BUDGETED DATA ERROR DETECTION FOR TABLE QUESTION ANSWERING

Anonymous authors
Paper under double-blind review

ABSTRACT

Table question answering (TQA) can fail because of errors in the underlying data, yet data errors remains a largely overlooked source of TQA failures. Error detection is typically performed *Offline*, targeting all data in the database. However, when detection relies on costly LLMs, limited budgets make it important to allocate detection effort effectively. Queries provide signals for focusing this effort on data that can affect the answer, motivating *Online*, query-aware detection. We formulate *budgeted error detection* for TQA and introduce an Online detection policy that leverages the query to interleave detection with execution. We develop a data error taxonomy and propose BED-TQA, a configurable benchmark for **b**udgeted **e**rror **d**etection derived from four TQA datasets. We evaluate detection policies across budgets, error distribution, query scope, database size and workload conditions. Given the same budget to scan all columns once, TQA with Online detection achieves up to 33.2% higher answer accuracy than Offline detection, while using 59.2% fewer LLM tokens . Online detection remains competitive across most settings, particularly under tight budgets. Offline becomes more cost-effective only when repairs can be reused across multiple queries. Overall, our results challenge the popular assumption that data errors should be detected and repaired Offline: Online detection can allocate budget more effectively toward relevant data.

1 INTRODUCTION

Table question answering (TQA) systems answer natural-language questions over tables, often by generating and executing SQL queries. Recent work has improved how these systems identify relevant tables and values, decompose complex questions, and generate queries (Pourreza & Rafiei, 2023; Wang et al., 2024; Talaei et al., 2024). Despite these advances, prior work largely assumes clean tables, leaving data errors underexplored (Gu et al., 2026). In real-world databases, values may be inconsistent as a result of corruption during migration or integration across heterogeneous sources. As a result, a query can execute without errors yet confidently return a wrong answer.

Data errors include missing values, formatting inconsistencies, and semantic variants such as synonyms and abbreviations (Ahmadi et al., 2025). Fig. 1 illustrates these errors in an employee table containing an abbreviation (`HR`) for `Human Resources`, and inconsistent salary, date and phone-number formats. For a query that sums monthly salaries for HR employees, both the department and salary errors require repair, while the date and phone-number formats do not affect the answer. The department error causes the equality predicate to exclude Ben's \$5K salary, returning \$10,000 instead of \$15,000. After repairing the department, their salary format must also be corrected before summation. This example illustrates that errors can differ in both their relevance to a query and the evidence needed to detect them, suggesting that detection effort should be targeted rather than applied uniformly.

Recent work shows that large language models (LLMs) can perform data cleaning across heterogeneous tasks (Narayan et al., 2022; Zhang et al., 2025), but their high inference cost makes broad error detection expensive. A conventional approach detects errors *Offline*, independently of any query (Mahdavi et al., 2019; Mahdavi & Abedjan, 2020; Rekatsinas et al., 2017), targeting all data in the database. Under a limited budget, however, this broad coverage can spend effort inspecting data irrelevant to the current query.

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.

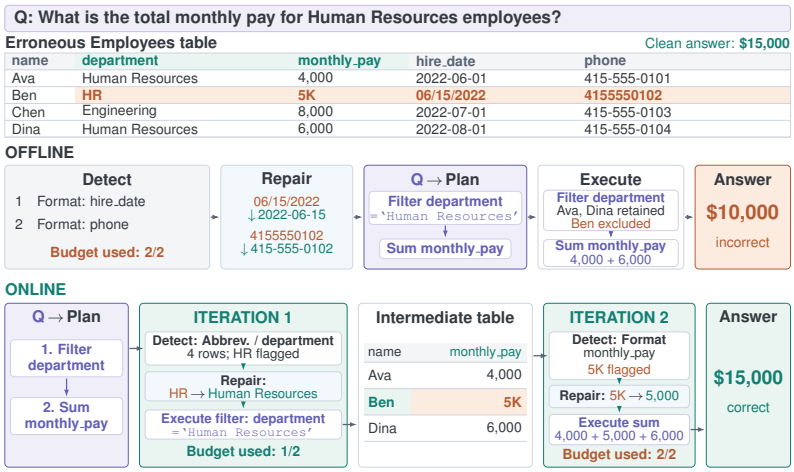

Figure 1: TQA with Online and Offline detection policies. Offline detects errors before execution, while Online interleaves detection with execution. With a two-column budget, Online focuses on the relevant abbreviation using query context, while Offline may inspect irrelevant columns.

The query can focus limited detection effort by providing two types of information: *scope*, which identifies relevant data, and *context*, which provides clues for how values should be interpreted. In the example above, the query identifies `department` as relevant and provides the literal `Human Resources` as evidence for interpreting `HR`. Building on this idea, we explore an *Online* error detection policy, which uses the query to guide detection rather than inspecting the entire database upfront. We further interleave detection with query execution, using intermediate results to progressively narrow the *scope* of detection to errors that can affect the answer. This enables detection effort to adapt as the query executes, rather than committing the full budget to a fixed scope upfront.

We formulate *budgeted error detection*, and compare how well *Offline* and *Online* policies allocate a limited detection budget. In our evaluation, the budget is the number of columns that can be scanned for errors. To study this problem systematically, we develop an error taxonomy for TQA and a configurable benchmark, BED-TQA, from four widely-used TQA datasets: MMQA (Wu et al., 2025), WikiSQL (Zhong et al., 2017), SQuALL (Shi et al., 2020) and BIRD (Li et al., 2023). BED-TQA allows us to inject controlled errors into the data and evaluate these strategies under varying detection budgets, error distributions, query scopes, database sizes and workload conditions. This enables us to characterize when Online detection is useful and when the broader coverage of Offline detection is more beneficial. Our contributions include:

- We formulate budgeted error detection for TQA and characterize the trade-off between Offline and Online detection.
- We develop an Online detection approach that use query scope and context, including interleaved detection and execution.
- We introduce an error taxonomy and a configurable benchmark BED-TQA based on four TQA datasets, and systematically evaluate detection policies across varied error and workload settings.

Our experiments show that, with a budget allowing one scan of all columns, TQA with Online detection achieves 33.2% higher exact answer accuracy than Offline detection, while using 59.2% fewer LLM tokens. Online detection remains competitive across most settings, while Offline detection becomes more cost-effective only when repairs are extensively reused across queries. We release our benchmark and code to support ongoing research.

2 Related Work

TQA spans SQL generation and multi-table, multi-hop reasoning (Yu et al., 2018; Li et al., 2023; Wu et al., 2025; Qiu et al., 2024). Approaches range from pre-trained table models (Herzig et al., 2020; Yin et al., 2020; Liu et al., 2021) and LLM-based SQL generation (Pourreza & Rafiei, 2023; Gao et al., 2024; Wang et al., 2023; Talaei et al., 2024) to systems that augment reasoning with tools

or intermediate tables (Cheng et al., 2023; Wang et al., 2024). These systems largely assume clean underlying data. Robustness work perturbs questions, schemas, or database representations, but primarily evaluates the query-generation pipeline (Chang et al., 2023). We instead fix the reference query and study errors in the underlying data. Our error taxonomy builds on a broader catalog of data errors (Bhadauria et al., 2026).

Data cleaning has a long history in probabilistic and constraint-based repair (Rekatsinas et al., 2017), automated detection and correction (Mahdavi et al., 2019; Mahdavi & Abedjan, 2020), and multi-table error detection (Ahmadi et al., 2025). Recent work also applies LLMs to data cleaning and wrangling (Zhang et al., 2025; Narayan et al., 2022). These approaches generally detect errors independently of a downstream query, corresponding to our *Offline* setting. For example, Auto-Test (Chen et al., 2025) learns semantic-domain constraints to identify erroneous cells without query context. Other work allocates cleaning effort based on downstream objectives such as model-training quality (Krishnan et al., 2016; Karlaš et al., 2021; Krishnan et al., 2017) or consistent query answering under integrity constraints (Arenas et al., 2002). These objectives differ from detecting errors that affect a specific query answer. We introduce an *Online*, query-aware setting and compare it with Offline detection under a limited budget.

Recent NL2SQL and TQA systems have used the question to guide data preparation. AutoPrep (Fan et al., 2025), for example, uses the question to plan table transformations, while Koukouvinis et al. (2026) study lexical and semantic mismatches between question expressions and database values. These approaches show the value of question context, but do not focus on detecting erroneous cells that affect a specific query answer. Other robustness work, such as Dr. Spider (Chang et al., 2023), perturbs questions and schema, while Liu et al. (2025) detects errors in generated SQL rather than underlying data. We instead study query-aware error detection and compare it with Offline detection across error types, query scope, database scale, detection budget, and workload reuse.

TQA benchmarks such as Spider, BIRD, and MMQA (Yu et al., 2018; Li et al., 2023; Wu et al., 2025) evaluate query generation and table reasoning. Cleaning datasets such as Hospital, Flights, Beers, and Rayyan pair dirty tables with reference values to evaluate cell-level detection and repair (Mahdavi et al., 2019; Mahdavi & Abedjan, 2020). DGov-X benchmarks by MaTElDa (Ahmadi et al., 2025) extend this to table collections with injected errors. These benchmarks evaluate cell-level correctness without linking errors to query answers. RADAR (Gu et al., 2026) is a related benchmark for end-to-end LLM-based reasoning over imperfect tables, but does not include ground-truth SQL queries. We instead add labeled corruptions and answer-supporting scopes to TQA instances that enable evaluation of detection and end-to-end answer quality.

3 Preliminaries

3.1 Problem Formulation

Let $\mathcal{D}^*$ denote a database such that executing a natural-language question Q over $\mathcal{D}^*$ produces the gold answer Y^*. Let $\mathcal{D}$ be a corrupted version of $\mathcal{D}^*$, where a set of cells c are erroneous if $\mathcal{D}[c] \neq \mathcal{D}^*[c]$. Given a predefined error taxonomy and set of detectors for each error type, a *detection action* applies a detector to a column. An action specifies the detector, target column, and query context provided to the detector. A *detection policy* π selects actions under a fixed budget B. To isolate error detection from query plan generation, we assume access to a fixed executable query plan q for each question Q, such that $Y^* = q(\mathcal{D}^*)$. Our goal is to identify a detection policy π that, within a budget of at most B detection actions, repairs $\mathcal{D}$ such that $q(\mathcal{D}) = Y^*$. When selecting a detection action, a policy takes the available execution state and remaining budget as input:

$$\pi(\mathcal{D}, Q, q, S_t, B_t) \rightarrow (f_{\theta_t}, a_t, x_t),$$

where S_t denotes the current execution state, B_t is the remaining detection budget, f_{θ_t} is the detector for error type θ_t, a_t is its target column, and x_t is the context supplied to the detector. Each selected action consumes one unit of budget, so the remaining budget is updated as $B_{t+1} = B_t - 1$.

We consider two detection settings. In *Offline detection*, actions are selected and applied to the input database before query execution. In *Online detection*, actions are selected during query execution and may operate on intermediate tables produced by earlier query-plan steps. Thus, the execution state available to the policy can evolve as detection and query execution proceed. We use *oracle*

Table 1: Error taxonomy for TQA.

Category	Error type	Description	Example
Syntactic	*Missing value*	A value is replaced by a null or placeholder	`N/A` instead of an amount
	Typographical error	A value contains a spelling or character error	`Ssarah` instead of `Sarah`
	Format inconsistency	A value uses an unexpected format	`03/25/2024` vs. `2024-03-25`
	Misfielded value	A value appears in the wrong column	200 in `remark` instead of `amount`
Semantic	*Synonym*	A value uses a different word with the same meaning	`finalized` vs. `completed`
	Abbreviation	A value uses a shortened form of the same entity	`US` vs. `United States`
Embedded value	*Embedded value*	A value is present but embedded in surrounding text	`200` vs. *total was $200*

repair, which restores every flagged cell to its known clean value. This isolates the effect of detection from repair quality, which is common practice in database literature (Rekatsinas et al. (2017)).

3.2 Error Taxonomy

We develop an error taxonomy for TQA inspired by the 35 error types cataloged by Bhadauria et al. (2026). We select a subset of seven error types that can be automatically detected without additional data sources, and organize them into three categories as shown in Tab. 1: syntactic, semantic and embedded value. Syntactic errors occur when values deviate from the expected type, format, or data pattern (e.g., value `05.29.2025` in a column with the expected format "`dd/mm/yyyy`"). Semantic errors use values that differ from the representation expected by the query (e.g., `HR` vs. `Human Resources`). Embedded-value errors require extracting values from surrounding text (e.g., extracting `5K` from a value "`monthly pay of 5K`"). Unlike prior work (Fan et al., 2025), our taxonomy covers a broader range of data errors for the task of TQA and can be easily extended to incorporate new error types.

4 BED-TQA: Benchmark Construction

Since existing TQA benchmarks lack cell-level data errors with ground truth for error types and clean values, we construct a synthetic benchmark, BED-TQA, by systematically injecting taxonomy-defined errors into existing benchmarks. This provides control over error type and distribution for evaluating error detection and its downstream impact on TQA.

Although errors may be injected anywhere in the database, we define the *answer-supporting scope* as the set of cells whose values can influence the query result. For example, in Figure 1, a query may use `department` to filter rows before counting them. Thus, `department` values can affect the answer even though they do not appear in the result. More generally, the scope includes data values that are referenced during filtering, joins, grouping, aggregation, or ordering. Building on query lineage and provenance (Cui et al., 2000; Buneman et al., 2001), we derive this scope by tracing these dependencies in the SQL query.

Datasets. We construct BED-TQA from subsets of MMQA (Wu et al., 2025), WikiSQL (Zhong et al., 2017), SQuALL (Shi et al., 2020), and BIRD (Li et al., 2023). We retain instances with executable reference queries and computable answer-supporting scopes. MMQA, WikiSQL, and SQuALL provide suitable cases for all seven error types in our taxonomy, while BIRD primarily contributes syntactic errors because its columns provide few opportunities to inject semantic errors.

Table 2: Overview of our mixed-error benchmark

Source	# Questions	# Databases	Error types	Answer change (%)
MMQA	141	141	7	56.9 [55.3–60.0]
WikiSQL	131	66	7	53.3 [52.7–54.0]
SQuALL	128	44	7	50.7 [49.3–51.3]
BIRD	124	7	4	67.1 [65.3–68.0]

Error Injection. For each TQA instance, we first profile the columns to identify column types and value patterns. We then select target cells from the answer-supporting scope and inject errors using transformations appropriate for each column and error type. We use deterministic or rule-based

transformations where possible, and LLM-based transformations for errors that require semantic interpretation We apply one error type per cell to keep error labels unambiguous and enable reliable per-type evaluation. We validate each generated corruption before including it in the benchmark. LLM-based verifiers check whether it matches the intended error type, and sampled semantic corruptions are manually reviewed for meaningful synonyms and abbreviations.

We corrupt approximately 10% of cells, distributed roughly uniformly across error types. As a controlled setting, this keeps most of the database clean while introducing enough meaningful errors. We treat this rate as experimental rather than representative of real-world prevalence, and vary corruption density and error composition in our sensitivity analysis. The benchmark framework supports configurable error types and distributions and can be adapted to other TQA datasets.

For ablations, we also construct single-error benchmarks using the same queries, with one error type injected at a time. The mixed-error benchmark evaluates budget allocation across co-occurring error types, while the single-error benchmarks isolate detection of individual types. Each benchmark record stores the original and corrupted values, error type, and information needed to restore the clean database. Tab. 2 summarizes the mixed-error benchmark. Across datasets, error injection changes answer for 49–68% of queries. We provide details of the injection transformations and other benchmark artifacts in App. A.

5 Budgeted Error Detection

We now describe the detectors and policies used to instantiate the framework in Sec 3.1. All policies use the same set of detectors. They differ in when detection is performed, how candidate actions are identified and prioritized under the detection budget, and what context is supplied to each detector.

Detector Bank. We implement one detector for each of the seven error types in our taxonomy. Each combines a deterministic analysis of the selected column with an LLM-based check, as summarized in Tab. 3. The deterministic stage identifies candidate values, while the LLM stage determines whether they exhibit the target error type. Across policies, we hold detector implementations, prompt templates, and decision thresholds fixed. The policy determines the target column and the context available to the detector, and therefore the evidence used by the LLM to determine whether a cell value exhibits the target error. Details on the detectors and thresholds are provided in App. I.

Table 3: Criteria for deterministic screening and LLM-based checks for each error type.

Detector	Deterministic screening	LLM-based check
Missing value	Look for missing-value markers or unusual placeholders	Use column semantics and value distributions to decide whether candidate values are actually missing
Typographical error	Look for rare values that are similar to common values	Decide whether the difference is a spelling or typing error
Format inconsistency	Look for values that do not follow the common format of the column	Infer the expected column format and decide whether the value is a genuine format error
Misfielded value	Look for values that seem to belong in another column based on overlapping column types	Use column semantics to check whether values were placed in the wrong columns
Synonym	Find suitable text values for comparison	Identify values with the same meaning, optionally using a question context
Abbreviation	Find suitable text values for comparison	Decide whether values are abbreviated forms of the same entity, optionally using a question context
Embedded value	Look for values that contain another value within extra text	Decide whether value contains another atomic value

Query Context Construction. To leverage query information, we refer to ground-truth SQL to isolate errors from query plan generation (Shahbazi et al., 2026). We parse the executable query into a logical plan and identify the columns involved in joins, filters, grouping, aggregation, ordering, and output, together with literals used in filter predicates. These elements define the *scope* of candidate detection actions, including the columns and cells that can affect the query result. In addition, they filter literals provide additional *context* for interpreting candidate values for selected detectors.

Action Selection. As depicted in Fig. 1, at each detection step, a policy constructs candidate detector-column actions for which the corresponding detector is applicable. It then prioritizes these candidates and executes selected actions until the detection budget is exhausted. Columns may be candidates for multiple detectors. So the budget gets allocated across both columns and error types. For Online policies, candidate actions are constructed separately at each query-plan step using the current intermediate table. Selected actions are executed before the corresponding query operator,

and the resulting table is used to construct candidates for the next step. Thus, Online action selection adapts to the current state of query execution.

5.1 Detection Policies

The policies differ in when detection is performed and how they use database and query information to select actions under the shared budget.

Offline, Query-agnostic Policy. The Offline policy uses only the database and schema. It ranks columns using data characteristics such as missing values, rare values, and unusual formats, and applies suitable detectors to the highest-ranked columns until the budget is exhausted. With sufficient budget, it can examine every applicable detector-column pair. Its main advantage is reuse: one Offline pass can support multiple queries over the same database. Its limitation is that it may spend detection budget on columns that are irrelevant to a particular query.

Online, Query-aware Policy. The Online policy uses the query plan to guide detection during query execution. At each query-plan step, it identifies candidate detector-column actions on the current intermediate table and prioritizes them according to their roles in the query, from highest to lowest: join keys filter columns, grouping and aggregation inputs, ordering columns, and projection columns. When a column has multiple roles, its highest applicable priority is used. The selected actions are executed before the corresponding query operator. Flagged cells are repaired using the oracle, and the operator is then applied to the repaired intermediate table. The resulting table becomes the input to the next query-plan step (as illustrated in Fig. 1). This process continues until query execution completes or the detection budget is exhausted.

For each selected action, the policy supplies the relevant query-plan fragment and filter literals as detector context. Thus, query information affects both *where* detection is performed and, for detectors that use semantic context, *how* candidate values are assessed.

6 Experiments

Our experiments evaluate detection policies under a limited budget through four research questions:

- RQ1: Under a limited budget, does Online policy improve end-to-end TQA accuracy over Offline policy?
- RQ2: How does performance vary with budget, query context, error distribution, and database size and workload conditions?
- RQ3: How does the cost of policies compare as repairs are reused across queries?
- RQ4: Under what conditions does each policy offer a favorable quality-cost trade-off?

6.1 Setup

Baselines. Our implementation of Offline policy follows the architecture of Cocoon (Zhang et al., 2025). We also evaluate a *Question-only* policy, based on AutoPrep (Fan et al., 2025), to separate the benefit of query-based relevance from interleaved execution. This policy uses an LLM to select target columns for detection using the question, schema and sampled column values. Since the original implementations cannot be directly applied to our error taxonomy and budgeted setting, we re-implement their core ideas within our framework.

Settings. All policies use the same set of detectors and `gpt-5-mini` as base model. To isolate the effect of policy choice, we use an oracle query plan and oracle repair. Unless stated otherwise, the default budget is $B = C$, where C is the number of columns in the database. Since multiple detectors may apply to each column, this budget does not imply exhaustive detection. We run each experiment on three benchmark seeds and report 95% bootstrap confidence intervals clustered by source database. We provide the list of other hyperparameters in App. J.

Metrics. We evaluate TQA using *answer accuracy* as the percentage of questions whose predicted answer exactly matches the ground truth. For detection quality, we report *recall* as the fraction of erroneous cells that are correctly identified, regardless of error type. We also report *typed F1*, which requires both the erroneous cell and its error type to be correctly identified. We estimate policy cost using the total LLM tokens consumed throughout the workflow.

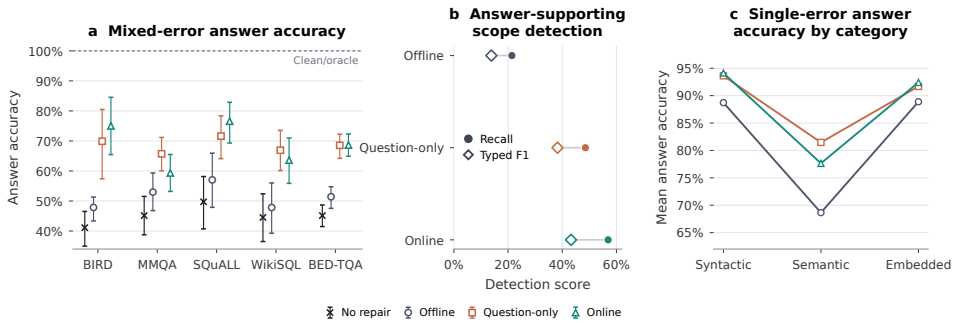

Figure 2: (a) Answer accuracy by dataset and benchmark average. (b) Detection quality on answer-supporting cells. Typed F1 requires both the error and its type to be correctly identified, while recall measures error detection regardless of type. (c) Mean answer accuracy on single-error benchmarks, grouped by error category. Error bars show 95% bootstrap confidence intervals.

6.2 RQ1: TQA ACCURACY AND ERROR DETECTION

Fig. 2(a) shows answer accuracy in the mixed-error setting. Error injection drops answer accuracy on BED-TQA to 0.451 (*No repair* in rightmost bars). Question-only and Online consistently outperform Offline across all datasets, achieving overall accuracies of 0.685 and 0.686, respectively, compared with 0.514 for Offline—an improvement of up to 33.2%. Online performs better on BIRD and SQuALL, while Question-only leads on MMQA and WikiSQL. The detection results on BED-TQA in Fig. 2(b) help explain this pattern. Within the answer-supporting scope, Online achieves the highest recall and typed F1, indicating that the query plan provides broader coverage of relevant columns. However, these additional candidates compete for the limited detection budget, resulting in lower or comparable answer accuracy to Question-only. In contrast, Question-only may miss columns implicit in the query, but can allocate its limited budget more effectively over a smaller candidate set.

We further examine policies over single-error types (Fig. 2(c)). Across the four syntactic error types, Online and Question-only policies achieve comparable accuracy and consistently outperform Offline policy. Semantic errors are more challenging for all policies, but both question-aware policies continue to outperform Offline. The gap between Online and Question-only again reflects the trade-off between detection coverage and budget allocation. We provide additional results on detection and answer accuracy, including breakdowns by the seven error types in App. C.

Takeaway. Given the same budget, policies that can leverage the query achieve higher answer accuracy than Offline. Although Online provides broader coverage of answer-supporting cells and higher detection quality, Question-only can sometimes achieve similar answer accuracy by concentrating on fewer relevant candidates.

6.3 RQ2: POLICY SENSITIVITY

We next examine how policy performance changes with key factors including budget, query context, error distribution, dataset size and query scope. Fig. 3 summarizes the results, with additional fine-grained analysis provided in App. D.

Effect of Detection Budget. As shown in Fig. 3(a), the relative advantage shifts from Question-only to Online as the detection budget increases. With a small budget, focusing on a narrow set of query-relevant columns leads to higher answer accuracy. As the budget grows to 32 columns, the broader scope and interleaved execution in Online policy becomes more beneficial, leading to 4.9% and 20.5% higher answer accuracy than Question-only and Offline, respectively.

Effect of Query Context. Fig. 3(b) isolates the effect of query context by fixing the candidate columns and detection budget. We progressively add the question, query plan and original filter literals, with literals revealed only in the final step. As shown, additional context has little effect on

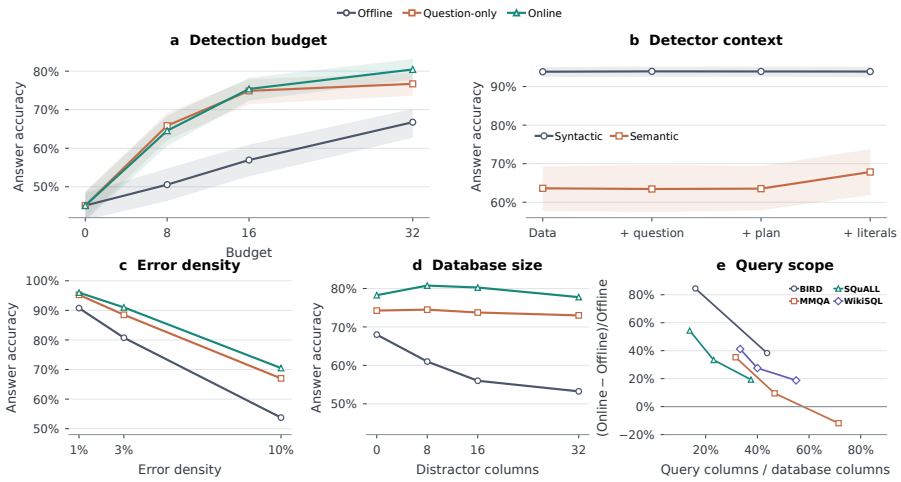

Figure 3: Sensitivity to key factors. Answer accuracy across (a) detection budgets, (b) query context, (c) error densities, and (d) database size and (e) query scope on BED-TQA. Shaded regions show 95% bootstrap confidence intervals.

syntactic error detection. In contrast, semantic error detection improves only when filter literals are provided, increasing accuracy by 6.8% and answer-supporting recall by 24%. The accuracy gain is larger for synonyms (8.1%) than abbreviations (3.8%).

Effect of Error Distribution and Database Size. As shown in Fig. 3(c), as target error density increases from 1% to 10%, Online's relative improvement in answer accuracy over Offline grows from 5.8% to 31.2%. We observe a similar trend when we expand the database with distractor columns (see Fig. 3(d)). We construct distractors by copying existing columns into separate single-column tables that are not accessed by the reference query. Adding distractors reduces answer accuracy by up to 32.4% for Offline, while Online and Question-only show more stable performance.

Effect of Query Scope. Fig. 3(e) shows that for queries that access a smaller fraction of the database, Online outperforms Offline policy by 35.3–84.5%. This gap narrows as queries cover a larger fraction of the database.

Takeaway. Policy performance is primarily shaped by budget and query scope. Question-only is more effective under tight budgets, while Online benefits from larger budgets. This advantage grows with higher error density and more distractor columns.

6.4 RQ3: Cost Efficiency

We compare the three policies in terms of token usage and examine how the cost of database-wide detection changes when repairs can be reused across queries. Fig. 4 summarizes the main results, with finer-grained results in App. E. Without reuse across queries, Online is the least expensive policy while achieving accuracy comparable to Question-only. Online consumes 3,324 tokens and costs $1.49 per 1,000 queries, compared to 5,773 tokens and $2.18 for Offline. This corresponds to a 59.2% cost reduction while retaining the 33.2% accuracy advantage. We next increase the number of questions sharing each source (A) to examine how reuse changes this trade-off. Offline incurs its cleaning cost once and spreads it across questions, reducing its per-question cost as $1/A$. In contrast, Question-only and Online incur their full detection cost for each question. Fig. 4 shows that Offline becomes increasingly cost-effective as reuse increases.

Takeaway. Online provides lower cost for low-reuse workloads, while Offline becomes more cost-effective as more questions share the same data source. However, greater workload reuse does not eliminate the answer-quality gap of Offline detection.

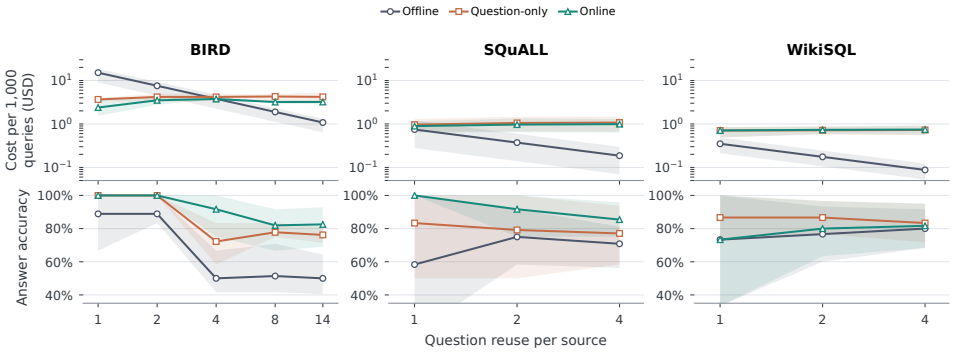

Figure 4: Cost and cumulative answer accuracy as questions reuse a corrupted source. Shaded regions show 95% bootstrap confidence intervals.

6.5 RQ4: Deployment decision framework

The cost-accuracy trade-off depends on budget, query scope, and workload conditions. We use these factors to guide policy selection. Given normalized budget B/C, expected questions per source A, and a target exact accuracy τ, a local audit estimates the accuracy and amortized cost of each policy and selects the least expensive policy that meets τ. If no policy meets the target, we do not make a recommendation. If no target is provided, we return the measured accuracy-cost Pareto set.

We next test whether this procedure generalizes to new databases. We freeze the policy select after auditing 180 instances and evaluate it, without refitting, on 720 instances from held-out WikiSQL and SQuALL physical sources corrupted using our BED-TQA framework. Combining $B/C \in \{0.5, 1, 2\}$, $A \in \{1, 4, 16\}$, and 21 values of τ gives 189 deployment decisions. The selector makes 141 recommendations and withholds 48 when no policy reaches the requested target. All recommendations are Pareto-efficient on the held-out data. The target is met in all WikiSQL cases and 63.1% of SQuALL cases.

Takeaway. A small local audit can evaluate policies under the expected budget, query scope, and reuse to select the least expensive policy that meets the accuracy target.

We also evaluate whether the main findings remain robust when relaxing key experimental assumptions. App. F, G, and H evaluate robustness to generated query plans (rather than oracle), choice of base LLM for detection, and a holistic repair (rather than oracle repair), respectively.

7 Conclusion and Future Work

Our study shows that data error detection in TQA pipelines have an inherent trade-off between answer accuracy and cost. We observe that question-aware policies outperform Offline detection in answer accuracy by focusing limited detection budgets on relevant data and providing context for semantic errors. Although Offline detection becomes more cost-effective when the workload is reused across many questions, question-aware and online detection policies consistently achieve higher answer accuracy. Our results therefore favor question-aware detection when answer correctness is the priority, with budget and query characteristics guiding the choice between Question-only and Online policies. To empirically compare these policies, we introduce BED-TQA, a configurable benchmark for error detection and repair in TQA. Future work includes expanding our evaluation to broader error taxonomies BED-TQA, and including different repair methods (Ding et al. (2025); Rekatsinas et al. (2017); Mahdavi & Abedjan (2020)).

AI Use Statement

Generative AI tools assisted with portions of the implementation and experimental workflow, proposing candidate semantic corruptions, checking results, and editing the manuscript. The au-

thors determined the research questions, methodological choices, benchmark acceptance rules, and final claims; reviewed the corruption-validation evidence; and verified the reported results against frozen artifacts. No unfrozen generation-time output served directly as evaluation ground truth: accepted semantic pairs were screened, adjudicated, and frozen before evaluation. LLMs also appear as evaluated detector and planning components, as described in the paper. The authors take responsibility for the final content.

Reproducibility Statement

The experimental setup, policies, budgets, metrics, and controlled factors are specified in Sections 4–6. The appendix documents source preparation, injection and provenance validation, detector prompts, uncertainty estimation, the four RQ-aligned main protocols, and the two robustness protocols. The supplementary artifact provides versioned configurations, seeds, frozen manifests and validation ledgers, analysis scripts, tests, and commands for regenerating the reported tables and figures. Upstream datasets are referenced by version and checksum and are acquired separately when redistribution terms require it.

References

Fatemeh Ahmadi, Marc Speckmann, Malte F Kuhlmann, and Ziawasch Abedjan. Matelda: Multi-table error detection. In *EDBT*, pp. 364–376, 2025.

Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Answer sets for consistent query answering in inconsistent databases. In *Proceedings of the International Conference on Logic Programming*, 2002.

Divya Bhadauria, Hazar Harmouch, Felix Naumann, Divesh Srivastava, and Lisa Ehrlinger. A catalog of data errors. *arXiv preprint arXiv:2604.09277*, 2026.

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *Database Theory – ICDT 2001*, pp. 316–330, 2001.

Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. Dr.Spider: A diagnostic evaluation benchmark towards text-to-sql robustness. In *International Conference on Learning Representations*, 2023.

Qixu Chen, Yeye He, Raymond Chi-Wing Wong, Weiwei Cui, Song Ge, Haidong Zhang, Dongmei Zhang, and Surajit Chaudhuri. Auto-Test: Learning semantic-domain constraints for unsupervised error detection in tables. *Proceedings of the ACM on Management of Data*, 3(3):1–27, 2025.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. Binding language models in symbolic languages. In *International Conference on Learning Representations*, 2023.

Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2):179–227, 2000.

Xiaoou Ding, Zekai Qian, Hongzhi Wang, Siying Chen, Yafeng Tang, Hongbin Su, Huan Hu, and Chen Wang. Uniclean: A scalable data cleaning solution for mixed errors based on unified cleaners and optimized cleaning workflow. *Proceedings of the VLDB Endowment*, 18(11):4117–4130, 2025.

Meihao Fan, Ju Fan, Nan Tang, Lei Cao, Guoliang Li, and Xiaoyong Du. AutoPrep: Natural language question-aware data preparation with a multi-agent framework. *Proceedings of the VLDB Endowment*, 18(10):3504–3517, 2025.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145, 2024.

Ken Gu, Zhihan Zhang, Kate Lin, Yuwei Zhang, Akshay Paruchuri, Hong Yu, Mehran Kazemi, Kumar Ayush, A Ali Heydari, Max Xu, et al. Radar: Benchmarking language models on imperfect tabular data. *Advances in Neural Information Processing Systems*, 38, 2026.

Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. TaPas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4320–4333, 2020.

Bojan Karlaš, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. Nearest neighbor classifiers over incomplete information: From certain answers to certain predictions. *Proceedings of the VLDB Endowment*, 2021.

Apostolos Koukouvinis, George Katsogiannis-Meimarakis, and Georgia Koutrika. Comparison and analysis of value linking in text-to-sql systems: [experiments & analysis]. *Proceedings of the ACM on Management of Data*, 4(3):1–26, 2026.

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

Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, and Eugene Wu. Boostclean: Automated error detection and repair for machine learning. *arXiv preprint arXiv:1711.01299*, 2017.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *arXiv preprint arXiv:2305.03111*, 2023.

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. TAPEX: Table pre-training via learning a neural SQL executor. *arXiv preprint arXiv:2107.07653*, 2021.

Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. NL2SQL-BUGs: A benchmark for detecting semantic errors in nl2sql translation. *arXiv preprint arXiv:2503.11984*, 2025.

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.

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

Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: Decomposed in-context learning of text-to-sql with self-correction. *arXiv preprint arXiv:2304.11015*, 2023.

Zipeng Qiu, Chenyue Li, You Peng, Guangxin He, Binhang Yuan, and Chen Wang. Tqa-bench: Evaluating llms for multi-table question answering. *arXiv preprint arXiv:2411.19504*, 2024.

Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: holistic data repairs with probabilistic inference. *Proceedings of the VLDB Endowment*, 10(11):1190–1201, 2017.

Nima Shahbazi, Seiji Maekawa, Nikita Bhutani, and Estevam Hruschka. From textual columns to query plans: A unified relational-semantic execution framework for hybrid query processing. *arXiv preprint arXiv:2604.02444*, 2026.

Tianze Shi, Chen Zhao, Jordan Boyd-Graber, Hal Daumé III, and Lillian Lee. On the potential of lexico-logical alignments for semantic parsing to SQL queries. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. CHESS: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*, 2024.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. MAC-SQL: A multi-agent collaborative framework for text-to-sql. *arXiv preprint arXiv:2312.11242*, 2023.

Zilong Wang, Hao Zhang, Chun-Liang Li, Julian Martin Eisenschlos, Vincent Perot, Zifeng Wang, Lesly Miculicich, Yasuhisa Fujii, Jingbo Shang, Chen-Yu Lee, and Tomas Pfister. Chain-of-table: Evolving tables in the reasoning chain for table understanding. *arXiv preprint arXiv:2401.04398*, 2024.

Jian Wu, Linyi Yang, Dongyuan Li, Yuliang Ji, Manabu Okumura, and Yue Zhang. MMQA: Evaluating LLMs with multi-table multi-hop complex questions. In *International Conference on Learning Representations*, 2025.

Pengcheng Yin, Graham Neubig, Wen tau Yih, and Sebastian Riedel. TaBERT: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 8413–8426, 2020.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 3911–3921, 2018.

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.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

A BENCHMARK CONSTRUCTION AND VALIDATION

We construct the benchmark by injecting typed, reversible errors, following our error taxonomy, into clean table question-answering instances. Each resulting benchmark instance includes a corrupted database, the query answer obtained from that database, a cell-level log of the injected changes, and a conservative set of cells that can affect the answer, which we refer to as the *answer-supporting scope*. We next describe each step of the benchmark construction pipeline.

A.1 SOURCE PREPARATION

We use the short-table stratum of MMQA, WikiSQL, SQuALL, and BIRD Wu et al. (2025); Zhong et al. (2017); Shi et al. (2020); Li et al. (2023). Each source is converted into named tables with stable row and column identifiers and materialized in a fresh SQLite database. We retain an instance only if its reference query executes successfully and its answer-scope can be computed under our supported provenance operator set.

Screening retains 428 of 465 MMQA candidates, 1,997 of 2,000 WikiSQL candidates, 1,876 of 2,000 SQuALL candidates, and 283 of 400 BIRD candidates. The WikiSQL and SQuALL pools are capped and not the full public releases. The reported exclusions arise from unsupported provenance constructs. For each dataset, we create three seeds each selecting 150 master questions. An *instance* is a question paired with a construction seed. Several questions may share a *physical source*, which is the underlying table or database.

A.2 APPLICABLE CELLS AND CORRUPTION OPERATORS

A deterministic profiler assigns each column a syntactic type—numeric, date, categorical, identifier, or free text—and, where available, a semantic label such as person, organization, location, code, address, or monetary value, etc. The profiler uses normalized source labels, recognized value formats, numeric prevalence, distinctness, and mean text length.

Each error type has eligibility rules based on the clean cell value and its column profile. A cell is eligible only if it meets the syntactic and semantic conditions in Table 4. An accepted proposal must change the stored value and use only previously unselected cells. The completed corrupted database must also pass the execution and restoration checks described below. The change log records the cell location, clean and dirty values, error type, seed, and information needed to reverse the operation.

The first four operators are deterministic given the eligible cells and seed. MMQA, WikiSQL, and SQuALL use all seven types. BIRD uses only missing-value, typographical, format/syntax, and misfielded errors because it lacks sufficient validated synonym and abbreviation capacity for the balanced semantic design. We therefore exclude BIRD from semantic claims.

A.3 SEMANTIC CANDIDATE VALIDATION

Synonyms, abbreviations, and embedded forms come from a pool specific to each clean value and column. Candidate generation receives the value, dataset, schema, column profile, and domain description, but not the question, reference query, or answer scope. Thus semantic candidates are generated independently of their usefulness to a question-aware policy. Placement subsequently uses answer scope to ensure semantic coverage.

We request three candidates per eligible cell from `gpt-5.6-terra` at low reasoning. A separate `gpt-5.6-sol` pass at medium reasoning checks the semantic relation, clean-to-dirty direction, and requested error type without seeing the generator identity. Deterministic and semantic checks reject synonyms that change denotation, resemble abbreviations, or exhibit inappropriate lexical overlap. Abbreviations must preserve the entity, satisfy the shortening threshold, and pass ambiguity checks; a conventional prefix is accepted only if independently validated. Both types reject candidates already present elsewhere in the column, where they would be indistinguishable from a legitimate stored value. Rejected proposals are discarded.

A human audit of candidate examples and a separate machine evidence checks whether the protocol operates as intended. The allocator may reuse an accepted pair in compatible cells, but cannot generate new forms or revise decisions. Representation frequency is retained for analysis but neither

Table 4: Corruption operators: eligible inputs, transformations, and required checks. Examples illustrate the transformations; semantic substitutions require approval in their specific column context.

Error type	Eligible input and transformation	Required checks
Missing value	Replace an observed value with a missing-value marker selected using the construction seed.	Reject cells already classified as missing. The replacement must be an empty string, `N/A`, `NULL`, `-`, `unknown`, or `-9999`.
Typographical	Edit one token by deleting or duplicating a character, swapping adjacent characters, or substituting an adjacent keyboard key; for example, `Paris` becomes `Pairs`.	The input is non-missing text of 2–80 characters containing a word of at least two letters. Date and identifier columns are excluded. An edit that leaves the value unchanged is replaced or rejected.
Format/syntax	Change the notation of a recognized date, phone number, percentage, currency, measurement, number, or structured code; for example, `25%` becomes `0.25`.	Both forms must parse to the same underlying value. Checks preserve the date, phone digits, numeric quantity, currency and amount, measurement and unit, or code identity, as applicable. Unparseable alternatives are rejected.
Misfielded	Exchange two distinct, non-missing values in different columns of the same row.	The columns must have different syntactic types, and neither may be an identifier. Each destination must contain the other's original value. Both cell changes are recorded, and reversing the swap must restore both originals.
Synonym	Replace a controlled categorical label, such as a status or role, with another expression of the same meaning; for example, `completed` becomes `finalized`.	The replacement must pass independent semantic validation as a synonym, not an abbreviation or a related concept. It must retain at least 75% of the original character length and pass the column-collision check below.
Abbreviation	Shorten a country, state, province, organization, or department long form; for example, `United States` becomes `US`.	The replacement must be more than 25% shorter and pass independent validation as a short form of the same entity. Expansion of an already abbreviated value is excluded. The column-collision check must also pass.
Embedded value	Add surrounding text to a non-missing atomic value in a non-free-text column; for example, `paid` becomes `status: paid`.	The original representation is at most 80 characters. The replacement must contain it as a contiguous substring, ignoring case, add at least three characters, and pass the column-collision check.

a majority-form assumption nor frequency alone determines which form is treated as erroneous. Prompts, model settings, raw responses, and adjudication records are reported in the artifacts.

A.4 Error allocation

For each dataset and seed, the allocator targets approximately 10% of all materialized cells, divided equally among the applicable error types. Corruption density is the percentage of all cells that are changed; the rate for an individual error type is the percentage of all cells changed by that type. A misfielded swap counts as two affected cells. Both rates use all cells as the denominator, although errors can be placed only in eligible cells. Each type therefore targets about 1.43% of all cells in the seven-type datasets and 2.5% in BIRD.

Allocation jointly considers error type, eligibility, answer-scope membership, and dependency role. Cells are selected without replacement, with at most one type per cell. Mandatory semantic coverage is resolved first, followed by the remaining equal-mass quotas. When synonyms and abbreviations compete for a cell, the scarcer abbreviation capacity takes priority; alternative syntactic fill orders avoid blocking later types. Insufficient supply would be reported as underfilling rather than replaced with another type.

Within these fixed quotas, every question with a validated in-answer-scope synonym or abbreviation pair receives at least one such corruption. For each of these two types, the allocator also targets at least 20% of strictly eligible in-answer-scope cells when the type quota permits. Remaining quota is filled from other eligible cells. There is no per-question operation cap or fixed ratio of in-answer-scope to out-of-answer-scope corruptions.

Why do we opt for 10% error rate uniformly distributed. We use a 10% overall corruption target with equal affected-cell shares across applicable error types to provide a controlled baseline for comparing detection policies. This setting preserves mostly clean database context while introducing substantial task difficulty: in the construction pool, approximately 49–68% of query answers change across datasets and seeds. Balancing the error types reduces confounding from their rel-

ative prevalence and ensures sufficient representation of less abundant semantic errors. We treat this mixture as an experimental reference setting, without assuming that it reflects real-world error frequencies or equal difficulty across types. We leave that for future work. Additionally we add separate density and error composition experiments assess sensitivity to these design choices.

Mixed and matched single-error instances. The mixed suite applies all selected error injections. Each matched single-error instance starts from the same clean master question and applies only the mixed-suite injection of the requested type. Its per-type corruption mass is therefore unchanged, allowing comparisons to isolate the effect of error composition.

A.5 Computing the answer-supporting scope

A cell can affect the query result without appearing in the answer. Its value may determine whether a row passes a filter, joins another table, contributes to an aggregate, or enters a limited result. Exhaustively testing every possible replacement value is impractical. We therefore compute a conservative answer-supporting scope from the dependencies of the executable reference query.

We parse the query into relational operators and propagate dependencies through its execution on the clean database, tracing them back to base-table cells. We distinguish three dependency classes:

Output dependencies.
Cells whose values are returned directly.

Value dependencies.
Cells used to compute a returned value, including inputs to arithmetic expressions and aggregates.

Gating dependencies.
Cells that determine whether a row contributes to the result, which group it belongs to, or where it appears in the ordering.

The scope includes cells from all three classes. The rules below also retain dependencies that currently prevent rows from contributing, since changes to those cells can bring previously excluded rows into the answer.

Selections. For each evaluated filter, we include the base cells used to evaluate its predicate. This applies to both passing and failing rows: changing a predicate value can remove a current contributor or admit a previously excluded row.

Joins. We include cells used in join conditions from both inputs. This covers matched keys, whose values establish existing matches, and unmatched keys, whose values can create new matches if changed. Including unmatched keys captures dependencies that would be missed by tracing only rows in the clean join result.

Aggregation and grouping. We include grouping attributes and the base values used by aggregate expressions. Cells used to evaluate `HAVING` are gating dependencies because they determine whether a group is returned, including groups that currently fail the condition. Predicate and join cells that control aggregate membership are retained even for rows currently excluded before aggregation.

Anti-joins. For supported anti-join constructs, we track cells that determine whether a matching row exists. Introducing a match can exclude an outer row, while removing its last match can admit it. The scope therefore accounts for both existing matches and the values whose changes could create a match.

Ordering and limiting. When ordering affects the answer, we include cells used by `ORDER BY`, including the inputs to computed sort expressions. With `LIMIT`, this also includes ordering dependencies of rows or groups outside the returned subset, since changing their sort values can move them across the cutoff. For example, when groups are ranked by an aggregate, the aggregate inputs of nonreturned groups can still affect which group is selected.

Nested queries. Dependencies of an inner query are propagated to the outer operators that use its result. For example, when an inner aggregate supplies a value to an outer filter, its input cells are tracked because changing that value can change which outer rows pass the filter.

Section A.6 describes the controlled operator tests and instance-level restoration checks used to validate this scope.

A.6 SCOPE VALIDATION

Validation addresses two questions: whether the dependency rules are implemented correctly, and whether the resulting scope is sufficient to recover the clean answer for each answer-changing corruption.

Operator-level tests. We test scope computation using small, purpose-built queries over tables with controlled values. For each test, we specify the expected answer-supporting cells and their dependency roles, then automatically check the computed scope against these expectations. We also change selected cell values and rerun the query to check that the scope captures dependencies that can alter the answer, including changes that make a previously excluded row relevant. Table 5 summarizes the principal cases.

Table 5: Operator-level scope validation. Tests check the expected dependencies and use controlled value changes to exercise membership, value, and ordering effects, including cells absent from the original result.

Construct	Controlled intervention	Required coverage and result
Selection	Change a predicate value so a passing row fails or an excluded row passes.	Predicate cells from both groups are in scope, and re-execution reflects the change in membership.
Join	Alter a matched key to break a join, or an unmatched key to create a match.	Join keys on both inputs are tracked, including unmatched keys; the expected joined rows disappear or appear.
Aggregation and grouping	Change an aggregate input, group key, or value controlling group membership or `HAVING`.	The scope covers both contributing values and membership gates; re-execution reflects the changed aggregate or returned groups.
Supported anti-joins	Introduce a matching value on the inner relation or remove an existing match.	Cells controlling the presence and absence of matches are tracked, and outer-row membership changes accordingly.
Ordering with a limit	Change ordering values so a row or group crosses the result cutoff.	Ordering dependencies include values capable of changing the selected rows or groups; re-execution returns the expected selection.
Nested queries	Change an inner value or membership condition used by an outer predicate or expression.	Dependencies reach the relevant base cells across query levels, and the outer result reflects the intervention.

Instance-level sufficiency check. For every corrupted instance whose answer differs from the clean answer, we restore all corrupted cells inside the computed scope and leave every out-of-scope corruption in place. We then rerun the reference query. The instance passes only if this partial restoration recovers the clean answer. A failure means that the computed scope is insufficient for that instance.

Passing establishes that the recorded in-scope corrections suffice for the observed corruption. It does not establish a minimal repair set or coverage of every possible future corruption. The scope can include redundant cells and dependencies that are inactive under the current values.

A.7 Resulting Benchmark and Error Populations

The construction pool contains 1,800 mixed instances, based on 150 questions from each dataset and three random seeds. Before the final evaluation, we exclude any question whose table or database was used during development or pilot experiments. This ensures that questions based on the same underlying data do not appear in both development and evaluation.

This removes 228 instances and leaves 1,572 mixed and 9,888 matched single-error instances. The mixed suite contains 585 distinct question identifiers across 249 physical-source clusters; 1,554 instances contain a corruption and 18 are clean controls (Table 6).

Table 6: Mixed-suite populations across all three seeds. Each count is a question-seed instance.

Source	Construction	Evaluation	Corrupted evaluation	Clean controls
MMQA	450	423	409	14
WikiSQL	450	393	393	0
SQuALL	450	384	380	4
BIRD	450	372	372	0
Total	1,800	1,572	1,554	18

Table 7 reports the error rate of each type within the answer-supporting scope. Every type uses the same denominator: all in-scope cells, including those left unchanged. We count each cell once per question-seed instance, regardless of how many dependency roles it has, and include clean controls. Counts are pooled across instances and seeds before calculating percentages. Across the confirmatory mixed suite, 15,603 of 150,104 in-scope cell occurrences are corrupted, for an overall in-scope error rate of 10.39%.

These rates describe realized placement after source exclusion. They can differ from the approximately 10% construction target, which applies to all materialized cells, and from the semantic placement target, which uses only eligible in-scope cells as its denominator.

Table 7: Error rates within the answer-supporting scope in the mixed suite (%).

Source	Missing	Typo	Format	Misfielded	Synonym	Abbrev.	Embedded	All types
BIRD	2.74	2.63	2.81	1.85	–	–	–	10.03
MMQA	1.46	1.20	1.92	1.14	1.05	0.86	1.47	9.09
SQuALL	1.43	2.44	1.10	1.88	3.50	0.60	1.38	12.33
WikiSQL	1.36	1.53	0.74	1.13	4.09	4.85	1.39	15.09

The injected errors also change query answers. Before development exclusions, the percentage of instances whose dirty answer differs from the clean answer ranges across seeds from 55.3–60.0% for MMQA, 52.7–54.0% for WikiSQL, 49.3–51.3% for SQuALL, and 65.3–68.0% for BIRD. Both changed and unchanged answers are retained.

A.8 Artifacts

The benchmark stores each clean instance once and records the changes needed to reproduce each corrupted version.

For each instance, the benchmark stores:

- the corresponding clean instance, natural-language question, and executable query;
- the query answers on the clean and corrupted databases;
- the location, original value, corrupted value, and error type of each changed cell;
- the answer-supporting cells and their roles in producing the answer; and
- the intended and realized corruption statistics, together with references to the relevant semantic validation records.

B HYBRID POLICY

Hybrid is another baseline policy that follows the routing rule motivated by an observation we made: Offline detection is sufficient for syntactic errors, while semantic errors need to use the query context available to Online. We assume that this routing is fixed before test evaluation and does not depend on per-instance ground-truth performance.

Hybrid uses the same total probe budget B as the other policies. Let α denote the share allocated to the Offline branch:

$$B_{\text{offline}} = \lfloor \alpha B \rfloor, \qquad B_{\text{online}} = B - B_{\text{offline}}.$$

Unless otherwise stated, we use the development-selected setting $\alpha = .75$, allocating approximately 75% of the budget to syntactic Offline detection and 25% to semantic Online detection. The split is fixed before test evaluation, unused detection actions are not transferred between branches, and sensitivity to α is evaluated separately.

When A questions share the same database, the Offline branch can be run once and amortized across questions, while the Online branch runs separately for each question. The resulting per-question cost is

$$C^{\text{per-query}}_{\text{Hybrid}}(q) = \frac{C_{\text{syntactic-offline}}}{A} + C_{\text{semantic-online}}(q).$$

C ADDITIONAL RQ1 RESULTS

This section provides additional results for the four policies on the mixed and single-error suites. Our primary metric is *answer accuracy* defined as the exact agreement with the clean reference answer after oracle repair. *Answer F1* measures the degree of overlap between the repaired answer and the ground-truth answer tuples. The oracle restores any corrupted cell detected by at least one detector, regardless of the predicted error type. We also report detection metrics to capture differences not reflected by answer accuracy. Unless otherwise noted, 95% confidence intervals are obtained by bootstrapping physical sources together with their questions and seeds, and dataset averages weight each applicable dataset equally.

C.1 MIXED-ERROR RESULTS

Table 8: Answer accuracy on the mixed suite at the default budget. No repair executes the dirty database directly; Oracle/clean restores every injected corruption.

System	BIRD	MMQA	SQuALL	WikiSQL	BED-TQA
No repair	0.411 [0.350, 0.465]	0.452 [0.388, 0.515]	0.497 [0.407, 0.581]	0.445 [0.365, 0.524]	0.451 [0.415, 0.487]
Offline	0.478 [0.434, 0.513]	0.530 [0.468, 0.593]	0.570 [0.479, 0.659]	0.478 [0.393, 0.560]	0.514 [0.475, 0.547]
Question-only	0.699 [0.574, 0.805]	0.657 [0.600, 0.712]	0.716 [0.641, 0.784]	0.669 [0.601, 0.735]	0.685 [0.642, 0.722]
Online	0.750 [0.655, 0.845]	0.593 [0.532, 0.655]	0.766 [0.693, 0.829]	0.636 [0.559, 0.710]	0.686 [0.649, 0.723]
Hybrid	0.478 [0.434, 0.516]	0.567 [0.508, 0.626]	0.643 [0.554, 0.718]	0.656 [0.581, 0.728]	0.586 [0.552, 0.618]
Oracle/clean	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]

C.2 SINGLE-ERROR RESULTS

Takeaway. The RQ1 results highlight how a limited detection budget interacts with policy allocation. Offline can spend detection actions on anomalous columns outside the current question's supporting scope. Online focuses on plan-referenced columns, but each column may require several detectors, and its fixed ordering can exhaust the budget before useful detector–column pairs are reached. Question-only can therefore outperform Online by concentrating detection actions on fewer question-linked columns, although those links may omit dependencies revealed by the plan. The WikiSQL results support this interpretation: Question-only leads at the default budget, while Online overtakes it when the budget doubles. Online's higher aggregate in-scope cell recall also need not translate into higher answer accuracy, because detecting additional answer-preserving corruptions may provide little benefit, while one remaining critical error can keep an answer incorrect. Hybrid reserves separate budgets for syntactic and semantic detection, but its fixed split can leave

Table 9: Mixed-suite detection at the default budget. P and R denote precision and recall. Table-wide metrics include all cells; answer-support metrics restrict evaluation to the conservative supporting scope. *Typed* metrics require the correct cell and error label. *In-scope cell recall* is the fraction of corrupted cells in this scope flagged by any detector, regardless of the predicted type.

		Table-wide typed			Answer-support typed			In-scope
Dataset	Policy	P	R	F1	P	R	F1	cell recall
BIRD	Offline	0.490	0.211	0.248	0.382	0.146	0.169	0.160
	Question-only	0.647	0.154	0.233	0.646	0.393	0.435	0.446
	Online	0.677	0.142	0.223	0.677	0.499	0.516	0.577
	Hybrid	0.465	0.180	0.223	0.319	0.126	0.141	0.141
MMQA	Offline	0.287	0.114	0.148	0.174	0.088	0.107	0.191
	Question-only	0.447	0.164	0.220	0.448	0.337	0.351	0.469
	Online	0.462	0.138	0.195	0.461	0.320	0.348	0.462
	Hybrid	0.330	0.137	0.174	0.322	0.157	0.192	0.260
SQuALL	Offline	0.355	0.168	0.202	0.235	0.185	0.190	0.337
	Question-only	0.495	0.159	0.214	0.455	0.374	0.381	0.554
	Online	0.531	0.131	0.187	0.541	0.519	0.499	0.708
	Hybrid	0.365	0.179	0.211	0.463	0.312	0.332	0.463
WikiSQL	Offline	0.269	0.094	0.127	0.113	0.087	0.088	0.168
	Question-only	0.465	0.190	0.247	0.402	0.380	0.363	0.473
	Online	0.436	0.192	0.239	0.389	0.419	0.367	0.530
	Hybrid	0.433	0.203	0.246	0.396	0.365	0.352	0.435
BED-TQA	Offline	0.350	0.147	0.181	0.226	0.127	0.138	0.214
	Question-only	0.513	0.167	0.228	0.488	0.371	0.382	0.485
	Online	0.526	0.151	0.211	0.517	0.439	0.432	0.569
	Hybrid	0.398	0.175	0.213	0.375	0.240	0.254	0.325

Table 10: *Answer F1* on the mixed suite at the default budget, measuring tuple overlap with the clean reference answer.

Dataset	Offline	Question-only	Online	Hybrid
BIRD	0.628	0.813	0.847	0.625
MMQA	0.645	0.755	0.701	0.680
SQuALL	0.589	0.722	0.776	0.658
WikiSQL	0.478	0.669	0.636	0.656
BED-TQA	0.585	0.740	0.740	0.655

Table 11: Single-error answer accuracy by dataset at the default budget.

Dataset	Error type	Dirty instances	No repair	Offline	Question-only	Online	Hybrid	Oracle/clean
BIRD	Missing	372	0.629	0.669	0.879	0.879	0.661	1.000
	Typographical	345	0.751	0.829	0.904	0.878	0.820	1.000
	Format/syntax	372	0.922	0.930	0.933	0.944	0.930	1.000
	Misfielded	290	0.700	0.731	0.879	0.924	0.728	1.000
MMQA	Missing	272	0.779	0.838	0.893	0.882	0.838	1.000
	Typographical	261	0.843	0.881	0.923	0.885	0.877	1.000
	Format/syntax	279	0.968	0.968	0.971	0.971	0.968	1.000
	Misfielded	159	0.799	0.830	0.887	0.862	0.830	1.000
	Synonym	190	0.758	0.774	0.905	0.816	0.811	1.000
	Abbreviation	201	0.701	0.761	0.836	0.796	0.806	1.000
	Embedded	278	0.791	0.824	0.867	0.856	0.867	1.000
SQuALL	Missing	330	0.864	0.879	0.918	0.973	0.879	1.000
	Typographical	300	0.820	0.877	0.957	0.950	0.890	1.000
	Format/syntax	321	0.994	0.994	0.994	0.994	0.994	1.000
	Misfielded	239	0.837	0.870	0.921	0.971	0.874	1.000
	Synonym	186	0.715	0.780	0.882	0.882	0.871	1.000
	Abbreviation	138	0.667	0.725	0.812	0.790	0.797	1.000
	Embedded	322	0.879	0.891	0.922	0.950	0.938	1.000
WikiSQL	Missing	204	0.980	0.985	0.990	0.990	0.985	1.000
	Typographical	192	0.953	0.969	0.969	0.990	0.969	1.000
	Format/syntax	147	0.980	0.980	0.986	0.986	0.980	1.000
	Misfielded	118	0.949	0.966	0.975	0.983	0.966	1.000
	Synonym	195	0.431	0.441	0.713	0.621	0.672	1.000
	Abbreviation	216	0.606	0.639	0.741	0.755	0.815	1.000
	Embedded	207	0.942	0.952	0.961	0.966	0.966	1.000

Table 12: Single-error answer F1 at the default budget. Values are unweighted means over applicable datasets, with 95% source-cluster bootstrap intervals. F1 gives partial credit for tuple overlap with the clean answer.

Error type	No repair	Offline	Question-only	Online	Hybrid
Missing	0.866 [0.840, 0.890]	0.885 [0.858, 0.906]	0.943 [0.926, 0.959]	0.955 [0.941, 0.970]	0.884 [0.857, 0.906]
Typographical	0.881 [0.854, 0.910]	0.918 [0.896, 0.940]	0.956 [0.941, 0.970]	0.950 [0.935, 0.965]	0.921 [0.899, 0.942]
Format/syntax	0.980 [0.969, 0.989]	0.981 [0.970, 0.990]	0.984 [0.973, 0.993]	0.986 [0.978, 0.993]	0.981 [0.971, 0.990]
Misfielded	0.856 [0.826, 0.881]	0.881 [0.848, 0.904]	0.933 [0.907, 0.953]	0.950 [0.935, 0.965]	0.880 [0.848, 0.903]
Synonym	0.654 [0.578, 0.720]	0.683 [0.612, 0.746]	0.845 [0.797, 0.890]	0.788 [0.728, 0.840]	0.803 [0.747, 0.855]
Abbreviation	0.698 [0.623, 0.761]	0.741 [0.680, 0.793]	0.818 [0.763, 0.866]	0.805 [0.747, 0.856]	0.833 [0.783, 0.875]
Embedded	0.894 [0.871, 0.915]	0.910 [0.889, 0.930]	0.934 [0.916, 0.951]	0.942 [0.926, 0.957]	0.940 [0.924, 0.956]

Table 13: Target-type detection on the same corrupted single-error instances. Each entry gives precision, recall, and F1, in that order, averaged over applicable datasets. The Instances column counts question-seed instances, not corrupted cells.

Error type	Dirty instances	Offline	Question-only	Online	Hybrid
Table-wide typed (precision / recall / F1)					
Missing	1,178	0.447/0.278/0.301	0.464/0.206/0.258	0.482/0.202/0.257	0.426/0.259/0.283
Typographical	1,098	0.392/0.322/0.317	0.381/0.261/0.277	0.319/0.194/0.214	0.387/0.306/0.304
Format/syntax	1,119	0.174/0.054/0.074	0.348/0.130/0.172	0.429/0.181/0.232	0.169/0.049/0.069
Misfielded	806	0.435/0.183/0.237	0.408/0.141/0.190	0.346/0.108/0.150	0.435/0.184/0.237
Synonym	571	0.070/0.048/0.046	0.369/0.323/0.326	0.260/0.220/0.223	0.278/0.234/0.238
Abbreviation	555	0.035/0.024/0.026	0.464/0.320/0.348	0.386/0.295/0.311	0.426/0.319/0.338
Embedded	807	0.127/0.055/0.069	0.183/0.069/0.090	0.186/0.072/0.092	0.182/0.073/0.092
Answer-support typed (precision / recall / F1)					
Missing	1,178	0.221/0.159/0.166	0.393/0.326/0.335	0.460/0.437/0.425	0.205/0.146/0.152
Typographical	1,098	0.154/0.159/0.149	0.286/0.303/0.280	0.258/0.279/0.256	0.150/0.155/0.145
Format/syntax	1,119	0.055/0.029/0.036	0.231/0.184/0.198	0.399/0.373/0.382	0.052/0.027/0.034
Misfielded	806	0.128/0.108/0.106	0.294/0.237/0.241	0.300/0.264/0.261	0.128/0.110/0.108
Synonym	571	0.041/0.046/0.042	0.362/0.357/0.357	0.254/0.247/0.248	0.276/0.268/0.270
Abbreviation	555	0.029/0.029/0.029	0.362/0.355/0.357	0.333/0.327/0.328	0.366/0.359/0.361
Embedded	807	0.030/0.021/0.024	0.111/0.086/0.093	0.166/0.132/0.141	0.169/0.134/0.143

useful detection actions unexecuted in one branch even when the other has unused capacity. Together, these findings suggest that identifying relevant columns is insufficient: performance also depends on which detectors reach those columns before the budget runs out.

D Additional RQ2 Results

These analyses examine how the context provided to the detectors and changes in key parameters affect each policy.

D.1 Detector context

The context experiment fixes the reference-plan candidate columns, detector family, corrupted instance, and budget. It changes only the evidence supplied to the detector, in four cumulative stages:

1. Data only: column values and detector-specific evidence.
2. Masked question: add the question, replacing value mentions with `[VALUE]`.
3. Redacted plan: add query structure and column roles, with literals still hidden.
4. Explicit literals: reveal the original values linked to the relevant columns.

Takeaway. Semantic detection benefits most from access to explicit literal values. Revealing these values improves answer accuracy by 6.8 percentage points and answer-supporting scope recall by 10.3 points in instances with answer-relevant semantic corruptions. In contrast, question or plan context alone provides little additional benefit, and syntactic detection remains largely unaffected by added context.

Table 14: Change in answer-supporting cell recall as context is added. Positive values indicate improved recall. Brackets show 95% source-cluster bootstrap intervals.

Group	Instances	Masked question minus data only	Redacted plan minus masked question	Explicit literals minus redacted plan
Syntactic	2,525	+0.001 [-0.002, +0.003]	+0.001 [-0.002, +0.006]	-0.001 [-0.005, +0.002]
Semantic	891	-0.004 [-0.013, +0.005]	+0.007 [+0.002, +0.013]	+0.065 [+0.037, +0.099]

Table 15: Answer accuracy at each context level by dataset and error family.

Group	Dataset	Instances	Data only	Masked question	Redacted plan	Explicit literals
Syntactic	BIRD	1,143	92.91%	92.91%	92.83%	92.74%
	MMQA	567	91.18%	91.53%	91.53%	91.53%
	SQuALL	640	95.31%	95.31%	95.31%	95.31%
	WikiSQL	175	96.00%	96.00%	96.00%	96.00%
Semantic	MMQA	273	70.33%	70.33%	70.33%	74.73%
	SQuALL	231	62.34%	62.34%	62.34%	68.83%
	WikiSQL	387	58.14%	57.62%	57.88%	59.95%

Table 16: Change in answer accuracy as each context level is added, by error type. Brackets show 95% source-cluster bootstrap intervals.

Error type	Instances	Masked question minus data only	Redacted plan minus masked question	Explicit literals minus redacted plan
Missing	738	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]
Typographical	560	+0.002 [0.000, +0.006]	-0.001 [-0.007, +0.004]	+0.003 [0.000, +0.008]
Format / syntax	663	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	-0.002 [-0.004, 0.000]
Misfielded	564	+0.002 [0.000, +0.008]	0.000 [0.000, 0.000]	-0.002 [-0.008, 0.000]
Synonym	472	-0.004 [-0.011, +0.004]	+0.002 [-0.004, +0.008]	+0.055 [+0.021, +0.097]
Abbreviation	419	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	+0.022 [+0.007, +0.040]
Embedded value	369	+0.002 [0.000, +0.007]	-0.002 [-0.007, 0.000]	+0.002 [0.000, +0.007]

D.2 Detection budget

We first examine how the detection budget affects each policy. All budget levels use the same mixed-suite instances, with larger budgets extending the same ordered sequence of detection actions. We consider both fixed budgets and budgets scaled by the number of database columns.

Table 17 reports results for fixed budgets of 8, 16, and 32 detection actions per question. Zero detection actions corresponds to the no-repair baseline. A policy may use fewer than the allowed number of detection actions if it runs out of applicable detector–column pairs.

We also scale the budget by database width in Table 18. For example, a budget of 0.5 allows one detection action for every two columns, while a budget of 2 allows two detection action per column. The *Full* setting removes the budget limit and evaluates every detector–column pair considered by each policy. Because the policies consider different pairs, Full may correspond to different numbers of detection actions.

Takeaway. Detection budget affects which query-conditioned policy performs better. With a tight budget, Question-only can benefit from concentrating its detection actions on a smaller scope. As more detection actions become available, Online can cover more of its broader plan-derived scope and generally becomes more effective.

D.3 Error density

We next examine how the policies respond as the number of corrupted cells increases. We use the same 100 instances per dataset at each density level. The corruption sets are nested, so errors present at lower densities remain as more errors are added. The reported densities are the actual percentages

Table 17: Answer accuracy at fixed detection budgets.

Dataset	Policy	Instances/ sources	0	8	16	32
BIRD	Offline	372/6	0.411	0.484	0.535	0.645
	Question-only	372/6	0.411	0.608	0.753	0.780
	Online	372/6	0.411	0.610	0.777	0.863
	Hybrid	372/6	0.411	0.489	0.538	0.637
MMQA	Offline	423/141	0.452	0.511	0.582	0.678
	Question-only	423/141	0.452	0.586	0.709	0.745
	Online	423/141	0.452	0.515	0.648	0.759
	Hybrid	423/141	0.452	0.522	0.608	0.723
SQuALL	Offline	384/39	0.497	0.544	0.591	0.685
	Question-only	384/39	0.497	0.682	0.734	0.745
	Online	384/39	0.497	0.685	0.781	0.786
	Hybrid	384/39	0.497	0.602	0.656	0.732
WikiSQL	Offline	393/63	0.445	0.483	0.570	0.662
	Question-only	393/63	0.445	0.758	0.799	0.799
	Online	393/63	0.445	0.771	0.809	0.809
	Hybrid	393/63	0.445	0.662	0.789	0.809
BED-TQA	Offline	–	0.451	0.506	0.569	0.668
	Question-only	–	0.451	0.659	0.749	0.767
	Online	–	0.451	0.645	0.754	0.804
	Hybrid	–	0.451	0.569	0.648	0.725

Table 18: Answer accuracy as the normalized detection budget increases.

Dataset	Policy	Instances/ sources	0.25	0.5	1	2	Full
BIRD	Offline	372/6	0.430	0.454	0.478	0.591	0.863
	Question-only	372/6	0.511	0.597	0.699	0.753	0.780
	Online	372/6	0.538	0.605	0.750	0.833	0.863
	Hybrid	372/6	0.425	0.454	0.478	0.589	0.858
MMQA	Offline	423/141	0.466	0.478	0.530	0.626	0.735
	Question-only	423/141	0.511	0.556	0.657	0.719	0.745
	Online	423/141	0.478	0.494	0.593	0.712	0.766
	Hybrid	423/141	0.475	0.489	0.567	0.664	0.768
SQuALL	Offline	384/39	0.531	0.536	0.570	0.630	0.747
	Question-only	384/39	0.596	0.667	0.716	0.742	0.745
	Online	384/39	0.589	0.664	0.766	0.786	0.786
	Hybrid	384/39	0.555	0.578	0.643	0.695	0.792
WikiSQL	Offline	393/63	0.450	0.455	0.478	0.545	0.662
	Question-only	393/63	0.491	0.529	0.669	0.794	0.799
	Online	393/63	0.478	0.506	0.636	0.809	0.809
	Hybrid	393/63	0.463	0.483	0.656	0.751	0.809

of corrupted cells and may vary slightly across datasets because of rounding and error eligibility. Some low-density instances contain no corruption, but they are retained to keep the comparison matched across density levels.

Takeaway. Accuracy decreases for all policies as corruption density increases. However, Question-only and Online generally maintain a larger advantage over Offline at higher densities, suggesting that focusing a fixed detection budget on query-relevant data becomes more useful as errors become more frequent.

Table 19: Answer accuracy as corruption density increases at the default detection budget.

Dataset	Density (%)	Instances	No repair	Offline	Question-only	Online	Hybrid
BIRD	1.30	100	0.830	0.850	0.940	0.950	0.850
	3.64	100	0.700	0.730	0.870	0.910	0.730
	10.15	100	0.400	0.520	0.680	0.730	0.530
MMQA	1.21	100	0.960	0.960	0.980	0.980	0.970
	3.51	100	0.830	0.860	0.900	0.880	0.870
	10.02	100	0.410	0.490	0.630	0.580	0.530
SQuALL	1.69	100	0.860	0.870	0.930	0.960	0.920
	4.70	100	0.730	0.770	0.860	0.920	0.840
	10.99	100	0.530	0.610	0.710	0.820	0.680
WikiSQL	0.99	100	0.950	0.950	0.960	0.950	0.970
	3.14	100	0.860	0.870	0.910	0.930	0.930
	10.94	100	0.500	0.530	0.660	0.690	0.700

D.4 Query scope

A query's column scope is the fraction of database columns referenced by its plan. Figure 5 groups queries by their scope within each dataset. It reports both answer accuracy and scope coverage, defined as the fraction of reference-plan columns that receive a detection action.

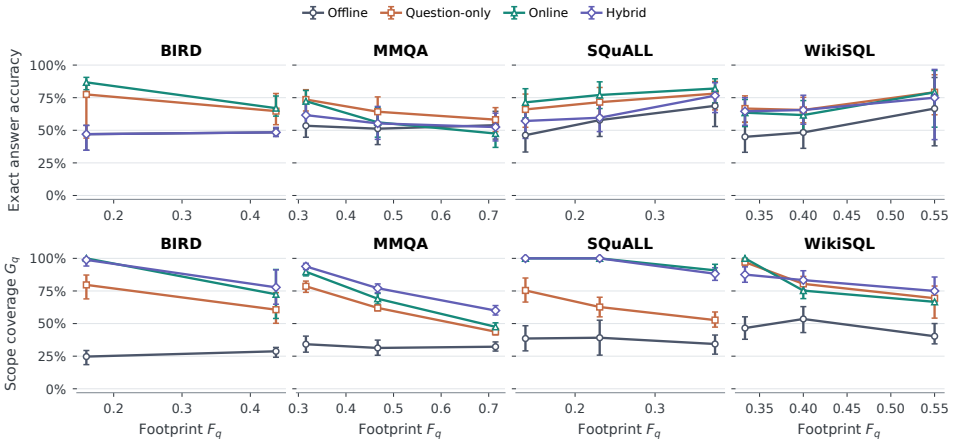

Figure 5: Answer accuracy and coverage of reference-plan columns across query scopes at the default budget. Group boundaries are specific to each dataset.

In the narrowest scope groups, Online exceeds Offline by 18.5–39.7 accuracy points. The gap generally becomes smaller as queries reference a larger fraction of the database. Because these groups come from naturally occurring queries, they may also differ in complexity and other characteristics, so this comparison should be interpreted as observational.

Takeaway. Query-conditioned cleaning is most useful when a query touches only a small part of the database. In these cases, Online can focus its detection actions on relevant columns, while Offline spreads the same limited budget across a much larger search space.

D.5 Database Size

We next test the effect of database width directly by adding 0, 8, 16, or 32 distractor columns outside the query scope. The question and original corruptions remain unchanged, and every instance receives the same budget of 32 detection actions.

Adding irrelevant columns mainly hurts policies that search more broadly. With 32 distractor columns, Offline loses 11–22 accuracy points across datasets and Hybrid loses 8–17 points. In

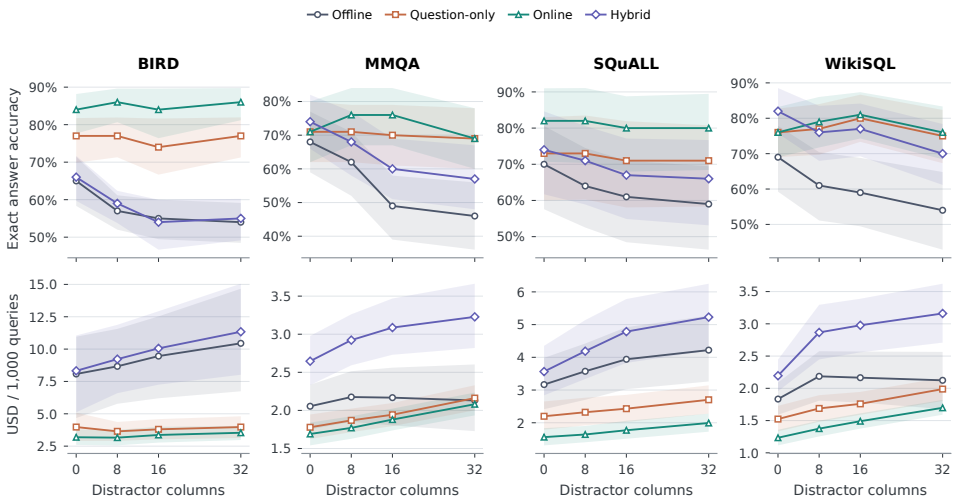

Figure 6: Accuracy and realized cost as distractor columns are added. The question and original corruptions remain unchanged, with a fixed budget of 32 detection actions.

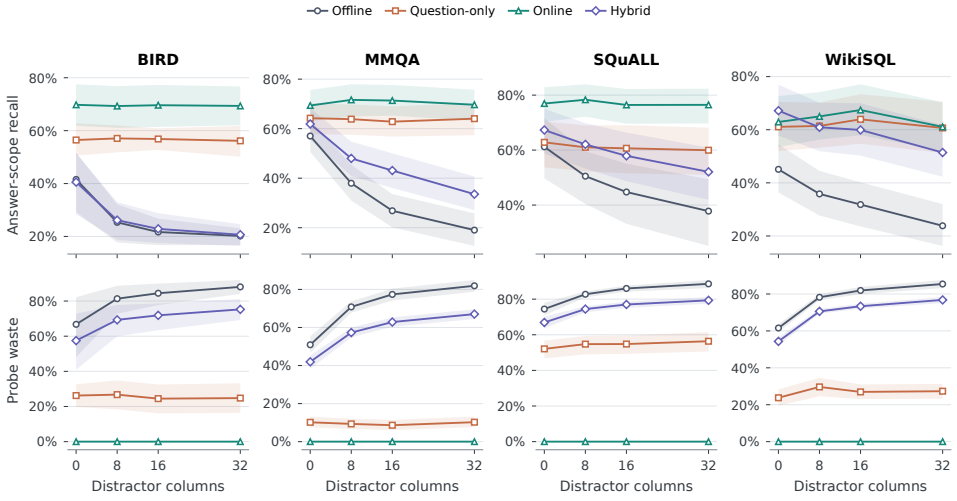

Figure 7: Answer-supporting cell recall and detection waste as distractor columns are added. Detection waste is the fraction of detection actions assigned to columns with no answer-supporting cells.

contrast, Question-only and Online change by at most two points because the added columns fall outside their query-relevant scope.

Takeaway. Irrelevant columns can waste a limited detection budget. Offline and Hybrid become less effective as distractor columns are added, while Question-only and Online remain relatively stable by focusing their detection actions on query-relevant columns.

D.6 Hybrid budget allocation

We next examine how Hybrid should divide its detection budget between its two branches. A syntactic share of 0 assigns the entire budget to the semantic Online branch, while a share of 1 assigns the entire budget to the syntactic Offline branch. Table 20 shows that the preferred split depends on the error composition. The highest observed accuracy occurs at a syntactic share of 0.75 when errors are syntactic-heavy, compared with 0.5 when semantic errors are more common. These values describe the tested instances rather than universal optimal settings.

Table 20: Hybrid answer accuracy and cost for different syntactic budget shares.

	Syntactic-heavy		Semantic-enriched	
Syntactic share	Answer acc.	USD / 1k	Answer acc.	USD / 1k
0	0.683	0.664	0.582	0.651
0.25	0.704	0.978	0.600	0.966
0.5	0.722	1.298	0.610	1.295
0.75	0.732	1.521	0.577	1.511
1	0.730	1.704	0.526	1.688

Takeaway. Hybrid's best budget split depends on the types of errors in the data and should be estimated from a small local sample.

E Additional RQ3 Results

We first report the cost and resource use of each policy when processing a single question. Table 21 breaks down accuracy and resource use at the default budget $B = C$. Cost includes detection, linking, ranking, and allocation calls.

Table 21: Per-dataset answer accuracy and mean resource use at $B = C$.

Dataset	Policy	Exact acc.	Calls/q	Tokens/q	Latency (s/q)	USD/1k
BIRD	Offline	0.478	9.46	15497	14.77	5.562
	Question-only	0.699	6.88	7076	10.52	2.826
	Online	0.750	6.38	6302	12.50	2.615
	Hybrid	0.478	9.90	15251	21.98	5.635
MMQA	Offline	0.530	4.70	2344	4.67	0.910
	Question-only	0.657	5.22	2613	6.62	1.270
	Online	0.593	3.77	2014	6.28	1.052
	Hybrid	0.567	5.70	3093	8.92	1.460
SQuALL	Offline	0.570	6.04	4024	7.65	1.781
	Question-only	0.716	6.53	3971	8.08	1.805
	Online	0.766	5.20	3095	8.55	1.446
	Hybrid	0.643	7.15	4782	12.54	2.184
WikiSQL	Offline	0.478	2.53	1227	2.57	0.489
	Question-only	0.669	3.87	1766	4.52	0.834
	Online	0.636	4.08	1886	5.79	0.885
	Hybrid	0.656	4.08	1893	5.88	0.898

Table 22 reports three forms of waste work: unused detection actions, findings outside the answer-supporting scope, and avoidable processing.

Detection waste is the fraction of detection actions assigned to columns with no answer-supporting cells. *Finding waste* is the fraction of flagged cells outside the conservative scope. *Processing waste* additionally accounts for work on irrelevant cells within partially relevant columns. For detection action p over cells I_p with cost C_p, we define:

$$W_{\text{proc}} = \frac{\sum_p C_p |I_p \setminus \mathcal{A}(q, \mathcal{D}^*)|/|I_p|}{\sum_p C_p}.$$

We next examine how cost changes when multiple questions share the same source. The reuse study contains 36 source–seed and 234 question–seed instances from BIRD, SQuALL, and WikiSQL. MMQA is excluded because each retained source contains only one question.

As reuse increases, Offline can spread its database-wide cleaning cost across multiple questions, while Question-only and Online repeat their cleaning work for each question. Table 23 reports the observed points at which Offline becomes as cheap as or cheaper than the other policies. These crossovers correspond to the tested reuse levels rather than estimated thresholds between them.

Finally, Figure 8 examines whether the effect of reuse also depends on query scope. It compares narrow- and broad-scope workloads within the same source and reuse level. The relationship varies

Table 22: Per-dataset single-query waste fractions at $B = C$.

Dataset	Policy	Probe waste	Finding waste	Processing waste
BIRD	Offline	0.678	0.802	0.810
	Question-only	0.157	0.408	0.409
	Online	0.000	0.211	0.216
	Hybrid	0.530	0.794	0.794
MMQA	Offline	0.448	0.497	0.661
	Question-only	0.033	0.172	0.262
	Online	0.000	0.030	0.055
	Hybrid	0.338	0.447	0.627
SQuALL	Offline	0.662	0.734	0.703
	Question-only	0.409	0.441	0.460
	Online	0.000	0.127	0.123
	Hybrid	0.513	0.681	0.627
WikiSQL	Offline	0.482	0.358	0.611
	Question-only	0.019	0.212	0.366
	Online	0.000	0.159	0.269
	Hybrid	0.331	0.320	0.568

Table 23: Observed Offline cost crossovers by dataset and comparison.

Dataset	Maximum A	vs. Question-only	vs. Online	vs. Hybrid
BIRD	14	4	8	4
SQuALL	4	1	1	1
WikiSQL	4	1	1	1

across datasets, suggesting that the number of questions sharing a source does not by itself determine when Offline becomes more cost-effective.

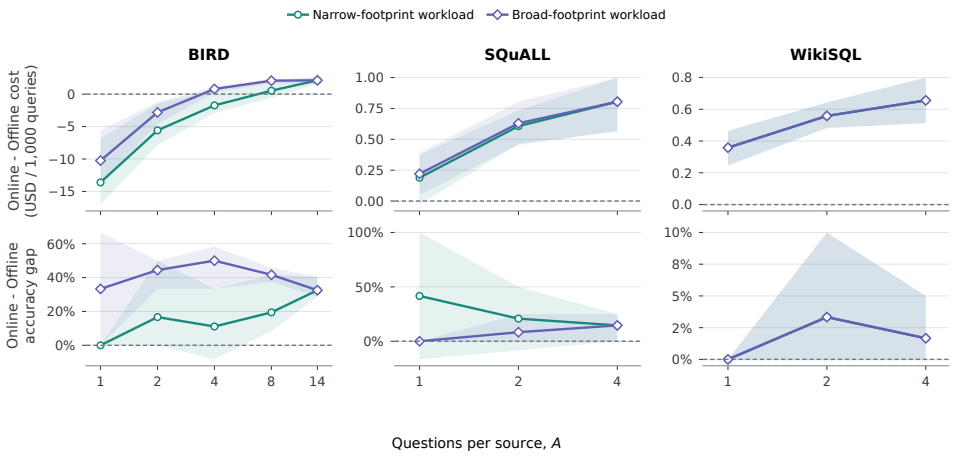

Figure 8: Online–Offline cost and accuracy gaps across reuse for narrow- and broad-scope workloads.

Takeaway. For small workloads, question-conditioned policies avoid much of the unnecessary work of database-wide cleaning. As more questions share the same source, Offline becomes cheaper per question because its cleaning cost is reused. However, the crossover depends on the dataset and workload, and lower cost does not necessarily mean higher answer accuracy.

F Robustness to Generated Plans

Our primary experiments use reference SQL to isolate the effect of error detection. Here, we instead generate SQL with `gpt-5-mini` from each corrupted database and use it both to guide Online and Hybrid detection and to compute the final answer. The planner receives the question, schema, and

a limited sample of table values, all from the corrupted database. We generate one query for each question–construction-seed instances and use the same query for Online and Hybrid. The detection budget is $B = C$.

Table 24: Plan and answer accuracy (%) with plans generated from corrupted databases. *Plan accuracy* measures whether the generated query returns the reference answer on clean data.

Dataset	Instances	Plan acc.	Answer acc. (before repair)	Answer acc. (after Online)	Answer acc. (after Hybrid)
BIRD	372	39.8	20.2	33.1	21.8
MMQA	423	61.2	24.8	34.5	33.3
SQuALL	384	61.5	32.8	49.5	43.5
WikiSQL	393	84.2	39.7	53.4	57.0
BED-TQA	1,572	61.7	29.4	42.6	38.9

Generated queries achieve 61.7% plan accuracy and 29.4% answer accuracy before repair (Table 24). Detection and oracle repair increase answer accuracy to 42.6% with Online and 38.9% with Hybrid. In comparison, their reference-plan accuracies are 68.6% and 58.6%, respectively. Thus, using generated plans reduces answer accuracy by 26.0 percentage points for Online and 19.7 points for Hybrid.

Answer-supporting detection recall decreases from 56.9% to 50.5% for Online, while Hybrid remains nearly unchanged (32.5% to 33.0%). However, similar detection recall does not guarantee a correct answer when the generated query itself contains an incorrect join, filter, or aggregation.

Takeaway. Detection and repair improve accuracy with generated plans, but query-generation errors remain a major source of failure. We therefore use reference plans in the primary experiments to isolate the effect of error detection.

G Detector-Model Robustness

We next test whether the results depend strongly on the detector model. We repeat the default-budget mixed-suite experiment on all instances using `gemini-3.8-flash`, while keeping the plans, prompts, oracle repair, and budget $B = C$ unchanged.

Table 25: Detector-model robustness at $B = C$; Δ is Gemini minus GPT-5 Mini.

(a) Dataset-macro accuracy and logical cost

Policy	GPT answer acc.	Gemini answer acc.	Δ answer acc. [95% CI]	GPT $/query	Gemini $/query
Offline	0.514	0.498	-0.016 [-0.028, -0.006]	$0.00219	$0.00727
Question-only	0.685	0.676	-0.009 [-0.034, $+0.016$]	$0.00168	$0.00570
Online	0.686	0.659	-0.027 [-0.042, -0.014]	$0.00150	$0.00495
Hybrid	0.586	0.571	-0.016 [-0.032, -0.002]	$0.00255	$0.00824

(b) Dataset-level answer accuracy change

Dataset	n_c	Offline	Question-only	Online	Hybrid
BIRD	372	-0.003	$+0.038$	-0.019	-0.005
MMQA	423	-0.012	-0.031	-0.028	-0.024
SQuALL	384	-0.023	-0.031	-0.018	-0.008
WikiSQL	393	-0.025	-0.013	-0.043	-0.025

Compared with `gpt-5-mini`, average accuracy changes by -0.016, -0.009, -0.027, and -0.016 for Offline, Question-only, Online, and Hybrid, respectively. Question-only and Online continue to achieve higher accuracy than Hybrid and Offline, although the exact differences and model-specific costs change.

Takeaway. Changing the detector model shifts absolute accuracy slightly but preserves the main policy-level pattern observed in the primary experiments while the answer accuracy and cost differences remain model-dependent.

H REPAIR ROBUSTNESS

Our primary experiments use oracle repair to isolate the effect of detection. We therefore run an additional experiment that replaces oracle restoration with a holistic repair agent while keeping the detector outputs fixed. The held-out stress test contains 120 instances, with 30 from each dataset.

For each flagged cell, the repair agent receives its column profile, value frequencies, same-row evidence, detector findings, and the context available to the corresponding policy. It does not receive clean values, answers, corruption records, or oracle repairs. Type-compatible repairs are applied when confidence is at least 0.9.

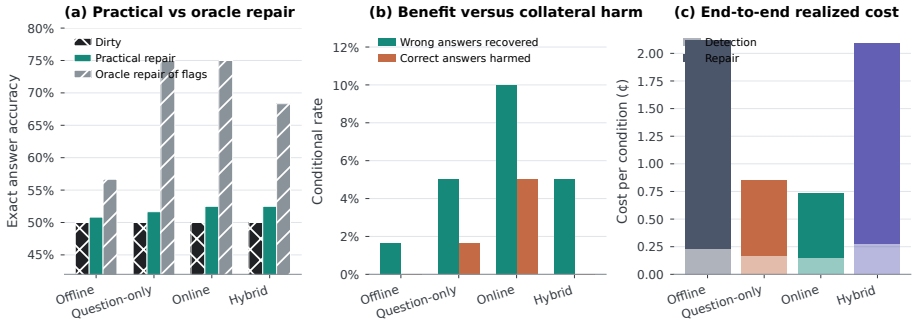

Figure 9: Held-out holistic-repair accuracy, collateral effects, and end-to-end cost.

Starting from an no-repair accuracy of 0.500, holistic repair increases answer accuracy to 0.508 for Offline, 0.517 for Question-only, and 0.525 for both Online and Hybrid. Online repairs 10.0% of initially incorrect answers but also changes 5.0% of initially correct answers into incorrect ones. Hybrid repairs 5.0% of initially incorrect answers without harming any initially correct answers in this sample.

Table 26: holistic repair and oracle restoration of the same detected cells by dataset (%).

Dataset	Offline	Question-only	Online	Hybrid
MMQA	50.0/60.0	50.0/73.3	50.0/63.3	50.0/63.3
WikiSQL	53.3/53.3	50.0/83.3	50.0/80.0	56.7/83.3
SQuALL	50.0/56.7	56.7/76.7	63.3/83.3	53.3/70.0
BIRD	50.0/56.7	50.0/66.7	46.7/73.3	50.0/56.7

Table 27: holistic-repair average differences from Offline with 95% cluster-bootstrap intervals.

Paired contrast	Δ answer accuracy (%)	Δ answer F1 (%)
Question-only − Offline	+0.8 [-2.7, +4.4]	-0.3 [-3.8, +3.0]
Online − Offline	+1.7 [-2.7, +5.9]	+0.5 [-2.9, +4.0]
Hybrid − Offline	+1.7 [+0.0, +4.1]	+1.1 [-1.1, +3.7]

Repair success also varies substantially by error type. Typographical and format/syntax errors are easier to restore than missing and misfielded values, while semantic repairs depend more strongly on the context available to the policy.

Takeaway. Replacing oracle repair with a holistic repair agent reduces the accuracy gains seen in the controlled experiments, showing that successful detection does not always translate into successful repair. Nevertheless, holistic repair improves over the dirty baseline for all four policies, and the experiment confirms that repair quality depends strongly on error type and available context.

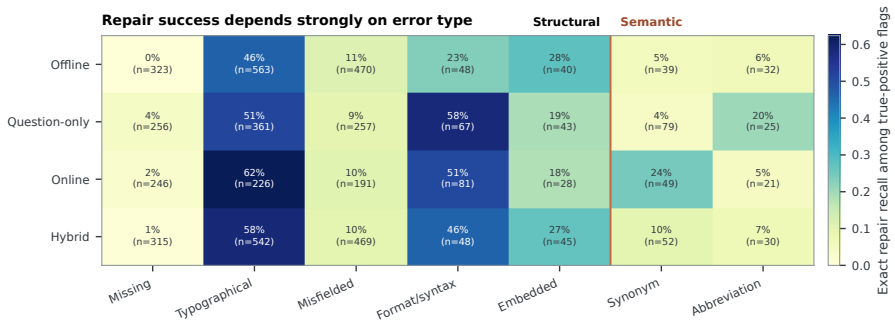

Figure 10: Exact restoration among true-positive detector flags by error type. Cells report recall and support n.

I Detector Implementation Details

All detectors use frozen syntactic and semantic profiles, value frequencies, and detector-specific evidence. Columns longer than 200 rows are processed in chunks along with finding from past iterations. A finding becomes a flag at confidence $s_c \geq .5$. Base models, prompts, and thresholds are shared across policies and only query scope, allocation, and permitted context differ. Table 28 describes implementation details on each detector.

Table 28: Implemented deterministic screens and LLM-based checks in the shared detector bank.

Detector	Deterministic screen	LLM-based check
Missing value	Normalize Unicode, whitespace, and case; match a fixed universal-sentinel list. Admit short ($\leq$ 25 characters), low-frequency ($\leq$ 10%) non-universal tokens for review, and separately detect word placeholders in predominantly numeric or date columns.	Given the column name, inferred semantics, a capped value histogram, and frequent normal contrasts, select tokens that denote missingness. Reject tokens absent from the histogram or covering at least 40% of the column.
Typographical	Require at least four eligible string or categorical values; skip identifier, date, time, phone, and amount columns. Rare forms within a capped Levenshtein distance of a frequent form become weak hints; at most 150 distinct values are retained.	Review every retained distinct value with its frequency and optional edit-distance hint. Flag clear spelling, keypress, or phonetic corruption even if the canonical form is absent; exclude valid rare values, identifiers, synonyms, and abbreviations.
Format/syntax	Exclude blanks, identifiers, and columns without a stable format contract. For date, phone, time, code, and status columns, build character-class skeleton and parsed-kind histograms, require a dominant structured pattern, and directly detect redundant trailing-zero formats.	Infer the majority skeletons and an anchored regex from the histogram and examples. Validate regex conformity and self-consistency, then combine skeleton, regex, and type-specific parsers to judge off-pattern values; abstain when the inferred format is weak or the violation share exceeds 40%.
Misfielded	Exclude key columns. Infer each column's dominant broad kind (number, date, or text) and mechanically identify same-row reciprocal cross-kind mismatches.	For same-kind columns with different inferred semantics, inspect at most six partner columns and 40 rows. Accept only an explicit bidirectional swap and validate every returned row and partner against the supplied sample.
Synonym	Require at least two eligible nonblank string or categorical forms; skip identifiers, single-character and numeric forms, garbled encodings, and typo-close pairs. Data-only scans are capped at 30 distinct values, with tighter guards for name and code columns.	Without an anchor, group co-present forms that have the same meaning, require the proposed canonical form to occur in the column, and apply relation-specific confidence gates. If a policy supplies a question mention or plan literal, test column values against that anchor. Reject abbreviations, related concepts, and character-level typos.
Abbreviation	Apply the same eligibility, cardinality, encoding, and typo guards as the synonym detector; code columns remain eligible only at low cardinality.	Without an anchor, group co-present complete and shortened forms under confidence gates. If a policy supplies an anchor, test column values against it. Reject synonyms, numeric word forms, ambiguous short forms, different entities, and character-level typos.
Embedded value	Require at least three rows and an atomic-cell majority of at least 60%. A candidate must be non-atomic, contain at least four words or 15 characters, and preserve a recoverable number, date, or existing atomic token; retain at most 25 candidates.	Distinguish an atomic value wrapped in surrounding text from a genuinely descriptive value. If the model call fails, the implementation falls back to the mechanically admitted candidate set.

J MODELS, PROMPTS, HYPERPARAMETERS, AND EXECUTION ENVIRONMENT

Table 29 records every hosted model used in our work, together with the local execution environment:

Table 29: Model and execution configuration.

Role	Model alias	Decoding and use
Primary detection and linking	`gpt-5-mini`	OpenAI through LiteLLM; seed 42 and minimal reasoning. Temperature was not transmitted. The completion cap was normally 2,048 tokens and 4,096 only for targeted retries of exhausted or invalid-JSON calls.
Holistic repair	`gpt-5-mini`	OpenAI through LiteLLM; seed 42, low reasoning, JSON-object responses, and at most 4,096 completion tokens. Repairs below confidence .9 are not applied; at most one contract-only partial retry is allowed.
Secondary detection	`gemini-3.8-flash`	Vertex AI through LiteLLM; seed 42, low reasoning, and at most 2,048 completion tokens. Temperature was not transmitted.
Semantic generation	`gpt-5.6-terra`	OpenAI Responses API; low reasoning and three candidates per eligible cell.
Semantic validation	`gpt-5.6-sol`	OpenAI Responses API; medium reasoning. Generator identity and output scores were withheld.
Difficulty screening	`gpt-5.6-terra`	OpenAI Responses API; low reasoning. The four context levels were evaluated as a diagnostic only.
Local execution	`Python 3.13.9`; `SQLite 3.51.0`	Apple M4 Pro host (12 CPU cores, 48 GB memory) running macOS 26.6.2; hosted inference is excluded.

J.1 PROMPT TEMPLATES

Prompt 1: Semantic corruption generation

Inputs: `<dataset domain>`, `<table>`, `<column>`, `<schema>`, `<clean value>`, and `<existing values>`. The question, SQL, plan, answer, policy output, and evaluation metric are not provided.

Instruction: For *synonymous representation*, produce a natural, lexically distinct word or phrase with exactly the same meaning in the supplied column domain. Do not abbreviate, expand an abbreviation, translate, merely change case, reorder words, or introduce a related but different concept. For *abbreviated representation*, produce a natural shortened form of the clean value. It must be meaningfully shorter, must not be a simple leading-prefix truncation, and must denote exactly the same value. The clean value must be the longer form.

Return: structured JSON containing the proposed value, proposed type, confidence, relation explanation, and any ambiguity.

Prompt 2a: Independent semantic validation

Independently judge `<clean value, candidate>` using only the dataset domain, table, column, schema, and existing column values. Confirm only exact task-level equivalence. Reject related concepts, opposites, hypernyms, unintended translations, typos, format changes, simple truncations, and candidates already present in the column. For abbreviations, verify that the candidate is the shortened corrupted form; for synonyms, reject abbreviation or expansion relations. Return JSON with relation validity, direction validity, predicted type, confidence, and a concise explanation. This call uses the validation model in Table 29.

Prompt 2b: Difficulty screening (separate call). Given exactly one declared context mode (data only; question and schema; plan columns without literals; or plan columns and literals), report separately whether the evidence identifies semantic equivalence and erroneous direction. Return false rather than using unavailable context. Do not use the ground-truth label, clean repair value, policy output, or evaluation metric. Include confidence and a short rationale. This call uses the screening model in Table 29.

Prompt 3: Question-only schema linking

System: You perform schema linking without generating or inspecting SQL. Return JSON only.

User: Link the question to database columns using only the supplied schema and sample values. Return JSON with key `links`; each link has `table`, `column`, `score` in [0, 1], and `question_value_mentions` copied verbatim from the question. Do not infer or produce SQL.

Payload: {"question": <question>, "schema": [<table, column, up to five distinct sample values>]}.

Prompt 4: Per-type detector interface

Common payload fields (normalized summary): `Column: <name>; inferred type: <type>; values/frequencies or format summary: <profile>; detector-specific evidence: <evidence>.` The exact profile and evidence fields vary by detector. Synonym and abbreviation prompts additionally receive permitted question or plan anchors. Offline receives no query context; Question-only uses its schema links and question mentions; Online uses the shared question mentions plus plan roles, fragments, and literals. Other detectors use query information only through policy selection and allocation.

Type	System instruction retained in the prompt	JSON output
Missing	Identify only tokens that encode no value, unknown, or not applicable; exclude rare-but-valid categories and omit universal sentinels handled deterministically.	`missing_tokens, reasoning`
Typographical	Flag only clear misspellings or keypress corruptions; exclude synonyms, abbreviations, valid rare entities, identifiers, and codes.	`corrupted, suspected_canonical`
Format/syntax	Infer the majority character-class format; return a strict anchored Python regex, or mark genuinely varied columns as free text.	`description, patterns, regex, multimodal/free-text, conformity, examples`
Misfielded	Flag a row only when a bidirectional same-row swap is evident; an unusual or rare value is insufficient.	`row, partner`
Synonym	Confirm only unambiguous same-meaning alternatives; exclude shortened forms and abbreviations.	`match/pairs/groups, canonical`
Abbreviation	Confirm only unambiguous shortened or contracted forms; exclude synonyms and numeric word forms.	`match/pairs/groups, canonical`
Embedded	Identify an atomic value wrapped in free text; exclude genuinely descriptive cells.	`row_indices`

Prompt 5: Generated-plan robustness

System: You generate one read-only SQLite query for a table-question-answering task. Use only the supplied question and schema. Return one JSON object with exactly one field, `sql`. Do not return markdown, explanations, DDL, DML, PRAGMA, or multiple statements.

User: Write the SQL query that answers the question. Use only listed tables and physical column names. Semantic labels explain opaque physical names. Question-matched values preserve exact database spelling and casing; representative values are unlinked column examples, not rows. Same-name joins are candidates, not guaranteed keys. Check projection, joins, filters, aggregation, grouping, ordering, and limits before returning the query.

Payload: {`"question": <question>, "schema": <enriched schema>, "dialect": "SQLite"`}. Reference SQL, answers, and dirty values are excluded.

Prompt 6: Semantic detector adjudication

Data-only synonym system instruction: Given unique column values, identify groups that represent the same concept through synonyms only—different words or representations with the same meaning. Do not include abbreviations or shortened forms. Be strict: group values only when the synonym relationship is unambiguous. Return JSON only.

Data-only abbreviation system instruction: Given unique column values, identify groups that represent the same concept through abbreviations only— shortened or contracted forms. Do not include synonyms or numeric word forms. Be strict: group values only when the abbreviation relationship is unambiguous. Return JSON only.

Question-anchor system instruction: A natural-language question mentions a concept, but the mention is not an exact SQL filter literal and is not automatically the database's canonical representation. Use the mention only as semantic evidence. Identify a column value only when it is an anomalous alternate representation relative to the column's other values and frequencies. Never flag the dominant compatible database form merely because it paraphrases the question. Return JSON only.

Plan-literal instruction: Given a linked SQL filter literal and the other values in its column, identify only an unambiguous synonym or abbreviation that the literal would exclude. Reject typos, merely related values, and different entities.

Payload and output by mode: Data-only receives the column name, inferred semantics, and unique values and returns groups with canonical form, variants, and confidence. Question-anchor receives a question mention plus column values and frequencies and returns a matched anomalous value, canonical alternative, and confidence. Plan-literal receives the linked literal plus other column values and returns a match, relation type where applicable, and confidence.

J.2 HYPERPARAMETERS

Table 30: Benchmark construction settings and validation thresholds.

Parameter	Setting
Sampling and corruption allocation	
Construction seeds	1701, 2701, and 3701.
Questions per dataset and seed	150 master questions.
Overall corruption target	Approximately 10% of all materialized cells.
Error-type allocation	Equal affected-cell shares across seven types for MMQA, WikiSQL, and SQuALL ($\approx 1.43\%$ of all cells per type), and four syntactic types for BIRD ($\approx 2.5\%$ per type).
Semantic placement target	For synonyms and abbreviations separately, 20% of eligible answer-supporting cells, subject to the type quota and available validated candidates.
Minimum semantic coverage	At least one in-scope synonym or abbreviation corruption for each question with an actionable validated candidate.
Cell reuse	At most one error type per cell. Misfielded swaps modify two cells and count as two affected cells.
Semantic generation and validation	
Candidate generator	`gpt-5.6-terra`, low reasoning; three candidates per request; 800-token output limit.
Independent validator	`gpt-5.6-sol`, medium reasoning; 700-token output limit.
Semantic acceptance	Relation, transformation direction, and requested error type must all pass. Confidence is recorded but not used as an acceptance threshold.
Transformation thresholds	
Typographical input	2–80 characters, including a word with at least two letters.
Synonym length	At least 75% of the original character length.
Abbreviation length	More than 25% shorter than the original.
Embedded value	Original value at most 80 characters; replacement adds at least three characters and retains the original as a contiguous substring, ignoring case.

Table 31: Primary detector configuration and internal thresholds.

Parameter	Setting
Shared inference and execution	
Primary model	`gpt-5-mini`, minimal reasoning; seed 42.
Temperature	Omitted from primary-model requests.
Completion limit	2,048 tokens; 4,096 for targeted retries after token exhaustion or invalid JSON.
Request attempts	Up to three per request.
Rows per chunk	At most 200; all chunks of the selected column are processed.
Final flag threshold	Confidence ≥ 0.5, after detector-specific checks.
Default confidence	Missing: 0.97; typographical: 0.90; format/syntax: 0.80; misfielded: 0.85; synonym: 0.68; abbreviation: 0.74; embedded: 0.80.
Detector-specific settings	
Missing value	Histogram capped at 30 distinct values with up to eight ordinary examples. Candidate tokens: ≤ 25 characters and $\leq 10\%$ frequency; LLM-proposed sentinels: $< 40\%$ frequency.
Typographical hints	Candidates have at least three characters; reference forms occur at least twice. Edit distance is $\lfloor L/6 \rfloor$, restricted to 1–2 edits; at most 25 hints.
Typographical scan	At most 150 distinct eligible values, ordered by frequency. The LLM may detect errors without an edit-distance hint.
Format/syntax evidence	At least four nonblank values, or ten for code/status columns; at most 40 format signatures. Dominance threshold: 70%, with a 40% top-three fallback; 85% for code/status columns.
Format/syntax acceptance	Minimum estimated conformity: 35% for date/phone/time and 80% for code/status columns. Abstain if more than 40% of nonblank values would be flagged.
Regex validation	Must match $\geq 60\%$ of nonblank values and $\geq 80\%$ of supplied matching examples, and reject $\geq 50\%$ of supplied violations when available.
Misfielded value	At least three nonblank values; inspect up to 40 rows and six partner columns; flag at most 60 rows per column and chunk.
Synonym/abbreviation scan	Values have at least two characters. Unanchored scans require 2–30 distinct eligible values, capped at five for name columns and 15 for code columns.
Synonym/abbreviation confidence	≥ 0.85 for unanchored grouping; ≥ 0.60 with a question mention or plan literal, increased to ≥ 0.80 when the anchor already occurs in the chunk.
Embedded value	At least three nonblank values and $\geq 60\%$ atomic values. Candidates require at least four words or 15 characters; at most 25 are checked.
Linking and allocation	
Schema-linker examples	Up to five distinct nonempty values per column.
Allocation seed	42 for reproducible ordering of tied actions.
Default detection budget	One detector–column detection actions per database column on average.
Hybrid budget split	75% to syntactic Offline detection, rounded down; the remainder to synonym, abbreviation, and embedded-value detection on plan-referenced columns. Unused detection actions are not transferred between branches.