Agentic AI System — Step-by-Step Full Technical Documentation

1. Project Overview

1.1 Goal of the System

The Agentic AI system is a semi-agentic AutoML and analytics platform built to:

- Let a user upload a tabular dataset (CSV / Excel / JSON).

- Automatically clean, analyze, and visualize the data.

- Optionally run an AutoML pipeline to train and select the best model.

- Expose clean analytics, model results, and prediction APIs for use in a frontend dashboard.

The system is multi-tenant: each user is isolated by user_id and versioned pipeline runs.

1.2 High-Level Architecture

The core architecture consists of:

- A Modal-deployed backend (modal_app.py) containing all business logic.

- A FastAPI application exposed via Modal's @modal.asgi_app wrapper.

- A shared Modal Volume mounted at /root/data for persistent storage.

- Pipeline modules under pipeline/ for ingestion, preprocessing, feature engineering, model training, and visualization utilities.

- A remote LLM (Groq) used for planning feature engineering, model selection, visualization plans, and final Markdown reports.

Data flows through ingestion → preprocessing → visualization → feature engineering → model training → reporting → prediction.

1.3 Data Flow Summary

End-to-end flow:

1) User calls POST /start with file info, target_column, and run_automl flag.

2) The backend ingests the file and saves a raw copy in the user uploads folder.

3) The preprocessing agent cleans the data and classifies feature types.

4) The visualization agent generates an LLM-based visualization plan and converts it into ready-to-plot chart data.

5) If run_automl is true, the Auto-AI agent also runs feature engineering, model selection, model training, and reporting.

6) The API returns references (version + endpoints) so the frontend can fetch schema, dashboard charts, and model results.

7) The /predict endpoint uses the stored model and configs to run inference on new rows.

2. Technologies Used

2.1 Programming Language and Runtime

- Language: Python 3.10.18.

- Runtime: Modal serverless container with a custom image built via modal.Image.

2.2 Core Libraries (Data & ML)

Data & ML stack:

- pandas: main data manipulation, file loading, and DataFrame operations.

- numpy: numerical operations, arrays, histograms, correlations.

- scikit-learn: preprocessing utilities, train/test split, metrics, RandomForest, GradientBoosting, KNN, LogisticRegression, LinearRegression.

- xgboost: XGBRegressor, XGBClassifier with early stopping.

- lightgbm: LGBMRegressor, LGBMClassifier with early stopping.

- catboost: CatBoostRegressor, CatBoostClassifier.

- imbalanced-learn (imblearn): SMOTE, SMOTENC, ADASYN, RandomOverSampler, RandomUnderSampler, SMOTEENN, SMOTETomek.

- StandardScaler: feature scaling in the feature engineering stage.

- PCA: dimensionality reduction for high-dimensional numeric features.

2.3 Infrastructure & API Libraries

Infrastructure and orchestration:

- modal: defines the serverless app, image, and mounted Volume; wraps FastAPI with @modal.asgi_app.

- FastAPI: web framework used to define HTTP endpoints for start, schema, dashboard, models, and predict.

- Uvicorn: ASGI server (used by Modal internally for running FastAPI).

- joblib: model serialization and saving of trained models.

- logging: structured logging across all modules for debugging and tracing.

- groq: client for calling LLM endpoints to plan feature engineering, model selection, visualization, and reporting.

2.4 Visualization & Reporting

- Custom JSON-based chart definitions and ready-to-plot data through visualizations_utils.py.

- Correlation matrices and top-correlation summaries.

- Markdown reports generated by the LLM and saved in the per-user reports folder.

The frontend is expected to render charts using the JSON structure output by the backend.

3. Codebase Structure

3.1 modal_app.py

modal_app.py is the main entrypoint and orchestration layer.

- Defines the Modal Image (dependencies), Volume (/root/data), and Modal app instance.

- Implements user- and version-aware orchestration functions:

- ingestion_agent

- preprocessing_agent

- visualization_agent

- feature_engineering_agenف

- model_training_agent

- auto_ai_agent

- prediction functions and utility loaders (load_deployed_model, etc.).

- Wraps a FastAPI app as fastapi_app **via @modal.asgi_app.**

- Defines the HTTP endpoints backing the public API.

3.2 Data_Ingestion.py

Data_Ingestion.py exposes a load_file(user_id, filename, url) function:

- Determines file type by extension (.csv, .xlsx/.xls, .json).

- Uses pandas.read_csv / read_excel / read_json.

- Drops unnamed index-like columns that start with 'Unnamed'.

- Logs dataset shape and throws errors for unsupported types or empty files.

- Returns a pandas DataFrame to the caller.

The ingestion agent in modal_app.py then saves this DataFrame into the user's folder.

3.3 utils_paths.py

utils_paths.py centralizes all filesystem path management. Functions include:

- BASE_DIR = '/root/data'.

- get_user_folder(user_id): root per-user folder.

- get_user_uploads(user_id): where raw uploaded files or references are kept.

- get_user_preprocessed(user_id): cleaned parquet files.

- get_user_engineered(user_id): engineered feature data.

- get_user_models(user_id): model files (.pkl) and metrics.

- get_user_visualizations(user_id): dashboard JSON and viz plans.

- get_user_dashboard_json(user_id): full path to dashboard_data.json.

- get_user_reports(user_id): report markdown files.

This ensures consistent, isolated storage for each user.

3.4 Preprocessing.py

Preprocessing.py implements the data cleaning pipeline used by preprocessing_agent:

- set_id_columns_as_index(df): sets ID-like columns as the index when appropriate.

- remove_duplicates(df): drops duplicate rows.

- handle_date_columns(df, target_column): converts date/time-like columns to pandas datetime, filters invalid ranges.

- handle_missing_data(df, col_threshold, row_threshold, target_column):

* Drops columns with too many missing values.

* Drops rows with too many missing values.

- detect_numeric_and_categorical(df, target_column): returns (numeric_continuous, numeric_categorical, categorical) and detects date columns.

- handle_outliers_iqr(df, categorical_columns, target_column, multiplier):

* If target is categorical, uses per-class IQR.

* Otherwise, uses global IQR to remove outliers.

- clean_data(df, target_column, config): orchestrates all steps according to a config object and returns cleaned df + inferred feature types.

- save_preprocessed(df, user_id): saves cleaned.parquet into the user's preprocessed directory.

3.5 Feature_Engineering.py

Feature_Engineering.py implements the feature engineering stage:

- expand_datetime_features(df, target_column): expands datetime columns into year, month, day, dayofweek and drops the original date columns (except the target if it is datetime).

- Encoding functions:

- one_hot_encode(df, categorical_columns)
- label_encode(df, categorical_columns)
- ordinal_encode(df, categorical_columns)
- frequency_encode(df, categorical_columns)
- target_encode(df, categorical_columns, target)

- polynomial_features(df, numeric_columns, degree): generates squared (and optionally cubic) terms.

- numeric_interactions(df, numeric_columns): generates pairwise product features.

- bin_numeric(df, numeric_columns, bins, strategy): discretizes numeric features using uniform or quantile bins.

- apply_pca(df, n_components): applies PCA to numeric features.

- handle_target_imbalance(df, target_column, strategy, ...): rebalances datasets using SMOTE, ADASYN, RandomOver/Under, SMOTEENN, SMOTETomek, etc.

- scale_numeric_features(df, target_column): scales numeric columns (excluding the target) using StandardScaler.

- run_feature_engineering(df, target_column, fe_config): orchestrates the above based on fe_config and returns a fully engineered DataFrame.

- save_engineered(df, user_id): persists engineered.parquet in the user's engineered folder.

All new column names are sanitized for compatibility with ML libraries.

3.6 Model_utils.py

Model_utils.py contains the model registry and training helpers:

- MODEL_CONFIG_REGISTRY: dictionary describing available models, their allowed parameters, and default parameters for both regression and classification.

- build_model_list_from_config(model_config, task_type): converts an LLM-specified config into a concrete list of models to train.

- agentic_train_test_split: robust train/test splitting with optional stratification, NaN handling, and guards for single-class issues.

- regression_metrics(y_true, y_pred): returns RMSE, MAE, R2.

- classification_metrics(y_true, y_pred): returns accuracy, precision, recall, F1.

- train_model: generic training function that:

- Instantiates the model.
- Configures special handling for XGBoost, LightGBM, and CatBoost (binary vs multi-class, early stopping).
- Trains the model with safe early stopping.
- Computes train/test metrics.
- Returns the trained model and metrics.

- Specific wrappers: train_random_forest_regressor, train_xgboost_classifier, etc., call train_model with correct metric functions.

- save_best_model(user_id, model, model_name): saves the trained model to the user's models folder.

- select_best_model(model_results, task_type): picks the best model based on RMSE or accuracy.

3.7 visualizations_utils.py

visualizations_utils.py provides utilities for dashboard data generation:

- json_safe(obj): safely converts numpy/pandas objects into JSON-friendly types.

- Chart data builders:

- generate_histogram_data(df, col)
- generate_bar_data(df, col, y_col)
- generate_scatter_data(df, x_col, y_col)
- generate_box_data(df, y_col, x_col)
- generate_pie_data(df, col)
- generate_heatmap_data(df, numeric_cols)

- Statistical aggregations:

- compute_summary_statistics(df, numeric_continuous, numeric_categorical, categorical)
- compute_correlation_matrix(df, numeric_continuous)
- compute_categorical_distributions(df, categorical_columns, numeric_categorical)

- build_ready_charts_from_plan(df, viz_plan, ...): converts an LLM-produced viz_plan into ready-to-plot chart entries.

- generate_dashboard_data(...): saves dashboard_data.json including metadata and charts.

4. End-to-End Pipeline Steps

4.1 Step 1 — Data Ingestion

4.1.1 Inputs:

- user_id: identifies the user.

- filename: original file name.

- url: path or URL pointing to the uploaded dataset.

4.1.2 Processing:

- modal_app.py calls load_file(user_id, filename, url).

- The file is read with pandas based on its extension.

- Unnamed index columns are dropped.

- Basic validation ensures the DataFrame is not empty.

4.1.3 Outputs:

- A pandas DataFrame returned to the ingestion agent.

- The DataFrame is saved to the user's uploads or preprocessed directory for traceability.

4.2 Step 2 — Preprocessing

4.2.1 Inputs:

- Raw DataFrame from ingestion.

- target_column name.

- Preprocessing configuration (missing value thresholds, outlier strategy, etc.).

4.2.2 Processing:

- set_id_columns_as_index: index set on ID-like columns.

- remove_duplicates: duplicate rows removed.

- handle_date_columns: convert date/time-like columns and clip invalid dates.

- handle_missing_data: drop columns and rows exceeding configured null thresholds.

- detect_numeric_and_categorical: infer numeric_continuous, numeric_categorical, categorical, and date columns.

- handle_outliers_iqr: remove outliers globally or within target categories.

4.2.3 Outputs:

- cleaned DataFrame.

- numeric_continuous, numeric_categorical, categorical, date_columns lists.

- cleaned.parquet saved per user.

- metadata.json capturing type information and configuration.

4.3 Step 3 — Visualization & Dashboard Data

4.3.1 Inputs:

- Cleaned DataFrame.

- Inferred feature types.

- Optional target_column.

4.3.2 Processing:

- The visualization agent collects schema and summary stats.

- It calls the LLM with a prompt describing the dataset and task to obtain a viz_plan of exactly 6 charts.

- build_ready_charts_from_plan turns each planned chart into concrete data using the helpers in visualizations_utils.py.

- Correlation heatmap is always included among the charts.

4.3.3 Outputs:

- dashboard_data.json containing:

- columns, target_column, feature type lists.
- charts: list of chart specifications with ready data.

- This file is consumed by the /dashboard API.

4.4 Step 4 — Feature Engineering (AutoML mode only)

4.4.1 Inputs:

- Cleaned DataFrame.

- target_column.

- fe_config: produced by LLM in AutoML mode.

4.4.2 Processing:

- expand_datetime_features: expand datetime into numeric components.

- Coding strategy: one_hot / label / ordinal / frequency / target encoding applied based on fe_config.

- Optional polynomial, interaction, binning, and PCA steps applied.

- Target imbalance handled using the chosen strategy (random over/under, SMOTE-family algorithms).

- Numeric features scaled via StandardScaler if enabled.

- Final column names sanitized.

4.4.3 Outputs:

- engineered DataFrame.

- engineered.parquet saved to per-user directory.

- fe_config.json saved alongside for reproducibility.

4.5 Step 5 — Model Training & Selection (AutoML mode only)

4.5.1 Inputs:

- Engineered DataFrame.

- target_column.

- model_config: produced by LLM (set of models and hyperparameters).

4.5.2 Processing:

- agentic_train_test_split splits the data into train and test (stratified for classification when feasible).

- For each enabled model in model_config:

- The specific train_* function (e.g., train_xgboost_classifier) is called.
- train_model configures, trains, and evaluates the model.
- Metrics (train and test) are collected.

- select_best_model picks the best-performing model based on:

- Lowest RMSE for regression tasks.
- Highest accuracy for classification tasks.

- The best model is serialized with joblib and saved via save_best_model.

4.5.3 Outputs:

- One or more model_v{version}_{model_name}.pkl files.

- Metric summaries per model (train/test metrics).

- Best model metadata returned to auto_ai_agent and later exposed via /models.

4.6 Step 6 — Reporting

4.6.1 Inputs:

- cleaned_summary: shape, missing overview, feature types.

- preprocess_config and fe_config.

- model training results (best model and leaderboard).

4.6.2 Processing:

- auto_ai_agent calls the LLM with a structured system prompt.

- The LLM generates a Markdown report summarizing:

- Data structure and quality.
- Feature engineering decisions.
- Model comparison and best model explanation.
- Any imbalance handling or caveats.

4.6.3 Outputs:

- report_v{version}.md saved under the user's reports folder.

- The report is returned via /models for frontend display.

4.7 Step 7 — Prediction

4.7.1 Inputs:

- user_id and version.

- Optional model_name.

- features: single-row feature mapping.

4.7.2 Processing:

- The prediction helper loads the stored model, preprocess metadata, and fe_config for that version.

- Incoming features are converted into a one-row DataFrame.

- Missing numeric values are coerced and filled; non-numeric are converted to strings.

- A dummy target column is added only for compatibility with feature engineering.

- run_feature_engineering is re-applied (with safeguards for target encoding).

- Final features are realigned to the exact set used during training (reindex with fill_value=0).

- The model's predict (and predict_proba for classifiers) is called.

4.7.3 Outputs:

- For regression: predicted numeric value and possibly residual-like info.

- For classification: predicted class plus class probabilities.

- Returned as JSON from the /predict endpoint.

5. API Layer and Input Criteria

5.1 POST /start

5.1.1 Purpose:

- Trigger ingestion, preprocessing, visualization, and optionally AutoML for a user dataset.

5.1.2 Request Body:

- user_id (string, required): identifier for the user; must be non-empty and consistent across future calls.

- filename (string, required): file name including extension (.csv/.xlsx/.xls/.json).

- url (string, required): path or URL where the backend can read the file.

- target_column (string, required): must match exactly one column in the ingested DataFrame.

- run_automl (boolean, required):

- false → analytics-only run (no model training).
- true → full AutoML (feature engineering + model training + report).

- objective (string, optional, default 'auto'): abstract objective string; can be used by the LLM but is optional.

5.1.3 Validation & Errors:

- If the file extension is unsupported or cannot be read → error.

- If the DataFrame is empty → error.

- If target_column is not found → error.

- On success, the response includes user_id, version, task type, and endpoint URLs.

5.2 GET /schema/{user_id}/{version}

5.2.1 Purpose:

- Return the cleaned schema and column types used for modeling and prediction.

5.2.2 Input Criteria:

- user_id (path): must correspond to an existing user folder.

- version (path, integer): must reference an existing cleaned_v{version}.parquet and metadata file.

- No request body.

5.2.3 Output:

- target_column name.

- Schema list: each item includes column name, pandas dtype, and logical input_type.

This schema should be used by the frontend to build input forms for /predict.

5.3 GET /dashboard/{user_id}/{version}

5.3.1 Purpose:

- Provide the ready-to-render chart configuration and data for the frontend dashboard.

5.3.2 Input Criteria:

- user_id (path): identifies the user.

- version (path): identifies the pipeline run.

- No body.

- Requires a dashboard_data_v{version}.json or fallback dashboard_data.json in the user's visualizations directory.

5.3.3 Output:

- charts: list of chart objects, each including type, axes, title, width, order, and data arrays.

- metadata: target_column, feature type lists, etc., as persisted by generate_dashboard_data.

5.4 GET /models/{user_id}/{version}

5.4.1 Purpose:

- Expose model training results and the business-facing report for a given version.

5.4.2 Input Criteria:

- user_id (path).

- version (path).

- Requires run_automl = true to have been used for that version.

- Models folder must exist for the user with model_v{version}_*.pkl and metrics.

5.4.3 Output:

- task: classification or regression.

- target_column.

- best_model: name, params, train/test metrics.

- models: full leader board of all candidate models.

- report_markdown: data and model summary for frontend display.

5.5 POST /predict

5.5.1 Purpose:

- Run single-row inference using the best or specified model from a previous AutoML run.

5.5.2 Request Body:

- user_id (string, required): must match the user that owns the model.

- version (integer, required): must have a successfully trained model.

- model_name (string, optional): if provided, must match one of the saved models for that version; otherwise the best model is auto-selected.

- features (object, required): a mapping from original column names (per /schema) to values for a single row.

5.5.3 Input Criteria & Conventions:

- It is recommended to send values for all important columns as indicated by /schema.

- Missing features are handled by:

  - creating any missing engineered columns during alignment and filling them with 0.

- Extra features are safely ignored after alignment.

- Target column should NOT be sent; the backend adds a dummy target internally when FE requires it.

- Numeric values are parsed with pandas.to_numeric; invalid values become NaN and then 0.

5.5.4 Output:

- For regression tasks: JSON with predicted_value and supporting metadata.

- For classification tasks: JSON with predicted_class and class probabilities.

- Errors are returned if the model or version cannot be found.

6. Deployment & Environments

6.1 Modal Image and Volume

- modal_app.py defines an Image with all required Python libraries pre-installed.

- A Modal Volume is mounted at /root/data to persist user files, models, logs, and dashboards across container restarts.

- Functions such as ingestion_agent, preprocessing_agent, auto_ai_agent, etc., run inside this container and share the same Volume.

- The FastAPI application is exposed via @modal.asgi_app as fastapi_app.

6.2 Logging and Monitoring

- All modules use the logging library to record progress, decisions, and errors.

- LOGS_DIR is set to /root/data/logs in modal_app.py so logs are persisted in the same Volume.

- Log messages include preprocessing statistics, feature counts, outlier removal counts, resampling stats, and model performance.

6.3 User Isolation and Versioning

- User isolation is achieved by maintaining separate folders for each user_id under /root/data/users.

- Versioning is implemented by naming files with v{version} and incrementing the version for subsequent /start runs.

- This allows multiple runs per user, each with its own preprocessed data, engineered features, models, dashboards, and reports.