

TabAutoSyn: A Generator-Agnostic Pipeline for Deployment-Ready Synthetic Tabular Data

Dmitrii Kuznechenko, Kirill Rapatskikh, Anna Kalyuzhnaya, Alexander Boukhanovsky

AI Institute, ITMO University, Saint Petersburg, Russia
dakuznechenko@itmo.ru, kirapatskikh@itmo.ru,
anna.kalyuzhnaya@itmo.ru, avbukhanovskii@itmo.ru

Abstract

Synthetic tabular data lets organisations share, rebalance, and stress-test sensitive in-house tables—but only if the result is actually fit for the downstream task. Yet a table that matches the real distribution on aggregate statistics can still be unfit for its downstream task: it loses the rare cases that matter, is not optimised for the end task, and silently breaks the deterministic relations that domain data must obey. We present TABAUTOSYN¹, an end-to-end, generator-agnostic pipeline that turns any base generator into deployment-ready synthetic data through four automated stages: generate, verify, augment, and curate. A multi-agent LLM module discovers functional dependencies and domain constraints from the real table and repairs or removes violating rows; a divergence-minimising stage rebuilds under-represented tails; and a curation stage whose every committed change is verified not to decrease a chosen downstream score, so the curated output is guaranteed at least as good as the input under fixed seeds. Across 22 public and proprietary datasets and eleven generators spanning five model families (GAN, VAE, diffusion, language-model, and graph), TABAUTOSYN delivers the best train-on-synthetic utility and the highest validity while preserving privacy. Because it is generator-agnostic, it attains these results on a commodity GPU at a fraction of the cost of the strongest LLM-based baselines, and it needs no per-dataset tuning—making it practical to run on sensitive, in-house data.

1 Introduction

Synthetic tabular data is widely used in industry to augment scarce or imbalanced datasets, share information without exposing sensitive records, and stress-test models before deployment. Modern generators – GAN - (Xu et al., 2019), diffusion- (Kotelnikov et al., 2023; Zhang et al., 2024), and LLM-

based (Borisov et al., 2023) – reproduce real data closely under common similarity measures. The difficulty appears downstream: a table that looks faithful in aggregate can still be unfit for the task it was generated for, and three recurring issues drive this gap. We validate it on public benchmarks and on proprietary in-house tables from regulated domains – precisely where synthetic data is most needed and least forgiving of constraint violations.

The first concerns rare events. Generators tend to oversmooth what they learn, so the extremes of the distribution thin out and infrequent classes are reproduced poorly (Seedat et al., 2024). When the rare cases are the ones that matter – fraud, churn, adverse clinical outcomes – a generator faithful to the bulk of the data is still of limited use. The second concerns control: a practitioner usually needs data optimised for a specific downstream objective, not merely faithful to the source distribution, yet distributional similarity says little about end-task performance (Hansen et al., 2023). The third concerns validity. Beyond per-column and pair-wise statistics, real tables obey deterministic relations among their columns; when a generator breaks them, it produces records that are individually impossible even though the column-level statistics remain intact (Umesh et al., 2025; Stoian et al., 2024). In regulated or safety-relevant settings, a single such inconsistency can render the data unusable.

These issues are typically handled separately, leaving practitioners to assemble generation, validation, and repair by hand. We present **TabAutoSyn**, an end-to-end pipeline that folds them into one automated workflow: it preprocesses the real data and generates an initial table, then an LLM stage infers functional dependencies and repairs or removes violations, a tail-augmentation stage rebuilds thinned-out extremes, and a curation stage drops samples that degrade a chosen quality metric. In this way, TabAutoSyn moves from “gener-

¹The code is available at <https://github.com/Samp0rt/TabAutoSyn>.

ate and hope” to “generate, verify, augment, and curate,” producing synthetic tables better aligned with deployment requirements and less dependent on manual post-hoc cleanup.

Intended use case. TabAutoSyn targets practitioners who need validity, tail coverage, and downstream utility simultaneously without committing to a single generator. In regulated or compliance-sensitive settings the challenge is not choosing a generator but ensuring that *any* generator’s output is deployable. Because TabAutoSyn wraps around an existing generator, a privacy-oriented model, for instance, retains its privacy properties while gaining validity and utility improvements. Our experiments instantiate the pipeline on two generators that span the full quality–cost spectrum—DDPM (weakest/cheapest) and TabNAT (strongest in our study)—so gains at both extremes bound expectations for any intermediate generator.

Contributions. (1) We present TabAutoSyn, a generator-agnostic pipeline that turns any base tabular generator into deployment-ready synthetic data through four automated stages – generate, verify, augment, and curate – with no per-dataset tuning. (2) We introduce a multi-agent LLM module that discovers eight classes of functional dependencies and integrity constraints from the real table and repairs or removes violating rows, raising constraint validity that aggregate fidelity metrics ignore. (3) We pair tail augmentation with a curation stage that accepts a change only when the downstream score does not decrease, directly targeting task utility rather than a distributional proxy; monotonicity is exact under fixed seeds and transfers to the test set across all 22 datasets. (4) Empirically, across 22 datasets and eleven generators TabAutoSyn attains the best utility and validity while preserving privacy, at a fraction of the cost of LLM-based baselines.

2 Related Work

Data Curation for Synthetic Tabular Data. Synthetic-data quality depends not only on the generator but on how data is profiled, filtered, and curated. Seedat et al. (2024) introduce CLLM, pairing LLM-based generation in low-data regimes with a post-hoc step that scores each sample by its learning dynamics (confidence and aleatoric uncertainty) and discards those that would harm downstream performance. Hansen et al. (2023) bring data pro-

files (easy, ambiguous, hard examples) into the generative setting, showing that near-identical statistical fidelity can mask large gaps in downstream utility, reframing curation as a first-class component rather than an afterthought.

LLM Agents for Tabular Data. A nascent line of work moves beyond single-pass generation toward agentic, feedback-driven LLM procedures. Ronval et al. (2025) propose TAGAL, a training-free workflow in which a generator and a critic LLM interact in a self-refinement loop, yielding data competitive with training-based generators while admitting external domain knowledge. Bendinelli et al. (2025) pair an LLM with code execution to iteratively clean tabular data, but repair existing data rather than synthesize new records. Overall, agentic generation of tabular data remains underexplored, with open questions around controllability, dependency preservation, and cost.

Preserving Dependencies and Integrity Constraints. Faithful tables must reproduce inter-column correlations and the deterministic relations that encode domain knowledge. Modern generators capture correlation structure with increasing fidelity: Zhang et al. (2024) propose TabSyn, a latent score-based diffusion model that captures inter-column relations and reduces pair-wise correlation error over GAN- (Xu et al., 2019), diffusion- (Kotelnikov et al., 2023), and LLM-based (Borisov et al., 2023) baselines, while GOGGLE (Liu et al., 2023) models relational structure via a learned graph. Yet statistical fidelity alone does not ensure logically valid samples. Stoian et al. (2024) add a differentiable layer that provably satisfies linear-inequality constraints, and Ge et al. (2021) enforce denial constraints during sampling. Targeting dependencies directly, Umesh et al. (2025) introduce the Q-function for logical dependencies and show current generators fail to preserve them, while Yang et al. (2025) condition each feature on its parents in an LLM-induced sparse dependency graph to reduce violations.

3 Methodology

TabAutoSyn is a sequential framework for constructing task-ready synthetic tabular datasets (Figure 1): it preprocesses the real data and trains a generator, validates and repairs the synthetic records with a multi-agent LLM module, augments under-represented tails, and finally curates against a downstream metric.

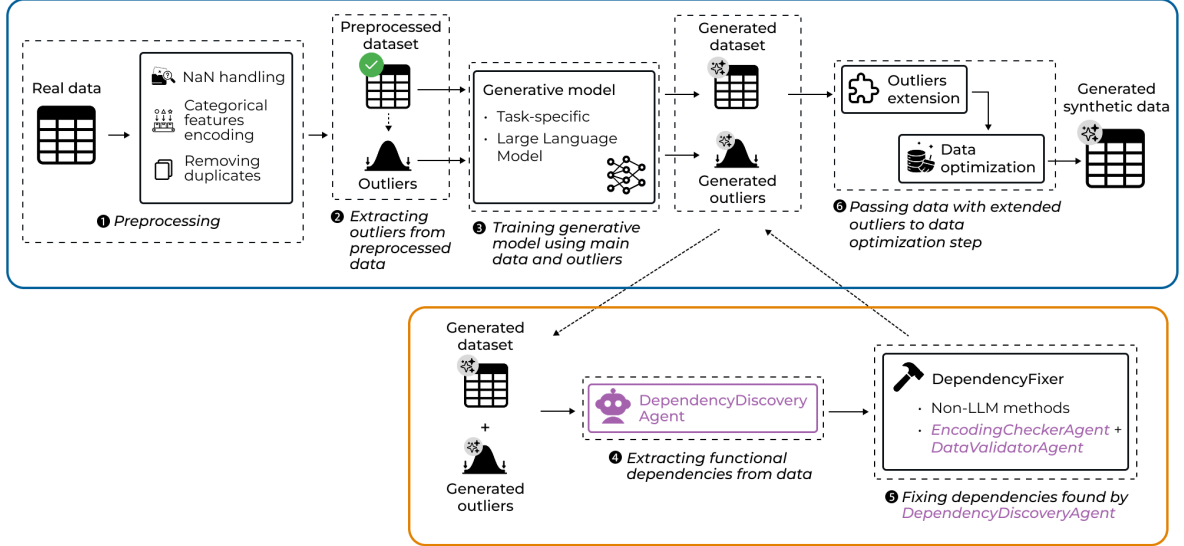


Figure 1: Overview of the proposed TabAutoSyn pipeline. The real dataset is first preprocessed and used to train a tabular generator, which produces an initial synthetic dataset. The generated data then passes through a multi-agent LLM module for functional dependency and constraint validation, followed by tail augmentation and metric-driven curation.

3.1 Data Preprocessing and Generation

TabAutoSyn first applies a lightweight preprocessing module that removes duplicate records, handles missing values, and encodes categorical variables, yielding a clean machine-readable table while preserving the statistical structure needed for generation and validation (details in Appendix A).

The pipeline then trains a tabular generative model on the prepared data. TabAutoSyn is *generator-agnostic*: it accepts any synthcity plugin or custom generator and optionally tunes hyperparameters with Optuna under the chosen evaluation criterion. In our experiments we instantiate it with two bases of very different cost: the lightweight DDPM from synthcity and TabNAT, the strongest baseline in our study. The generator is also used to populate under-represented tail regions, which we defer to the tail-augmentation stage (Section 3.3).

3.2 Constraint and Dependency Discovery and Repair

The next stage of **TabAutoSyn** enforces structural validity in two steps: *discovery*, which operates only on the real data to identify relations that should also hold in the synthetic table, and *repair*, which uses them to correct invalid synthetic records.

Discovery. The real dataset is processed by the DatasetProcessor (Figure 3), which extracts

dataset- and feature-level metadata (feature types, value distributions, cardinalities, missing-value patterns, and correlation signals). This metadata, together with the feature names and a short textual description of the dataset – provided by the user or, if absent, generated automatically by an LLM agent (Opus 4.6) – is passed to the DependencyDiscovery agent, which returns a structured (JSON) specification of the expected constraints.

The discovery stage operates in two complementary modes. The first is purely data-driven: the agent detects from metadata that a low-cardinality column partitions numeric ranges, or that an arithmetic identity links columns. The second leverages world knowledge to validate a statistically detected candidate—for example, metadata may show that `state_code` perfectly co-occurs with `bird`, but confirming which bird belongs to which state requires external knowledge. We view this as a strength rather than a confound: statistical miners detect that a mapping exists but cannot distinguish genuine domain rules from coincidences in the sample, nor supply correct replacement values when repairing violations.

We target eight dependency categories common in real tabular data: *mappings*, *dependent ranges*, *single-feature rules*, *value ranges*, *arithmetic relations*, *logical dependencies*, *temporal ordering*, and *unique value combinations* (full definitions

and examples in Appendix B). The eight categories were identified through iterative analysis of both enterprise and public datasets, prioritising dependency types that appear frequently and are violated often enough to affect deployment. Notable exclusions—soft or probabilistic dependencies, multi-table integrity constraints, cyclical dependencies, and higher-order constraints—are deliberate scope boundaries. The architecture is extensible: adding a new category requires only a new prompt entry in the DependencyDiscovery template and a corresponding repair function in DataFixer, while the overall pipeline structure remains unchanged.

Repair. The DataFixer module (Figure 4) then applies the discovered specification to the synthetic data. Most categories are repaired by deterministic, non-LLM procedures. The exception is *dependent ranges*: simple cases (fewer than three dependent columns) are also handled algorithmically, while complex ones invoke a two-agent LLM procedure (Claude Haiku 4.5) in which one agent infers the encoding scheme of the affected features and a second locates violating records and proposes corrected values. Repaired rows are merged back, yielding a constraint-consistent synthetic set for the subsequent stages.

We note that generator-agnosticism in the discovery-and-repair stage holds by construction: DependencyDiscovery operates exclusively on real-data metadata, while DataFixer and the two LLM repair agents treat synthetic rows as a plain table with no access to generator internals. The generator enters the pipeline only through a standard fit/generate interface, so there is no mechanism through which these stages could behave differently depending on the generator.

3.3 Tail Augmentation

Deep generative models concentrate probability mass near the centre of the distribution and under-populate the tails, so rare classes and rare events—often the ones that matter in production—are reproduced poorly. We mitigate this by augmenting only where the synthetic set is deficient: we score each real point by its proximity to the synthetic distribution along three complementary views: the Mahalanobis distance S_{mahal} , the mean distance to its k nearest synthetic neighbours S_{knn} , and the negative log-likelihood under a density estimated on the synthetic data S_{density} . After

normalisation we combine them into

$$S_{\text{comb}} = w_1 S_{\text{mahal}} + w_2 S_{\text{knn}} + w_3 S_{\text{density}},$$

where the non-negative weights w_1, w_2, w_3 balance the three views; we select them by grid search over the weight simplex (step 0.1) using Jensen–Shannon divergence on development splits drawn from training data; the held-out test set is never used. The resulting weights are fixed for all 22 datasets. The grid also serves as a sensitivity analysis: the objective varies within a narrow range across the simplex, confirming robustness to the exact weight choice. Taking the poorly reproduced set as the upper quantile $D_{\text{real}}^{\text{poor}} = \{i : S_{\text{comb}}[i] > Q_p(S_{\text{comb}})\}$, we run additional generation with the same base generator conditioned exclusively on $D_{\text{real}}^{\text{poor}}$, yielding a pool of tail candidates for the under-covered regions.

Rather than appending this pool wholesale, we add from it selectively, treating tail augmentation as a divergence-minimisation problem against the real data. The objective is a configurable divergence $\mathcal{D}(D_{\text{real}}, D_{\text{syn}})$, set to the Jensen–Shannon divergence by default, with Sinkhorn–Wasserstein, Kolmogorov–Smirnov, and mutual-information-matrix variants also available. Starting from D_{syn} , candidates are added iteratively and each is accepted only if it lowers $\mathcal{D}$ by at least a small threshold; an addition budget and an early stop on the first round without an accepted candidate bound the number of objective evaluations. As every committed addition strictly decreases the divergence, the augmented set is guaranteed to be at least as aligned with the real data as the original, and the procedure runs without manual tuning.

3.4 Data Curation

A synthetic table typically contains rows that, while individually plausible, collectively degrade a model trained on the data; our curation stage removes them. We frame curation as black-box optimization: given a synthetic set S and a held-out slice of real data R , find a subset $S' \subseteq S$ maximizing a downstream score $f(S', R)$, where f trains models on S' and evaluates them on R . This train-on-synthetic, evaluate-on-real objective targets utility for the end task rather than a distributional proxy. The scorer f is fully pluggable—any model-scoring callable is accepted, so one can curate against a production metric—and defaults to the summed ROC-AUC of a logistic-regression and

an XGBoost classifier (summed R^2 of a linear and an XGBoost regressor for regression), pairing a linear and a tree-based learner to avoid overfitting to one model family. As f is evaluated many times, a lightweight XGBoost-only fast scorer prescreens candidates before the full objective is paid.

We curate in three stages of increasing granularity, each accepting a change only if the verified objective does not decrease. A confidence pre-filter drops the lowest-confidence quantile of rows (scored by a reference model trained on real data) in one rollback-guarded cut; block-level backward elimination then greedily removes the single best score-improving block at several dataset-adapted block sizes, with a forward-recovery pass that restores any block whose reinstatement helps; finally, targeted refinement removes individual rows and pairs, starting from the lowest-confidence ones. Patience-based early stopping bounds the work, so cost scales with the data rather than an exhaustive search. Retaining all classes and a per-class minimum, removing at most a fixed fraction of rows, and dropping duplicates, the procedure runs without oversight; since every committed change is re-verified against the full objective under fixed seeds, monotonicity is exact: the curated set is at least as good as the un-curated set on the held-out training slice, and this guarantee transfers to the test set insofar as the training slice is representative—a condition confirmed empirically across all 22 datasets, where test-set scores after post-processing were never lower than the unprocessed baseline.

4 Experimental Setup

4.1 Datasets

We evaluate on 22 tabular datasets, summarised in Table 6: 15 classification tasks (binary and multi-class) and 7 regression tasks. They vary widely in size, dimensionality, and feature types, providing a broad testbed for synthetic-data generation. The benchmark combines *public* and *proprietary* datasets. The public datasets are standard tabular benchmarks obtained from OpenML and the UCI repository (Vanschoren et al., 2014; Kelly et al., 2023) and span common domains such as census, finance, medical, and image-derived tabular data. The proprietary datasets—oil_lithology, oil_netpay, flood, social_net_age, and social_net_interest—come from real oil-and-gas, hydrological, and social-network applications; they cannot be publicly released and let us

evaluate the pipeline under realistic, sensitive deployment conditions whose distributions differ from curated academic benchmarks. Each dataset has a single prediction target. We use a fixed 80/20 train/test split performed *externally* before the pipeline is invoked. TabAutoSyn receives only the 80% training portion; the 20% test set is never exposed to any pipeline stage. The split is not visible in the released code because TabAutoSyn is designed to accept whatever data it is given—in deployment, that is the full table. For the experiments reported here, we split externally and passed only the training portion. Curation optimises its downstream score against a held-out slice of this *training* data (not the test set), so the evaluation protocol remains uncontaminated.

4.2 Compared Synthetic Generation Models and Evaluation Metrics

We benchmark our pipeline against a broad and representative set of tabular generative models spanning the main modelling paradigms: a GAN-based generator (CTGAN) and a VAE-based generator (TVAE) (Xu et al., 2019); diffusion- and score-based generators (STaSy, CoDi, TabDDPM, TabSyn, TabDiff, TabNAT) (Kim et al., 2023; Lee et al., 2023; Kotelnikov et al., 2023; Zhang et al., 2024; Shi et al., 2025; Zhang et al., 2025a); autoregressive language-model generators (GReaT, GraDe) (Borisov et al., 2023; Zhang et al., 2025b); and a graph-based generator (GOGGLE) (Liu et al., 2023), covering all main modelling families. For every baseline we use the official implementation with its default (or paper-recommended) hyperparameters and no per-dataset tuning, so observed differences reflect the methods themselves rather than unequal tuning effort.

We evaluate along four axes that mirror the deployment concerns of Section 1, all implemented in *synthcity* (Qian et al., 2023) for reproducibility unless noted otherwise. (i) *Downstream ML utility* under the train-on-synthetic, test-on-real protocol: ROC-AUC for classification and R^2 for regression, each obtained with a linear model, XGBoost, and an MLP. (ii) *Statistical fidelity*: the inverse Kullback–Leibler divergence and the Kolmogorov–Smirnov complement on marginals, the joint Wasserstein distance, and PRDC coverage. (iii) *Privacy and detectability*: the nearest-neighbour distance ratio, distance-to-closest-record protection, membership-style identifiability, and a detection AUC measured with XGBoost, where an

ideal score of 0.5 corresponds to random-guessing accuracy, meaning the classifier cannot distinguish synthetic records from real ones. Grouping detection under privacy follows the synthcity framework (Qian et al., 2023), which we adopt for reproducibility. (iv) *Validity*, the fraction of records that violate the discovered dependencies, illustrated on a geospatial dataset (Figure 2). We additionally report per-dataset *compute cost* (Table 10). Unless stated otherwise, each number is a mean over datasets with a 95% bootstrap confidence interval; for our pipeline we also report the change relative to its own base generator, isolating the contribution of the pipeline from that of the generator.

5 Results

We instantiate TabAutoSyn on two base generators of very different strength and cost—the lightweight DDPM and TabNAT, the strongest baseline in our study—and report, alongside each absolute score, the change the pipeline induces over its own base. This isolates the contribution of TabAutoSyn from that of the underlying generator and tests our central claim: that the pipeline makes a *cheap* generator competitive with, and a *strong* one better than, the best dedicated synthesisers.

Downstream utility. Table 1 reports utility under the train-on-synthetic, test-on-real protocol. On its weak DDPM base, TabAutoSyn improves every metric and repairs the catastrophic regression behaviour of raw diffusion, turning models far worse than predicting the mean into genuinely useful ones (XGBoost $R^2 +6.37$). Applied to TabNAT, the pipeline yields the single best generator in the study – the top entry in every column. The gains are largest exactly where unconstrained generators fail, the regression targets, because curation optimises the end-task score directly rather than a distributional proxy.

Statistical fidelity. These utility gains do not come at the expense of fidelity (Table 2). TabAutoSyn improves its DDPM base on all four measures and roughly halves the joint Wasserstein distance, while the TabNAT variant matches or sets the best value on every column. That a utility-driven objective also tightens fidelity is expected: pruning task-harmful rows removes off-manifold ones as a side effect.

Validity. Aggregate fidelity hides the violations that block deployment, so we measure them di-

rectly: for several datasets we specify a ground-truth dependency and report the fraction of satisfying synthetic rows (Tables 3–9), spanning a dependent-range rule, two dictionary functional dependencies, an arithmetic relation, and a geographic polygon constraint. The gains over a weak base are dramatic: raw DDPM satisfies the `us_location` state–bird mapping in only 5% of rows and the adult education mapping in 34%, whereas TabAutoSyn (DDPM) repairs both to 100%, matching the best generators of any cost. The arithmetic relation (`page_blocks: height × length = area`) is the sharpest case: every baseline fails almost completely—the strongest, GraDe, reaches only 0.57—whereas TabAutoSyn lifts preservation to 0.79 (DDPM) and 0.92 (TabNAT). This improvement comes entirely from the verify-and-repair stage (Section 3.2), not from the base generator: DependencyDiscovery identifies the identity `height × length = area` from the dataset metadata, and DataFixer deterministically recomputes area for every violating row. While the repair itself is deterministic once the constraint is known, the non-trivial contribution is discovering the constraint automatically and applying it without breaking other dependencies. On the hardest *continuous* constraint—coordinates inside the correct state polygon—no generator is perfect, yet TabAutoSyn (TabNAT) leads all eleven baselines (0.83) and (DDPM) lifts its base from 0.11 to 0.73; Figure 2 shows this on Connecticut, cutting the raw DDPM out-of-polygon rate from 84.6% to 21.1% (DDPM) and 9.1% (TabNAT). Invisible to marginal statistics, these relations decide whether a record is usable in a regulated pipeline; the same mechanism enforces all eight dependency classes of Section 3.2.

Privacy. TabAutoSyn does not trade privacy for utility (Table 7). On the three identifiability/detection measures whose ideal is 0.5, both variants stay close to the ideal and at least as safe as their base, and nearest-neighbour distance ratios remain high, indicating the curated data is neither a memorised copy nor trivially separable from real records.

Cost. Finally, the pipeline is inexpensive to run (Table 10). Because TabAutoSyn is generator-agnostic, its GPU footprint is simply that of its base generator: on DDPM it needs well under a cent of GPU time per dataset, more than two orders of magnitude below the LLM-based generators GReaT (\$2.28) and GraDe (\$1.50) that it

Method	Linear (AUC) $\uparrow$	XGB (AUC) $\uparrow$	MLP (AUC) $\uparrow$	Linear (R^2) $\uparrow$	XGB (R^2) $\uparrow$
CTGAN	0.77 $\pm$ 0.07	0.76 $\pm$ 0.08	0.56 $\pm$ 0.09	-0.22 $\pm$ 0.47	-1.24 $\pm$ 1.61
TVAE	0.87$\pm$0.05	0.89 $\pm$ 0.05	0.68$\pm$0.07	0.35 $\pm$ 0.22	0.33 $\pm$ 0.25
Goggle	0.72 $\pm$ 0.08	0.68 $\pm$ 0.06	0.50 $\pm$ 0.13	-18.64 $\pm$ 22.95	-0.02 $\pm$ 0.55
DDPM	0.80 $\pm$ 0.09	0.80 $\pm$ 0.10	0.59 $\pm$ 0.11	-9.10 $\pm$ 14.50	-6.25 $\pm$ 10.43
GReaT	0.83 $\pm$ 0.07	0.82 $\pm$ 0.09	0.61 $\pm$ 0.11	0.20 $\pm$ 0.29	-0.16 $\pm$ 0.66
GraDe	0.79 $\pm$ 0.08	0.82 $\pm$ 0.08	0.62 $\pm$ 0.11	0.15 $\pm$ 0.21	-0.42 $\pm$ 1.14
STaSy	0.79 $\pm$ 0.07	0.81 $\pm$ 0.08	0.59 $\pm$ 0.11	0.39 $\pm$ 0.42	0.24 $\pm$ 0.67
CoDi	0.86 $\pm$ 0.06	0.90 $\pm$ 0.06	0.60 $\pm$ 0.11	-4.78 $\pm$ 7.99	-7.06 $\pm$ 11.46
TabSyn	0.87$\pm$0.06	0.91$\pm$0.05	0.61 $\pm$ 0.10	0.26 $\pm$ 0.54	-0.07 $\pm$ 0.91
TabDiff	0.86 $\pm$ 0.05	0.90 $\pm$ 0.05	0.62 $\pm$ 0.11	0.45 $\pm$ 0.30	0.10 $\pm$ 0.75
TabNAT	0.87$\pm$0.06	0.91$\pm$0.04	0.62 $\pm$ 0.11	0.51$\pm$0.26	0.41 $\pm$ 0.41
TabAutoSyn (DDPM)	0.87$\pm$0.06 (+0.06)	0.91$\pm$0.05 (+0.11)	0.64 $\pm$ 0.11 (+0.05)	-1.25 $\pm$ 2.88 (+7.85)	0.12 $\pm$ 0.90 (+6.37)
TabAutoSyn (TabNAT)	0.88$\pm$0.05 (+0.00)	0.92$\pm$0.04 (+0.01)	0.66 $\pm$ 0.12 (+0.04)	0.51$\pm$0.29 (+0.00)	0.64$\pm$0.23 (+0.23)

Table 1: ML-utility of synthetic data (mean $\pm$ 95% bootstrap CI half-width over datasets). Bold = best per column (within $\pm$ 0.01). For the highlighted “Ours” row the value in coloured parentheses is the change vs the base generator (**green** = gain, **red** = loss).

Method	Inv. Kullback–Leibler divergence $\uparrow$	Kolmogorov–Smirnov test $\uparrow$	Wasserstein distance $\downarrow$	PRDC coverage $\uparrow$
CTGAN	0.87 $\pm$ 0.04	0.80 $\pm$ 0.04	0.58 $\pm$ 0.19	0.62 $\pm$ 0.11
TVAE	0.78 $\pm$ 0.06	0.85 $\pm$ 0.04	0.42 $\pm$ 0.17	0.78 $\pm$ 0.08
Goggle	0.75 $\pm$ 0.10	0.79 $\pm$ 0.05	0.49 $\pm$ 0.18	0.58 $\pm$ 0.12
DDPM	0.74 $\pm$ 0.12	0.79 $\pm$ 0.08	1.96 $\pm$ 1.48	0.62 $\pm$ 0.17
GReaT	0.80 $\pm$ 0.07	0.81 $\pm$ 0.04	<i>n/a</i> [†]	0.78 $\pm$ 0.09
GraDe	0.82 $\pm$ 0.06	0.87 $\pm$ 0.03	<i>n/a</i> [†]	0.84 $\pm$ 0.06
STaSy	0.89 $\pm$ 0.05	0.86 $\pm$ 0.04	0.48 $\pm$ 0.21	0.75 $\pm$ 0.13
CoDi	0.83 $\pm$ 0.06	0.79 $\pm$ 0.05	0.61 $\pm$ 0.27	0.72 $\pm$ 0.13
TabSyn	0.90$\pm$0.05	0.90 $\pm$ 0.04	0.40 $\pm$ 0.17	0.91 $\pm$ 0.04
TabDiff	0.91$\pm$0.05	0.91$\pm$0.03	0.41 $\pm$ 0.17	0.91 $\pm$ 0.04
TabNAT	0.91$\pm$0.05	0.92$\pm$0.03	0.37 $\pm$ 0.16	0.93$\pm$0.04
TabAutoSyn (DDPM)	0.84 $\pm$ 0.0 (+0.11)	0.85 $\pm$ 0.0 (+0.06)	1.01 $\pm$ 0.5 (+0.94)	0.82 $\pm$ 0.1 (+0.20)
TabAutoSyn (TabNAT)	0.91$\pm$0.0 (+0.00)	0.92$\pm$0.0 (+0.00)	0.33$\pm$0.1 (+0.04)	0.94$\pm$0.0 (+0.01)

Table 2: Statistical fidelity of synthetic data (mean $\pm$ 95% bootstrap CI half-width over datasets). Bold = best per column (within $\pm$ 0.01 of the optimum). For the highlighted “Ours” rows the value in coloured parentheses is the *improvement* vs the base generator – positive means closer to ideal regardless of metric direction (**green** = better, **red** = worse). [†] Generator emitted samples orders of magnitude outside the data range; the raw value (here $\gg 10^2$) is treated as out-of-range and excluded from the best-in-column comparison.

outperforms on utility. The only added cost is a one-off LLM-API validation fee (mean \$0.59 per dataset)—a deliberate trade-off that grows with table width and violation count—so even the *total* cost stays below both LLM baselines.

6 Conclusion

We presented **TabAutoSyn**, an end-to-end pipeline that makes synthetic tabular data fit for deployment rather than merely faithful in aggregate. By composing *any* base generator with a multi-agent LLM stage for dependency discovery and repair, a divergence-minimising tail-augmentation stage, and a curation stage that accepts changes only when they do not decrease the target metric, it addresses the three failure modes of Section 1 – rare-case

loss, lack of task control, and constraint violation – in one automated workflow. Across 22 public and proprietary datasets and eleven generators it attains the best downstream utility and the highest validity while preserving privacy, at a fraction of the cost of LLM-based baselines. Because it needs no per-dataset tuning and never touches the held-out real data, it is straightforward to apply to sensitive, in-house tables; natural extensions include execution-feedback repair and exposing the curation objective to a team’s own production or fairness metric.

Limitations

TabAutoSyn relies on a hosted LLM for dependency discovery and complex repair; cost, latency,

and exact outputs therefore depend on the provider and may drift across model versions, which can affect reproducibility on closed APIs.

The LLM layer in dependency discovery is an implementation choice rather than a methodological necessity. Whether it is justified depends on the diversity of incoming datasets, the variety of dependency types, and the tolerance for per-dataset engineering. For well-formalised settings with known schemas and a fixed set of constraint types, deterministic rule miners are viable and cheaper. The pipeline targets a different regime—heterogeneous datasets arriving with minimal metadata where constraint types are unknown in advance. In this setting the LLM handles the long tail of dependency patterns at modest cost (\$0.59 per dataset on average, Table 10). This trade-off should be weighed against the practitioner’s deployment context.

Dependency discovery is hypothesis-driven from metadata and a small data sample, so it can miss true constraints or propose spurious ones; confidence thresholds and deterministic repair for simple types reduce but do not eliminate this.

The eight dependency categories cover the types most frequently encountered and most often violated in our enterprise and public benchmarks. Cyclical dependencies (where column A depends on B and B depends on A) are a genuine gap: the current repair stage processes categories independently and does not detect or resolve circular constraint chains. Soft or probabilistic dependencies and multi-table integrity constraints are also out of scope. The architecture is designed to be extensible—adding a category requires a new prompt entry and a corresponding repair function—but coverage should be expanded as new deployment settings demand it.

We do not report precision and recall for the dependency-discovery stage because a complete ground-truth constraint set is not well-defined: one would need to certify not only which constraints hold but that no others hold, and for an arbitrary dataset this enumeration is open-ended. Instead we evaluate discovery through its downstream effect. Validity tables (3–5, 8–9) test known constraints and confirm they are discovered and repaired; utility gains (Table 1) guard against false discoveries, since spurious repairs would distort the distribution and harm downstream scores. The consistent gains across both base generators indicate that false discoveries, if any, cause no measurable harm.

Per-run API cost grows with the number of

constraint-violating rows and with table width, so very large or very wide tables can raise the LLM portion of the cost beyond the averages we report. Both the augmentation and curation stages assume access to a representative held-out slice of real data to score against, which may be scarce in the lowest-data regimes.

The tail-augmentation stage re-fits the base generator on $D_{\text{real}}^{\text{poor}}$ and samples new rows; no real row is copied verbatim, and deduplication during curation removes exact duplicates. Both pipeline variants match or exceed their base generator on all aggregate privacy measures (Table 7). However, this is aggregate evidence. Because tail augmentation concentrates generative capacity on a small real-data subset, it may increase memorisation risk for that subset. Subset-level privacy auditing—evaluating memorisation metrics specifically on the tail population—is warranted and should be standard practice when deploying the pipeline on sensitive data.

Our validity evaluation uses explicitly specified ground-truth dependencies on a subset of datasets (Tables 8, 9, and Appendix C); constructing such constraint sets for every benchmark—and for soft or probabilistic dependencies—remains manual and incomplete. Finally, we study single-table data only; relational/multi-table and time-series synthesis are out of scope and left to future work.

Acknowledgements

This work supported by the Ministry of Economic Development of the Russian Federation (IGK 000000C313925P4C0002), agreement No139-15-2025-010.

References

- Tommaso Bendinelli, Artur Dox, and Christian Holz. 2025. Exploring LLM agents for cleaning tabular machine learning datasets. In *ICLR 2025 Workshop on Foundation Models in the Wild*.
- Vadim Borisov, Kathrin Seßler, Tobias Leemann, Martin Pawelczyk, and Gjergji Kasneci. 2023. Language models are realistic tabular data generators. In *The Eleventh International Conference on Learning Representations*.
- Chang Ge, Shubhankar Mohapatra, Xi He, and Ihab F. Ilyas. 2021. [KAMINO: Constraint-aware differentially private data synthesis](#). *Proceedings of the VLDB Endowment*, 14(10):1886–1899.
- Lasse Hansen, Nabeel Seedat, Mihaela van der Schaar, and Andrija Petrovic. 2023. Reimagining synthetic

- tabular data generation through data-centric AI: A comprehensive benchmark. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023) Track on Datasets and Benchmarks*.
- Markelle Kelly, Rachel Longjohn, and Kolby Nottingham. 2023. The UCI machine learning repository. <https://archive.ics.uci.edu>.
- Jayoung Kim, Chaejeong Lee, and Noseong Park. 2023. STaSy: Score-based tabular data synthesis. In *The Eleventh International Conference on Learning Representations*.
- Akim Kotelnikov, Dmitry Baranchuk, Ivan Rubachev, and Artem Babenko. 2023. TabDDPM: Modelling tabular data with diffusion models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 17564–17579. PMLR.
- Chaejeong Lee, Jayoung Kim, and Noseong Park. 2023. CoDi: Co-evolving contrastive diffusion models for mixed-type tabular synthesis. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 18940–18956. PMLR.
- Tennison Liu, Zhaozhi Qian, Jeroen Berrevoets, and Mihaela van der Schaar. 2023. GOGGLE: Generative modelling for tabular data by learning relational structure. In *The Eleventh International Conference on Learning Representations*.
- Zhaozhi Qian, Bogdan-Constantin Cebere, and Mihaela van der Schaar. 2023. Synthcity: Facilitating innovative use cases of synthetic data in different data modalities. *arXiv preprint arXiv:2301.07573*.
- Benoît Ronval, Pierre Dupont, and Siegfried Nijssen. 2025. TAGAL: Tabular data generation using agentic LLM methods. In *Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2025)*, volume 2841 of *Communications in Computer and Information Science*. Springer.
- Nabeel Seedat, Nicolas Huynh, Boris van Breugel, and Mihaela van der Schaar. 2024. Curated LLM: Synergy of LLMs and data curation for tabular augmentation in low-data regimes. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 44060–44092. PMLR.
- Juntong Shi, Minkai Xu, Harper Hua, Hengrui Zhang, Stefano Ermon, and Jure Leskovec. 2025. TabDiff: A mixed-type diffusion model for tabular data generation. In *The Thirteenth International Conference on Learning Representations*.
- Mihaela Cătălina Stoian, Salijona Dyrnishi, Maxime Cordy, Thomas Lukasiewicz, and Eleonora Giunchiglia. 2024. How realistic is your synthetic data? constraining deep generative models for tabular data. In *The Twelfth International Conference on Learning Representations*.
- Chaithra Umesh, Kristian Schultz, Manjunath Mahendra, Saptarshi Bej, and Olaf Wolkenhauer. 2025. Preserving logical and functional dependencies in synthetic tabular data. *Pattern Recognition*, 166:111702.
- Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. 2014. OpenML: Networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60.
- Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. 2019. Modeling tabular data using conditional GAN. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*.
- Shuo Yang, Zheyu Zhang, Bardh Prenkaj, and Gjergji Kasneci. 2025. Doubling your data in minutes: Ultra-fast tabular data generation via LLM-induced dependency graphs. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.
- Hengrui Zhang, Liancheng Fang, and Philip S. Yu. 2025a. TabNAT: A continuous-discrete joint generative framework for tabular data. In *Proceedings of the 42nd International Conference on Machine Learning*, *Proceedings of Machine Learning Research*. PMLR.
- Hengrui Zhang, Jiani Zhang, Balasubramaniam Srinivasan, Zhengyuan Shen, Xiao Qin, Christos Faloutsos, Huzefa Rangwala, and George Karypis. 2024. Mixed-type tabular data synthesis with score-based diffusion in latent space. In *The Twelfth International Conference on Learning Representations*.
- Zheyu Zhang, Shuo Yang, Bardh Prenkaj, and Gjergji Kasneci. 2025b. Not all features deserve attention: Graph-guided dependency learning for tabular data generation with language models. *arXiv preprint arXiv:2507.18504*.

7 Appendix

A Data Preprocessing

Before generation, the real dataset is passed through a preprocessing module that produces a clean, machine-readable table while preserving the statistical structure needed for generation and validation. The module removes duplicate records, handles missing values, and encodes categorical variables when present. The goal is not feature engineering but a consistent representation: tabular generators are sensitive to noise and inconsistent feature encodings, so this step reduces spurious variation that would otherwise propagate into the synthetic data and into the downstream dependency-discovery stage.

B Dependency Categories

The DependencyDiscovery agent (Section 3.2) targets eight categories of dependencies that we empirically found to be common in real tabular datasets:

1. **Mapping:** values of one feature determine values of another, e.g. a city uniquely corresponding to a postal region.
2. **Dependent ranges:** the valid range of one variable depends on the value of another variable.
3. **Single-feature rule constraints:** a required format or fixed pattern for a single feature, e.g. an email format or an identifier pattern.
4. **Value ranges:** a feature is restricted to a valid numerical or categorical domain.
5. **Arithmetic dependencies:** a feature equals an arithmetic expression of others, e.g. a total amount equal to price multiplied by quantity.
6. **Logical dependencies:** conditions linking categorical and numerical features, e.g. a field that must be present or take a specific value under a given condition.
7. **Temporal dependencies:** strict chronological order between date or time fields.
8. **Unique value combinations:** the uniqueness of a combination of features, e.g. an order-item pair.

C Additional Validity Results

Tables 3, 4, and 5 report column-dependency preservation for three further constraints discussed in Section 5: a dependent-range rule (Iris), a dictionary functional dependency between state code and state bird (US Location), and a dictionary functional dependency between education and education-num (Adult).

D Additional Figures and Tables

Method	Score
CTGAN	0.61 ± 0.277
STaSy	0.67 ± 0.471
GraDe	0.90 ± 0.084
TVAE	0.91 ± 0.116
Goggle	0.98 ± 0.027
TabSyn	0.98 ± 0.029
DDPM	0.98 ± 0.034
GReaT	0.99 ± 0.010
CoDi	0.99 ± 0.014
TabDiff	0.99 ± 0.015
TabNAT	0.99 ± 0.021
TabAutoSyn (DDPM)	1.00 ± 0.000
TabAutoSyn (TabNAT)	1.00 ± 0.000

Table 3: Column dependency preservation score on the Iris dataset. The tested dependency is: for target = 0, petal length ≤ 2.5 and petal width ≤ 0.75 ; for target = 1, petal length ≤ 6 and petal width ≤ 2 ; for target = 2, petal length ≤ 8 and petal width ≤ 3 .

Method	Score
STaSy	0.03
DDPM	0.05
Goggle	0.06
TVAE	0.67
CTGAN	0.75
CoDi	0.98
TabSyn	0.98
TabDiff	0.98
TabNAT	0.98
GReaT	1.00
GraDe	1.00
TabAutoSyn (DDPM)	1.00
TabAutoSyn (TabNAT)	1.00

Table 4: Column dependency preservation score on the US Location dataset. The tested dependency is a dictionary-based functional dependency: for each generated pair (code,bird), the dependency is preserved if code exists in the reference dictionary and birds_dict[code] = bird.

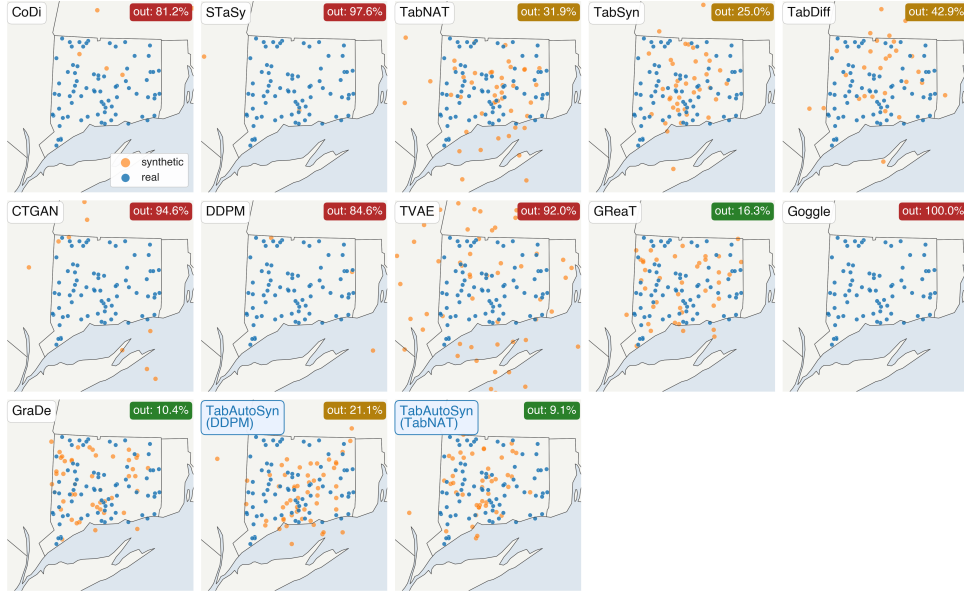


Figure 2: Connecticut panel from the per-state evaluation on `us_location`: synthetic samples (orange) overlaid on real held-out points (blue), one generator per panel. The chip reports the *out-of-polygon rate* – the share of `state_code=CT` synthetic rows whose coordinates fall outside the actual state polygon (**green** $\leq 20\%$, **amber** $\leq 50\%$, **red** $> 50\%$). Our **TabAutoSyn** pipeline sharply cuts the out-of-polygon rate of the raw DDPM base from 84.6% to 21.1% (DDPM) and 9.1% (TabNAT).

Method	Score
STaSy	0.000
Goggle	0.084
DDPM	0.342
CTGAN	0.393
CoDi	0.727
TabDiff	0.962
TabSyn	0.970
TVAE	0.972
TabNAT	0.998
TabAutoSyn (DDPM)	0.998
GReaT	1.000
GraDe	1.000
TabAutoSyn (TabNAT)	1.000

Table 5: Column dependency preservation score on the Adult dataset. The tested dependency is a dictionary-based functional dependency between education and education-num: for each generated pair (education, education-num), the dependency is preserved if education exists in the reference dictionary and its mapped value equals education-num.

Dataset	Target	Task
Adult	income	classification
Breast cancer	target	classification
Credit_g	target	classification
Oil lithology	Lithology	classification
Heloc	RiskPerformance	classification
Iris	target	classification
Magic04	class	classification
Nursery	target	classification
Page blocks	target	classification
Pendigits	target	classification
Phoneme	target	classification
Travel	Target	classification
US location	bird	classification
Social net	top1_interest	classification
Wine	target	classification
Bodyfat	target	regression
Cpu small	target	regression
Diabetes	target	regression
Oil netpay	Netpay	regression
Flood	L	regression
Seattle_housing	price	regression
Social net age	age	regression

Table 6: Datasets used in the benchmark: target column and task type.

Method	DCR (≈ 0.5)	NN-identifiability (≈ 0.5)	Detection XGB (≈ 0.5)	NNDR $\uparrow$
CTGAN	0.48 ± 0.03	0.65 ± 0.07	0.96 ± 0.03	0.90 ± 0.03
TVAE	0.49 ± 0.04	0.44 ± 0.07	0.91 ± 0.04	0.84 ± 0.05
Goggle	0.49 ± 0.07	0.55 ± 0.08	0.96 ± 0.03	0.88 ± 0.03
DDPM	0.51 ± 0.04	0.58 ± 0.10	0.82 ± 0.06	0.87 ± 0.04
GReaT	0.52 ± 0.04	0.40 ± 0.04	0.83 ± 0.06	0.83 ± 0.04
GraDe	0.49 ± 0.04	0.40 ± 0.05	0.80 ± 0.05	0.82 ± 0.05
STaSy	0.52 ± 0.04	0.44 ± 0.07	0.86 ± 0.06	0.85 ± 0.04
CoDi	0.54 ± 0.05	0.41 ± 0.06	0.88 ± 0.06	0.84 ± 0.04
TabSyn	0.54 ± 0.04	0.38 ± 0.04	0.74 ± 0.05	0.83 ± 0.04
TabDiff	0.52 ± 0.04	0.39 ± 0.05	0.72 ± 0.05	0.83 ± 0.04
TabNAT	0.55 ± 0.05	0.35 ± 0.03	0.61 ± 0.03	0.81 ± 0.04
TabAutoSyn (DDPM)	0.52 ± 0.04 (-0.01)	0.46 ± 0.07 (+0.04)	0.76 ± 0.05 (+0.06)	0.86 ± 0.04 (-0.01)
TabAutoSyn (TabNAT)	0.54 ± 0.04 (+0.00)	0.37 ± 0.05 (+0.02)	0.61 ± 0.04 (+0.00)	0.82 ± 0.04 (+0.01)

Table 7: Privacy / detection metrics (mean $\pm$ 95% bootstrap CI half-width over datasets). The first three have ideal ≈ 0.5 (bold = closest to 0.5 per column, within ± 0.01); NNDR mean is higher-is-better (bold = highest). For the highlighted “Ours” row the coloured parentheses give the improvement vs the base generator – toward 0.5 for the ≈ 0.5 metrics ($|\text{base} - 0.5| - |\text{new} - 0.5|$) and the raw gain for NNDR (green = better, red = worse).

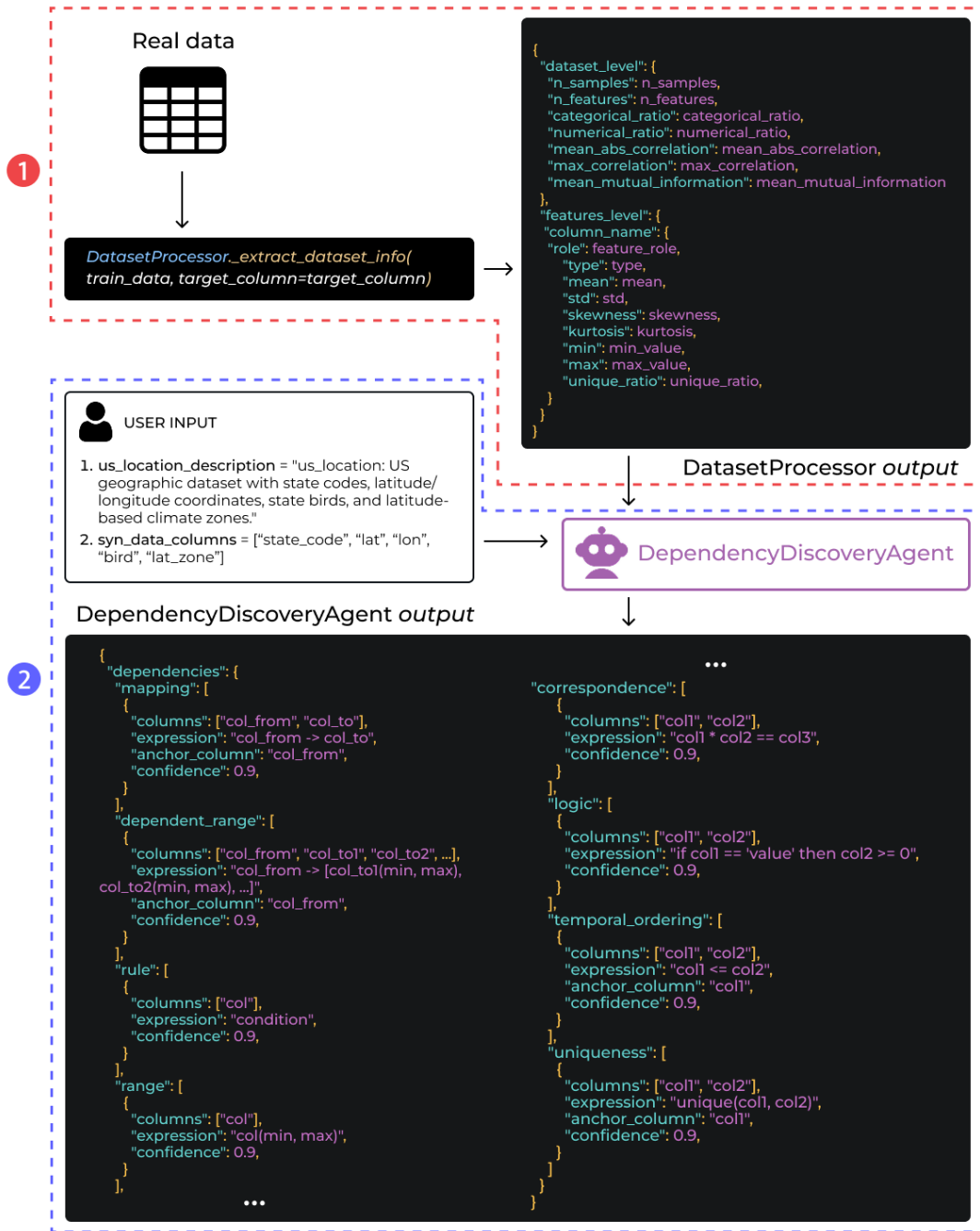


Figure 3: Overview of the dependency discovery stage. First, the real dataset is processed by the DatasetProcessor, which extracts dataset-level and feature-level metadata, including feature types, statistics, cardinalities, and correlation-based characteristics. This metadata is combined with the user-provided dataset description and feature semantics, and then passed to the DependencyDiscoveryAgent. The agent returns a structured JSON specification of the discovered dependencies and constraints, including mappings, dependent ranges, rule constraints, value ranges, arithmetic correspondences, logical dependencies, temporal ordering, and uniqueness constraints.

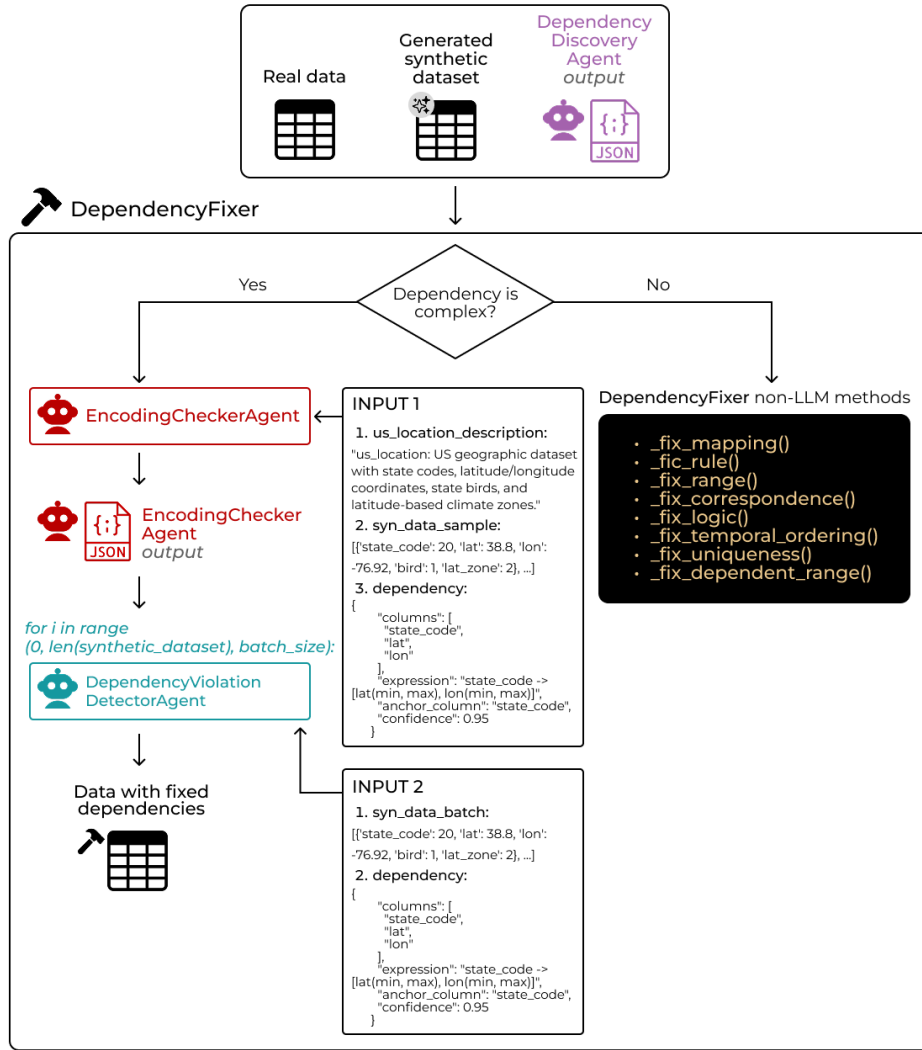


Figure 4: Overview of the dependency fixing stage. The module receives the real dataset, the generated synthetic dataset, and the structured output of the DependencyDiscovery agent. If a dependency is not complex, it is repaired using deterministic non-LLM procedures. Complex dependencies are processed by the EncodingCheckerAgent, which identifies encoded features and their likely encoding scheme, followed by the DependencyViolationDetectorAgent, which detects and fixes invalid synthetic records in batches.

Method	Score
STaSy	0.01 ± 0.023
Goggle	0.02 ± 0.031
DDPM	0.11 ± 0.103
CTGAN	0.11 ± 0.158
TVAE	0.16 ± 0.189
CoDi	0.60 ± 0.263
TabDiff	0.70 ± 0.109
TabSyn	0.71 ± 0.140
TabAutoSyn (DDPM)	0.73 ± 0.154
TabNAT	0.77 ± 0.158
GReaT	0.82 ± 0.068
GraDe	0.82 ± 0.081
TabAutoSyn (TabNAT)	0.83 ± 0.128

Table 8: Column dependency preservation score on the US Location dataset. The tested dependency is geographical consistency: for each state s , rows with `state_code` corresponding to s must have coordinates (`lon`, `lat`) located inside or on the border of the geographic polygon of state s .

Method	Score
CTGAN	0.000
TVAE	0.000
Goggle	0.000
STaSy	0.000
CoDi	0.000
TabDiff	0.001
DDPM	0.002
TabSyn	0.002
TabNAT	0.060
GReaT	0.275
GraDe	0.573
TabAutoSyn (DDPM)	0.792
TabAutoSyn (TabNAT)	0.923

Table 9: Column dependency preservation score on the Page Blocks dataset. The tested dependency is an arithmetic dependency between height, length, and area: for each generated row, the dependency is preserved if $\text{height} \times \text{length} = \text{area}$.

Method	GPU time (s)	GPU cost (\$)	LLM cost (\$)	Total (\$)
TVAE	8	0.004	–	0.004
CTGAN	14	0.008	–	0.008
Goggle	51	0.028	–	0.028
TabSyn	92	0.051	–	0.051
DDPM	7	0.006	–	0.006
TabDiff	240	0.132	–	0.132
TabNAT	359	0.197	–	0.197
CoDi	561	0.309	–	0.309
GReaT	4,144	2.279	–	2.279
GraDe	2,727	1.500	–	1.500
TabAutoSyn (DDPM)	7	0.006	0.587	0.589
TabAutoSyn (TabNAT)	359	0.197	0.813	1.011

Table 10: Compute cost per dataset (mean over the benchmark), every generator billed at the same rate of \$1.98 per GPU-hour (RTX 5080). For each baseline the cost is its own logged train+sample wall-clock time $\times$ this rate. **TabAutoSyn** runs a lightweight base generator (the DDPM / TabNAT baseline) and adds a one-off LLM-API validation/repair pass; we list the base-generator GPU cost on the same GPU and that API fee (from the pipeline run summaries) separately. TabAutoSyn’s GPU footprint is thus that of a cheap DDPM – \$0.006, over $700\times$ below the LLM-based SOTA generators GReaT (\$2.28) and GraDe (\$1.50) – and even its *total* cost is $2.5\text{--}3.9\times$ lower than both, while it attains the best ML-utility (Table 1).

E Multi-Agent LLM Prompts

For full reproducibility, we list below the verbatim prompt templates used by the multi-agent LLM module (Section 3.2). Each agent receives one template with runtime values substituted for the \$placeholder fields (e.g. \$columns, \$statistics, \$sample); the schema placeholders (\$output_schema, \$encoding_schema) are replaced by the JSON schema the agent must conform to.

Dependency Discovery Agent

DependencyDiscovery agent: find functional dependencies in a tabular dataset from its metadata and statistics (no raw rows).

Input: domain_description, column names, aggregated statistics from real data.

CATEGORIES (expression format -> example):

- mapping: `A -> B` -> country_code -> currency
- dependent_range: `A -> [B(min, max), ...]` -> department -> [salary(min, max)] or product_category -> [weight(min, max), price(min, max)]
- rule: ONE atomic Python-executable condition per rule (no libraries) -> "@" in email, len(str(phone)) == 10

FORBIDDEN in rule expressions: keywords `if`, `for`, `while`, `and`, `or`.

If you need multiple conditions, create SEPARATE rule entries for each.

- range: `col(min, max)` -> [temperature(-40, 55), humidity(0, 100)]
- correspondence: arithmetic formula using ONLY column names, numeric constants, and operators (+, -, *, /, ==, <=, >=, <, >) -> unit_price * quantity == total_amount

Use `==` for equality (not `=`).

FORBIDDEN in correspondence expressions: natural-language words, descriptions, or phrases (e.g. "is proportional to", "depends on");

Unicode math symbols (~, <=, >=, x, /, etc.);

approximate relations -- only exact equalities/inequalities.

If you cannot write an exact formula, do NOT create a correspondence entry.

- logic: `if COND then CONSTRAINT` (one or more rules) -> if status == "retired" then age >= 55; if discount > 0 then has_coupon == True
- temporal_ordering: `A <= B` -> hire_date <= termination_date
- uniqueness: `unique(A, B)` (at least 2 columns required) -> unique(order_id, product_id)

confidence: 1.0 = certain, 0.7-0.9 = strong hypothesis, 0.4-0.6 = plausible.

DETECTION STRATEGY -- how to systematically find dependencies:

1) **CRITICAL -- dependent_range is the HIGHEST-PRIORITY check.**

For EVERY column with low cardinality (<= 10 unique values) -- including integer-coded class labels, target variables, category columns, and ordinal codes -- you MUST check whether it partitions ANY numeric column into narrower sub-ranges.

How: compare per-category min/max (from statistics) against the global min/max. If the global range is noticeably wider than any single category's range, that is a dependent_range.

A single categorical column can (and often does) constrain ALL numeric columns at once -- emit one dependent_range entry listing every constrained numeric column.

Example: a "species" column with 3 categories constrains body measurements

-> species -> [height(min, max), weight(min, max), wingspan(min, max)].

- DO NOT skip this check -- it is the single most valuable dependency type.
- 2) For every pair of categorical columns, ask: does one uniquely determine the other? If the cardinality ratio suggests a many-to-one relationship -> mapping.
 - 3) For every numeric column, check if its min/max or domain is inherently bounded by physical or domain logic -> range or rule.
 - 4) For every group of numeric columns, check if an arithmetic identity links them -> correspondence.
 - 5) For every categorical column paired with another column, check if specific category values require specific values/states in the other column -> logic.
 - 6) Look for PAIRED columns with related names (e.g. X and X_Duration, X_count and X_amount).
- If one column is a count/quantity and the other is a derived metric (duration, amount, rate), check: does count == 0 imply metric == 0? Does count > 0 imply metric > 0? If yes -> logic.
- Example: if num_orders == 0 then total_spent == 0.

Rules:

- Use statistics to support hypotheses (MI, correlation, cardinality, min/max, top_values).
- Be thorough: better to propose a weak dependency than miss a real one.
- Pay special attention to categorical -> numeric relationships: they are the most common source of dependent_range dependencies and are easy to miss.
- Empty category lists are fine. Do NOT invent columns not in the input.
- Expressions must NOT contain raw statistics or summary values. For data-driven bounds use symbolic placeholders like `col(min, max)` -- the fix function will compute actual values from real_df at runtime.
- Output ONLY valid JSON.

INVALID expression examples (NEVER produce these):

- [x] rule: "lot_size_units == 0 or lot_size_units == 1" -- contains `or`, split into separate rules.
- [x] rule: "size >= 250 and size <= 11010" -- contains `and`, use `range` category instead.
- [x] correspondence: "size is proportional to beds and baths" -- natural language, not a formula.

OUTPUT SCHEMA

\$(output_schema)

INPUT

- 1) Domain: \$domain_description
- 2) Columns: \$columns
- 3) Statistics: \$statistics

Return ONLY valid JSON conforming to OUTPUT SCHEMA.

Dataset-Info Generator Agent

You are UserDFInfoGeneratorAgent.

Task:

Given dataset columns and a small chunk of real rows, infer what this dataset describes.

INPUT

- 1) Columns: \$columns
- 2) Real data chunk: \$real_data_chunk

STRICT OUTPUT FORMAT:

- Return ONLY one plain string (no JSON, no Markdown, no code fences).
- Output must be exactly one sentence in this format: <dataset_name>: <dataset description sentence>.
- `dataset\_name` must be short, lowercase, snake_case.
- The description must be concise but dependency-oriented and specific to the provided chunk.
- Do not output extra lines, explanations, or surrounding quotes.

CONTENT REQUIREMENTS FOR THE DESCRIPTION SENTENCE:

- Mention the domain/entity of the dataset.
- Explicitly state what the target-like column represents (if any obvious label/class column exists).

- Mention 2-4 important columns (or column groups) and their likely role.
- Include likely logical constraints useful for dependency discovery (for example: code->name mappings, category->numeric range constraints, uniqueness keys, simple if-then logic).
- Prefer concrete wording from observed columns/rows; avoid generic phrases like "tabular data with several features".
- Keep it to one sentence using semicolons to separate parts.

Example format:

us_location: US geography-by-state dataset; target bird is the official state bird label; state_code maps to state identity while lat/lon encode location; lat_zone constrains plausible latitude ranges and co-varies with climate-like attributes; state_code with state name behaves as near-unique mapping.

Encoding Checker Agent

You are the EncodingChecker agent.

Return ONLY one top-level JSON object with EXACTLY these keys: `encoding\_detected`, `encoding\_kind`, `confidence`, `readable\_mapping` (no other keys).

- `encoding\_detected`: true if any column in the sample needs decoding for downstream checks; false if all relevant values are already human-readable / non-coded.
- `encoding\_kind`: dominant kind across coded columns (`label`, `ordinal`, `opaque`, `custom`); use `none` when encoding_detected is false; use `mixed` when multiple kinds apply.
- `confidence`: 0.0-1.0 for the overall decoding (lower if any mapping is uncertain).

`readable\_mapping`: for each column that uses a non-obvious encoding (label/ordinal/code flags, etc.), `readable\_mapping[column\_name][raw\_value\_string]` = concrete human meaning derived from the INPUT sample rows together with \$dataset_info and \$dependency (use same -row fields, e.g. lat/lon, to infer geography).

Rules:

- Include inner objects under `readable\_mapping` only for columns that need decoding. Omit columns that are already plain text or raw coordinates with no code layer. If nothing needs decoding, set encoding_detected=false,

encoding_kind=none, confidence high, readable_mapping={}.

- For every included column, map EVERY distinct raw value that appears in \$sample for that column (string keys; e.g. integer 9 -> key "9"). No empty inner objects. No placeholders like "TBD" or "sample needed".
- Do not invent values that never appear in \$sample.
- Use bidirectional evidence from dependencies: dependent columns can help decode anchor codes, and anchor values can help decode dependent codes. Infer mappings from row-level co-occurrence patterns across the whole sample.
- Prefer mappings that are globally consistent across rows. Example: if multiple rows with similar lat/lon ranges repeatedly co-occur with the same state_code, decode that state_code to the matching state (e.g., Alaska).

OUTPUT SCHEMA (return ONLY valid JSON, no Markdown)
\$encoding_schema

INPUT

1) Sample (rows covering all distinct anchor values, full table columns):
\$sample

2) Dataset info (single string, may be JSON or text summary):
\$dataset_info

3) Dependency:
\$dependency

ANSWER

Return ONLY JSON matching OUTPUT SCHEMA.

Dependent-Range Batch Detector Agent

You are validating dependency consistency.

Anchor: \$anchor_column = \$anchor_value (\$anchor_encoded).
Dependent columns to validate: \$dependent_columns.

Task:

- 1) Decide whether dependent values are valid for this anchor.
- 2) If invalid, provide corrected dependent values in the same order.

CORRECTION QUALITY RULES:

- Corrections must be logically valid for the given anchor.

- Across invalid rows in this batch, corrected value tuples MUST be diverse.
- Do NOT assign one identical default tuple (e.g., state center) to every invalid row.
- Do NOT return duplicate `corrected\_values` tuples for different invalid rows unless the valid anchor-specific range has only one plausible value.
- Avoid creating duplicate samples: the tuple `[ \$anchor\_value ] + corrected\_values` should differ from other corrected rows and, when possible, from already valid rows in this batch.
- If several invalid rows share the same anchor, spread corrections across different plausible anchor-specific values instead of collapsing them to one center/default.
- Keep diversity realistic: values should stay within plausible anchor-specific ranges.

STRICT OUTPUT REQUIREMENTS:

- Return ONLY valid JSON (no markdown, no prose, no code fences).
 - Return EXACTLY this schema and key names:
- ```
{
 "is_valid": true_or_false,
 "corrected_values": [value_for_dep_1, value_for_dep_2, ...],
 "reason": "short reason based only on provided row and anchor"
}
```
- Include no additional keys.
  - Do not argue, justify, or debate; answer the task directly and concisely.
  - `corrected\_values` must follow this exact dependent column order: \$dependent\_columns.
  - If `is\_valid` is true, set `corrected\_values` to [] and still provide `reason`.
  - Do not request more information. Use only the provided row.

\$n\_rows rows to check are: \$batch\_rows

## F LLM Context Boundary

A key design decision in TabAutoSyn is that no LLM agent ever receives the full dataset serialised into its context. All full-table computations are performed by the DatasetProcessor, a non-LLM Python module that extracts dataset- and feature-level statistics over the complete data. What each agent sees is as follows.

**UserDfInfoGenerator** (Dataset-Info Generator) receives at most 10 sampled rows and the column-name list. It produces a short textual description of the dataset when the user does not supply one.

**DependencyDiscovery** receives the short textual description (user- or agent-provided), the column-name list, and the aggregated statistics from DatasetProcessor. It does not receive any

raw data rows.

**EncodingChecker** receives the short dataset description and a compact sample assembled as 5-row slices—one slice per `anchor_column` of a discovered `dependent_range` dependency—together with that dependency’s JSON specification from `DependencyDiscovery`.

**DependencyViolationDetector** (Dependent-Range Batch Detector) operates only on synthetic rows already under repair. For each unique anchor value it takes the matching rows from the current output dataframe, splits them into windows of `batch_size` (default 10), and sends only the dependency sub-table (the relevant columns, including the anchor) as JSON records, together with the `EncodingChecker`’s anchor encoding and the dependent-column list.

Thus, LLMs see compact samples, metadata, and—where needed—small synthetic batches, while full-table statistics remain a cheap Python computation. This design keeps the pipeline scalable to enterprise-sized tables and underpins the cost advantage reported in Table 10.