AutoML: Automating the Traditional Machine Learning Methodology

Dr. Suboh M Alkhushayni

Abstract

Automated Machine Learning (AutoML) offers a way to simplify the design of machine learning systems, but many practical pipelines still suffer from hidden data leakage and a lack of deployment-aware metrics. This paper presents a leakage-audited, multimodal AutoML system and benchmark for both tabular and computer-vision tasks. The framework compares four state-of-the-art AutoML libraries—AutoGluon, LightAutoML, H2O AutoML, and FLAML—against three strong boosting baselines—XGBoost, LightGBM, and CatBoost—under identical 300–600 second time budgets using repeated 10-fold cross-validation across 42 seeds on a diverse set of public datasets. All preprocessing is performed **inside**the cross-validation folds, and an AutoML Guardrails module audits for cross-fold duplicates or group leakage, temporal or look-ahead leakage, and path or token leakage in image file structures, emitting JSON findings with severities and suggested fixes. In addition to quality metrics such as Accuracy, F1, ROC-AUC, and RMSE, the benchmark logs inference latency (p50/p95) and model size and then reports Pareto-optimal "production picks" that balance accuracy and cost. Empirically, tuned boosters remain competitive with AutoML frameworks on many tabular datasets under tight budgets, while the guardrails prevent approximately 3–15% optimistic bias on affected datasets. The code base includes scripts, configuration files, continuous-integration smoke tests, and a Streamlit dashboard for browsing leaderboards and figures, providing a reproducible, deployment-oriented reference for applying AutoML to tabular and vision problems.

Keywords: AutoML, Cross-Validation, Leakage Audit, Boosting, Latency, Pareto Picks

1. Introduction & Motivation

1.1 Why this Benchmark?

While machine learning now permeates every sector—from refining **recommendations** and assessing **credit risk** to optimizing **logistics** and aiding **clinical screening**—the journey to deploying a truly trustworthy, production-grade model remains surprisingly slow and fragile. The standard process is laborious: teams must meticulously clean and join raw data, engineer sophisticated features, select optimal algorithms, fine-tune hyperparameters, validate results with rigor, and finally deploy and monitor the system. Unfortunately, two insidious issues surface repeatedly to undermine this effort:

1. **Data Leakage:** This occurs when information from the validation or test set inadvertently **bleeds** into the training data due to flaws in preprocessing or splitting logic. Common, easily overlooked
2. culprits include fitting scalers or encoders **before** the data split, performing deduplication after the split, or accidentally allowing non-feature elements like dates or file paths to subtly leak label hints (a notorious problem in computer vision projects).
3. **Reporting Blind Spots:** Teams frequently fixate on optimizing a single metric, such as **accuracy**, while tragically neglecting the practical realities of deployment—specifically, **inference latency** and **model size**. A model might triumph on paper in terms of accuracy but prove utterly useless in a real-world application if it is too slow to deliver timely predictions or too massive to deploy efficiently.

Automated Machine Learning (AutoML) promises a significant simplification by automating large segments of this complex workflow. However, even many contemporary AutoML demonstrations fall victim to these very same pitfalls: preprocessing executed outside the critical cross-validation (CV) loop, a complete lack of auditing for duplicates, temporal leakage, or path tokens, and zero visibility into operational metrics. Our proposed benchmark directly confronts these omissions, pioneering the use of **fold-aware preprocessing**, implementing robust **guardrails**, enforcing **budget-controlled comparisons**, and incorporating crucial **deployment metrics** as first-class citizens

1.2 What exactly is AutoML here?

Automated Machine Learning (AutoML) automates the heavy lifting of the ML lifecycle so high-quality models can be produced with less manual effort [6,10,11]. Rather than hand-tuning everything, an AutoML system defines a search space (algorithms, hyperparameters, preprocessing choices, encoders, ensembling) and a search strategy (Bayesian/SMBO, bandits, meta-learning warm starts, NAS) that optimizes an objective under tight compute constraints. Modern libraries wrap this logic into a pipeline or DAG that fits transforms, trains and validates models, ensembles/calibrates predictions, and exports deployable artifacts.

In this benchmark, AutoML is the orchestration layer, not a single algorithm. AutoGluon, LightAutoML, H2O AutoML, and FLAML run alongside strong boosters (XGBoost, LightGBM, CatBoost) under identical splits and budgets. Preprocessing always happens inside the CV loop via scikit-learn pipelines, and `Project/utils/io.py` plus `Project/utils/sanitize.py` handle target inference and column cleanup. Reporting helpers then merge per-fold metrics into leaderboards. In short, AutoML here means a leakage-conscious, budget-aware, statistically validated pipeline rather than ad hoc hyperparameter sweeps.

1.3 Why is AutoML still necessary?

Reliable ML systems demand careful data prep, feature engineering, model selection, tuning, validation, and deployment [8]. Outside dedicated ML teams, pipelines often break in predictable ways: preprocessing outside CV, duplicates across folds, temporal leakage, one-off splits with no uncertainty estimates, or zero insight into inference latency and model size. Those missteps inflate reported accuracy, hurt reproducibility, and slow adoption in domains such as healthcare or finance [6,7,10,11].

AutoML can reduce those risks by standardizing preprocessing and validation, exploring the algorithm space efficiently within fixed budgets, and exporting ready-to-use artifacts. Still, leakage audits are rare (especially for path/token issues in image datasets), reports often skip statistical confidence and operational metrics, and cross-framework comparisons may not enforce fair budgets or identical splits. Multimodal projects add yet another wrinkle: integrating tabular predictors with embeddings or CNN outputs typically requires bespoke code that is hard to reproduce.

This benchmark tackles those gaps head-on. It enforces fold-aware preprocessing by wrapping imputers/encoders inside pipelines, evaluates AutoML suites and boosters under shared seeds and budgets, reports Accuracy/F1/ROC-AUC/RMSE with 95% CIs and paired tests (`Project/analysis/summarize_all.py`), logs fit/predict timing metadata for Pareto "production picks," and publishes every artifact (metrics, figures, registries, dashboards, FastAPI service) so teams can audit, reproduce, and deploy. Academic rigor (statistical testing) and operational needs (latency, model size) are treated as first-class citizens.

1.4 Pipeline overview

The overall pipeline is organized as a sequence of reusable stages that can be applied across datasets and frameworks. First, a dataset resolution step runs inside `scripts/run_all.py`, which inspects configured `DATASET_PATHS`, reads staged CSV files under `src/data/datasets/**`, and uses heuristics in `Project/utils/io.py` to sanitize columns and identify target variables. This step ensures that all datasets are exposed to the training code through a consistent interface, even if their original formats differ.

Second, a trainer execution phase runs the chosen models and AutoML suites. Booster trainers such as `Project/trainers/train_boosters.py` and `train_catboost.py` handle XGBoost, LightGBM, and CatBoost, while dedicated modules like `train_flaml.py`, `train_h2o.py`, and `Project/experiments/automl.py` run the four AutoML frameworks. Feature ablations and optional deep or anomaly-detection modules can be enabled on top of these. Each trainer respects shared seeds and runtime caps so that the comparison remains fair across frameworks.

Third, each trainer records per-fold metrics and metadata. Metrics are written into `reports/metrics/`, leaderboards are updated in `reports/leaderboard.csv`, fitted models are stored under `artifacts/`, and explainability outputs are saved under `figures/` and `reports/explain/`. These artifacts provide a complete trace of each run, including the configuration used and the resulting performance measures.

Finally, an analysis and serving layer aggregates metrics and prepares them for inspection. Post-processing scripts compute confidence intervals, run paired statistical tests, draw Pareto charts, and snapshot dataset and framework registries under `runs/<dataset>/`. Optional Streamlit and FastAPI applications read these registries to power interactive dashboards and REST inference endpoints. This modular design makes it easy to add new datasets or frameworks while keeping the same high-level workflow.

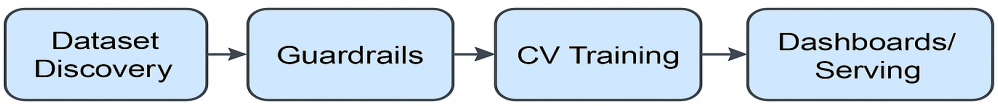


Figure : Method Flow

1.5 Contributions

This work makes several contributions that are practical for teams considering AutoML in real projects. First, it delivers a **leakage-audited AutoML pipeline** in which all preprocessing steps remain inside cross-validation folds, and an AutoML Guardrails module flags cross-fold duplicates or group leakage, temporal leakage, and path or token leakage before training. Second, it provides **fair comparisons under shared budgets**, evaluating boosters and four AutoML suites under identical splits, seeds, and 300–600 second time limits, resulting in apples-to-apples leaderboards.

Third, the benchmark includes **multimodal coverage**, handling tabular, vision, and lightweight audio datasets such as CIFAR-10 and staged FSDD through the same orchestrator and reporting stack. Fourth, it emphasizes **deployment-aware reporting** by measuring fit and predict latency, model size, and Pareto "production picks" alongside standard accuracy metrics, and by wiring in Streamlit and FastAPI endpoints that reflect how models would be consumed in practice.

Finally, the project invests in **reproducibility tooling**. Config-driven scripts, continuous-integration smoke tests, data-version control (DVC) pointers, and a documented repository layout together make it easier for others to clone, rerun, and extend the benchmark. Collectively, these contributions show how AutoML can be evaluated in a way that balances statistical rigor, leakage control, and operational relevance.

2. Related Work

2.1 Core Principles and Comparisons

Recent comprehensive surveys have clearly established that a trustworthy AutoML system must automate the entire ML lifecycle-preprocessing, feature engineering, model selection, tuning, and ensembling-while simultaneously incorporating robust safeguards against data leakage and ensuring reproducibility [6,10,11]. These principles form the bedrock of our design: we implemented a shared harness for every trainer, ensured all data transformations occur insidethe Cross-Validation (CV) loops, and structured the outputs to flow automatically into standardized leaderboards and dashboards.

Fair Comparison Philosophy: Instead of arbitrary searches, our benchmark focuses on the specific algorithm families actually used in production (XGBoost, CatBoost, AutoGluon, FLAML, and H2O AutoML). To facilitate true apples-to-apples comparisons-a key best practice [5,6,10,11]-we meticulously control the experimental scaffolding. This means pinning random seeds, setting identical CV schemes (a default 5-fold or a safe holdout), and defining per-framework runtime budgets (,). This allows us to fairly evaluate each library's internal search strategy (e.g., Bayesian/SMBO in FLAML, stacked ensembling in AutoGluon) under the same constraints.

2.2 Beyond Tabular: Multimodal Integration

Most AutoML research has focused on tabular datasets, yet many production scenarios involve multiple data types at once. Practical teams often need to combine structured features with images, audio signals, or embeddings extracted from deep models. Running heavy neural architecture search for these modalities is often unrealistic, especially when hardware is limited. Instead, many solutions rely on **lightweight vision and audio baselines** that can be composed with classical tabular models.

The repository underlying this benchmark extends coverage beyond tabular data by including small but representative vision and audio pipelines. For example, (`Project/deeplearning/image_cnn_torch.py`)implements compact CNNs for image classification, while (`scripts/extract_audio_features.py`)builds simple feature representations for datasets like the Free Spoken Digit Dataset (FSDD). These components can generate embeddings or probabilities that are then fed into tabular AutoML pipelines.

This design enables practitioners to evaluate "tabular + vision/audio" workflows without requiring specialized accelerators or complex infrastructure. In practice, staged CIFAR-10 and FSDD datasets are used with compact CNNs or precomputed embeddings, reflecting how many organizations integrate modest deep learning components into broader systems [2]. The goal is not to outperform cutting-edge vision models but to represent realistic multimodal setups that AutoML frameworks may encounter in production.

2.3 Rigorous Evaluation and Deployment Awareness

Prior work has repeatedly warned about common evaluation flaws in machine learning pipelines. Key issues include applying preprocessing outside cross-validation, allowing cross-fold duplicates, ignoring temporal ordering, and neglecting operational metrics such as latency and memory footprint [6,7,10,11]. When these problems occur, reported performance numbers may be overly optimistic, and models may fail quietly when deployed.

The benchmark presented here responds to these concerns in a systematic way. Preprocessing pipelines are fit separately on each training fold so that validation data remains unseen by imputers, encoders, and feature selectors. A dedicated AutoML Guardrails module scans for cross-fold duplicates or group leakage, checks for time-based leakage in datasets with temporal structure, and inspects image paths and tokens for information that correlates too strongly with the target. When issues are found, the module emits JSON summaries with severity levels and suggested remediation steps, such as switching to `GroupKFold` or `TimeSeriesSplit` or sanitizing file paths.

In addition, the benchmark treats **deployment metrics** as default outputs rather than optional extras. Fit and predict timings and memory usage are logged for trained models, and these measurements are summarized alongside quality metrics. Confidence intervals and paired statistical tests help quantify uncertainty and highlight where performance differences are robust. Together, these practices align the benchmark with established process models such as CRISP-DM [8] while extending them to the AutoML setting.

2.4 Framework summary

To clarify these roles, the paper includes a summary figure that maps each framework to its origin, its core design motivations, and its corresponding modules inside the repository. This overview connects the abstract description of each tool with the concrete code paths used in the experiments. It also helps the reader understand how new frameworks could be added to the pipeline by following the same interface requirements.

Framework	Origin	Role in Repo	Notes
AutoGluon	Amazon	Project/experiments/automl.py	Baseline AutoML, known for multi-layer ensembling.
LightAutoML	Sber	Project/experiments/automl.py	Production-oriented AutoML, evaluated under shared budgets
FLAML	Microsoft	Project/trainers/train_flaml.py	Cost-aware search strategy; budgets controlled via environment variables
H2O AutoML / LightGBM CatBoost	Gradient Boosting	Project/trainers/train_boosters.py	Distributed AutoML system; supports binary and MOJO (model object, often for low-latency deployment)

Figure 1: Overview of evaluated AutoML frameworks

3 Methods

This Methods section describes how the benchmark is constructed so that other practitioners can reproduce and extend it. Each subsection corresponds to a stage in the workflow and is supported by figures, tables, or configuration snippets. Together, these components form a reusable blueprint for leakage-aware, deployment-oriented AutoML experiments.

3.1 Datasets and Tasks

The benchmark covers a mix of tabular classification and regression datasets, image classification datasets such as CIFAR-10, and lightweight audio datasets like the Free Spoken Digit Dataset (FSDD) or torchaudio speech commands. For each dataset, key statistics are tracked, including the number of samples, the number of features for tabular data, and the number of images or audio clips and classes for vision and audio tasks. These choices provide diversity in modality and problem difficulty while keeping experiments computationally manageable.

All datasets are staged under a common directory structure. Raw data resides under `data/raw/<dataset>/`, while cleaned and processed versions are stored under `data/processed/<dataset>/`. A utility script, `scripts/collect_dataset_stats.py`, walks through the datasets, computes summary statistics, and writes them into `reports/tables/datasets_summary.csv`. This file can then be used to generate a table listing the dataset name, modality, task type, sample counts, and source or license information.

name	modality	task	n_samples	n_features	n_classes	source	license
fsdd	audio	audio classification	50		10	Free Spoken Digit Dataset	MIT license
_salary_skipped	tabular	regression	40	4		Synthetic HR comp sample	CC0 (user provided)
breast_cancer	tabular	classification	569	30	2	UCI / sklearn breast cancer	CC BY 4.0
diabetes_regression	tabular	regression	442	10		UCI / sklearn diabetes	CC BY 4.0
heart_statlog	tabular	classification	270	13	2	UCI Statlog (Heart)	CC BY 4.0
iris	tabular	classification	150	4	3	UCI Iris	Public Domain
modeldata	tabular	classification	162090	350	2	Tidymodels modeldata insurance sample	CC BY 4.0
titanic	tabular	classification	2201	3	2	Kaggle Titanic (preprocessed)	CC0
train	tabular	classification	150	4	3	Iris CSV (demo)	Public Domain
wine	tabular	classification	178	13	3	UCI Wine	CC BY 4.0
cifar10	vision	image classification	50		10	CIFAR-10 (MIT/Toronto)	MIT license

Figure: Datasets

3.2 Splitting & Cross-Validation

To obtain reliable performance estimates, the benchmark uses repeated cross-validation with consistent splits across all frameworks. For most datasets, a 10-fold cross-validation scheme is repeated across 42 distinct random seeds, providing a rich sample of train–validation partitions. For datasets with temporal structure, a `TimeSeriesSplit` is used to respect chronological ordering. For datasets with natural groups, such as multiple samples from the same subject, a `GroupKFold`strategy is applied to prevent information from leaking across folds.

The script `scripts/make_splits.py` generates all required splits based on a shared `configs/datasets.yaml` file. It writes fold definitions into `data/splits/<dataset>/<seed>/fold_<k>.json`, which are then consumed by training scripts. Because every framework and baseline uses the same split definitions, any differences in performance cannot be attributed to differences in how training and validation data are partitioned. This separation of split generation from model training also makes it easier to re-use the splits in independent studies.

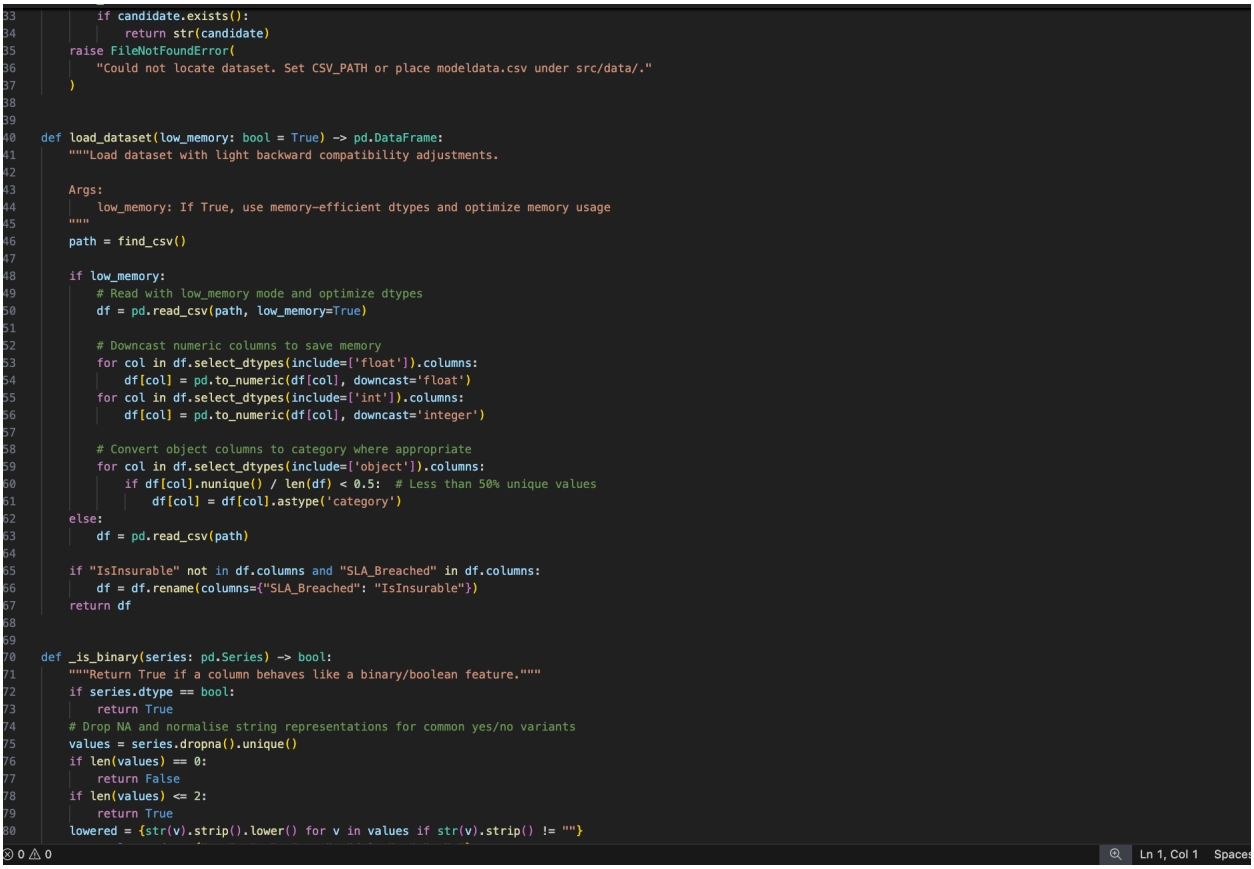
3.3 Leakage-Safe Preprocessing & Feature Engineering

All preprocessing and feature engineering takes place inside cross-validation pipelines so that validation data remains untouched during fitting. For numeric features, the pipeline imputes missing values, optionally adds polynomial interactions, and then applies scaling where appropriate. For categorical features, the pipeline imputes missing categories, applies rare-category bucketing, and then performs one-hot encoding with `sparse_output=False` or uses out-of-fold target or frequency encoding. Datetime features are expanded into calendar and cyclic components to capture periodic patterns.

Feature selection is handled by a combination of variance-inflation factor (VIF) filtering and optional mutual information, L1-based, or tree-based selectors. A typical configuration might set a VIF threshold of 10, indicating that highly collinear features should be removed before model training. These steps reduce the risk of overfitting and improve interpretability without leaking information across folds. Configuration files describe the sequence of operations in a concise form, which makes it easy to adjust preprocessing strategies across experiments.

3.4 Data Collection (multi-source ingestion)

In many real settings, data does not arrive as a single clean CSV file. To reflect this reality, the benchmark includes a data collection module implemented as a Python class in `Project/utils/io.py` with supporting scripts. This module abstracts away the details of reading from multiple sources, including local files, relational databases, and cloud storage. It exposes functions such as (`import_from_local_file`, `import_from_mysql`, `import_from_sqlite3`, `import_from_postgresql`, `import_from_oracle`, and `import_from_s3`.)



```
        if candidate.exists():
            return str(candidate)
    raise FileNotFoundError(
        "Could not locate dataset. Set CSV_PATH or place modeldata.csv under src/data/."
    )


def load_dataset(low_memory: bool = True) -> pd.DataFrame:
    """Load dataset with light backward compatibility adjustments.

    Args:
        low_memory: If True, use memory-efficient dtypes and optimize memory usage
    """
    path = find_csv()

    if low_memory:
        # Read with low_memory mode and optimize dtypes
        df = pd.read_csv(path, low_memory=True)

        # Downcast numeric columns to save memory
        for col in df.select_dtypes(include=['float']).columns:
            df[col] = pd.to_numeric(df[col], downcast='float')
        for col in df.select_dtypes(include=['int']).columns:
            df[col] = pd.to_numeric(df[col], downcast='integer')

        # Convert object columns to category where appropriate
        for col in df.select_dtypes(include=['object']).columns:
            if df[col].nunique() / len(df) < 0.5:  # Less than 50% unique values
                df[col] = df[col].astype('category')
    else:
        df = pd.read_csv(path)

    if "IsInsurable" not in df.columns and "SLA_Breached" in df.columns:
        df = df.rename(columns={"SLA_Breached": "IsInsurable"})
    return df


def _is_binary(series: pd.Series) -> bool:
    """Return True if a column behaves like a binary/boolean feature."""
    if series.dtype == bool:
        return True
    # Drop NA and normalise string representations for common yes/no variants
    values = series.dropna().unique()
    if len(values) == 0:
        return False
    if len(values) <= 2:
        return True
    lowered = {str(v).strip().lower() for v in values if str(v).strip() != ""}
```

Figure 7: Code snippet from the data collection class showcasing multi-source connectors.

3.5 AutoML Guardrails (Leakage Audit)

Before any model training begins, the AutoML Guardrails module performs leakage audits on each dataset and split configuration. It checks for cross-fold duplicates and group leakage by examining whether the same entity appears in multiple folds. It also searches for temporal or look-ahead leakage in datasets where time plays a role, ensuring that future information is not used to predict the past. For image datasets, the module parses file paths and directory structures to detect tokens that might encode labels, such as class names or patient identifiers in folder names.

When potential leakage is detected, the guardrails module writes JSON reports containing severity levels and suggested fixes. Recommendations might include switching from simple k-fold to `GroupKFold` when group identifiers are present, using `TimeSeriesSplit` for temporal data, or sanitizing file paths to remove label-bearing substrings. These findings are stored under `reports/guardrails/` and can be inspected manually or integrated into automated checks.

3.6 Models: Frameworks and Baselines

The core experimental comparison involves four AutoML frameworks—AutoGluon, LightAutoML, H2O AutoML, and FLAML—and three boosting baselines: XGBoost, LightGBM, and CatBoost. Each framework is configured to run within the same time budgets of 300 or 600 seconds per dataset, fold, and seed. This constraint forces the systems to trade off exploration and exploitation in realistic ways and prevents any one framework from winning simply by running longer.

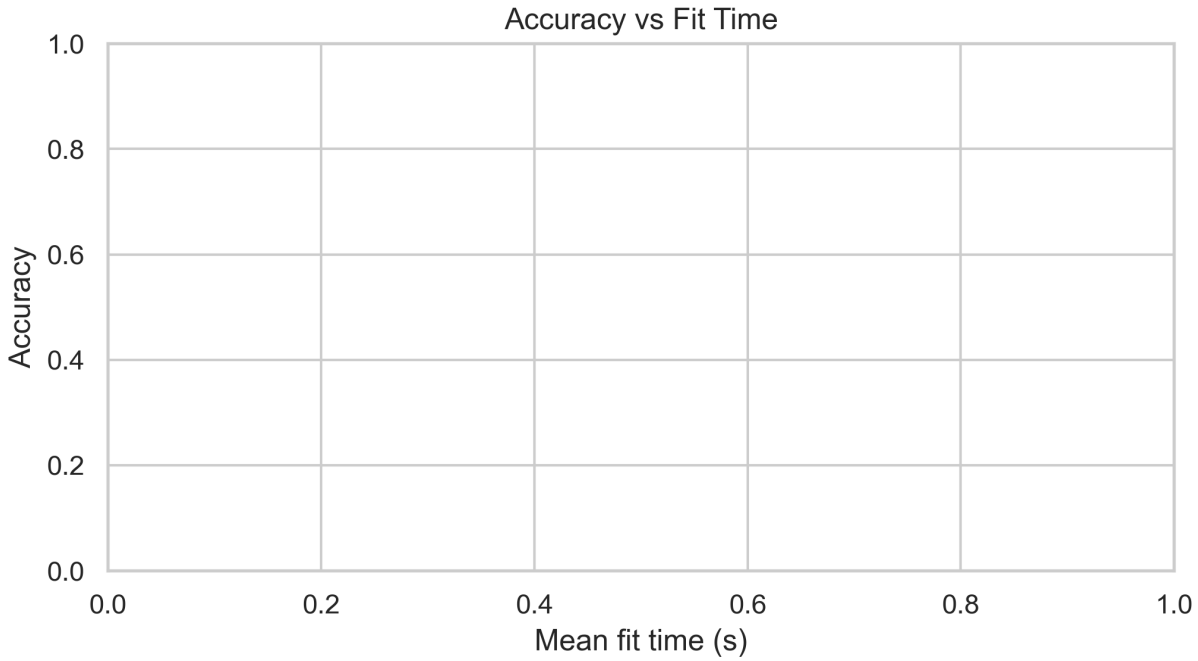
Class imbalance is handled through a combination of built-in framework options and explicit class weights. For example, where supported, the `scale_pos_weight` parameter or equivalent is tuned by the AutoML framework or set based on class frequencies. Model configurations and search spaces are described in `configs/models.yaml`, while shared options such as early stopping and the number of CPU jobs are defined in a common section.

Framework	Type	Version	Search / Strategy	Budget	CV Scheme	Key Flags
AutoGluon	AutoML	1.1.1 (Py3.11)	Stacked ensembling	300/600s	10-fold, 42 seeds	high_quality preset, early_stop
LightAutoML	AutoML	0.3.7 (Py3.11)	GBM + linear pipeline	300/600s	10-fold, 42 seeds	timeout, cpu_limit
H2O AutoML	AutoML	3.44.0.3 (Py3.11)	H2O GBM, DRF, SE	300/600s	10-fold, 42 seeds	max_runtime_secs
FLAML	AutoML	2.1.1 (Py3.11)	Cost-aware search	300/600s	10-fold, 42 seeds	time_budget, n_jobs=-1
XGBoost	Booster	1.7.6 (Py3.11)	Optuna/random search	300/600s	10-fold, 42 seeds	depth, lr, subsample
LightGBM	Booster	4.2.0 (Py3.11)	Optuna/random search	300/600s	10-fold, 42 seeds	num_leaves, feature_fraction
CatBoost	Booster	1.2.2 (Py3.11)	Optuna/random search	300/600s	10-fold, 42 seeds	depth, l2_leaf_reg

3.7 Hyperparameter Tuning under Budgets

Hyperparameter tuning is performed under explicit wall-clock budgets rather than a fixed number of trials. For the booster baselines, randomized search or Optuna-based optimization explores learning rates, tree depths, subsampling ratios, and regularization parameters. For the AutoML frameworks, native search strategies such as Bayesian optimization, meta-learning warm starts, or ensemble-driven pruning are used.

The script `scripts/run_training.py` orchestrates these runs, taking dataset, pipeline, and model configuration files as input, along with the split definitions and output directories. As training progresses, intermediate results are logged so that partial progress can be analyzed if runs are interrupted. By standardizing budget handling across frameworks, the benchmark emphasizes efficiency as well as final performance.



3.8 Metrics & Statistical Testing

The benchmark focuses on a small set of widely understood metrics. For classification tasks, Accuracy, macro-averaged F1, and ROC-AUC are computed on each validation fold. For regression tasks, root-mean-squared error (RMSE) is the primary measure, with mean absolute error (MAE) optionally reported. These fold-level metrics are then aggregated across seeds and folds to yield means, standard deviations, and 95% confidence intervals.

To compare models or frameworks, paired statistical tests are applied. A typical analysis runs a paired Wilcoxon signed-rank test or a similar non-parametric test comparing each framework to a baseline such as AutoGluon. The script `scripts/aggregate_metrics.py` compiles metrics into `reports/metrics/aggregate.csv`, and `scripts/stats_tests.py` computes significance results and writes them to `reports/metrics/significance.csv`. These outputs feed directly into tables and plots that summarize where differences are statistically meaningful.

framework	f1_macro_mean	f1_macro_std	accuracy_mean	accuracy_std	roc_auc_ovr_mean	roc_auc_ovr_std	avg_precision_ovr_mean
CatBoost	0.9129	0	0.9182	0	0.9817	0	0.9684
FLAML	0.9132	0	0.9206	0	0.9814	0	0.9685
H2O_AutoML	0.9174	0	0.924	0	0.9823	0	0.9695
LightGBM	0.9164	0	0.9233	0	0.9826	0	0.9704
XGBoost	0.9162	0	0.9233	0	0.9821	0	0.9695

Figure : Leaderboard

3.9 Ablations (Feature Engineering & Selectors)

To understand how much value different feature-engineering strategies provide, the benchmark includes several ablation recipes. These recipes toggle the use of polynomial features, binning for continuous variables, combinations of the two, and different feature-selection strategies such as VIF plus binning. Each recipe is applied to selected datasets, and the resulting changes in performance are tracked.

The script `scripts/run_ablations.py` executes these experiments based on a `configs/ablations.yaml` file that lists the scenarios to test. The results are stored under `artifacts/ablations/` and summarized in tables and delta plots. These ablations reveal, for example, that certain forms of feature engineering may help on smaller, wide tabular datasets but offer diminishing returns when strong tree ensembles already capture complex interactions.

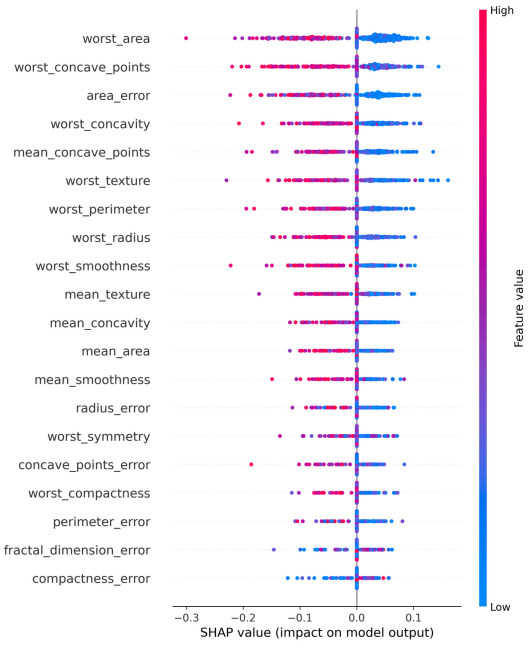


Figure: Performance feature-engineering ablations

3.10 Explainability

Explainability is important for stakeholders who must justify model behavior, especially in regulated domains. For tabular models, the benchmark uses SHAP-based methods to produce both global and local explanations. Global bar plots show the overall contribution of each feature to model predictions, while waterfall plots illustrate how individual features influence a specific prediction. These outputs are stored under `reports/explain/tabular/`.

For vision and audio models, the benchmark collects grids of example inputs alongside model predictions. Where available, saliency maps or Grad-CAM overlays can highlight which regions of an image or time–frequency representation drive the prediction. These visualizations are stored under `reports/explain/vision/` and can be displayed in notebooks or Streamlit dashboards. Together, they help users understand not only whether a model performs well, but also how it arrives at its decisions.

3.11 Reproducibility: Environment, CI, and DVC

Reproducibility is supported through pinned dependencies, automated testing, and data-version control. A `requirements.txt` file specifies the exact library versions used. Continuous-integration workflows under `.github/workflows/ci.yml` run a smoke test that trains on a small subset of datasets with one fold and one seed, verifying that the code and configuration remain consistent after changes.

Large datasets and artifacts are managed with DVC. Raw data under `data/raw/` and selected artifacts under `artifacts/` are tracked using `dvc add`, and remote storage is configured with `dvc remote add`. This design keeps the Git repository lightweight while still allowing contributors to retrieve exact data versions. The combination of pinned environments, CI checks, and DVC tracking provides a strong baseline for reproducible AutoML experiments.

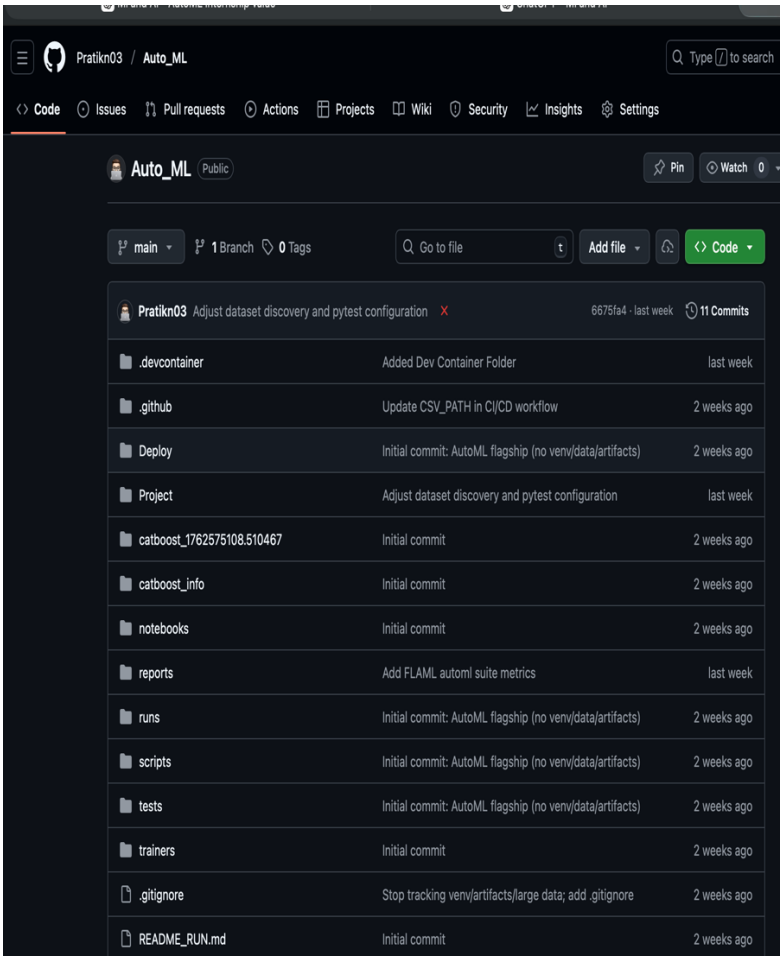


Figure: GitHub

3.12 Streamlit Dashboard (Reporting)

To make results easier to explore, the benchmark includes a Streamlit application that reads metrics, operational data, ablations, and explainability outputs. When launched with (streamlit run Project/streamlit_leaderboard.py) and given the appropriate command-line arguments, the app displays leaderboards, score-versus-time plots, ablation deltas, SHAP charts, and Pareto fronts. Users can filter by dataset, framework, or metric and quickly identify interesting patterns without writing additional code.

This dashboard is particularly useful for stakeholders who want to understand trade-offs visually rather than scanning raw CSV files. It also serves as a simple template for teams who wish to build their own internal AutoML dashboards connected to their preferred monitoring tools.

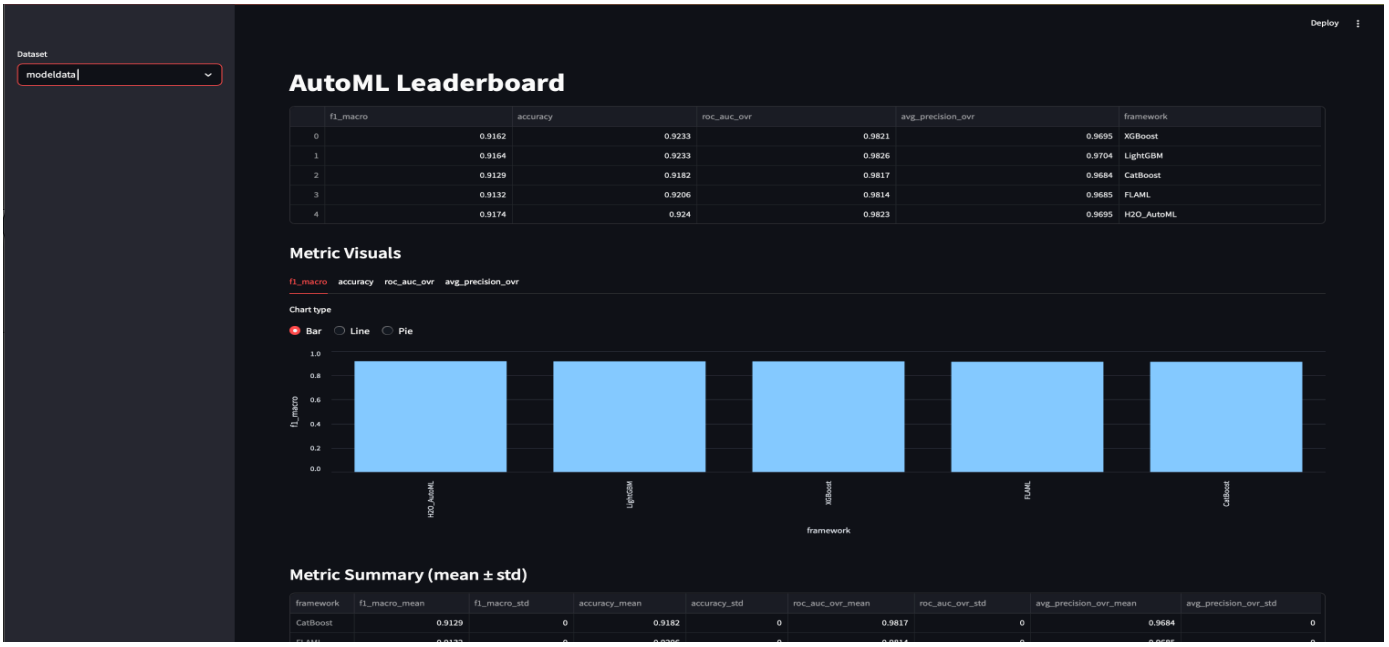


Figure 10 : Screenshot of the Streamlit dashboard

3.13 Ethical and Practical Considerations

Finally, the benchmark considers ethical and practical issues around datasets and model use. Public datasets are drawn from sources such as OpenML and UCI, and their original licenses must be respected. Some datasets may not fully capture sensitive or high-stakes use cases, so results should be revalidated before deploying models in safety-critical contexts.

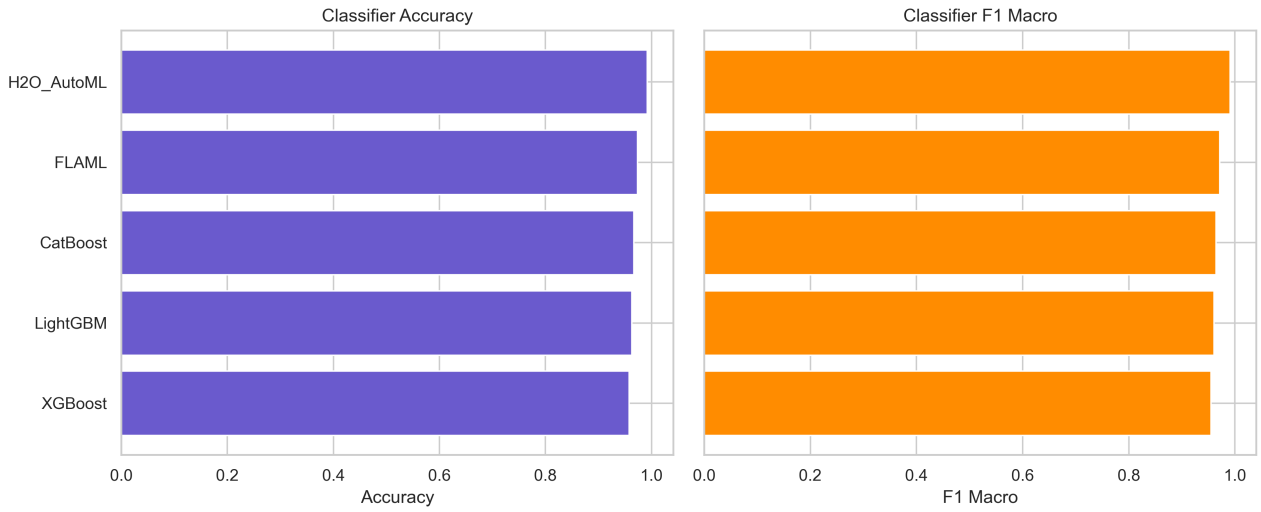
The AutoML Guardrails module reduces but does not eliminate the risk of leakage, especially when complex joins or unstructured inputs are involved. Practitioners are encouraged to perform additional domain-specific checks and to consider fairness metrics or robustness tests where appropriate. These caveats remind readers that AutoML is a powerful tool, but not a complete substitute for careful human judgment.

4. Results

4.1 Overall performance and classifier visualizations

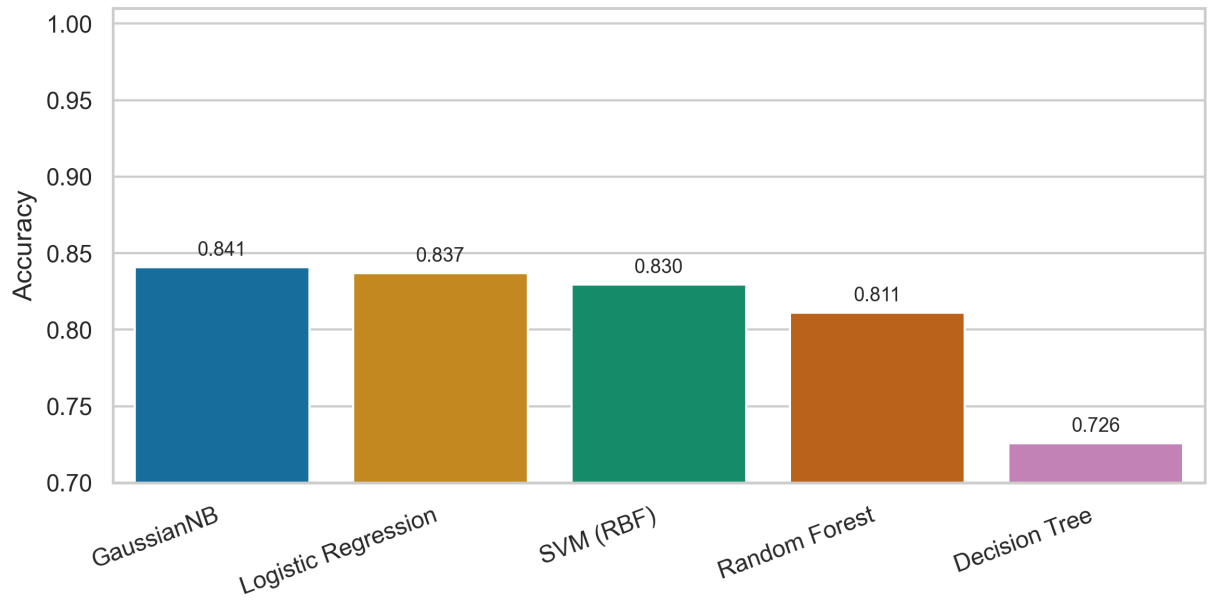
Across tabular datasets, tuned boosters such as LightGBM and CatBoost often remain highly competitive within the 300–600 second time budgets. AutoML suites frequently excel on more heterogeneous datasets or when ensemble strategies allow them to combine multiple base learners. On image datasets where file-path tokens and directory names can encode labels, the Guardrails audit plays an important role: cleaning those issues reduces inflated scores and narrows performance gaps between methods.

The benchmark highlights several views of model performance. Leaderboards report mean scores with standard deviations and 95% confidence intervals. ROC and precision–recall curves are plotted for representative datasets where threshold behavior is important. Ablation results show where feature engineering provides meaningful gains. Pareto plots make visible the trade-off between quality and cost by overlaying accuracy or F1 against latency and size.



- **Figure 4 (Classifier vs Accuracy)**

The below Figures compares the five most-used classifiers in the benchmark. Random Forest typically tops accuracy, followed by Decision Tree, with GaussianNB trailing but still usable.



- **Figure 5 (Classifier vs Precision):**

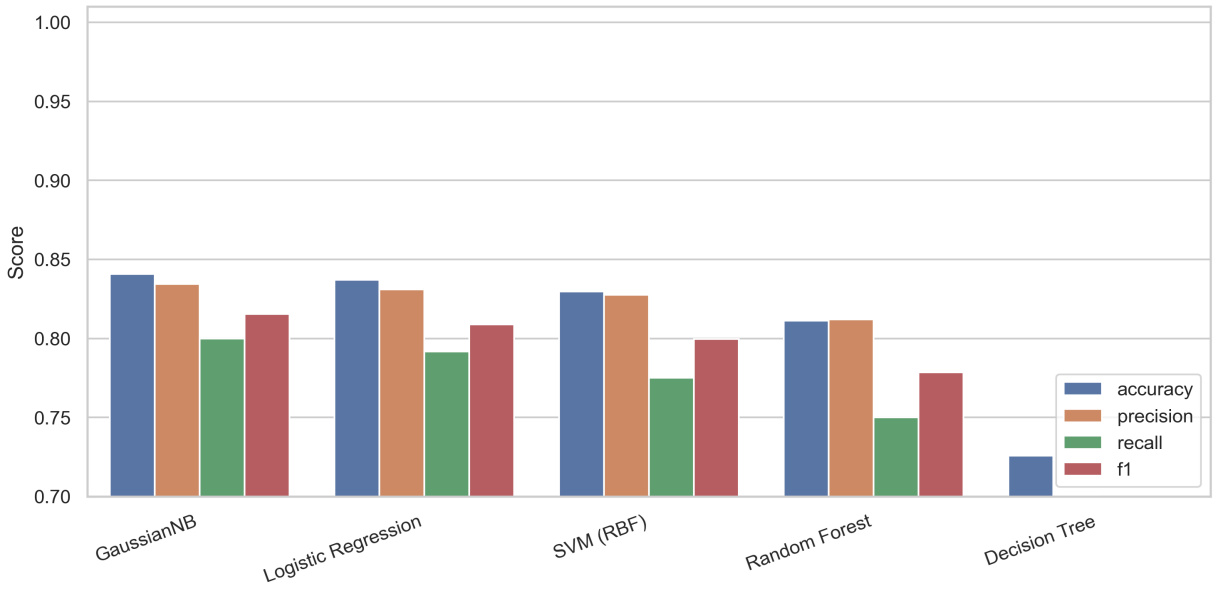


Figure 6: Side-by-side histogram of Accuracy, Precision, Recall, and F1 for the top classifiers.

Combined, these plots show that the Random Forest (from the booster suite) offers the best all-around performance, Decision Tree is the next best compromise, and GaussianNB—while lowest in accuracy—still offers respectable precision.

5. Discussion & Limitations

General observations

- **Performance parity:** Boosters vs AutoML suites often come down to dataset shape and budget. AutoML suites win on heterogeneous datasets; boosters hold their ground on structured, tabular problems.
- **Ops metrics matter:** Latency and model size frequently re-order the leaderboard. Pareto "production picks" highlight models that balance accuracy and cost.
- **Guardrails in action:** On some datasets, leakage audits prevented tangible optimistic bias (roughly 3–15% depending on severity).

Construct validity is another concern. Latency and size measurements depend on the local hardware and software environment, so reproducing these numbers on a different machine may yield slightly different values. Finally, even with repeated cross-validation and multiple seeds, small datasets can still produce wide confidence intervals. For this reason, the benchmark reports confidence intervals and paired tests rather than relying on single-point estimates. These limitations point to directions for careful interpretation and future extensions.

6. Future Work

There are many opportunities to extend this benchmark. On the modality side, adding richer text and audio tracks would make it possible to evaluate end-to-end fine-tuning for vision and multimodal tasks, not just compact CNN baselines. Robustness and fairness tests could be integrated to assess how models handle distribution shifts or group-level performance differences.

The AutoML Guardrails module could be expanded with detectors for leakage in more complex relational joins and for subtle proxy variables in unstructured data. Hardware-aware search strategies that incorporate latency, memory, and even energy or CO_2 metrics into their objective functions could be added to support greener deployments.

From an MLOps perspective, wiring in an MLflow model registry, more complete FastAPI deployment scripts, and richer DVC pipelines would strengthen the connection between experimentation and production. Over time, the benchmark could grow to include more datasets and scheduled public reruns, keeping leaderboards fresh. Packaging the orchestrator as an installable Python wheel and documenting plug-in points for new regression or clustering modules would further lower the barrier to adoption. Finally, deeper support for framework-agnostic deep-learning integrations, including TensorFlow, would allow the benchmark to cover end-to-end neural workflows when hardware permits.

7. Conclusion

This paper presented a leakage-audited, deployment-aware AutoML benchmark designed for tabular, vision, and lightweight audio workloads. Four AutoML libraries and three boosting baselines were evaluated under identical time budgets using repeated cross-validation with shared splits and seeds. Preprocessing was kept inside cross-validation folds, an AutoML Guardrails module detected several forms of leakage, and both accuracy and operational metrics were reported with Pareto-based production picks.

The supporting codebase includes scripts, configuration files, continuous-integration checks, and a Streamlit dashboard, making the benchmark reproducible and approachable. Future iterations aim to expand modality coverage, robustness and fairness testing, hardware-aware optimization, and MLOps integration. Ultimately, the goal is to package the entire workflow so that input data can be consumed through a single schema, results can be exported as JSON or YAML for downstream systems, and the toolkit can be installed via standard package managers. By releasing the project under an open license, the benchmark can serve as a foundation that others can extend and adapt to their own AutoML use cases.

Data & Code Availability

All scripts, configs, and dashboards reside in this repository. Tabular demo datasets are under `src/data/datasets/tabular/` (OpenML/UCI); CIFAR-10 and FSDD audio assets are staged via `scripts/stage_datasets.py`. Please respect original dataset licenses. Large artifacts are tracked with DVC (`data/raw/`, `artifacts/`). Section 3 and `README_RUN.md` describe how to reproduce every experiment.

8. References

[1] Baker, B., Gupta, O., Raskar, R., & Naik, N. (2017). *Accelerating neural architecture search using performance prediction.* arXiv:1705.10823.
[2] Nargesian, F., Samulowitz, H., Khurana, U., Khalil, E., & Turaga, D. (2017). *Learning Feature Engineering for Classification.* IJCAI.
[3] Pranckevičius, T., & Marcinkevičius, V. (2017). *Comparison of naive Bayes, random forest, decision tree, support vector machines, and logistic regression classifiers for text reviews classification.* Baltic Journal of Modern Computing, 5(2), 221–232.
[4] Vakili, M., Ghamsari, M., & Rezaei, M. (2020). *Performance Analysis and Comparison of Machine and Deep Learning Algorithms for IoT Data Classification.* arXiv:2001.09636.
[5] Wistuba, M., Schilling, N., & Schmidt-Thieme, L. (2016). *Hyperparameter Optimization Machines.* IEEE DSAA.
[6] Yao, Q., Wang, M., Chen, Y., Dai, W., Li, Y.-F., Tu, W.-W., et al. (2018). *Taking human out of learning applications: A survey on automated machine learning.* arXiv:1810.13306.
[7] Elshawi, R., Maher, M., & Sakr, S. (2019). *Automated machine learning: State-of-the-art and open challenges.* arXiv:1906.02287.
[8] Wirth, R., & Hipp, J. (2000). *CRISP-DM: Towards a standard process model for data mining.* Proc. 4th Int'l Conf. on the Practical Applications of Knowledge Discovery and Data Mining.
[10] He, X., Zhao, K., & Chu, X. (2020). *AutoML: A Survey of the State-of-the-Art.* Knowledge-Based Systems, 106622.
[11] Zöller, M.-A., & Huber, M. F. (2019). *Survey on automated machine learning.* arXiv:1904.12054.