AutoML Assistant for Tabular Data — Project Guide

A complete, beginner■friendly walkthrough of the repository, agents, RAG store, fallbacks & safety, and the full runtime flow from CSV input to a polished Markdown + PDF report.

What this project is

- An “AutoML assistant” for tabular data.
- You give it a CSV (or a Kaggle dataset), choose the target column, and it will:
  - Profile your data (columns, types, missingness, class balance).
  - Plan a modeling pipeline (preprocessing + model).
  - Retrieve house rules & examples via RAG (from a local Chroma vector DB) to guide the LLM.
  - Generate step■by■step Python code with an LLM (a clean “code map”).
  - Execute that code safely to train a model (or fall back to strong baselines).
  - Evaluate on a hold■out split (accuracy/F1 or RMSE/R^2).
  - Summarize everything into Markdown and a PDF, with embedded/linked plots.
- Safety by default: sandboxed execution, IO/tuning guards, and a deterministic baseline if LLM code under■performs.

Big concepts used (and why)

- LangGraph orchestration keeps the process deterministic and modular—each “agent” focuses on one job.
- RAG with Chroma DB grounds the LLM with rules/patterns/snippets for consistent, policy■compliant generation.
- LLM Codegen writes preprocessing/modeling/plotting code; we clean it and run it in a guarded environment.
- Scikit■learn baselines act as a robust fallback (LogReg/RandomForest; Ridge/RFR).
- ReportLab builds a polished PDF with page■broken, scaled plots.
- Streamlit is the UI for loading data, selecting a target, running, viewing, and downloading outputs.

Repository tour — every file & its role

app.py (Streamlit app)

- Upload CSV or load Kaggle; choose target; run AutoML.
- Guards: file■size limit and environment flags (ALLOW_IO, ALLOW_TUNING, ALLOWED_DATA_DIR).
- Invokes create_graph(); collects outputs in session state.
- Displays profile, metrics, baseline CV, generated code map, and plots (step_*n*_plot.png).
- Builds Markdown + ReportLab PDF with safe image sizing and page breaks.

execution.py (CLI helper)

- Run the graph headlessly outside Streamlit—useful for batch/CI.

ingest_rules.py (RAG ingestion)

- Reads rules_data.json and embeds them with OpenAI Embeddings into a local Chroma collection (“rules”).
- Run whenever rules change to refresh the vector store.

main_runner.py (batch entry)

- Headless runner mirroring Streamlit’s flow.

prompts.py

- Houses planning/code■gen prompts and safety/contract rules (e.g., save plots as step_*n*_plot.png).

llm_codegen.py

- Helper around ChatOpenAI to synthesize Python code for a step; strips backticks/bullets; normalizes newlines; used by pipeline_builder.

Agents (the LangGraph 'nodes')

agents/graph_orchestrator.py

- create_graph() wires the agents in order: intake → profile → planning → retrieval (RAG) → pipeline_builder → execution → evaluation → summary.
- Maintains shared state dict as it flows between nodes.

agents/intake_agent.py

- Normalizes DF + target, checks basic shape, seeds an initial profile shell.

agents/profile_agent.py

- Computes schema, dtypes, numeric/categorical columns, missingness, target type, and class balance; writes state['profile'].

agents/planning_agent.py

- LLM drafts an ordered step plan using prompts.py contracts.

agents/retrieval_agent.py (RAG with Chroma)

- Fetches relevant rules/snippets from Chroma and merges them into state['rules']/state['context'].

agents/pipeline_builder.py

- Turns plan + RAG context into executable code; cleans it; enforces IO/plot contract; numbers plot files; builds state['code_map'].
- Implements BaselineSelector (LogReg/RandomForest or Ridge/RFR) for a deterministic fallback.

agents/execution_agent.py

- Safely executes generated code with restricted globals and honors ALLOW_IO/ALLOW_TUNING/ALLOWED_DATA_DIR; captures plot info; returns model with _origin='ai' on success.

agents/evaluation_agent.py

- Creates a fresh hold■out; evaluates AI and baseline; picks the winner; records metrics and confusion matrix/labels for classification.

agents/summary_agent.py

- Builds the final Markdown: Dataset, Quick EDA, Final Model (Origin + Algorithm), Hold■out Metrics, Code Artifacts, Visualizations, and Reproducibility Notes.

RAG store & assets

- **rules_data.json** with curated rules/snippets;
- **chroma_store/** is the local persistent Chroma DB directory created by ingest_rules.py.

Fallbacks & safety — clearly spelled out

- Code sandboxing via execution_agent with restricted builtins and safety flags.
- Deterministic baselines ensure usable results even if LLM code fails or is weak.
- Plots saved to step_n_plot.png and later page■broken & size■constrained in the PDF.
- Correct task/metrics selection via profile_agent + evaluation_agent.
- MD/PDF cleanliness: inline base64 for UI; sanitized file links for downloads; proper image insertion in PDF.

End■to■end runtime flow

A) User input

- Upload CSV or provide Kaggle slug/URL.
- Pick target column.

B) LangGraph pipeline

- intake → profile → planning → retrieval (RAG) → pipeline_builder → execution → evaluation → summary.

C) Streamlit output

- Displays profile/metrics/plots; provides Markdown & PDF downloads with correct image handling.

How to extend or customize

- Update rules_data.json and re■run ingest_rules.py to change policies.
- Adjust prompts.py contracts for stricter codegen or different model families.
- Modify baselines in pipeline_builder.py to suit your domain.
- Tweak ReportLab styling or add a cover page in app.py’s PDF section.

TL;DR

- Plans → retrieves rules → generates code → runs safely → evaluates → reports.
- RAG with Chroma keeps the LLM grounded; baselines guarantee usable results; Streamlit + ReportLab produce a clean UI and outputs.