DemandPrep: Demand-Driven Data Preparation via Agentic Action Allocation and Operator-Grounded Execution

Zekai Qian
Harbin Institute of Technology, China
qzk010728@gmail.com

Xiaoou Ding*
Harbin Institute of Technology, China
dingxiaoou@hit.edu.cn

Hongzhi Wang
Harbin Institute of Technology, China
wangzh@hit.edu.cn

Chen Wang
Tsinghua University, China
wang_chen@tsinghua.edu.cn

ABSTRACT

Agentic data systems can invoke many data-preparation operators, but they still need a task-level decision about whether a downstream demand calls for preservation, selective data-side intervention, or an all-cleaner regime. We present DemandPrep, a demand-driven data preparation framework that separates upstream action allocation from operator-grounded execution. The allocator decides whether a candidate unit should remain unchanged or enter a data-side action family such as Repair, Delete, or Replace, and the executor realizes selected actions through reusable operator registries (cleaners, filters, value estimators) and emits concrete operations with provenance. We evaluate DemandPrep on five native-error datasets and 40 mixed-error injected settings. Against named fixed cleaning methods, DemandPrep achieves the highest average utility and ranks first or second on almost every aggregate row. Where DemandPrep leads, relative gains over the strongest fixed cleaner reach 6.8% on native rows and 13.7% on injected rows. The selective branch improves over No-op on 6 of 10 aggregate rows and never falls below it after validation, while winning or tying FullCleaners on 9 of 10 rows with an average gain of 0.0254. The results confirm that preparation is not monotone in cleaning strength. Some tasks demand preservation, some selective intervention, and some all-cleaner execution.

VLDB Workshop Reference Format:
Zekai Qian, Xiaoou Ding, Hongzhi Wang, and Chen Wang. DemandPrep: Demand-Driven Data Preparation via Agentic Action Allocation and Operator-Grounded Execution. VLDB 2026 Workshop: ADS 2026: The Joint Workshop on Agentic Data Systems and Data-Centric AI (The 1st ADS & 3rd DATAI).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/qzkinhit/DemandPrep-artifact.

1 INTRODUCTION

Data-centric AI has made data quality a first-order design problem rather than a preprocessing detail [13, 14], and recent surveys connect data quality and governance to the changing value of data as an asset [4]. Dirty, missing, inconsistent, or biased records can dominate downstream model behavior even when the model architecture is unchanged [1, 17, 18]. Existing data cleaning research has developed many ways to detect and repair erroneous cells, including probabilistic inference, contextual repair, rule reasoning, and mixed-error workflow coordination [6, 7, 20, 21, 23, 28]. These methods provide concrete operators for data-centric systems, but they do not by themselves determine whether a downstream task is better served by preserving a data unit, operating on it, committing all available cleaners, or exposing a model-side adaptation path.

Existing cleaning pipelines implicitly assume that applying more operators to more suspicious units is uniformly beneficial, leaving the question of *which* intervention a given task actually warrants unaddressed before any operator workflow is compiled. In practice, this assumption does not hold because the intervention decision is not monotone in the amount of cleaning. Repairing every suspicious cell may increase cell-level correctness but can erase rare patterns or overfit to an imperfect reference. Deleting all suspicious rows may remove harmful records but can collapse sample diversity. Automatic replacement is often lower cost, yet it can propagate systematic errors when the value estimator is misaligned with the target metric. Conversely, leaving data unchanged is not always a weak baseline. It is a rollback outcome when validation finds limited data-side gain. FullCleaners is the opposite regime, indicating that applying all available cleaners has higher validation utility than selective pruning.

As a concrete instance of this demand, consider a tabular dataset containing typographical errors, missing values, outliers, and dependency violations, used to train a downstream classifier. A conventional cleaning pipeline asks which cleaner to run and in what order. A task-driven allocator instead asks which records or cells are worth changing for the downstream model. A typo in a high-impact feature warrants repair. A row with implausible values warrants deletion. A missing value in a low-sensitivity feature can be safely replaced by an automatic estimator. A rare but valid boundary example warrants preservation even if a detector flags it as suspicious. If validation shows that none of these edits improves the task metric, the allocator falls back to the No-op regime. If the complete all-cleaner workflow is aligned with the task, it admits FullCleaners rather than pruning beneficial operators. Figure 1 summarizes this action-allocation boundary.

This boundary exposes three challenges for agentic data preparation.

(1) Task demand cannot be inferred from repair correctness alone. The same detected issue may require different actions under different downstream metrics, models, budgets, or validation splits. The allocator therefore learns intervention intensity from task feedback instead of from a fixed ordering of suspicious cells.

*Corresponding author.

Proceedings of the VLDB Endowment.

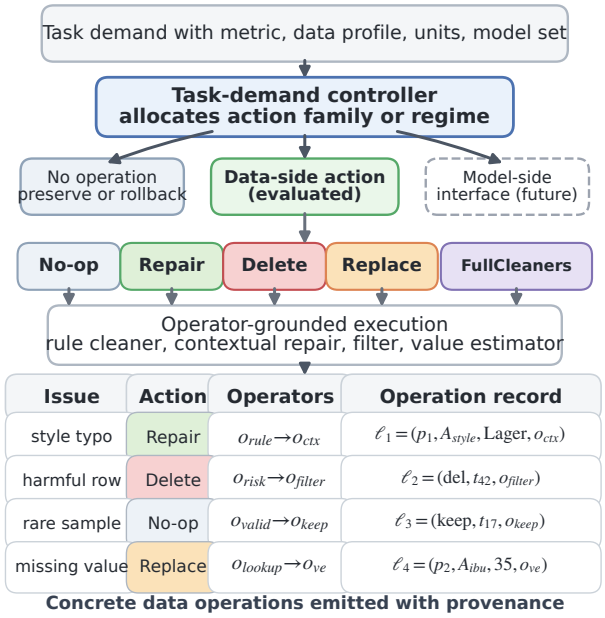

Figure 1: Action-allocation boundary of DemandPrep. An action-allocation agent chooses no operation, a data-side action, or a model-side interface. The evaluated data-side branch then routes data-side action families through operator execution and emits concrete repair, deletion, preservation, and replacement operations with provenance.

(2) Action decisions and operator execution live at different granularities. Choosing no operation, data-side intervention, or a model-side adaptation interface is a top-level action decision. Choosing Repair, Delete, Replace, or No-op specifies the intent of a data-side preparation decision, but it does not determine which rules, contextual signals, estimators, or filters should be ordered into a workflow, nor which concrete value update or row operation should be emitted.

(3) Feedback updates both regimes and operators. A prepared dataset is treated as a candidate version. Validation may accept selective execution, roll back to No-op, or promote FullCleaners. The same feedback can also update which actions and operators are trusted later, without requiring new manual labels just to train the allocation policy.

DemandPrep addresses these challenges with an allocator and executor loop. An action family captures the semantic intent of a data-side intervention such as No-op, Repair, Delete, or Replace. An operator is an executable procedure under that intent, such as a contextual cleaner, a rule-based repairer, a value-estimation routine, or a filtering operator. The allocator assigns action families according to task demand, the executor realizes each selected action as an ordered operator workflow and materializes the resulting data operations, and downstream validation selects the final execution regime and feeds utility back to both layers. The tabular preparation instantiation studied here uses a learned action allocator trained without acquiring new human ground-truth labels. The policy uses existing operator outputs, cached cleaned versions, and downstream validation feedback, but does not request new manual labels to learn which action should be taken.

The paper makes four contributions.

(1) **A demand-driven preparation formulation.** We formulate preparation around task demand rather than around a fixed cleaning objective. At the action-allocation level, an agent may decide that a demand calls for no operation, data-side intervention, or a model-side adaptation interface. This paper instantiates and evaluates the data-side branch for tabular data preparation, where the control boundary is an action family or workflow regime. It includes No-op rollback when the data side offers limited measurable gain, selective action allocation when only part of the emitted data operations is useful, and FullCleaners when all available cleaners are beneficial. This formulation makes preservation and full-cleaner execution explicit decisions of the agent rather than weak baselines or external competitors.

(2) **A zero-new-label allocator and executor architecture.** We design a two-level agent in which a hierarchical action allocator learns whether to select a unit for data-side intervention and how much intervention it needs without requesting additional human labels. An operator-level executor then compiles the selected action into ordered operator workflows and concrete data operations. The allocator uses downstream utility, budget, detector, model, and execution-history signals. The executor uses action-indexed operator registries so that each action family has its own compilation network.

(3) **An auditable execution trace.** DemandPrep records action-level, workflow-level, operator-level, and operation-level history for each prepared data version. The trace supports rollback, promotion to FullCleaners, operator-weight updates, and reuse across future preparation demands. We evaluate the interface through tabular data preparation, where the implemented action families are preservation, repair, deletion, and replacement.

(4) **Empirical evidence for demand-driven intervention.** We evaluate five native-error datasets and 40 mixed-error injected settings against fixed cleaners, DemandPrep regimes, and oracle diagnostics. Against named fixed cleaning methods, DemandPrep achieves the highest average utility (0.5372) and ranks first or second on almost all aggregate rows. Where DemandPrep leads, relative gains over the strongest fixed cleaner reach 6.8% on native rows and 13.7% on injected rows. The selective branch improves over No-op on 6 of 10 aggregate rows and never falls below it after validation, and wins or ties FullCleaners on 9 of 10 rows with an average gain of 0.0254. Across 40 injected settings, the selective branch reaches or exceeds the best non-oracle reference in 21 cases and triggers rollback in 18. The action traces show that native Tax is deletion-heavy, native Beers and Flights are replacement-heavy, and injected Tax shifts toward a No-op/Delete mixture.

2 RELATED WORK

Prior work on data preparation differs in the decision boundary it exposes before execution. Fixed cleaning systems decide how to produce a repaired table. Workflow systems decide how to coordinate operators across mixed error types. Task-aware preparation

systems use downstream feedback, but typically decide which sample, pipeline, or repair strategy to apply. We survey these three lines and a related line on data-centric AI, noting where each boundary stops.

Fixed data cleaning operators. Classical cleaning work studies how to detect and repair data errors using data-level evidence [1]. Cleaning techniques have also been surveyed and specialized for particular data types, including IoT streams and time-series databases [8, 9]. Constraint-based systems reason over denial constraints, dependencies, and violation structures to search for consistent repairs [3, 30]. Rule- and similarity-based methods also drive feature engineering for entity matching, where description-similarity rules generate flexible matching features over records [33]. Scalable cleansing systems organize rules and operators into executable workflows for large data [16]. Probabilistic and Bayesian systems integrate constraints, co-occurrence statistics, and uncertainty to infer likely cell repairs [27, 28], while optimal-transport and conditional-independence approaches target more specific statistical violations [25]. Learning-based, retrieval-based, and language-model-assisted systems further expand the signal space by using contextual representations, transferred correction evidence, clean corpora, or LLM reasoning [10, 11, 20, 21, 24, 35]. These methods provide concrete repair, detection, imputation, and filtering operators. Their decision boundary, however, is usually a repaired cell, a repaired table, or a fixed cleaning pipeline optimized by data-level criteria. They do not decide whether the downstream task should preserve, repair, delete, or replace a unit, nor whether the workflow should commit all available cleaners.

Operator coordination and mixed-error workflows. Real tables rarely contain a single isolated error type. A repair produced for one attribute may become the source evidence for another operator, and two operators may propose conflicting values for the same target attribute. Existing systems study multi-signal integration, dependency-aware execution, and workflow optimization [6, 7, 16, 22, 28, 30, 31]. Recent mixed-error systems further show that unified cleaner interfaces and optimized workflows can improve repair scalability. This line of work establishes that operators must expose their scope, target, cost, output, and provenance so that a workflow can reason about dependencies and conflicts. The remaining gap is the control boundary above the workflow. Coordinating cleaners answers how a repair workflow should run once repair is desired. It does not by itself answer whether the task demands repair at all, or whether deletion, low-cost replacement, preservation, rollback, or all-cleaner execution is the better task-level decision.

Task-aware data preparation. Another line connects data cleaning or preparation with downstream machine learning. ActiveClean prioritizes records for cleaning under model training feedback [18], BoostClean searches over detector and repair combinations for model performance [17], and GoodCore studies data-efficient learning over incomplete data [2]. CPClean focuses on certain predictions over incomplete information [15]. Context-aware and pipeline-level systems construct or optimize preparation plans for ML tasks [12, 22, 31]. DemandClean is closest in spirit in treating cleaning as action selection under data authenticity, diversity, and model tolerance [26]. These systems establish the central premise that downstream utility should guide data preparation. The control boundary remains different. They usually clean selected records, choose a pipeline, optimize an order, or target a specific prediction setting. DemandPrep makes the per-unit action family explicit and treats No-op rollback, selective execution, and FullCleaners promotion as explicit task-level outcomes rather than as external baselines.

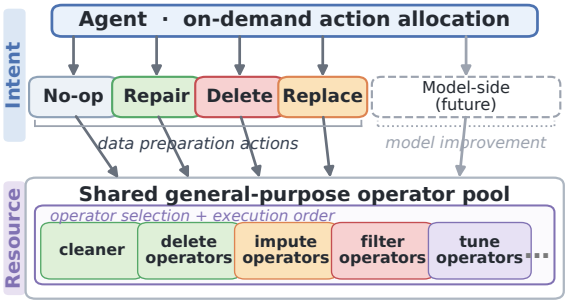

Figure 2: Layered design behind DemandPrep. The agent makes an on-demand allocation across action families (intent layer), and every selected family draws operators from the same shared general-purpose pool (resource layer). Data-preparation actions and the extensible model-improvement branch reuse cleaners, filters, value estimators, and tuners through this pool rather than reimplementing them.

Data-centric AI and extensible preparation actions. Data-centric AI argues that model behavior can be improved by changing data as well as by changing models [13, 14], and data-quality-aware MLOps emphasizes that data interventions must be tracked and evaluated in operational ML workflows [29]. Related preparation operators include uncertainty-driven imputation, imputation-order optimization, and task-aware recommendation of imputation algorithms [5, 34, 39], data valuation and selection [19, 37], augmentation operators such as mixup and CutMix [36, 38], and diversity-aware tabular data generation that uses LLMs to expand data heterogeneity and rare patterns [32]. Model-side adaptation also belongs to the broader action space, because a task may be better served by changing an estimator or training configuration than by editing the data. This paper does not evaluate non-cleaning actions or model-side adaptation. Such choices are treated as future action families whose units, admissible operations, costs, outputs, and provenance must be registered before the allocator can select them. The evaluated instance is tabular data preparation, where this boundary is instantiated by preservation, repair, deletion, and replacement.

3 PROBLEM FORMULATION

We formalize the layer that sits between a downstream task and a library of preparation operators. At the action-allocation level, an agent may decide whether a demand should trigger no operation, data-side intervention, or model-side adaptation. This paper focuses on data-side action allocation for tabular data preparation, where cleaning operators such as repair, deletion, and replacement serve as the evaluated action families. Let D be a dirty training dataset for task T with evaluation metric U. The user or task interface may

also provide a candidate model set $\mathcal{M}$, such as random forest, SVM, gradient boosting, or a task-specific estimator. A data profiler, detector, cached prepared version, or prior workflow output produces candidate preparation units $E = \{e_1, \ldots, e_n\}$. A unit can be a cell, a row, a feature group, or a structured violation. In the tabular preparation instance, it is usually a detected cell-level error. The agent observes a state s_i for each unit, chooses a data-side action family a_i, and asks an executor to realize the selected family by ordering operators and emitting concrete data operations. The output is a workflow W, a prepared dataset D_W, and an execution trace τ_W. Figure 2 summarizes this layered view.

For example, in the Hospitals task used later, T is measure-code classification, U is accuracy, E contains candidate cells or violations from upstream detection, and $\mathcal{A}$ contains preservation, repair, deletion, and replacement. In the Tax-10K task, T is rate regression, so a suspicious record may be more useful as a deletion candidate than as an individual cell repair.

Definition 1 (Task demand). *A task demand is a tuple*

$$q = (T, U, \mathcal{M}, D, E, \mathcal{A}, \Omega, B, \mathcal{P}),$$

where T and U define the downstream task and metric, $\mathcal{M}$ is the optional candidate model set, E is the preparation-unit set, $\mathcal{A}$ is the action-family space, $\Omega = \{\Omega_a \mid a \in \mathcal{A}\}$ is the action-indexed operator registry, B is the human or execution budget, and $\mathcal{P}$ contains policy constraints such as whether deletion is allowed. Evidence sources such as detector scores, rule violations, cached cleaned versions, validation outputs, or prior execution traces are encoded in the state s_i. Governance regimes are defined separately because they are commit decisions after execution rather than fields of the user demand.

The tuple is deliberately wider than a dirty table and a detector output. It makes the downstream objective, candidate models, action space, operator registry, and policy constraints visible before execution, so the same suspicious cell can be handled differently when the budget, task metric, candidate model, or admissible operators change.

Definition 2 (Action family). *An action family $a \in \mathcal{A}$ denotes the semantic intent of a preparation decision. It is not an executable program. In the evaluated instantiation, the data-side action space is*

$$\mathcal{A}_{\text{clean}} = \{\textsc{No-op}, \textsc{Repair}, \textsc{Delete}, \textsc{Replace}\},$$

corresponding to preservation, correction, removal, and low-cost substitution. The action space is an interface rather than a closed list. Adding a family requires specifying its preparation units, state features, admissible operators, cost model, validation protocol, and provenance schema. We evaluate the tabular data preparation instance, but the allocator and executor communicate through this extensible action and operator boundary.

The action family is local to a preparation unit. A workflow still needs a dataset-level decision after execution, because a selective candidate may fail validation, while applying all available cleaners may be better than selective pruning. This motivates a separate regime variable.

Definition 3 (Execution regime). *An execution regime $\rho \in \mathcal{G}$ denotes the workflow-level decision after action execution and validation. In the current instantiation,*

$$\mathcal{G}_{\text{clean}} = \{\textit{rollback}, \textit{selective}, \textit{fullcleaners}\}.$$

The rollback regime returns the current data version and corresponds to dataset-level No-op when edits do not improve utility. The selective regime commits only the data operations emitted by action-allocated workflows. The FullCleaners regime commits the output produced by applying all available cleaners when all-cleaner execution is beneficial.

A regime selects which data version should be committed, but it still does not specify what the executor emits. We first define the operation record that appears in the workflow and provenance trace.

Definition 4 (Data operation). *A data operation ℓ is the auditable edit or preservation record emitted by an action workflow for a preparation unit. It records the selected action family, the scope of affected tuples or cells, the target attribute when applicable, the proposed value when applicable, the originating operator, execution cost, and commit status. In repair or replacement, we write*

$$\ell = (a, p, A_t, v', o, c_\ell),$$

where p is the predicate scope, A_t is the target attribute, v' is the candidate value, and o is the operator that generated the value. When the action and cost are clear, we use the shorter form $\ell = (p, A_t, v', o)$. A deletion operation records a scoped removal such as $\ell = (\textsc{Delete}, p, o, c_\ell)$, and a No-op operation records intentional preservation of the unit. This object is the unit that can be committed, pruned, rejected, or traced back to an operator.

The executor contract formalizes how a selected action family is realized as an ordered operator workflow.

Definition 5 (Operator execution). *For each action family a, Ω_a denotes the concrete operators that can realize it. Each operator exposes a scope, input requirements, source attributes, target attributes, cost, output type, and provenance fields. The execution function $g(e_i, a_i, \Omega_{a_i})$ does not choose a single operator. It constructs or reuses an action-specific operator network, orders applicable operators, generates candidate operations ℓ, and appends accepted operations to workflow W.*

Finally, the allocator is the policy that chooses among these action families under the task demand.

Definition 6 (Demand-driven policy). *The allocator is a policy $\pi_\theta(e_i, s_i, q) \to a_i$ that maps an issue, a task-aware state, and a demand specification to an action family. The policy is demand-driven because the same issue may receive different actions under different downstream metrics, budgets, or operator availability.*

The optimal tuple $(\rho^*, \pi^*, W^*, M^*)$ maximizes downstream utility after execution while accounting for human and execution costs.

$$\begin{aligned} \max_{\rho,\pi,W,M} \quad & U(M, T, D_{\rho,\pi,W}) \\ & - \lambda C_{\text{exec}}(\rho, W) - \gamma C_{\text{human}}(\rho, W) \\ \text{s.t.} \quad & \rho \in \mathcal{G},\ \pi \in \Pi, \\ & W \in \mathcal{W},\ M \in \mathcal{M}, \end{aligned} \tag{1}$$

subject to the allowed action space, operator availability, mutual exclusivity for each preparation unit, and any user-specified budget

or policy constraints. If $\mathcal{M}$ contains a single model, Equation 1 reduces to the fixed-model setting. In the current instantiation, human cost matters because ground-truth repair is expensive and often unavailable at deployment time. Under zero-new-label training, the allocator learns the policy from existing operator outputs, cached preparation results, and downstream validation feedback, without requesting new manual labels.

This formulation differs from operator-only cleaning. A fixed cleaner directly searches for a repair D' that improves data-level correctness. DemandPrep searches over action families, operator realizations, and workflow regimes with respect to downstream utility. Downstream validation adds a decision layer around this objective. When a candidate execution lowers the validation utility relative to the reference data, the final decision can roll back to No-op. When applying all available cleaners is consistently better than selective pruning, the system can expose FullCleaners as the appropriate execution regime. Rollback means that the data side offers limited measurable gain for the current task, while FullCleaners means that the task benefits from all-cleaner execution.

4 DEMANDPREP FRAMEWORK AND EXECUTION LOOP

Figure 3 shows the framework for the evaluated data-side loop. The section follows the execution order of the system. Step 1 converts a downstream request into a typed demand. Step 2 trains an action allocator over the candidate units without requesting new manual labels. Step 3 executes the learned actions by indexing into action-specific operator registries, ordering the operators, and committing concrete data operations. Step 4 records action, workflow, operator, and operation trace, and the downstream-validation panel feeds utility back to both the allocator and the operator weights, which can accept the selective candidate, roll back to the reference data, promote FullCleaners, or update the allocation policy.

4.1 Step 1 Demand Specification

Conventional data preparation pipelines embed the preparation intent implicitly. A cleaner runs because it was configured, not because the task was interrogated. The first step of DemandPrep makes the intent explicit by resolving the preparation need into the typed demand tuple q before any operator is invoked. This explicit representation is what gives the system its agentic character. The allocator reasons over a legible description of what the task demands rather than inheriting a fixed operator schedule, and the same legibility lets the downstream-validation feedback in Step 4 update the demand and the allocator policy in interpretable units. A request such as preparing the hospital table for measure-code prediction is not actionable until these fields are instantiated. Once they are, a repair operator, a row filter, and a value estimator can all be considered by the same allocator without being treated as interchangeable tools.

The tuple q includes the downstream task and target column, the validation metric, optional candidate downstream models, candidate preparation units, action families, action-indexed operator registries, budget, and policy constraints. These fields let a future system distinguish data-side actions from model-side adaptation, because both the candidate model set and admissible action families are visible before execution. In the current tabular preparation instance, the candidate units are detected cells and additional cells suggested by differences between dirty data and cached cleaning outputs. The evidence sources used for state encoding, denoted by $\mathcal{S}$ in the algorithms, include detector confidence, rule violations, feature importance from the downstream model, cached repair values, and execution histories. The operator registry includes cleaning operators, value estimators, row filters, and pass-through operators.

Each operator is registered by a small schema containing source scope, target scope, input requirements, output type, cost, and provenance fields. For example, a repair operator may expose $A_s \rightarrow A_t$, a detector function, and a value generator. A delete operator exposes a row scope and a retention constraint. A replacement operator exposes a candidate-value source and admissible domain. This contract lets the allocator reason over action families without knowing the internal implementation of every operator, and lets the executor later build action-specific operator networks.

4.2 Step 2 Training the Action Allocator without New Labels

The second step trains a task-aware action allocator without requesting any new manual annotations. The challenge is supervisory. A reinforcement-learning allocator needs to know whether each chosen action improved the prepared dataset, which normally requires ground-truth corrections that no one provides at deployment time. DemandPrep derives this signal from the dirty dataset itself. Detector outputs partition each training table into a candidate-error set and a relatively clean residual. Controlled errors are then injected into the residual at known positions, so the reference value of each injected cell is known by construction rather than by annotation. The allocator is trained over these injected positions while detector-flagged cells in the original table supply additional state context, and downstream validation utility on a held-out split provides the reward. No human is asked to label a real dirty cell, which is what zero-new-label means in our setting. When candidate downstream models are provided, the demand interface can select or fix the evaluation model used by the allocator. The reported experiments fix one downstream model per dataset, so the learned policy is evaluated as data-side action allocation rather than as model selection. At inference time the trained allocator is applied to genuinely detected issues on unseen data, with no synthetic substrate present.

Training is cast as a sequential decision problem over E. Each candidate unit is represented by a state vector rather than by its raw value alone. In the tabular preparation instance, the state vector contains local signals such as issue type, column type, detector confidence, feature importance, boundary distance, and column error rate. It also contains global data-quality signals such as sample retention and distributional retention, budget signals such as remaining human budget and remaining issue ratio, and execution-history signals such as whether an operator candidate exists and how similar operators behaved previously. The policy then learns the following mapping from state and demand to an action family.

$$\pi_\theta(e_i, s_i, q) \rightarrow a_i \in \{\text{No-op}, \text{Repair}, \text{Delete}, \text{Replace}\}.$$

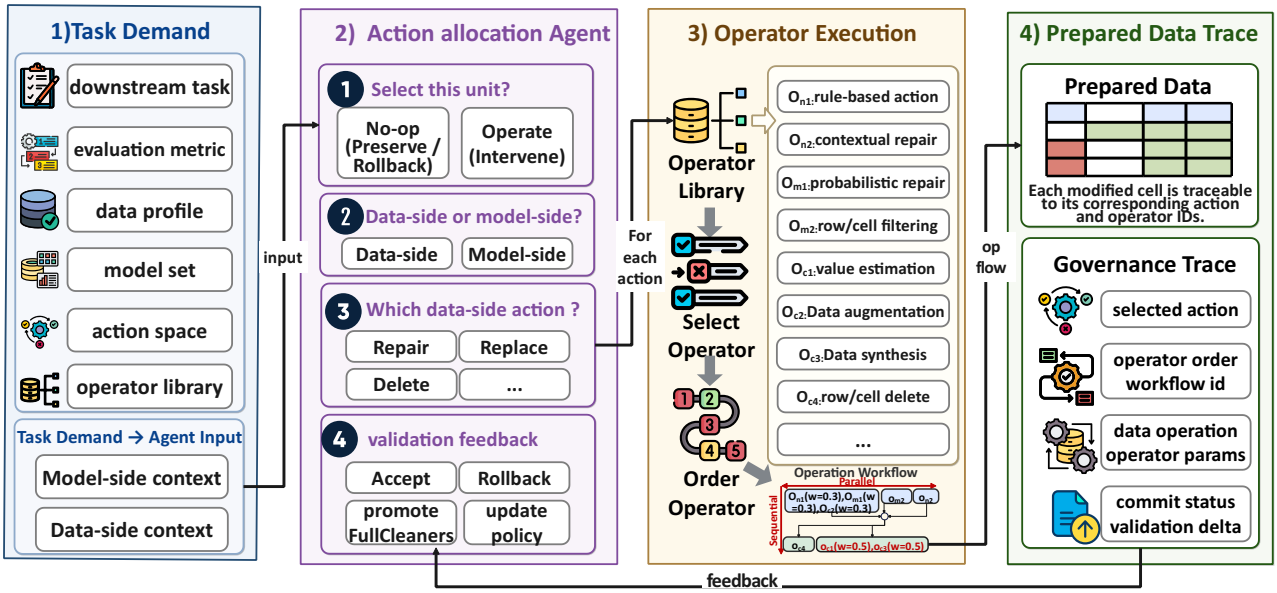

Figure 3: DemandPrep framework for the evaluated data-side loop. A task demand drives an action-allocation agent that decides, per candidate unit, whether to operate and on which side, then which action family to apply. The selected action is realized by ordering operators from a shared library, producing prepared data together with an execution trace that records the selected action, operator order, parameters, commit status, and validation delta. The downstream-validation panel and feedback edge can accept selective execution, roll back to No-op, promote FullCleaners, or update the allocation policy.

The action family is intentionally coarser than an operator workflow. Repair means that a correction is demanded. It does not yet decide which rule, dependency, or contextual cleaners should be ordered, nor which candidate value should be committed. Replace means that a lower-cost estimate is acceptable. It does not hard-code which lookup or estimator should run first. Delete means the issue scope is more harmful than useful under the task demand. No-op is a first-class decision when the detector is uncertain, the task is tolerant, or intervention would remove useful boundary cases.

The current allocator uses a hierarchical Dueling Double DQN design. Stage 1 performs cost-free triage and selects among No-op, Delete, and *enter intervention*. Stage 2 is triggered only for intervention candidates and selects between low-cost replacement and costly repair. This prevents a flat action space from degenerating into always repairing whenever a reference value is available. Each stage has its own value and advantage heads,

$$Q_j(s,a) = V_j(s) + A_j(s,a) - |\mathcal{A}_j|^{-1} \sum_{a'} A_j(s,a'),$$

where $j \in \{1,2\}$ denotes the stage. Target networks and replay buffers are maintained separately, because Stage 1 mainly learns whether intervention is needed, while Stage 2 learns how much cost the intervention should spend.

The reward combines three signals and is shaped to prevent the policy from collapsing into uniform repair. A shaping reward gives dense guidance from issue priority, feature importance, detector confidence, deletion risk, and budget slack. A periodic proxy reward measures the change in downstream validation utility after a batch of decisions, subtracting the cost of the actions taken. A terminal reward compares the final prepared candidate against the baseline data version while penalizing lost samples and spent human budget. The cost term is essential because Repair, although it may yield the strongest data-level correction, is the action whose executor realization in Step 3 can require human verification of candidate corrections, while Replace and Delete incur only computational cost. The allocator therefore learns to reserve Repair for units where the expected utility gain offsets that downstream human cost. When the remaining budget is exhausted, Repair is automatically downgraded to a zero-cost Replace or No-op according to policy constraints. This training procedure makes action allocation a task-conditioned policy rather than a fixed ranking of suspicious cells.

Algorithm 1 summarizes the training loop. The environment is built from existing data versions and operator evidence, not from newly requested manual labels. The allocator observes how candidate actions change downstream validation utility and learns a cost-aware policy over action families.

A typical decision illustrates the two-stage structure. If a candidate unit lies in a low-importance column with weak detector confidence, Stage 1 can select No-op and avoid unnecessary execution. If the unit is high-impact, Stage 1 enters intervention. Stage 2 then chooses Repair when the evidence store contains reliable correction evidence and the remaining budget allows it, or Replace when the task can tolerate a lower-cost estimate. This is why the policy is trained over action families rather than concrete cleaners. The allocator learns how much intervention the task demands before the executor orders operators and emits the concrete operation.

Algorithm 1 Zero-New-Label Action Allocation Training

Require: Demand q, dirty data D, units E, signals $\mathcal{S}$, operators Ω, validation split, episodes N
Ensure: Action allocator π_θ

Build evidence store C from cached data versions, detector outputs, and operator candidates
Select or fix downstream evaluator $M \in \mathcal{M}$ for utility U
Initialize two-stage Q networks Q_1, Q_2, targets $\bar{Q}_1, \bar{Q}_2$, replay buffers $\mathcal{B}_1, \mathcal{B}_2$
for episode = 1 to N **do**
  $D^{(0)} \leftarrow \textsc{SampleVersion}(D, C)$, $B_{rem} \leftarrow B$, and $\tau \leftarrow \emptyset$
  for each preparation unit $e_i \in E$ **do**
    $s_i \leftarrow \textsc{Encode}(D^{(t)}, e_i, q, \mathcal{S}, B_{rem}, \tau)$
    $z_i \leftarrow \textsc{EpsGreedy}(Q_1, s_i, \epsilon)$ over {No-op, Delete, *intervene*}
    if z_i = *intervene* **then**
      $a_i \leftarrow \textsc{EpsGreedy}(Q_2, s_i, \epsilon)$ over {Replace, Repair}
      if a_i = Repair and ($B_{rem} = 0$ or $\textsc{NoEvidence}(e_i, C)$) **then**
        $a_i \leftarrow \textsc{Downgrade}(\textsc{Replace}, \text{No-op}, \mathcal{P})$
      end if
    else
      $a_i \leftarrow z_i$
    end if
    $(D^{(t+1)}, c_i, \tau_i) \leftarrow \textsc{SimulateOrReplay}(D^{(t)}, e_i, a_i, \Omega_{a_i}, C)$
    $r_i \leftarrow \textsc{ShapeReward}(s_i, a_i, c_i, q)$ and store transition in $\mathcal{B}_1$ or $\mathcal{B}_2$
    Periodically add proxy reward $\Delta U(M, T, D^{(t+1)}) - \lambda c_i$
    Update Q_1, Q_2 from replay buffers and softly update $\bar{Q}_1, \bar{Q}_2$
  end for
  Add terminal reward from $U(M, T, D^{(T)})$, retention, and spent budget
  Save θ if validation utility improves, breaking ties by lower human cost
end for
return best historical allocator π_θ

Algorithm 2 Inference-Time Action-Workflow Execution

Require: Demand q, data D, units E, allocator π_θ, operator registries Ω, validator (M, U)
Ensure: Prepared data D_ρ, workflow W_ρ, provenance trace τ_ρ

$D_{sel} \leftarrow D$, $W_{sel} \leftarrow \emptyset$, and $\tau_{sel} \leftarrow \emptyset$
$D_{full} \leftarrow \textsc{ApplyAllCleaners}(D, \Omega)$ and $D_{noop} \leftarrow D$
for each preparation unit $e_i \in E$ **do**
  $s_i \leftarrow \textsc{Encode}(D_{sel}, e_i, q, \mathcal{S}, B, \tau_{sel})$
  $a_i \leftarrow \pi_\theta(e_i, s_i, q)$
  $G_{a_i} \leftarrow \textsc{BuildOperatorGraph}(\Omega_{a_i}, D_{sel}, e_i, \tau_{sel})$
  if a_i = No-op **then**
    Record pass-through decision for e_i in τ_{sel}
  else
    $\mathcal{K}_i \leftarrow \textsc{GenerateCandidates}(G_{a_i}, D_{sel}, e_i)$
    $\mathcal{O}_i \leftarrow \textsc{SearchOperations}(\mathcal{K}_i, G_{a_i}, q)$
    $(D_{sel}, W_{sel}, \tau_{sel}) \leftarrow \textsc{CommitIfImproves}(\mathcal{O}_i, D_{sel}, W_{sel}, \tau_{sel})$
    Update operator weights from accepted, rejected, and conflicting candidates
  end if
end for
$u_{rollback} \leftarrow U(M, T, D_{noop})$ and $u_{selective} \leftarrow U(M, T, D_{sel})$
$u_{fullcleaners} \leftarrow U(M, T, D_{full})$
$\rho \leftarrow \arg\max_{\rho \in \{rollback, selective, fullcleaners\}} u_\rho$
return data, workflow, and trace associated with ρ

4.3 Step 3 Inference-Time Action Execution

At inference time, the learned allocator assigns an action family to each candidate unit for a new demand. The executor realizes these decisions by drawing operators from the shared general-purpose pool of rule-based cleaners, contextual repair signals, value estimators, row filters, and pass-through, then selecting and ordering them under the constraints of the assigned action family. DemandPrep does not invent new low-level cleaners. It treats traditional data-preparation methods as composable resources and lets the action allocation decide which of them, in what order, should produce the concrete data operations. In the tabular instance, No-op is realized as traceable pass-through, Repair as an ordered repair workflow, Delete as row- or cell-scope filtering subject to retention constraints, and Replace as an ordered value-estimation workflow. This is the core of the preparation agent. The allocator decides what the task demands, and the executor decides how to compose existing preparation operators to deliver it.

Algorithm 2 gives the inference procedure. The allocator only selects action families. The executor then indexes into the corresponding operator registry, builds an action-specific operator graph, orders applicable operators, and searches executable operations under that graph.

The executor maintains a separate operator network per action family, so that Repair, Replace, Delete, and No-op each compile through their own action-indexed registry rather than through a shared cleaner pipeline. This keeps the interface extensible. A new action family is added by registering its operator network without changing the others. For concreteness we describe the Repair network in detail, since it has the most heterogeneous signals and the deepest dependency structure. The other action families follow the same interface. Each repair operator is represented as $o = [\text{detect}_o, f_o]$ from A_s to A_t, where A_s is the source scope, A_t is the target scope, detect_o estimates whether a tuple remains erroneous, and f_o generates candidate values. The executor first separates single-scope operators from multi-attribute operators. Single-scope operators can run as a parallel preprocessing layer. Multi-attribute operators form a dependency graph. If $A_{t_i} \cap A_{s_j} \neq \emptyset$, then the output of o_i can affect the evidence used by o_j. If two operators share the same target scope, their repairs can conflict. Cycles are merged into a block, independent blocks are scheduled by topological level, and operators inside a block are ordered by their current weights.

Within a repair block, the executor generates candidate operations rather than immediately editing the table. For low-quality tuples, it collects source-attribute values as predicates that locate the affected records. From high-quality tuples, cached repaired data, or external knowledge, it collects candidate target values. A candidate repair operation has the form $\ell = (\text{Repair}, p, A_t, v', o, c_\ell)$, where p is the predicate scope, v' is the proposed value, o is the originating operator, and c_ℓ is its cost. For example, if an action selects Repair for a violation involving $A_s \rightarrow A_t$, dependency and context signals may propose different values for A_t under the same predicate p. The search procedure evaluates candidates by the weighted residual detector score plus modification cost. It applies the best candidate only if it reduces the block objective, updates the data and quality scores, removes redundant candidates with the same predicate or target, and stops when no candidate further improves

the block. This gives the Repair action a concrete compilation path through operator association, block construction, operator ordering, candidate generation, repair search, and weight update. The committed output is not a cleaner name alone. It is an operation such as assigning a repaired value to a target attribute or removing a tuple under a recorded predicate.

The other action families use the same execution interface with different operator networks. Replace orders lower-cost estimators such as dependency-based lookup, neighborhood estimation, cached cleaned-value lookup, or domain-constrained imputation when exact repair is not demanded or not available. Delete materializes a row- or cell-scope filtering operation subject to retention constraints, so that one uncertain signal is not enough to remove a row. No-op records an intentional preservation decision rather than silently skipping the unit.

Once the action-allocated workflows finish, the executor holds one prepared version D_{sel}. To close the inference loop, two reference versions are materialized alongside it. $D_{noop} = D$ preserves the dirty input, and D_{full} is the output of applying the full operator pool without selective pruning. Downstream validation reads utility on a held-out split for all three and commits the version that maximizes it. This is not a comparison against external baselines but a diagnostic over the data side. When D_{noop} wins, data-side editing offers no measurable utility for the current task, indicating limited preparation headroom. When D_{full} wins, running every available operator outperforms cost-aware allocation, signaling that the raw data quality is the bottleneck and the priority is broad cleaning rather than refining the policy. When D_{sel} wins, the demand-driven allocation justifies its cost. The same three outcomes are reported per dataset in Section 5, where they serve as a readout of how much preparation the task actually warrants.

4.4 Step 4 Provenance and Feedback

The fourth step closes the loop by making every committed data operation traceable to its source. The trace records which action family was selected for the unit, which operator in the executor's network generated the candidate value, and which validation signal accepted or rejected it. This is what makes DemandPrep interpretable as an agent rather than as an opaque policy. Each prepared value can be explained back to an allocator decision and an executor compilation path, and the same record carries feedback into Steps 2 and 3 so that the next inference run is cheaper and more accurate.

The trace stores the preparation-unit identifier, selected action family, stage decisions, ordered operator workflow, operation identifier ℓ, source scope, target scope, predicate, dirty value, candidate value, final value, input data version, output data version, execution cost, and validation delta. For a Repair action, provenance is recorded at four granularities. An action record explains why the unit entered Repair rather than No-op, Delete, or Replace. A workflow record stores the ordered operators and the dependency block that produced the candidates. An operator record identifies the source-target dependency $A_s \rightarrow A_t$ and the signal that generated a candidate. An operation record stores the concrete ℓ together with its status, which can be committed, pruned as redundant, rejected for not reducing the block objective, or overwritten by a higher-weight candidate. In the current implementation this information is materialized as a decision log, a deferred repair plan, a repair-source log, an operator execution trace, and a run-level metric file.

Feedback derived from the trace flows in two directions. Feedback to Step 2 updates the allocator. Observed utility deltas, cost accumulation, and the prevalence of rollback or FullCleaners outcomes refine the Q-value estimates and the shaping priors that govern action preference, the same priors that bias Repair, Replace, Delete, and No-op under issue priority, feature importance, deletion risk, and budget slack. The policy becomes more task-conditioned over runs without any new manual labels. Feedback to Step 3 updates the operator weights used during candidate search. Operators whose candidates survive conflict resolution and improve the block or downstream objective gain priority in later repair blocks, and operators that produce overwritten, conflicting, or harmful candidates are down-weighted or skipped. This concentrates the executor's search on operators that have paid off on similar units, so the candidate set explored at inference shrinks toward operators with empirical support rather than expanding with the registry size.

These two reactions vindicate the agentic separation between allocator and executor. The same trace explains the agent's behavior cell-by-cell, supports audit and rollback, and feeds back as cost-aware policy updates and operator-weight updates without conflating the two layers. The agent becomes more efficient and more accurate on the next demand by reading its own history, not by being retrained from scratch.

4.5 Execution Decisions Exposed by the Loop

The four steps expose four design decisions worth stating before the experiments. Allocation and execution are kept separate so that specialized cleaning operators remain reusable resources rather than committing the agent to a single cleaner's objective. No-op and FullCleaners are modeled as positive regimes rather than absent decisions, defining the conservative and intensive ends of the same action-allocation space. Action families are the extension boundary, so a new family enters the loop by registering its units, signals, admissible operations, costs, validation protocol, and provenance schema, without redesigning the existing operators. Preparation is therefore evaluated by downstream utility, execution cost, traceability, and the ability to recover from harmful interventions rather than by cell-level repair accuracy alone, which motivates the native-error, injected-error, and diagnostic-control analyses in Section 5.

5 EXPERIMENTAL EVALUATION

The evaluation tests the central claim of demand-driven preparation. The claim is that allocating task-aware action families before any operator is run yields better downstream utility than running a fixed cleaning pipeline, and that the allocator correctly identifies tasks for which no operation, selective intervention, or all-cleaner execution is the right answer. We design the experiments to challenge this claim from three angles. Native-error datasets test whether the agent competes with strong cleaning systems when the errors are

Table 1: Summary of datasets and downstream tasks. C and R denote classification and regression.

Data	Rows	Col.	C%	R%	Err.	Task
Beers	2.4K	10	13.94	100.00	M/F/V	C-style
Flights	1.1K	7	15.45	56.42	M/F/V	C-delay
Hospitals	1.0K	19	2.68	40.70	T/V	C-measure
Rayyan	642	11	10.25	84.58	M/T/F/V	C-lang.
Tax-10K	10K	15	0.11	1.65	T/F/V	R-rate

real and heterogeneous. Mixed-error injection across eight perturbation rates stresses the policy under controlled but heterogeneous corruption, exposing how the allocator shifts among Repair, Replace, Delete, and No-op as the error mix changes. Diagnostic controls such as rollback to the dirty input, FullCleaners, oracle deletion, and ground-truth repair anchor each result against conservative, intensive, and clean-reference references, so reported gains can be read against what the data side could in principle deliver.

We organize the analysis around three questions. **RQ1** asks whether the final execution output competes with fixed cleaning methods on native errors. **RQ2** asks whether the same control loop remains useful under controlled mixed-error injection. **RQ3** asks when the agent should preserve, selectively intervene, or commit the all-cleaner regime, and whether action traces explain these regimes.

5.1 Experimental Setup

We evaluate DemandPrep on five datasets that together cover the error-type and task-type diversity used in the cleaning literature, namely Beers, Flights, Hospitals, Rayyan, and Tax-10K. The first four are standard cleaning benchmarks with heterogeneous native errors (missing values, typos/outliers, format inconsistencies, and attribute-dependency violations), and Tax-10K adds a large-scale regression target. Each dataset is reported under two regimes, namely native errors as released and the same table under mixed-error injection. Table 1 lists the task settings, where C% and R% denote cell-level and record-level error rates on the task rows and M, T, F, V abbreviate the four error families above. The Tax case is evaluated on a clustered 10K subset rather than a random sample, so records sharing key-like structure remain together. The Hospitals target is measure-code classification, which avoids the saturated ownership-label split and gives data-side actions a measurable downstream effect. Downstream utility is measured by accuracy for classification and R^2 for Tax regression, with higher values being better in both cases. Each table is split into a training partition for allocator learning and a held-out partition for downstream validation. The reported experiments use a singleton downstream model per dataset, so the candidate-model interface in Section 3 is exercised as fixed-model task demand rather than as model selection.

The comparison spans three groups, namely fixed cleaning baselines, the three DemandPrep execution regimes, and oracle diagnostic references. For each fixed cleaner, we train and evaluate the same downstream model directly on the cleaned table that the cleaner produces, so the protocol compares data preparation outputs rather than internal cell-level objectives. The three DemandPrep regimes correspond to the three diagnostic outcomes introduced in Section 4. No-op preserves the dirty input and is selected when validation indicates limited measurable data-side gain. FullCleaners commits the output of applying all available cleaners. The selective branch commits only the operations emitted by action-allocated workflows. Reporting them separately is not a comparison of DemandPrep against itself but a readout of which data-side condition each task is in. The oracle diagnostics use clean-reference information that is unavailable at deployment time. Oracle deletion removes rows containing known true errors and tests whether aggressive authenticity improvement helps after sample loss. Full ground-truth repair replaces all known erroneous cells with clean values and tests whether maximum cell-level correctness is aligned with task utility. The reported native setting contains 45 baseline, regime, and reference evaluations. The reported injected setting contains 40 preparation cases and 360 such evaluations.

Table 2 organizes the evaluated methods by their decision boundary before downstream evaluation. The distinction matters because the methods differ in both available evidence and controllable actions. Fixed cleaners output a cleaned table under a prescribed repair logic, while DemandPrep separates demand-conditioned action allocation from general-purpose operator orchestration. The upstream allocator decides whether and how a unit should be operated on, and the executor invokes reusable cleaners or filters under the selected action family before committing the prepared data version.

For injected data, we follow the mixed-error construction used in data cleaning benchmarks. For each dataset, non-key attributes and applicable constraints are independently perturbed with typographical or outlier edits, missing values, and rule-violation values. We use eight cell- or rule-level injection rates between 0.25% and 2.0%. Dataset names with * denote arithmetic means over these eight injected settings, and no dataset is reported at a single favorable injection level. Since a tuple may contain several attributes and participate in multiple constraints, record-level error rates can be much higher than the nominal cell- or rule-level rate. Table 3 reports the measured cell-level and record-level intensity of the injected benchmark tables. The non-Tax rows use the corresponding injected benchmark table before downstream target filtering, so the table characterizes the corruption source rather than the final trainable rows. Across the 40 reported tables, the average cell-level error rate is 1.58%, the average record-level error rate is 17.78%, and the hardest setting reaches 50.90% erroneous rows. The resulting 40 cases require the allocator to avoid unnecessary changes under mild corruption and select demanded interventions when mixed errors affect downstream utility.

5.2 Performance on Native Errors

Table 4 compares the final DemandPrep execution output with fixed cleaning outputs. The DemandPrep column reports the validation-selected regime among conservative No-op rollback, selective action allocation, and FullCleaners execution. Table 5 separates these regimes. The values are downstream task utilities after training on each prepared table. Accuracy is used for classification and

Table 2: Taxonomy of evaluated methods by decision boundary. The decision boundary describes the layer controlled by each method. Evidence lists the information used for this decision, and Action summarizes the expressed operation or regime.

Method	Decision Boundary	Evidence	Action	Oracle
Baran [20]	cell repair	transferred cell context	repair	No
BigDansing [16]	rule-plan execution	quality rules and UDFs	repair/filter	No
Holistic [3]	repair-set selection	denial constraints	repair	No
HoloClean [28]	probabilistic repair	constraints and co-occurrence	repair	No
Horizon [30]	dependency-plan execution	discovered dependencies	repair	No
DemandPrep	action allocation	task demand and general-purpose operators	multi-action	No
No-op	data-version commit	dirty input only	no-op	No
FullCleaners	all-cleaner execution	all available cleaners	full repair	No
OracleDel	true-error row deletion	clean-reference error rows	delete	Yes
GTRepair	true-cell repair	clean-reference erroneous cells	repair	Yes

Table 3: Cell-level and record-level intensity of the injected benchmark tables used by the reported settings. Each row summarizes the eight injected settings used for the aggregate injected result of that dataset. Tax is measured on the clustered 10K subset.

Data	Ver.	Avg. C%	Range C%	Avg. R%	Range R%
Beers	8	1.30	0.62 to 1.98	12.50	6.18 to 17.93
Flights	8	1.89	0.44 to 3.35	11.14	3.03 to 19.23
Hospitals	8	1.95	0.41 to 3.47	31.00	8.30 to 50.90
Rayyan	8	0.88	0.42 to 1.34	9.72	4.90 to 14.20
Tax-10K	8	1.87	0.45 to 3.19	24.55	6.40 to 40.15
Mean	40	1.58	0.41 to 3.47	17.78	3.03 to 50.90

Table 4: Comparison with fixed baseline cleaning methods. The DemandPrep column reports the final execution output selected among No-op rollback, selective allocation, and FullCleaners execution. When a fixed cleaner does not produce an applicable output for a row, we report the corresponding No-op utility as a conservative fallback. Injected rows follow the aggregation protocol in Section 5. Red and blue mark the best and second-best downstream utilities in each row, with ties included.

Data	DemandPrep	Baran	BigD.	Hol.	HoloC.	Horiz.
Beers	**0.1929**	**0.1807**	0.1607	0.1526	0.1511	0.1540
Flights	**0.9854**	**0.9615**	0.9297	0.9312	0.2454	0.7970
Hospitals	**0.8665**	0.8392	0.8075	0.8002	**0.8498**	0.8457
Rayyan	**0.1538**	0.1183	0.1205	**0.1817**	0.0685	0.0825
Tax-10K	**0.5756**	0.5753	**0.5762**	0.5751	0.4720	0.3294
Beers*	**0.1821**	0.1728	0.1720	**0.1736**	0.1724	0.1718
Flights*	**1.0000**	0.9357	0.9910	**0.9923**	0.2706	0.9099
Hospitals*	0.8571	0.8532	**0.8583**	**0.8655**	0.8460	0.7858
Rayyan*	**0.1700**	0.1489	0.1214	**0.1889**	0.0395	0.1377
Tax-10K*	**0.3883**	**0.3414**	-6.5793	-1.6333	-5.6384	0.1318

R^2 for Tax regression, and higher is better. Across the table, DemandPrep has the highest mean utility among the named fixed cleaners (0.5372) and is ranked first or second on almost every row, with Hospitals* as the only third-place case. The best and second-best values in each row are marked in **red** and **blue**, respectively, with ties included.

On native errors, DemandPrep is best among fixed cleaned outputs on Beers, Flights, and Hospitals, and second on Rayyan and Tax-10K. The gains over the strongest fixed cleaner are 0.0122 on Beers, 0.0239 on Flights, and 0.0167 on Hospitals, corresponding to relative improvements of 6.8%, 2.5%, and 2.0%. The two second-place cases are also informative. DemandPrep is only 0.0006 below BigDansing on Tax-10K, but 0.0279 below Holistic on Rayyan, where a fixed constraint-oriented output is better aligned with the language-classification target. On Flights, fixed cleaning outputs vary widely. Baran remains close to DemandPrep, but HoloClean and Horizon reduce downstream utility by 0.7400 and 0.1884 relative to DemandPrep. A high-quality cell repair method is not automatically aligned with the task metric.

Finding 1. Native errors require demand-driven action allocation, and intervention intensity itself is a decision variable. The native rows make two claims at once. First, no single fixed cleaner dominates across tasks. The strongest fixed cleaner changes from Baran on Beers to Holistic on Rayyan to BigDansing on Tax-10K, and DemandPrep is the only method that is first or second on every native row. This is the empirical case for demand-driven allocation, since a fixed pipeline cannot be the right answer for tasks that disagree about what counts as a useful intervention. Second, even when the right action family is chosen, the right intensity is task-specific. Table 5 shows the selective policy improving over FullCleaners by 0.1194 on Flights, 0.0313 on Hospitals, and 0.0267 on Rayyan, while FullCleaners is best or tied on Beers and Tax-10K. DemandPrep also matches No-op on Flights, Hospitals, and Rayyan when validation finds that committing edits yields no measurable utility. Running every available cleaner is therefore neither uniformly safer nor uniformly more useful than running a calibrated subset, and preservation is a positive outcome rather than a null result. Both observations argue that preparation should be controlled at the action-family level by a policy that selects intensity per task, not at the cleaner level by a fixed objective.

5.3 Performance on Injected Errors

The injected aggregate rows in Table 4 use the perturbation protocol and record-level intensities reported in Table 3. They test whether the action allocator remains useful when the error distribution is controlled but heterogeneous. DemandPrep is best on Beers*, Flights*, and Tax-10K*, with relative gains of 4.9%, 0.8%, and 13.7% over the strongest fixed cleaner on those rows. On Rayyan*, the final system promotes FullCleaners and becomes second, 0.0189 below Holistic. On Hospitals*, DemandPrep is 0.0084 below Holistic and 0.0012 below BigDansing, showing a concrete boundary where a fixed cleaner remains slightly better under the complete all-rate average. In the Tax-10K* row, several fixed cleaned outputs degrade the regression utility sharply, while DemandPrep preserves a positive utility close to the stable references.

The injected rows also show the boundary of the current implementation. Holistic is strongest on Hospitals* and Rayyan*, and FullCleaners is strongest among the reported DemandPrep regimes on Rayyan* in Table 5. When one fixed cleaning workflow or the complete all-cleaner workflow is consistently aligned with a task and error distribution, the executor can reuse it without forcing selective pruning.

Finding 2. The same allocator transfers across error mixes by re-weighting action families, and the trace makes every shift inspectable. The injected rows test whether the policy generalizes across heterogeneous corruption without retraining or retuning. Across the 40 injected settings, the selective branch reaches or exceeds the best non-oracle reference in 21 cases, improves over No-op in 17, and triggers rollback in 18. The action traces explain how a single learned policy produces this spread. On injected Beers, the allocator settles on a repair/replace-heavy mix at 44.7% Repair and 48.6% Replace. On injected Rayyan, Replace dominates at 60.3%. On injected Tax, the mix shifts to 37.1% No-op and 36.5% Delete. The same architecture is therefore producing four qualitatively different behaviors, namely repair-oriented, replacement-oriented, mixed, and preservation/deletion-oriented, all driven only by downstream feedback. This is exactly the behavior that the action-family abstraction in Section 4 predicts, and the trace makes the predictions auditable. Every shift in the allocation histogram can be traced back to the per-unit state features (detector confidence, feature importance, budget slack) that the allocator actually observed. A fixed cleaning pipeline cannot match this because it commits to a single intervention pattern before seeing the error distribution. DemandPrep can because it commits to nothing below the action-family level until validation reads back.

5.4 Action Allocation and Feedback Analysis

Table 5 reports the three DemandPrep regimes side by side with the two oracle diagnostics. The selective column is the action-allocated output before any validation rollback or FullCleaners promotion, so the row-by-row comparison shows where validation later moves the system. When OracleDel has no applicable deletion output for a row, the table reports the No-op utility.

Finding 3. The three regimes form a per-task diagnostic of data-side headroom. Reporting No-op, selective, and FullCleaners as distinct outcomes is not an ablation over regime variants but a readout of how much preparation the data side can deliver for the current task. Across the ten native/injected aggregate rows in Table 5, DemandPrep-Sel is never below No-op, improves over

Table 5: DemandPrep operating regimes and diagnostic comparison. No-op is the conservative rollback regime, FullCleaners is the all-cleaner execution regime, and Sel. is the selective action-allocation branch after validation rollback to No-op but before promotion to FullCleaners. OracleDel and GTRepair use clean-reference information and are included only to interpret action choices. Red and blue mark the best and second-best values in each row.

Data	OracleDel	FullCleaners	Sel.	GTRepair	No-op
Beers	0.1511	**0.1929**	**0.1929**	**0.1901**	0.1511
Flights	0.9391	0.8660	**0.9854**	**1.0000**	**0.9854**
Hospitals	0.7907	0.8352	**0.8665**	**0.8456**	**0.8665**
Rayyan	0.1282	0.1271	**0.1538**	**0.1540**	**0.1538**
Tax-10K	**0.5755**	**0.5756**	**0.5756**	**0.5755**	0.5751
Beers*	0.1724	**0.1790**	**0.1821**	0.1773	0.1724
Flights*	**0.9968**	0.9027	**1.0000**	**1.0000**	**1.0000**
Hospitals*	0.8125	0.8413	**0.8571**	0.8510	**0.8564**
Rayyan*	0.1104	**0.1700**	0.1278	**0.1414**	0.1241
Tax-10K*	0.3878	0.3861	**0.3883**	0.3881	**0.3882**

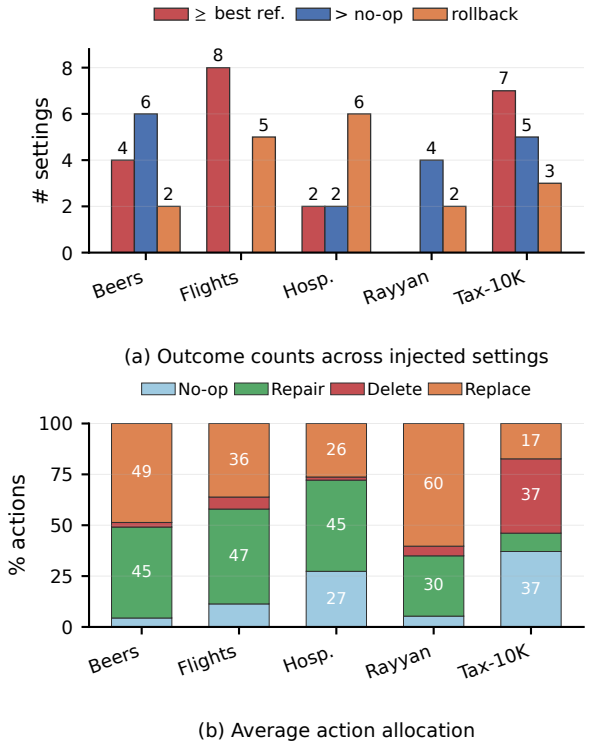

Figure 4: Injected-case behavior of the selective branch across the five reported datasets. Panel (a) counts, out of eight injected settings per dataset, when it reaches or exceeds the best non-oracle reference, improves over No-op, or rolls back a harmful candidate. Panel (b) reports the average action allocation from the decision logs over the same eight settings.

it on six rows, and ties it on four. It also wins or ties against FullCleaners on nine rows, with an average utility gain of 0.0254. The single counter-row, Rayyan*, is the case where applying all available cleaners beats cost-aware allocation. Read regime by regime, the table maps each task to a specific data-side condition. A tie with No-op (Flights, Hospitals, Rayyan natively) means validation chose preservation because data-side editing did not yield measurable utility, indicating that the preparation headroom on those tasks is small even though the data are visibly dirty. A FullCleaners win (Rayyan*) means the dataset is degraded enough that committing every available cleaner outperforms cost-aware allocation, signaling that broad cleaning, rather than policy refinement, is the immediate priority. A selective win (Beers*, Hospitals*, Tax-10K*) means demand-driven allocation justifies its cost over both conservative preservation and intensive all-cleaner execution. GTRepair is not an absolute upper bound either. The final DemandPrep output matches or exceeds GTRepair on most rows, including Hospitals and Hospitals*, where full repair is worse than selective preparation. Feedback is therefore usable rather than terminal. DemandPrep treats every prepared table as a candidate version that can update future action preferences and operator weights.

The action logs report the allocation behavior behind these outcomes. In the native runs, the candidate policy assigns 57.8% of actions to Replace, 21.2% to Repair, 13.0% to Delete, and 7.9% to No-op. This aggregate hides strong dataset dependence. Native Beers and Flights are dominated by Replace (95.2% and 61.2%). Native Hospitals is repair-heavy (67.3% Repair), while native Rayyan splits between No-op and Delete (47.6% and 52.4%). Native Tax is almost entirely deletion in the candidate trace (99.6% Delete), which matches the intuition that a sparse set of structurally problematic records can be more harmful than individual cell noise for the regression target. In the injected setting, the action mix shifts to 40.4% Repair, 34.8% Replace, 19.7% No-op, and 5.1% Delete. The trace evidence indicates that action intensity is data- and task-conditioned. The same framework can become conservative, repair-oriented, replacement-oriented, or deletion-oriented depending on downstream feedback.

Figure 4 summarizes the same behavior across the injected settings, and the mixes are internally consistent when read against dataset scale and task type. Beers is the smallest table (2.4K rows), where dropping a row is expensive relative to the available sample, so the policy stays away from Delete and prefers low-cost replacement that retains the unit. Hospitals is a 1K classification target with rule-violation noise, where repair preserves both row count and label structure. Tax-10K is the largest table with a sparse population of structurally problematic records, where deletion is cheap on a per-sample basis and pays off more than per-cell repair for the regression target. Together with the diagnostic readout in Finding 3, this gives a per-dataset picture of which action family is doing the work and which data-side condition each task is actually in.

6 CONCLUSION

DemandPrep frames data preparation as a task-conditioned decision problem rather than as an upstream pipeline that produces a cleaner table. An allocator decides, per unit, what intervention the task warrants among preservation, repair, deletion, and low-cost replacement. An executor realizes that decision by selecting and ordering reusable preparation operators from the shared pool, and downstream validation reads back whether the result improves the task and feeds the answer to both layers. The criterion for changing a value therefore shifts from "does this match a cleaning objective" to "does this improve the downstream task." This is the data-centric AI commitment taken at the system level. If data quality matters because it shapes model behavior, then the layer that decides how data is changed has to read model behavior back as its objective.

The native and injected experiments substantiate this framing on three counts, reported as Findings 1 through 3. These are the necessity of demand-driven allocation, the inspectable transfer of one policy across error mixes, and the diagnostic readout of the three regimes. The implemented scope is tabular data preparation with a fixed downstream model per dataset and four action families. The paper does not claim a natural-language planning interface, a benchmark over non-cleaning action families, or an evaluated model-side adaptation policy. Relative to the lines surveyed in Section 2, DemandPrep answers each of them with a different concrete capability. It is transferable against fixed cleaners, because one allocator re-weights action families across datasets and error mixes without retraining (Finding 2). It is demand-conditioned against workflow systems, because whether to intervene and at what intensity is decided before any operator runs rather than after repair is already committed (Findings 1 and 3). And it is extensible against task-aware preparation, because existing cleaners, estimators, and filters enter the loop as registered operators under shared action families rather than being redesigned for each task. The contribution to data-centric AI is therefore a control layer above the existing operator catalogue rather than a new operator. A new action family such as data augmentation, feature engineering, or model-side adaptation joins by registering its units, admissible operations, costs, validation protocol, and provenance schema under the same allocator-executor loop. Under this view, model-centric and data-centric improvements are not opposing strategies but action families competing on the same downstream-utility criterion, with the allocator deciding which is justified for the current demand.

ACKNOWLEDGMENTS

This work is supported by National Science and Technology Major Project (2025ZD1607102), the National Natural Science Foundation of China (NSFC) (62572151, 62232005), the National Key Laboratory of Data Space Technology and System (QZQC2026042), the Natural Science Foundation Project of Heilongjiang Province of China (YQ2024F005), and the Heilongjiang Provincial Young Talents Support Program for Science and Technology Innovation (2025QNTJ001).

REFERENCES

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting Data Errors: Where are we and what needs to be done? *Proc. VLDB Endow.* 9, 12 (2016), 993–1004. https://doi.org/10.14778/2994509.2994518

[2] Chengliang Chai, Jiabin Liu, Nan Tang, Ju Fan, Dongjing Miao, Jiayi Wang, Yuyu Luo, and Guoliang Li. 2023. GoodCore: Data-effective and Data-efficient Machine Learning through Coreset Selection over Incomplete Data. *Proc. ACM Manag. Data* 1, 2 (2023), 157:1–157:27. https://doi.org/10.1145/3589302

[3] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, Brisbane, Australia, 458–469. https://doi.org/10.1109/ICDE.2013.6544847

[4] Xiaoou Ding, Genglong Li, Yafeng Tang, Chen Liang, Tianren Yu, Muyun Zhou, Yida Liu, Zekai Qian, Zixuan Song, and Hongzhi Wang. 2025. A Survey on Data Asset Value Change Estimation and Appreciation with Data Governance. *Big Data Min. Anal.* 8, 6 (2025), 1369–1387. https://doi.org/10.26599/BDMA.2025.9020034

[5] Xiaoou Ding, Yanshuo Liu, Zhounan Chen, Hongzhi Wang, Chen Wang, and Jianmin Wang. 2025. TARImpute: Task-Aware Auto-Recommender System for Missing Value Imputation Algorithms with Clustering Case Studies. *Proc. VLDB Endow.* 18, 12 (2025), 5343–5346. https://doi.org/10.14778/3750601.3750667

[6] Xiaoou Ding, Zekai Qian, Hongzhi Wang, Siying Chen, Yafeng Tang, Hongbin Su, Huan Hu, and Chen Wang. 2025. UniClean: A Scalable Data Cleaning Solution for Mixed Errors based on Unified Cleaners and Optimized Cleaning Workflow. *Proc. VLDB Endow.* 18, 11 (2025), 4117–4130. https://doi.org/10.14778/3749646.3749681

[7] Xiaoou Ding, Zekai Qian, Hongzhi Wang, Zhe Sun, Siying Chen, Hongbin Su, and Huan Hu. 2025. UniClean: A Multi-Signal Fusion Pipeline for Optimizing Data Cleaning Workflow. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, Hong Kong, China, 4600–4603. https://doi.org/10.1109/ICDE65448.2025.00362

[8] Xiaoou Ding, Yichen Song, Hongzhi Wang, Donghua Yang, Chen Wang, and Jianmin Wang. 2024. Clean4TSDB: A Data Cleaning Tool for Time Series Databases. *Proc. VLDB Endow.* 17, 12 (2024), 4377–4380. https://doi.org/10.14778/3685800.3685879

[9] Xiaoou Ding, Hongzhi Wang, Genglong Li, Haoxuan Li, Yingze Li, and Yida Liu. 2022. IoT data cleaning techniques: A survey. *Intell. Converged Networks* 3, 4 (2022), 325–339. https://doi.org/10.23919/ICN.2022.0026

[10] Eduardo Souza dos Reis, Mohamed Abdelaal, and Carsten Binnig. 2024. Generalizable Data Cleaning of Tabular Data in Latent Space. *Proc. VLDB Endow.* 17, 13 (2024), 4786–4798. https://doi.org/10.14778/3704965.3704983

[11] Mohamed Y. Eltabakh, Zan Ahmad Naeem, Mohammad Shahmeer Ahmad, Mourad Ouzzani, and Nan Tang. 2024. RetClean: Retrieval-Based Tabular Data Cleaning Using LLMs and Data Lakes. *Proc. VLDB Endow.* 17, 12 (2024), 4421–4424. https://doi.org/10.14778/3685800.3685890

[12] Haotian Gao, Shaofeng Cai, Tien Tuan Anh Dinh, Zhiyong Huang, and Beng Chin Ooi. 2024. CtxPipe: Context-aware Data Preparation Pipeline Construction for Machine Learning. *Proc. ACM Manag. Data* 2, 6 (2024), 231:1–231:27. https://doi.org/10.1145/3698831

[13] Johannes Jakubik, Michael Vössing, Niklas Kühl, Jannis Walk, and Gerhard Satzger. 2024. Data-Centric Artificial Intelligence. *Bus. Inf. Syst. Eng.* 66, 4 (2024), 507–515. https://doi.org/10.1007/S12599-024-00857-8

[14] Mohammad Hossein Jarrahi, Ali Memariani, and Shion Guha. 2023. The Principles of Data-Centric AI. *Commun. ACM* 66, 8 (2023), 84–92. https://doi.org/10.1145/3571724

[15] Bojan Karlas, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. 2020. Nearest Neighbor Classifiers over Incomplete Information: From Certain Answers to Certain Predictions. *Proc. VLDB Endow.* 14, 3 (2020), 255–267. https://doi.org/10.5555/3430915.3442426

[16] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. BigDansing: A System for Big Data Cleansing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, Melbourne, Victoria, Australia, 1215–1230. https://doi.org/10.1145/2723372.2747646

[17] Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, and Eugene Wu. 2017. BoostClean: Automated Error Detection and Repair for Machine Learning. arXiv:1711.01299 http://arxiv.org/abs/1711.01299

[18] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. 2016. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *Proc. VLDB Endow.* 9, 12 (2016), 948–959. https://doi.org/10.14778/2994509.2994514

[19] Andreas Loizou and Dimitrios Tsoumakos. 2025. Chunked Data Shapley: A Scalable Dataset Quality Assessment for Machine Learning. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management, CIKM 2025, Seoul, Republic of Korea, November 10-14, 2025*, Meeyoung Cha, Chanyoung Park, Noseong Park, Carl Yang, Senjuti Basu Roy, Jessie Li, Jaap Kamps, Kijung Shin, Bryan Hooi, and Lifang He (Eds.). ACM, Seoul, Republic of Korea, 1958–1967. https://doi.org/10.1145/3746252.3761305

[20] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning. *Proc. VLDB Endow.* 13, 11 (2020), 1948–1961. http://www.vldb.org/pvldb/vol13/p1948-mahdavi.pdf

[21] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Raha: A Configuration-Free Error Detection System. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, Amsterdam, The Netherlands, 865–882. https://doi.org/10.1145/3299869.3324956

[22] Sedir Mohammed, Felix Naumann, and Hazar Harmouch. 2025. Step-by-Step Data Cleaning Recommendations to Improve ML Prediction Accuracy. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*, Alkis Simitsis, Bettina Kemme, Anna Queralt, Oscar Romero, and Petar Jovanovic (Eds.). OpenProceedings.org, Barcelona, Spain, 542–554. https://doi.org/10.48786/EDBT.2025.43

[23] Wei Ni, Xiaoye Miao, Xiangyu Zhao, Yangyang Wu, Shuwei Liang, and Jianwei Yin. 2024. Automatic Data Repair: Are We Ready to Deploy? *Proc. VLDB Endow.* 17, 10 (2024), 2617–2630. https://doi.org/10.14778/3675034.3675051

[24] Wei Ni, Kaihang Zhang, Xiaoye Miao, Xiangyu Zhao, Yangyang Wu, Yaoshu Wang, and Jianwei Yin. 2025. ZeroED: Hybrid Zero-Shot Error Detection Through Large Language Model Reasoning. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, Hong Kong, China, 3126–3139. https://doi.org/10.1109/ICDE65448.2025.00234

[25] Alireza Pirhadi, Mohammad Hossein Moslemi, Alexander Cloninger, Mostafa Milani, and Babak Salimi. 2024. OTClean: Data Cleaning for Conditional Independence Violations using Optimal Transport. *Proc. ACM Manag. Data* 2, 3 (2024), 160. https://doi.org/10.1145/3654963

[26] Zekai Qian, Xiaoou Ding, Chen Wang, and Hongzhi Wang. 2025. DemandClean: A Multi-Objective Learning Framework for Balancing Model Tolerance to Data Authenticity and Diversity. *Proc. VLDB Endow.* 18, 12 (2025), 5339–5342. https://doi.org/10.14778/3750601.3750666

[27] Jianbin Qin, Sifan Huang, Yaoshu Wang, Jing Zhu, Yifan Zhang, Yukai Miao, Rui Mao, Makoto Onizuka, and Chuan Xiao. 2024. BClean: A Bayesian Data Cleaning System. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, Utrecht, The Netherlands, 3407–3420. https://doi.org/10.1109/ICDE60146.2024.00263

[28] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (2017), 1190–1201. https://doi.org/10.14778/3137628.3137631

[29] Cédric Renggli, Luka Rimanic, Nezihe Merve Gürel, Bojan Karlas, Wentao Wu, and Ce Zhang. 2021. A Data Quality-Driven View of MLOps. *IEEE Data Eng. Bull.* 44, 1 (2021), 11–23. http://sites.computer.org/debull/A21mar/p11.pdf

[30] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. 2021. Horizon: Scalable Dependency-driven Data Cleaning. *Proc. VLDB Endow.* 14, 11 (2021), 2546–2554. https://doi.org/10.14778/3476249.3476301

[31] Shafaq Siddiqi, Roman Kern, and Matthias Boehm. 2023. SAGA: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications. *Proc. ACM Manag. Data* 1, 3 (2023), 218:1–218:26. https://doi.org/10.1145/3617338

[32] Yafeng Tang, Xiaoou Ding, Jianzhuo Du, Zishuo Yan, Zhuang Ma, Zheng Liang, Zekai Qian, and Hongzhi Wang. 2025. Exploring the Heterogeneity of Tabular Data: A Diversity-aware Data Generator via LLMs. arXiv:2512.21915 [cs.LG] https://arxiv.org/abs/2512.21915

[33] Yafeng Tang, Zheng Liang, Hongzhi Wang, Xiaoou Ding, Tianyu Mu, and Huan Hu. 2025. Description-Similarity Rules: Towards Flexible Feature Engineering for Entity Matching. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 1–15. https://doi.org/10.1109/ICDE65448.2025.00008

[34] Jianwei Wang, Ying Zhang, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2024. Missing Data Imputation with Uncertainty-Driven Network. *Proc. ACM Manag. Data* 2, 3 (2024), 117. https://doi.org/10.1145/3654920

[35] Mengyi Yan, Yaoshu Wang, Yue Wang, Xiaoye Miao, and Jianxin Li. 2024. GIDCL: A Graph-Enhanced Interpretable Data Cleaning Framework with Large Language Models. *Proc. ACM Manag. Data* 2, 6 (2024), 236:1–236:29. https://doi.org/10.1145/3698811

[36] Sangdoo Yun, Dongyoon Han, Sanghyuk Chun, Seong Joon Oh, Youngjoon Yoo, and Junsuk Choe. 2019. CutMix: Regularization Strategy to Train Strong Classifiers With Localizable Features. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, Seoul, Korea, 6022–6031. https://doi.org/10.1109/ICCV.2019.00612

[37] Chi Zhang, Huaping Zhong, Kuan Zhang, Chengliang Chai, Rui Wang, Xinlin Zhuang, Tianyi Bai, Jiantao Qiu, Lei Cao, Ju Fan, Ye Yuan, Guoren Wang, and Conghui He. 2025. Harnessing Diversity for Important Data Selection in Pretraining Large Language Models. https://openreview.net/forum?id=bMC1t7eLRc

[38] Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. 2018. mixup: Beyond Empirical Risk Minimization. https://openreview.net/forum?id=r1Ddp1-Rb

[39] Jiaxuan Zhang, Haitao Yuan, Jianing Si, Nan Jiang, and Shangguang Wang. 2025. Think Twice Before Imputation: Optimizing Data Imputation Order for Machine Learning. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, Hong Kong, China, 1362–1374. https://doi.org/10.1109/ICDE65448.2025.00106