/FN feedback. This incremental editing protocol reduces oscillation and makes each iteration attributable. If performance changes, the controller can relate the change to a small number of modified clauses.

Syntax and runtime failures are handled separately from semantic cleaning errors. The Debugger receives the current script and execution log, then returns a patched script. It may fix imports, parsing edge cases, missing columns, type errors, and file I/O issues. It is not allowed to invent a new cleaning strategy unless the failure shows that the current strategy cannot be implemented under the program contract. The debugger is instructed to preserve repair intent, and subsequent cell-level evaluation checks the patched program.

A.3 EXECUTION-GUIDED VALIDATION AND CONTROL

Execution supplies the evidence used to revise candidate programs. Failed executions are routed to the Debugger, and outputs with an incompatible shape are rejected by the evaluator. Schema and row-order preservation are also requirements of the generation contract. Executable outputs are compared against the labeled development rows to identify correct repairs, missed errors, and incorrect changes.

For iteration t, the evaluator returns

$$F_t = (TP_t, FP_t, FN_t, \Delta_t, M_t), \tag{10}$$

where TP_t, FP_t, and FN_t are cell-level repair counts of true positives, false positives, and false negatives. Δ_t summarizes regressions relative to the best program, and M_t stores per-column failure messages. Development-set precision, recall, and F1 are computed as $P_t = TP_t/(TP_t + FP_t)$, $Q_t = TP_t/(TP_t + FN_t)$, and $F_1 = 2TP/(2TP + FP + FN)$.

Among executable programs, the controller tracks the best development-set score.

$$\mathcal{P}^* = \arg\max_{\mathcal{P}_t} F1_t. \tag{11}$$

The stopping rule fires when the best score reaches the target threshold, the iteration budget is exhausted, or no improvement has been observed for a patience window. Otherwise, the candidate, feedback, and best-program summary are passed to the Review Agent.

The review memo is structured rather than conversational. For false negatives, it distinguishes missed mechanisms from implementation bugs and cases with insufficient evidence. For false positives, it identifies over-broad transformations and missing guards. For regressions, it points to the clause that should be restored from $\mathcal{P}^*$. The memo ends with a column-level action table using labels such as KEEP, MODIFY, REMOVE, or ACCEPT. This makes the next strategy update grounded in executed behavior rather than in the LLM's self-assessed confidence.

A.4 ROLE-SPECIALIZED AGENTS AND MEMORY

TABCLEAN uses multiple role-specialized agents because different stages require different context and have different failure modes. This design follows the broader pattern of tool-grounded LLM systems, where LLMs propose plans or programs while deterministic tools perform execution and validation (Qi et al., 2024; Zhang et al., 2025; Song et al., 2025; Yang et al., 2024; Chen et al., 2024; Shinn et al., 2023). In TABCLEAN, agents do not share an unfiltered transcript. Instead, the FSM maintains a role-aware memory buffer $\mathcal{B}$ containing the current profile, diagnosis, strategy, candidate program, best program, execution feedback, and review history.

Diagnose Agent. The Diagnose Agent receives the profile, development-set errors, error taxonomy, and skills library. It outputs a structured diagnosis containing column-level error types, representative examples, candidate dependencies, evidence levels, and repair opportunities. It also marks unsupported columns or mechanisms whose correct repairs would require external knowledge.

Strategize Agent. The Strategize Agent turns the diagnosis into an executable repair plan. The plan specifies target columns, selected skills, guards, application order, and skip decisions. In later iterations, the agent receives the review memo and best prior strategy, then performs an incremental

update rather than replanning from scratch. This preserves validated repairs and limits the number of simultaneous changes.

Coding Agent and Debugger. The Coding Agent translates the plan into Python under the program contract in Section A.2. The Debugger is invoked only after syntax or runtime failure. It patches the current script using execution logs while preserving the high-level strategy.

Rule-Based Decider. The Decider computes development-set scores, tracks the best successfully evaluated program, and applies the stopping criteria. Its decisions use measured output quality rather than an agent's confidence in its own proposal.

Review Agent. The Review Agent closes the semantic feedback loop. It receives the executed program, cell-level FP/FN summaries, regression information, and the current best program. Its output explains why the candidate failed or improved, then converts that explanation into actionable edits for the next strategy.

Agent Memory Passing the full interaction trace to every agent is noisy and expensive (Shi et al., 2023; Kuratov et al., 2024; Song et al., 2025). Outdated code, verbose logs, and stale reasoning distract agents as prompts grow across iterations. TABCLEAN instead retains only high-value context. The FSM state serves as a role-aware working memory that stores the current profile, diagnosis, strategy, generated program, execution feedback, evaluation history, and the best validated program so far. Each agent receives only the context needed for its role. The Diagnose Agent observes the profile, sampled errors, structural signals, and cleaning skills. The Strategize Agent receives the diagnosis, recent feedback, evaluation summary, and best prior strategy. The Coding Agent receives the current strategy, relevant skills, profile, best code, and recent implementation feedback. The Review Agent receives the executed code, cell-level FP/FN feedback, diff summary, and iteration dashboard. This role-scoped memory reduces prompt cost while letting later iterations preserve successful repairs and avoid repeating failed ones.

B REPAIR TAXONOMY AND SKILLS

B.1 ERROR TAXONOMY

TABCLEAN structures diagnosis with an explicit taxonomy. The current catalog includes typographical errors (T), missing values (MV), format inconsistencies (FI), violated attribute dependencies (VAD), out-of-distribution values (OD), semantic equivalence or canonicalization issues (SE), unit and scale inconsistencies (UT), truncation or extraction artifacts (TE), schema and type inconsistencies (SI), and composite or misaligned fields (CM). The taxonomy is not meant to be complete. Its role is to force each proposed repair to state the evidence it requires. For example, an MV repair needs a deterministic source for the missing value, while an SE repair needs evidence that several spellings denote the same entity.

B.2 EVIDENCE CONSTRUCTION

TABCLEAN constructs evidence to guide both what to repair and when to apply the repair. Column statistics and structural patterns expose candidate mechanisms. Annotated dirty-clean examples identify target conventions and distinguish observed errors from valid variation. Together, these inputs guide the synthesis of transformations and their guards. The loop adds execution feedback to test and revise these choices.

B.2.1 ADAPTIVE DEVELOPMENT-SET SIZER

The development set should expose recurring errors without becoming a second full-labeling task. TABCLEAN simulates sequential annotation in a fixed random row order using the benchmark's clean references. After each batch of ten rows, the sizer checks whether at least 100 rows and 100 dirty cells have been observed, each observed error-bearing column has at least ten dirty cells, and no new error-bearing column has appeared in the last 100 rows. Sampling stops when all conditions hold or a dataset-size-dependent cap is reached. These checks use errors encountered in sampled rows, not coverage of all errors in the table.

A column remains undercovered when the development set contains too few labeled dirty cells to support reliable diagnosis and validation for that column. Downstream agents treat such columns conservatively. They may still propose guards for deterministic format errors, but they avoid broad dependency or semantic repairs unless concrete evidence is available.

B.2.2 DATASET PROFILER

The profiler computes compact statistics over the dirty input table. It records each column's cardinality, missingness, frequent values, and sampled values. Structural analysis identifies candidate entity-group keys and columns that are approximately constant within a group. These summaries expose recurring patterns without requiring agents to inspect every row. They complement annotated examples, which establish the intended clean values.

The Diagnose Agent uses these summaries and examples to infer candidate repair mechanisms. For example, group-level regularity can suggest a dependency repair when labels confirm that minority values are erroneous. Frequent and sampled time strings can reveal multiple representations, while annotated targets identify the intended format. The profile supplies observations, and synthesis interprets them with the skills. Execution then tests whether the inferred transformation and guard behave as intended.

B.2.3 EVIDENCE LEVELS

The Diagnose Agent classifies repair opportunities by evidence level.

- **Deterministic evidence**. The dirty pattern and clean target are mechanically inferable, such as date, time, numeric, unit, or whitespace normalization with an unambiguous target format.
- **Table-backed evidence**. The repair follows from cross-row or cross-column regularities, such as majority values within an entity group or near-functional dependencies.
- **Weak or external evidence**. The correct value cannot be inferred from the development set and table profile alone, such as sparse address repairs or ambiguous entity names.

The synthesis instructions prioritize the first two levels and direct agents to skip weakly supported repairs. This evidence discipline is implemented through prompts and skills, with resulting behavior checked on development examples. It does not formally guarantee that every generated guard or repair is correct.

B.3 EVIDENCE AND SKILLS FOR GUARD SYNTHESIS

The cleaning-skills library $\mathcal{K}$ guides the translation from evidence to repair mechanisms. Its natural-language skills specify required evidence, guard conditions, and candidate transformations. The Strategize Agent instantiates these skills for the observed columns and target conventions, and the Coding Agent expresses the resulting plan in Python. The skills guide generation and review through prompts, while execution feedback tests the resulting behavior. Table 8 shows representative skills.

Table 8: Representative cleaning-skills used by TABCLEAN. Each skill instructs agents to connect a transformation to supporting evidence and applicability guards.

Skill and error type	Required evidence, guard, and transformation
FORMATNORMALIZE **Error type:** FI	**Required evidence:** Dirty and clean examples reveal a target representation, such as date, time, or numeric format. **Guard:** Value is parseable under supported formats and does not already match the target format. **Transformation:** Parse and emit canonical string.

Table 8: Representative cleaning-skills used by TABCLEAN (continued).

Skill and error type	Required evidence, guard, and transformation
UNITSCALEREPAIR **Error type:** UT	**Required evidence:** Numeric values show a consistent scale or unit mismatch with clean examples. **Guard:** Value falls in the suspicious range or has an explicit unit marker. **Transformation:** Convert to target unit or scale.
MAJORITYFDREPAIR **Error type:** VAD	**Required evidence:** A key or column group nearly determines the target column in the development profile. **Guard:** Group majority exists, support exceeds threshold, and current value disagrees. **Transformation:** Replace by supported group majority.
ALIASCANONICALIZE **Error type:** SE	**Required evidence:** Aliases are supported by repeated examples or profile-level canonicalization evidence. **Guard:** Value belongs to a validated alias set and target canonical form is unique. **Transformation:** Map alias to canonical value.
MISSINGFILL **Error type:** MV	**Required evidence:** A deterministic source exists in another column, group, or profile-backed dependency. **Guard:** Target value is missing and source value is available and unambiguous. **Transformation:** Fill missing value from source.
COMPOSITESPLITMERGE **Error type:** CM	**Required evidence:** Examples show repeated composite-field structure or extraction artifact. **Guard:** Value matches the composite pattern and target fields are empty or inconsistent. **Transformation:** Extract, split, merge, or realign fields.

A guarded repair usually combines several guard families.

$$\mathcal{G} = \mathrm{g}_{\mathrm{dirty}} \wedge \mathrm{g}_{\mathrm{target}} \wedge \mathrm{g}_{\mathrm{evidence}} \wedge \mathrm{g}_{\mathrm{scope}}. \tag{12}$$

The dirty-pattern guard $\mathrm{g}_{\mathrm{dirty}}$ checks whether the value matches a supported error pattern. The target-negative guard $\mathrm{g}_{\mathrm{target}}$ prevents already-clean cells from being modified. The evidence guard $\mathrm{g}_{\mathrm{evidence}}$ represents support conditions derived from labels or profile statistics. Some evidence fixes constants or mappings during synthesis, while table-dependent conditions such as group support are checked during execution. The scope guard $\mathrm{g}_{\mathrm{scope}}$ restricts the repair to the intended column, group, or value range. Table 9 summarizes these guard families.

Table 9: Guard families used to limit repair scope and reduce over-correction.

Guard family	Purpose
Dirty-pattern guard	Requires the value to match an observed or profile-supported dirty pattern.
Target-negative guard	Skips values that already satisfy the target canonical representation.
Evidence guard	Requires sufficient development-set or profile support before applying a repair.
Scope guard	Restricts the repair to the intended column, key group, type range, or parseable domain.
Anti-regression guard	Added after validation feedback to block transformations that produced false positives.

The first strategy is intentionally conservative. It prioritizes deterministic repairs because they can be validated with few examples and typically admit precise guards. Later strategies expand coverage only when validation feedback identifies persistent false negatives and the review memo explains them as missed supported mechanisms. Conversely, when a candidate introduces false positives, the next strategy narrows or removes the offending clause, often by adding anti-regression guards.

C EVALUATION DETAILS

Datasets. Table 2 summarizes six benchmarks spanning diverse domains, table sizes, and errors, including typos, format inconsistencies, dependency violations, and missing values. Our adaptive sizer determines development-set sizes. To assess repair performance on newly released data, Section 5.2 additionally evaluates AddressTable (Ahmadi et al., 2026), which was released after the model version used by all LLM methods (`gpt-5-mini-2025-08-07`).

Baselines. We compare with five systems: HoloClean combines constraints and statistical signals (Rekatsinas et al., 2017); Baran learns corrections from data and labeled examples (Mahdavi & Abedjan, 2020); Cocoon synthesizes and validates cleaning code (Zhang et al., 2025); IterClean iterates over detection, verification, and repair (Ni et al., 2024); and AutoDCWorkflow generates OpenRefine operations through iterative quality inspection (Li et al., 2025). HoloClean receives oracle error locations as it cannot detect errors, so its results measure repair given known errors.

Human input. Table 10 lists supervision requirements. TABCLEAN uses development labels to guide repair synthesis and evaluate candidate programs.

Table 10: Required human input per dataset.

System	Required Human Input	Avg. Time
HoloClean	Denial constraints / FDs (∼5–10 rules)	1–2 h
Baran	Cell labels on 20 tuples (active learning)	5–10 min
Cocoon	None	0
IterClean	Cell labels on 5 seed tuples	2–5 min
AutoDCWorkflow	None	0
TABCLEAN	Cell labels on adaptive dev set	1–2 h

Metrics. We measure cell-level correction precision, recall, and F1. For original, final, and ground-truth values d, o, and g, TP counts $d \neq g$ and $o = g$; FP counts $o \neq d$ and $o \neq g$; and FN counts $d \neq g$ and $o \neq g$. A wrong repair therefore counts as both FP and FN. We compute $P = TP/(TP + FP)$, $R = TP/(TP + FN)$, and $F1 = 2TP/(2TP + FP + FN)$, assigning zero for a zero denominator. We also report end-to-end wall-clock runtime, token usage, and API cost.

Implementation. TABCLEAN is a Python prototype whose generated programs preserve string values, schema, row order, and table shape. Runtime includes profiling, synthesis, LLM calls, execution, debugging, and evaluation.

Models. TABCLEAN and the LLM baselines use GPT-5-mini, with one model shared across all agents. Appendix E.2 examines model sensitivity.

D Complete Effectiveness and Efficiency Results

D.1 Standard Benchmarks

Table 11: Cleaning results (P = precision, R = recall, F = F1). Scores use the cell-level correction metric in Section 5.1. Scores are rounded to two decimals, so small nonzero values can appear as 0.00.

System	hospital			flight			beers		
	P	R	F	P	R	F	P	R	F
HoloClean	0.63	**0.91**	0.74	0.00	0.00	0.00	0.00	0.00	0.00
Baran	**0.92**	0.54	0.68	0.34	0.19	0.25	0.80	0.78	0.79
Cocoon	0.09	0.42	0.15	0.00	0.00	0.00	0.04	0.03	0.04
IterClean	0.67	**0.91**	**0.77**	0.00	0.00	0.00	0.69	0.60	0.64
AutoDCWorkflow	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TabClean	0.88	0.57	0.69	**0.74**	**0.71**	**0.72**	**0.89**	**0.89**	**0.89**

System	tax			rayyan			movies		
	P	R	F	P	R	F	P	R	F
HoloClean	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Baran	**1.00**	0.73	0.84	0.26	0.22	0.24	**0.95**	0.67	0.79
Cocoon	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
IterClean	0.03	0.01	0.02	0.00	0.00	0.00	0.18	0.05	0.07
AutoDCWorkflow	**1.00**	0.72	0.84	0.00	0.00	0.00	0.10	0.81	0.17
TabClean	**1.00**	**0.97**	**0.98**	**0.97**	**0.82**	**0.89**	0.92	**0.85**	**0.88**

Table 11 compares repair precision, recall, and F1 on six standard datasets. TabClean has the highest reported F1 on five datasets. It corrects more Tax errors than AutoDCWorkflow at similarly high precision. The two systems also achieve similar recall on Movies, but their precision is 0.92 and 0.10, respectively. On Hospital, TabClean has precision of 0.88 and recall of 0.57, leaving many errors unrepaired. Some of AutoDCWorkflow's date conversions and category merges produce incorrect Movies values, which lower its precision. HoloClean selects repairs from candidate values. After its own value normalization, these candidates include the correct value for 98.8% of Hospital errors but at most 19.6% on the other datasets. Cocoon's low precision reflects both changes to originally correct values and unsuccessful repairs. On Hospital, capitalization and missing-value conversions account for many false positives. On Beers, unit and ABV transformations often fail to recover ground-truth values, while missing-value conversions introduce additional false positives. These errors reduce precision under exact-string evaluation, whereas unresolved errors limit recall. IterClean sometimes makes incorrect changes to free-form fields.

D.2 Evaluation on a Newly Released Benchmark

Table 12: AddressTable repair quality.

Method	Precision	Recall	F1
HoloClean (oracle error locations)	0.010	0.015	0.012
Baran	0.165	0.096	0.122
Cocoon	0.092	0.055	0.069
IterClean	0.727	**0.438**	**0.546**
AutoDCWorkflow	0.244	0.149	0.185
TabClean	**0.815**	0.210	0.334

Table 12 compares repair quality on AddressTable (Ahmadi et al., 2026), which was released after the model version used by all LLM methods, `gpt-5-mini-2025-08-07`. We randomly sample 1,000 rows from the official `full-named` subset to limit IterClean's inference cost. TabClean

has the highest precision at 0.815, but its recall is 0.210. IterClean performs better as it uses contextual LLM inference to repair address variations, but it uses 5,915 LLM calls, compared with 18 for TABCLEAN.

D.3 RUNTIME EFFICIENCY AND LLM COST

D.3.1 END-TO-END RUNTIME

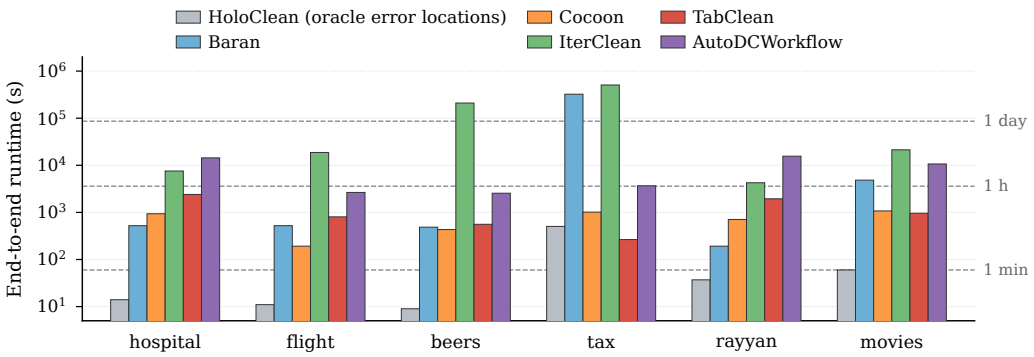


Figure 4: End-to-end wall-clock runtime (seconds, log scale, lower is better). Dashed lines mark 1 min, 1 h, and 1 day.

Figure 4 reports total cleaning time, including LLM calls and program execution. On the 200,000-row Tax table, TABCLEAN takes 4.42 minutes, compared with 89.8 hours for Baran. Its LLM calls generate and refine a program, which then cleans the table in Python without further LLM calls. Tax takes less time than the smaller Hospital table, so row count alone does not explain the runtime.

D.3.2 TOKEN USAGE AND API COST

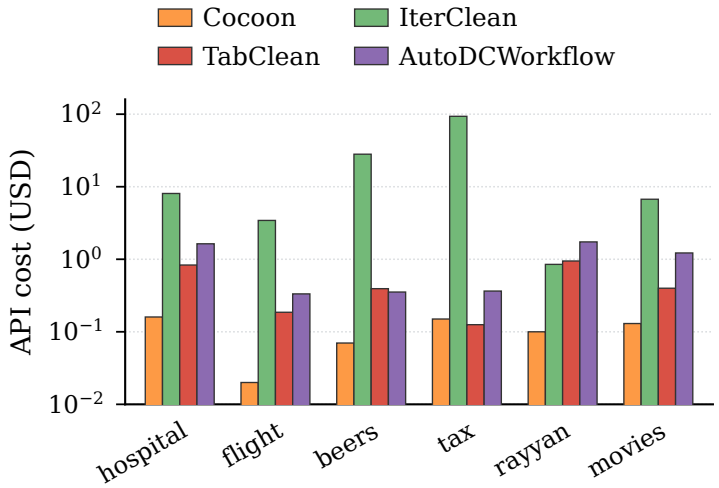


Figure 5: API cost per dataset for the LLM-based systems (US dollars, log scale).

Figure 5 reports API cost, and Table 13 reports input and output token counts. TABCLEAN costs an average of $0.48 per dataset, with its lowest cost on Tax. Cocoon often costs less but has poor F1 scores. IterClean's average API cost across the six standard datasets is $48.8\times$ that of TABCLEAN.

Table 13: LLM token usage (thousands of tokens).

	Cocoon		IterClean		AutoDCWorkflow		TABCLEAN	
Dataset	In	Out	In	Out	In	Out	In	Out
hospital	14	78	3,727	3,571	1,034	758	552	361
flight	5	14	1,303	1,553	114	156	396	56
beers	8	32	7,560	13,096	190	161	261	173
tax	20	72	87,251	36,643	165	168	185	47
rayyan	12	50	1,008	303	577	815	604	412
movies	21	65	13,807	1,633	428	575	784	127

Although IterClean achieves higher F1 on Hospital (0.77 vs. 0.69), its API cost on this dataset is $9.7\times$ that of TABCLEAN (8.07 vs. 0.83).

D.3.3 COST PER CORRECT REPAIR AND PROGRAM REUSE

We also report the API cost per 100 correctly repaired cells. This measure is $\mathrm{Cost}_{100} = 100C_{\mathrm{run}}/TP$, where C_{run} is the run's API cost and TP is its number of correct repairs. Each method's cost and repair count come from the same run and test rows. On Tax, TABCLEAN costs $\$1.07 \times 10^{-4}$ per 100 correct repairs, compared with $\$4.17 \times 10^{-4}$ for AutoDCWorkflow and $\$6.68$ for IterClean. This difference combines lower spending with more correct repairs.

E SENSITIVITY ANALYSES

E.1 DEVELOPMENT-SET BUDGET AND SAMPLING

Table 14: Development-set sensitivity.

Dataset	$0.5\times$ **adaptive**	**Adaptive**	$2\times$ **adaptive**	**Random 1%**
hospital	0.583	0.692	0.846	0.134
flight	0.702	0.723	0.704	0.682
beers	0.922	0.887	0.887	0.848
tax	0.986	0.983	0.987	0.996
rayyan	0.917	0.889	0.926	0.845
movies	0.898	0.879	0.905	0.884
Average F1	0.835	0.842	0.876	0.731

Table 14 compares the adaptive development-set size with half that size, double that size, and random 1% sampling. Each setting excludes its own labeled rows from testing.

Doubling the adaptive budget raises average F1 by 3.4 points. Halving the budget lowers Hospital F1 from 0.692 to 0.583 but improves F1 on Beers, Tax, Rayyan, and Movies. More labels therefore do not consistently produce better results in these runs. On Hospital, random 1% sampling yields an F1 of 0.134, compared with 0.692 under the adaptive budget. On Tax, it uses 13.3 times as many labeled rows for an F1 increase from 0.983 to 0.996. In these runs, the adaptive budget therefore offers a better balance between repair quality and annotation effort across the two table sizes.

E.2 MODEL SENSITIVITY

Figure 6 compares five models across six datasets, with one model shared by all agents in each configuration. GPT-5-mini uses the quality runs from Table 11 and has the lowest mean API cost at $0.48 per dataset. Among other alternatives, Sonnet 4.6 achieves the highest mean F1 at $1.39 per dataset. Opus 4.8 costs $11.63 per dataset but has lower F1. Higher cost therefore does not consistently improve quality. Model choice makes the largest difference on Hospital and Rayyan. Tax scores are identical, while Beers and Movies scores are similar. On Rayyan, Haiku 4.5 has

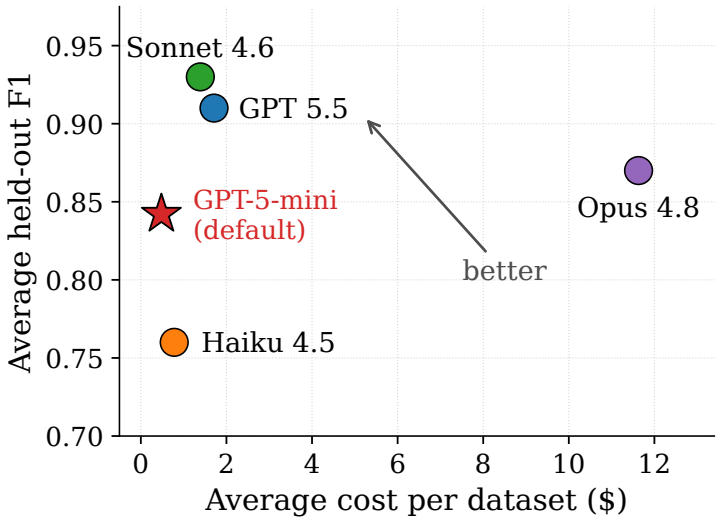


Figure 6: Model sensitivity across six datasets. Higher F1 and lower API cost are better.

precision of 0.87 but recall of only 0.09, leaving most errors unrepaired. Its reported precision alone therefore gives an incomplete picture of its cleaning performance.

F GENERATED-PROGRAM CASE STUDY

Figure 7 gives a simple example to show how labeled examples determine the required values in a generated Tax program. For `rate`, the program converts `7.0` to `7`. Its guard requires digits before the decimal point and only zeros after it. Integer strings and decimals with a nonzero fractional part remain unchanged.

```
rate_dirty_re = re.compile(r'^[0-9]+\.0+$')
# matches dirty whole-number floats
rate_clean_re = re.compile(r'^[0-9]+$')
# recognizes already-clean integers
zip_digits_re = re.compile(r'^[0-9]+$')
# limits zip repair to digit-only cells

for idx, val in df["rate"].items():
    s = val.strip()
    # dirty pattern:  "7.0", "10.00"
    # guard:  exact whole-number float, not clean int
    if s and rate_dirty_re.fullmatch(s) \
         and not rate_clean_re.fullmatch(s):
        df.at[idx, "rate"] = s.split(".", 1)[0]

for idx, val in df["zip"].items():
    s = val.strip()
    # dirty pattern:  leading-zero digit string
    # guard:  pure digits, starts with 0, nonempty result
    if s and zip_digits_re.fullmatch(s) \
         and s.startswith("0"):
        stripped = s.lstrip("0")
        if stripped:
            df.at[idx, "zip"] = stripped
```

Figure 7: Shortened excerpt of a real synthesized *tax* cleaning program. Red comments mark dirty patterns, blue comments mark guard predicates, and green lines mark the clean transformations.

For `zip`, the program converts `00627` to `627` by removing leading zeros. Its guard requires a digit-only string beginning with zero and a nonempty result after stripping. This rule matches the benchmark's required values and is not a general postal-code normalization rule. Once generated, these checks and replacements run without further LLM calls. The program needs to be checked again if later data require different formats or contain different error patterns.

G AGENT PROMPTS AND BASELINE ADAPTATIONS

We give the role interfaces and abridged prompt excerpts for TABCLEAN, the complete added or modified AutoDCWorkflow instructions in factored form, and the flat-agent prompt contract. The unabridged templates, conditional context blocks, and retry messages are provided in the accompanying source supplement, `supplement/prompts_full.md`. Excerpts combine selected passages without changing their wording, apart from line wrapping and typography; tables summarize interfaces rather than quote prompts. Named placeholders denote runtime inputs.

G.1 TABCLEAN AGENTS AND RULE-BASED CONTROL

Each role receives one assembled prompt string, with its role instruction at the beginning and no separately authored system message. Shared evidence comprises the skills library K, error catalog E, dirty-table profile P, structural analysis S, data peek V, and labeled development errors L. L contains dirty–correct pairs indexed within the development sample; the later exact-string runner serializes these as JSON. Table 15 specifies which fields each role receives. The templates match the archived Rayyan rerun; the earlier prototype omits the shared application policy described below.

Table 15: Agent inputs and required outputs. Optional iteration context is supplied only when available.

Role	Input fields	Output format
Diagnose	K, E, P, S, V, L.	Natural language: error distribution, per-column diagnosis, evidence quality, dependencies, repair feasibility.
Strategize	K, S, L, diagnosis; dashboard, latest review and feedback, best strategy.	Natural-language plan with ordered skills, target formats, guards, entity/group decisions, and skips. Revisions use KEEP UNCHANGED, MODIFY, ADD NEW, REMOVE.
Coding	K, P, S, strategy; best code, latest feedback, dashboard.	One complete Python code block reading `INPUT_PATH` and writing `OUTPUT_PATH`.
Review	K, L, current and previous strategies, executed code, feedback, cell differences, dashboard.	Natural language: FN causes, FP causes, regressions, prioritized fixes, revisions, keep/change decisions, and a per-column status table.
Debugger	Complete working script and execution error (up to 2,000 characters).	One complete fixed Python code block.

Diagnose: role and output excerpt.

```
You are a data quality diagnostician. Your job is to analyze a dirty
    dataset,
classify the error types present in each column, and identify which
    cleaning skills are relevant.
Output a structured natural-language diagnosis document. Do NOT output
    JSON or code.
```

The diagnosis classifies evidence as HIGH/MEDIUM/LOW and repair feasibility as DETERMINISTIC/EVIDENCE-BASED/UNCERTAIN, using the development examples and table evidence.

Strategize: role and iteration-specific excerpts.

```
You are a data cleaning strategist. Based on the diagnosis, available
    skills,
and all prior iteration feedback, design a cleaning strategy for this
    dataset.
```

The first call establishes deterministic repairs; subsequent calls partition changes and preserve validated behavior. The following excerpts are from the initial and revision tasks, respectively:

```
- Focus ONLY on DETERMINISTIC repairs -- transformations where the fix
    is a mechanical rule
  with zero ambiguity (format normalization, suffix/unit stripping,
    encoding repair, etc.).

- Add at most 1-2 new EVIDENCE-BASED repairs per iteration. Do not
    attempt too many new
  things at once -- each new technique should be validated before
    adding more.
For every transform, specify the guard condition that prevents false
    positives:
- What condition must be true BEFORE the cell is modified?
- Give a concrete example of a value that should NOT be changed and
    explain why the guard blocks it.
```

Coding: role and preservation excerpts.

```
You are a Python code generator for data cleaning. Translate the
    strategy below
into a complete, executable Python script.
- Read: df = pd.read_csv(INPUT_PATH, dtype=str, keep_default_na=False)
- Save: df.to_csv(OUTPUT_PATH, index=False)
- Keep DataFrame shape unchanged (same rows, same columns).
- Do NOT hardcode specific row indices.
- For columns/techniques marked KEEP UNCHANGED in the strategy, copy
    the corresponding
  logic from the best code VERBATIM. Do not rephrase, reorder, or
    "improve" working code.
Output ONLY a complete ```python code block.
```

Best-code preservation is inserted when a retained program has F1 greater than zero. The full prompt also requires guarded changes, dirty/clean example checks, and a final anti-regression check.

Review: role and status-line excerpts.

```
You are a data cleaning reviewer and debugging expert. Analyze the
    evaluation
results by comparing the STRATEGY (what was planned), the CODE (what
    was implemented), and
the RESULTS (what happened). Your analysis must be actionable for the
    next iteration.
[column_name] | STATUS: working/failing/regressed/stuck | ACTION:
    KEEP/MODIFY/ADD/REMOVE/ACCEPT | WHY: [one sentence]
```

FN analysis distinguishes a correct strategy with a code bug, a wrong strategy, and insufficient evidence. FP analysis specifies missing guards; regressions identify logic to restore. The full classification rules also use `improving`, `RESTORE`, and `RETHINK`; the output is passed as text. Best-iteration information reaches Review through feedback and the dashboard, not a separate best-code body.

Debugger: development template.

```
Fix this Python script. Error:
{error_feedback}
```python
{working_code}
```
IMPORTANT: INPUT_PATH and OUTPUT_PATH are already defined. Do NOT
    redefine them.
{_REPAIR_APPLICATION_POLICY}
Output the COMPLETE fixed script in a ```python block.
```

The full-table variant starts with "Fix this script for a LARGER dataset." Both execution stages allow three attempts and at most two Debugger calls, with 120/300-second development/full-table timeouts.

Shared application policy. Strategize, Coding, and Debugger additionally require applying repairs that pass their existing confidence thresholds and guards, without manual approval or proposal-only output. Uncertain proposals remain excluded. If switches are used, `APPLY_PROPOSALS` defaults to `1` and `DRY_RUN` to `0`, while explicit caller opt-outs are respected.

Decider rules. Decider uses no LLM. A strictly higher unrounded development F1 replaces the incumbent code and strategy and resets the stagnation count; a valid tie or regression increments it. Stop if current F1 rounded to four decimals reaches 0.95, the iteration count reaches 10, or stagnation reaches 3. Failed evaluations consume an iteration but do not update the incumbent or stagnation count. Otherwise, run Review $\rightarrow$ Strategize $\rightarrow$ Coding. Apply the best retained code, or the current candidate if no incumbent exists. The later exact-string evaluator counts dirty-to-wrong edits as both FP and FN; the earlier evaluator counted them only as FN. These are evaluator-version differences, not LLM decisions.

G.2 AutoDCWorkflow: Added and Modified Instructions

We retain the released AutoDCWorkflow stages (Li et al., 2025). The following reproduces *all* added or replaced instruction text, with shared clauses factored exactly. Substitute the setting-specific entries below literally; these documentary slots are not sent to the model. Other angle-bracketed fields denote runtime inputs. Each fresh conversation contains the system prefix plus the stage suffix, the fixed examples when applicable, and the inherited stage user prompt.

Full-table purpose and system prefix. The replacement purpose is:

```
Clean the entire table: inspect every original column for erroneous
    values, format inconsistencies, type errors, and missing values that
    can be resolved from <evidence>. Preserve every row, row order,
    original column, and valid value. <label-policy> Apply only
    corrections supported by <support>; otherwise retain the value.
```

The added system prefix is:

```
You are running the released AutoDCWorkflow on a <setting>.
    <availability> Execute only the current workflow subtask. Do not
    preempt later workflow stages or add code, operations, or stopping
    flags unless this subtask requests them. Preserve valid values and
    do not invent known correct values.
```

Slot	Zero-label	Same-label
`evidence`	the dirty table alone	the dirty table and supplied development examples
`label-policy`	There are no labeled examples or reference clean values.	Labeled examples come only from the fixed TabClean development subset.
`support`	the observed data and domain meaning	the observed data, supplied examples, and domain meaning
`setting`	full-table, zero-label cleaning task	full-table cleaning task with fixed development labels
`availability`	No ground-truth or development labels are available.	Only the supplied development labels are available; held-out labels are unavailable.

Stage-specific system suffixes (exact text).

Stage	Added instruction
Operation selection	Output only one fenced operation name from the six permitted choices. Do not generate its arguments or a stopping flag.
Quality explanation	Explain the specified quality judgement. Output prose only; do not generate a cleaning program or choose a new operation.
Objective summary	Output exactly two sentences: Summary and `**Data Cleaning Objective:**`. Do not output code, operation names, or stopping flags.
Quality vote	Judge whether this column is sufficiently clean under the stated purpose; output exactly one fenced True or False flag. Do not output cleaning code.
`mass_edit` arguments/repairs	Output exactly one JSON code block containing only the requested edits list. Each object has from (a list of strings) and to (a string). No additional code, operations, or stopping flags.
Other subtask	Follow the output format requested by the current subtask only.

For `regexr_transform` arguments and execution repairs, the complete suffix is below. It overrides the inherited user prompt's `Python Expression:` label.

```
Generate a single-cell Jython 2.7 expression, compatible with Python
    2.7, inside one fenced code block marked jython. The only input
    variable is value. Do not use Python 3 features such as f-strings,
    annotations, re.fullmatch, or keyword-only arguments. Use re.match
    with an explicit end anchor where a full match is needed. Use
    unicode(value) when conversion is needed; preserve Unicode and
    leading zeros where meaningful. Allowed imports: re, math, datetime,
    calendar, string, unicodedata, decimal. No filesystem, network,
    process, reflection, row/cell object access, or external data. End
    the program with an explicit top-level return statement returning
    the corrected value; return the original value when no supported
    transformation applies. Do not create provenance columns, flags,
    manual-review files or deferred repair proposals. Output only the
    code block, with no Boolean flag or separate operation name.
```

Same-label user message. Each frozen erroneous development cell contributes one JSON-encoded `"<column>" | "<dirty value>" | "<correct value>"` line. Unlisted cells are not declared correct; row identifiers and held-out labels are absent. Examples remain fixed throughout the run.

```
Additional labeled examples from a small development subset of this
    table:
column | dirty value | correct value
<example lines>
Use these examples only as evidence about possible cleaning patterns.
    Do not copy row positions or assume that every value occurrence has
    the same label.
```

Recovery user messages (complete templates). Execution repair uses a fresh conversation with the Jython suffix (and fixed examples in the same-label setting). Inputs are the target column, purpose, original/latest expressions, actual error, JSON diagnostics, and the first 20 distinct current-column values. The other two repairs append to the existing conversation. Limits are five additional execution/argument repairs and three stopping-format repairs.

```
The generated expression could not execute correctly in OpenRefine
    Jython 2.7. Correct only its execution compatibility while retaining
    the intended cleaning operation. Return one Jython 2.7 code block
    using only value.
Column: <column>
Purpose: <PURPOSE>
Original expression:
<original_expression>
Last expression:
<current_expression>
Actual parser/runtime error:
```

```
<error>
Error details:
<error_feedback(details)>
Observed dirty/intermediate examples:
<JSON list of observed values>
```

Argument-format repair:

```
Correct only the output syntax. Return exactly one JSON code block
    containing a list of objects with from (list of strings) and to
    (string). Parser error: <error>
```

Stopping-format repair:

```
Restate your stopping decision in the required format: exactly one
    fenced True or False flag. Do not output two alternatives.
```

Literal inherited-prompt fix and controller changes. The quality-explanation sample suffix changes from `{col_str}logger` to `{col_str}`. Separately, the controller visits every original column, protects Flight's synthetic index, preserves table structure, and allows ten iterations per column. Any `True` among three native quality votes stops that column; budget exhaustion and invalid votes are not positive judgements. Jython programs are validated, previewed, and applied or rolled back; compatibility repair receives execution errors, not external scores.

G.3 Flat Iterative Agent

The flat baseline combines diagnosis, strategy, and coding in one call. Its actual role instruction is:

```
You are one full-context data-cleaning agent. In this single response,
    diagnose
the remaining errors, choose repair mechanisms, and output a complete
    Python cleaning script.
This is refinement call {iteration} of at most {MAX_ITERATIONS}.
```

Each call includes the full skills library, taxonomy, profile, structural analysis, data peek, and JSON-encoded development errors. Optional blocks provide `best_code`, `latest_code` if different from the best, and `latest_feedback`. There is no separate system prompt, full conversation history, or Debugger call. The following excerpts specify the program contract:

```
- Read strings with pd.read_csv(INPUT_PATH, dtype=str,
    keep_default_na=False, encoding="utf-8-sig").
- Preserve table shape, all column names/order, and row order. Write
    OUTPUT_PATH without an index.
- Do not hard-code row indices, memorize development examples as a
    lookup table, access labels or other files, or make network/model
    calls.
- Use evidence-backed, guarded transformations. Leave ambiguous values
    unchanged.
- This is an automated cleaning run: APPLY every repair whose evidence
    thresholds and guards pass to the saved output by default. Do not
    leave eligible changes only in proposals or require manual approval.
- Keep those thresholds and guards; automatic application does not
    authorize uncertain repairs.
- Refine the prior code when development validation identifies a
    defect, preserving working logic.
- Validation uses exact strings: no evaluator-side trimming, case
    folding, or normalization. Every changed-but-wrong cell counts as
    FP, and dirty-to-wrong also remains FN.
Output only one complete ```python code block.
```

The remaining contract sets identical global/environment paths, permits only standard-library, pandas, and NumPy imports, and adopts the application defaults above. Feedback supplies development metrics and per-column errors, or execution failure for the next call. Selection follows Decider's improvement, rounding, failure, and stopping rules; held-out evaluation occurs only after generation.