ChatGPT

Autonomous Data Science Assistant Project: Getting Started and Implementation Plan

1. Define the Project Scope and Goals

Begin by clearly outlining what your **Autonomous Data Science Assistant** (AutoML Agent) is expected to do. In essence, this project involves creating an AI-driven tool that can take a raw dataset (e.g. a CSV file and a specified target column) and perform the end-to-end data science workflow with minimal human input [1]. The agent should be capable of:

- **Data Ingestion & Understanding:** Loading the dataset and identifying the problem type (classification vs. regression) based on the target variable.
- **Exploratory Data Analysis (EDA):** Analyzing feature distributions, correlations, and summary statistics; detecting issues like missing values or outliers; and producing initial visualizations (histograms, box plots, etc.).
- **Data Cleaning & Preprocessing:** Handling missing data (e.g. imputation or removal), encoding categorical variables, scaling or transforming features if needed (for example, log-transforming a skewed distribution), all with minimal manual guidance.
- **Automated Model Training:** Trying multiple modeling approaches to find the best model. This could involve using an AutoML library to automatically select algorithms and tune hyperparameters, or programmatically training a few candidate models (e.g. Random Forest, XGBoost, Neural Network) for comparison. The agent should evaluate model performance (accuracy, RMSE, etc.) and select the winner.
- **Results Summarization:** Compiling a report of findings – including key insights from EDA (with plots), what preprocessing was done, which model performed best and why, and the performance metrics. Ideally, the agent uses an LLM to **narrate** these results in plain English (e.g. explaining that *"the income feature was highly skewed, so a log transform was applied,"* or *"the Random Forest achieved 95% accuracy, likely due to capturing nonlinear effects"*).

Defining these capabilities sets a clear target for your implementation. It also helps to explicitly list what the agent will **not** cover to keep the project scope manageable (for example, you might limit scope to tabular data and supervised ML tasks only, excluding time-series or NLP tasks). This clarity will guide all subsequent steps and prevent scope creep. The ambition is to demonstrate an **"AI mini-data-scientist"** that autonomously handles a project from start to finish, showcasing cutting-edge AutoML and AI orchestration skills – a highly impressive feat in 2025 when many companies are experimenting with autonomous AI agents [2].

2. Set Up the Tech Stack (Using Free Tools)

With the scope defined, assemble the tools and environment you'll use. Since cost is a concern, prioritize free and open-source resources:

- **Programming Environment:** Python will be the core language (which is perfect given your background). Use a Jupyter Notebook for development and demonstration. This allows interactive development and easy display of dataframes, charts, and Markdown reports. For a more polished interface later, consider a simple web app framework like **Streamlit** (pure Python,

great for data apps) to accept file uploads and display results – but this can be a second step once the core logic is working.

- **Data and ML Libraries:** Install standard data science libraries – pandas for data manipulation, numpy for numerical operations, and matplotlib/seaborn for plotting. Ensure you have scikit-learn as well, since many AutoML tools build on it. For the AutoML component, you can use **Auto-sklearn**, which is open-source and automates model selection and hyperparameter tuning [3]. Auto-sklearn is essentially a drop-in replacement for a scikit-learn model – you initialize an `AutoSklearnClassifier` or `AutoSklearnRegressor` and call `.fit()` on your data, and it takes care of trying many models under the hood [4]. It also does some preprocessing for you (e.g. automatic one-hot encoding of categorical features and other pipeline steps) [5]. Alternative free AutoML libraries include **TPOT** (uses genetic programming to find good pipelines) or **H2O AutoML** (which can train a suite of models and ensemble them). Any of these can work, but Auto-sklearn is a good start for Python integration.
- **Large Language Model (LLM):** A crucial part of this project is using an LLM to drive the agent's reasoning and even generate code. Since you need a free solution, you have a few options: if OpenAI's API (e.g. GPT-3.5 or GPT-4) is accessible to you with a free trial or credits, that would be the simplest path (cloud-based and powerful). Otherwise, consider an open-source LLM that you can run locally. Models like **Code Llama** (a version of Meta's Llama2 tuned for code) or **StarCoder** by Hugging Face could be suitable for code-generation tasks without cost. Keep in mind running these locally requires sufficient computing resources (a decent GPU or using a smaller model on CPU). You can also use online hosted inference APIs for open models (Hugging Face offers free limited access to some models). In summary, choose an LLM that can reliably understand instructions and produce Python code. Using a cloud API (if free) might yield better results due to the advanced capabilities of models like GPT-4, but an open model ensures you won't run into usage limits or costs.
- **Agent Orchestration Framework (optional):** While not strictly required, using a framework like **LangChain** can greatly simplify building an agent that interacts with the LLM and tools. LangChain provides structures for prompt templates and even the concept of tools/agents that can decide actions. For example, LangChain can let you define a *Python REPL tool* which the LLM can use to execute code, enabling a loop of "think in natural language → generate code → run code → observe output → continue". This is exactly the kind of reasoning loop an autonomous data science agent would need. You can certainly implement the logic yourself (via manual prompts and `exec()` calls in Python), but LangChain or similar frameworks (like Microsoft's Guidance or the Python `openai` library with function calling) can handle a lot of the heavy lifting. Given your limited time and the desire to demonstrate orchestration skills, integrating LangChain could be impressive. Just remember it's free and open-source, so it fits your constraints.

Make sure to test your environment by running a simple example: for instance, try a quick Auto-sklearn run on a small dataset to ensure it's working, and a quick OpenAI API call or local model invocation to ensure the LLM is set up. Getting these pieces working early will avoid surprises later.

3. Plan the Agent's Workflow and Logic

Now design how the agent will actually proceed from raw data to final report. This means breaking the overall task into a sequence of concrete steps or sub-tasks that the agent (and underlying functions) will execute. A logical workflow for the autonomous assistant might look like this:

1. **Dataset Input & Problem Identification:** The agent accepts the dataset (file path or uploaded file) and the name of the target variable. The first task is to determine the nature of the problem. This can be done by checking the target column: if it's numeric (regression) or categorical

(classification) – or even binary classification vs multi-class. This step sets the stage for what kind of analysis and models to consider.

2. **Basic Data Exploration:** Next, the agent should quickly scan the dataset. Compute basic info like the number of rows and columns, data types of each column, and a glimpse of the first few rows. Also, identify any obvious issues: e.g., how many missing values in each column, or if any column has a suspicious single value for all entries, etc. This gives the LLM (or your code) an overview to reason about.
3. **LLM-Driven Planning:** With the initial dataset summary in hand, have the LLM generate a plan of action. You might prompt the LLM with something like: *"You are an AI Data Science assistant. We have a dataset with X rows and Y columns. The target is 'LoanDefault' which is categorical. Many features are numeric, one is text. Plan an analysis: what steps should we do (EDA, preprocessing, modeling)?"* The LLM should output a series of steps it thinks are appropriate. For example, it might say: *"1. Plot distributions of each numeric feature. 2. Check class balance for the target. 3. Impute missing values (perhaps median for numeric, mode for categorical). 4. Encode categorical variables. 5. Try a Random Forest, an XGBoost, and a Logistic Regression. 6. Evaluate with cross-validation and choose the best model. 7. Generate a summary report with findings."* This LLM-generated plan can guide the next stages. (If the LLM's plan seems off or too ambitious, you can always adjust or constrain it with a better prompt. In practice, you might write a prompt that already biases it to the steps you know you can implement, to avoid getting unusable suggestions.)
4. **Exploratory Data Analysis (EDA):** Follow the plan for EDA. This likely includes: for each numeric feature, plotting a histogram (to see distribution shape and outliers); for categorical features, plotting bar charts of value counts; maybe plotting a correlation matrix heatmap for numeric features to see multicollinearity; possibly cross-tabs or boxplots comparing features to the target (to see which features are predictive). You can implement a library like `pandas_profiling` (now called **YData Profiling**) to automate some of this EDA, but since you want to showcase the agent's *autonomy*, it might be better to have the agent do it step by step, possibly via the LLM generating the code for each plot. For example, the agent could loop through numeric columns and for each one, prompt the LLM: *"Generate Python code (using matplotlib/seaborn) to plot the distribution of the feature 'Age'"*, execute the code to produce the chart, and save it. This showcases the LLM writing code for EDA tasks. (If using LangChain, you might instead have the LLM in an agent role where it decides "I should create a histogram of Age" and directly uses a plotting tool.) Ensure that after generating these plots, you collect some descriptive statistics too (mean, median, std, etc. for numeric features, and frequency tables for categorical). These stats and plots will be inputs for the report.
5. **Data Cleaning & Preprocessing:** Based on the EDA findings, the agent should address data quality issues. This could include: imputing missing values (the strategy could be decided by the LLM or you could have a default – e.g., median for numeric, mode for categorical, or drop if too many missing); removing or winsorizing outliers if they would skew the model; encoding categorical variables into numeric (one-hot encoding for nominal categories, or ordinal encoding if appropriate). If certain features are highly skewed, the agent might decide (or the LLM might suggest) applying transformations (like log or Box-Cox) to make them more Gaussian. Another preprocessing step is feature selection or dimensionality reduction, though for an MVP you might skip this unless the dataset is very high-dimensional. **Auto-sklearn** actually automates many of these steps internally – for example, it will one-hot encode categoricals and scale features as needed [5]. Still, implementing some of it explicitly is a good learning exercise and can be guided by the LLM's reasoning. You can again leverage the LLM here: e.g., *"We have 3 categorical columns: Color, State, and ProductType. Write Python code to one-hot encode these features using pandas or scikit-learn."* Then execute the generated code. The combination of your oversight and the LLM's code generation can handle this stage.

6. **Model Selection & Training (AutoML):** This is the core of the AutoML aspect. Now that data is prepared, the agent will either invoke an AutoML library or try a set of models. Using **Auto-sklearn** is straightforward – you initialize `AutoSklearnClassifier` (for classification) or `AutoSklearnRegressor` (for regression) and call `model.fit(X_train, y_train)` [6]. Under the hood, Auto-sklearn will evaluate many pipelines and models (using techniques like Bayesian optimization and ensemble selection) to find a high-performing model for the data [3]. It saves you from manual algorithm selection and hyperparameter tuning, which is exactly the point of the project. You can set a time limit for the autoML search (to avoid running forever – e.g. 5 or 10 minutes for experimentation) [7]. Alternatively, if you prefer not to rely entirely on AutoML black-box, you could have the agent (via LLM) explicitly decide on a few algorithms to try. For example, the LLM might reason: *"Since this is a classification task with mostly numeric features, I will try Random Forest, XGBoost, and a Logistic Regression as a baseline."* You can then programmatically train these three models (using scikit-learn or XGBoost library) and compare their performance. This approach shows more of your manual ML skills but is less "fully automated" than using something like H2O's complete AutoML which tries dozens of models. A middle ground could be using **TPOT**, which evolves pipeline combinations of preprocessors and models – but note that TPOT can be slow. For an MVP, Auto-sklearn is a good choice for simplicity.
7. **Model Evaluation & Selection:** Whichever route you choose, the agent should evaluate models on a validation set or via cross-validation. If using Auto-sklearn or H2O, they usually keep an internal validation or provide a leaderboard of models. If you manually train models, evaluate each with appropriate metrics (accuracy, F1, or AUC for classification; RMSE or MAE for regression) using a hold-out set or CV. Identify the best model from these results. This best model can be saved or kept for the final report. Collect any additional info that might be useful for the report, such as feature importance from a tree-based model or coefficients from a linear model, to help interpret the model's behavior.
8. **Autonomous Reasoning & Error Handling:** Throughout steps 4–7, where the LLM is generating code or making decisions, you need to manage the fact that LLM outputs can sometimes be wrong or produce errors. Implement a loop for error handling: after you execute an LLM-generated code snippet, catch any exceptions. If an error occurs, feed the error message back to the LLM with a prompt like *"The code resulted in the following error, please fix it: {error_traceback}"*. This allows the LLM to debug and produce a corrected snippet [8]. For example, if the LLM tries to plot a histogram using a column name it assumed, but your actual dataframe column name is different, the execution will error – you capture that and prompt the LLM to correct itself (perhaps your prompt can remind it to use the exact column names from the data summary you provided). This iterative generate-and-test approach is key to a robust autonomous agent [9] [10]. If using LangChain, a lot of this can be handled by an Agent with a tool: the agent's LLM will decide an action like "use Python tool to do X", and if that fails, the next LLM response can incorporate the error message. In effect, the agent debugs itself. By designing the agent to handle such loops, you increase reliability. Also, put some guardrails: for instance, you might sandbox the execution environment (to ensure the LLM doesn't execute dangerous code). If using `exec()` in Python, be cautious and maybe strip out built-in functions that could be harmful. However, if you keep the tasks focused on data manipulation and modeling, the risk is low. Logging every step is a good idea – have the agent print or record each decision and action (like "LLM decided to impute missing Age with median" or "LLM generated code to train RandomForest"). This not only helps in debugging the agent, but also is great to include in your project report to illustrate the agent's autonomous reasoning.
9. **Result Synthesis and Report Generation:** After the modeling is done, the agent should compile all the insights and results into a coherent report. This is where you heavily use the LLM for its strength in language generation. Gather the key pieces of information: EDA insights (e.g. distributions, any interesting correlations), what preprocessing was done, how many models

were tried and which was best, and the performance of the best model. You might structure the report as sections: **Data Overview**, **EDA Findings**, **Modeling Approach**, **Results**. For each section, feed relevant data to the LLM and ask it to draft a summary. For example, *"Summarize the data exploration findings: we found Age is skewed right (see histogram), Income has 5% missing values, etc. Explain any actions taken."* The LLM can output a paragraph that reads like a human analyst's commentary. Similarly for the modeling results: *"Explain the model results: which model won, what was its accuracy, and possible reasons. Also mention feature importance if applicable."* The narrative the LLM produces will make the report easy to read for a hiring manager and shows that you can leverage AI to generate human-like insights [11]. Be sure to include the visualizations in the report – since this might ultimately be a Jupyter Notebook or a PDF, you can have the images (histograms, etc.) embedded with captions that the LLM or you write. If using a Notebook, you could simply display the plots in sequence with markdown text in between. For a PDF or markdown report, you might save the plots as image files and reference them. The **final report** should tell the story of the data and the model's performance in a concise way, as if a data scientist wrote it. This demonstrates the *auto-report-generation* aspect of your project.

10. **User Interaction (Optional for MVP):** If you aim to make this a bit interactive (bonus points, since you mentioned interest in a web app), consider wrapping the agent in a simple interface once the core logic works. Using **Streamlit** is a great choice since you only know Python – you can create a small app where the user uploads a CSV and enters the target column name, then your backend code (the agent) runs and displays the resulting analysis and report. Streamlit can display DataFrame heads, charts, and text easily. This turn it into a demo-able web application without you having to write any JavaScript or HTML. That said, a polished UI is not necessary to prove the concept – a well-documented Jupyter Notebook showing a couple of example runs (maybe one on a classification dataset like Titanic, and one on a regression dataset like Boston Housing) can be just as effective for your resume/portfolio. The key is that you can **demonstrate the agent working autonomously** through those examples.

By outlining the workflow as above, you have a clear step-by-step game plan. Each of these steps can be mapped to code modules or functions you'll implement.

4. First Steps: Environment and Data Setup

With the plan in mind, start implementing step by step. **The first thing to do is set up your development environment and get a simple end-to-end run without the LLM integration, to ensure the ML pipeline works.** For example, pick a sample dataset and do a manual (non-LLM) run of loading data -> basic cleaning -> AutoML training -> output results. This will flush out any issues with libraries and give you baseline code to later hand over to the LLM agent.

- **Environment Setup:** Create a new Python environment if needed (using venv or Conda) and install the necessary packages: `pandas`, `numpy`, `matplotlib`, `seaborn`, `scikit-learn`, `auto-sklearn`, `tpot` (if you want to try), `streamlit` (if you plan to add UI), and for LLM integration either `openai` (for OpenAI API) or libraries for running local models (like `transformers` from Hugging Face). If using LangChain, install `langchain` and ensure you have access to an LLM model or API.
- **Data Selection:** Choose one or two **example datasets** to work with initially. Classic choices: the Titanic survival dataset (a classification with a mix of numeric and categorical features and some missing values) – great for demonstrating cleaning and model training; or the Boston Housing/ California Housing dataset for regression; or any dataset you're comfortable with. Using well-known datasets is useful because you can sanity-check the agent's results against known benchmarks or expectations (for instance, you know roughly the accuracy people get on Titanic,

or the important features in housing prices). Load one of these datasets into your environment to use as a test case throughout development.

- **Basic Pipeline without LLM:** Code up a quick script or notebook cells that perform a **manual version** of the pipeline on the dataset. For example:
- Read the CSV (using `pandas.read_csv`).
- Do a little manual EDA: `df.info()`, `df.describe()`, maybe a couple of static plots.
- Handle missing values in a straightforward way (just as a placeholder for now).
- If classification, encode categoricals (pandas `get_dummies` is quick for this).
- Split into train/test.
- Run Auto-sklearn for, say, 60 seconds on the training set.
- After it finishes, get the best model (Auto-sklearn has `model.show_models()` or `model.leaderboard()` to inspect what it found). Evaluate it on the test set and print accuracy/MSE.
- Print or jot down a few observations like "X features, Y rows, best model was RandomForest with accuracy 0.78" etc.

Doing this manually ensures all the pieces (data handling, autoML, etc.) are working. It also gives you insight into what parts are time-consuming or tricky – for instance, Auto-sklearn might take a while or might require a specific data format; maybe you find you need to install a system library for it to work. Better to resolve those now. Moreover, this manual run provides a **template** for the LLM to follow. You now know the general sequence of code that needs to be executed, which will help when you start prompting the LLM to generate code for these tasks.

5. Implement Modular Functions for Each Task

Refactoring the above pipeline, start writing **helper functions** or classes for each major component of the workflow. This makes it easier to let the LLM agent call these functions, or to debug parts in isolation. Some suggested functions/modules:

- `load_data(filepath) -> DataFrame`: reads the CSV and maybe prints basic info.
- `explore_data(df) -> dict_of_insights`: generates EDA results. For instance, calculate summary stats, maybe return a dictionary like `{"num_rows": ..., "num_cols": ..., "columns": [...], "missing_counts": {...}, "feature_types": {...}, "plots": [list of filepaths to generated plots]}`. (In practice, generating and saving plots to files can be part of this function. You might have this function call subroutines like `plot_histograms(df)` internally.)
- `clean_data(df) -> df_cleaned`: handles missing values and encoding. Perhaps allow this to be guided by a strategy input (so you can apply different strategies or what the LLM suggests). For example, parameters like `strategy_numeric='median'` etc., or simply decide within the function.
- `train_models(df, target_column) -> model_or_results`: this could either wrap an AutoML call or orchestrate training multiple models. If using Auto-sklearn, this function would prepare `X, y`, split data (if you want a hold-out set aside from what AutoML does internally), then fit the `AutoSklearnClassifier` and return the fitted model along with evaluation metrics. If doing manual model comparisons, this function would train each model and compute metrics. Either way, have it return the **best model** and its performance.
- `generate_report(insights, model, metrics) -> str_or_markdown`: takes the outputs from exploration and modeling and uses them to produce a nicely formatted report. This is where you will likely call the LLM to help craft the narrative. The `insights` could be the

dictionary from `explore_data`, and `metrics` could include things like best model name and its score, etc. The output might be a Markdown string that you can display or save.

Designing your code in these chunks is useful for two reasons: (1) you can unit-test or run them independently (e.g., run `explore_data` on a DataFrame and see what it returns, make sure plots are saved correctly), and (2) when you integrate the LLM agent, you could either invoke these functions directly from the agent or even have the LLM decide which function to call when (if using an advanced agent setup). For example, LangChain could allow the LLM to choose a tool called "explore_data" which triggers that Python function. The functions encapsulate the heavy lifting, while the LLM provides the brain to decide the sequence and interpret results.

Implement and test each function with your example dataset. This is pure coding work – for instance, ensure `clean_data` actually fills NaNs and encodes categoricals correctly by checking `df_cleaned.isnull().sum()` or the new columns after encoding. For the `train_models` step, try a very short Auto-sklearn run to see it returns a model. You might save a simple artifact like the model's score or the model object itself. At this point, you're building a conventional pipeline, which is fine – the "autonomy" will come when hooking up the LLM.

6. Integrate the LLM for Planning and Code Generation

With a working pipeline in modular form, now focus on the AI orchestration aspect. The goal here is to have an LLM "drive" those functions or even generate code for parts of them, making the process autonomous. There are a couple of integration patterns you can choose (or even combine):

- **Plan & Execute:** Use the LLM to create a high-level plan (as mentioned in step 3), then programmatically execute that plan. For example, after the LLM provides a plan (perhaps as a list of steps), map those steps to your functions. If the LLM's plan says "Do EDA", you call your `explore_data` and maybe then feed some results back to the LLM for the next step. This approach is simpler but somewhat rigid – you're interpreting the plan yourself.
- **LLM Agent with Tools:** A more advanced (but very cool) approach is to set up the LLM as an **agent** that can decide on actions and use tools. In this context, your Python functions (exploration, cleaning, training, etc.) become "tools" that the agent can invoke. LangChain supports this pattern: you define a Tool with a name, description, and a function. For instance, a tool named "PreviewData" that calls a function to show `df.head()` and returns the output, or "TrainAutoML" that calls your training function. The LLM is given access to these tools and a goal (e.g., "Analyze this dataset and report results"). It will then dynamically choose a tool, supply arguments, and you get the result, and it continues the conversation. For example, the agent might think: *"To start, I should examine the data."* It calls the PreviewData tool, gets output (like column names and some stats), then the LLM might next decide *"Now I should perform EDA."* It calls an EDA tool (which perhaps returns paths to plots or text summary). Then it might say *"I see some missing values; I should clean data."* – calls cleaning tool, and so on, until it decides to train models and then summarizes. This ReAct (Reason+Act) loop continues until the task is done [12] [11]. The benefit of this approach is that it truly shows an autonomous chain of thought driven by the LLM, and you only have to monitor or correct if it goes off track. The downside is it's more complex to implement and requires careful prompt engineering to keep the agent on task (you have to describe what each tool does and when to use it in the prompt). Given your project's ambition, this is worth attempting if time permits, as it will *really* impress viewers. LangChain's documentation and examples of agents (like using the Python REPL or Pandas DataFrame agent) could be very useful guides [13].

No matter which pattern you choose, you will use prompts to have the LLM generate either plans or code. Some tips for effective prompts and usage:

- **Provide context:** Supply the LLM with information about the data (column names, types, sample values) before asking it to make decisions. For instance, "The dataset has columns: Age (numeric), Gender (categorical: M/F), Income (numeric, skewed distribution)... The target is Purchase (binary 0/1)." This helps the LLM make informed suggestions (it might say "consider one-hot encoding Gender").
- **Constrain the output:** When asking for code, be specific. e.g., "Generate Python code using pandas to impute missing values in columns X, Y, Z with their median." The more specific, the less cleanup needed. If you're using OpenAI's API, you can also use the function-calling capability where the LLM returns structured data (though for code generation, plain text might be fine).
- **Iterate and refine prompts:** Don't expect the LLM to get everything perfect on first try. You might find it produces a complicated plotting code when you just wanted a simple histogram. If so, adjust your prompt or consider simpler tasks. Also, if you find the LLM often making a certain mistake (like using a wrong variable name), you can insert a prior step where you explicitly tell it the variable names in use.
- **Use the LLM for narrative, code for heavy computation:** A good balance is to let the LLM handle the "thinking and explaining" parts, while relying on proven libraries for actual computation. For example, for EDA you might not need the LLM to calculate mean or missing count (you can do that in code and just have the LLM explain the results). This saves tokens and reduces error. Use the LLM in areas where flexibility and language are needed (planning, explaining, writing code glue), and use raw Python where the task is straightforward.

As you integrate the LLM, test incrementally. For instance, first try a prompt where the LLM is asked to produce a simple function or code snippet (like a function to one-hot encode) and run it to see if it works. Then move to the larger sequence. Each integration point (like one for EDA, one for model training decisions, one for report writing) should be tested independently before chaining them, so you can pinpoint issues. Remember, debugging an AI agent can be tricky – you have to debug the prompts or the logic, not a deterministic script – so small steps are key.

7. Automated Model Training with AutoML Integration

When the pipeline reaches the model training step, ensure that your AutoML integration is working properly and efficiently:

- If using **Auto-sklearn**, you will have had it working in the manual pipeline. Now, you might expose it through the LLM agent. For example, you could allow the LLM to choose training parameters: *"Use AutoSklearnClassifier for 5 minutes on the training set."* or it might simply call a function that you wrote which runs Auto-sklearn. One thing to be careful about: Auto-sklearn's fitting process can be lengthy (even 5 minutes may feel long if the agent is interactive). To keep the demo snappy, use a small dataset or explicitly limit time and resources (e.g., `AutoSklearnClassifier(time_left_for_this_task=120, per_run_time_limit=30, n_jobs=4)`). This tries to finish in 2 minutes which is reasonable for demo purposes [7].
- If you opted for manual model selection by the agent, make sure you implemented training for each suggested model. For instance, if the LLM plan says "train a Random Forest and an XGBoost," your code should handle instantiating those (with some default hyperparameters or maybe even have the LLM suggest hyperparameters) and evaluating them. This can be done in a loop or as separate functions (`train_random_forest`, `train_xgboost`, etc.). It's extra work, but it shows you understand different algorithms. You might still use the LLM to generate parts of this code for you – e.g., ask it to write the training and evaluation code given the dataset.

But since model training code is often boilerplate, you might have these pre-coded for reliability and just have the LLM choose which to run.

- **Evaluating and Choosing the Best Model:** After training, gather the results. If using an AutoML library, it likely has the best model accessible directly (Auto-sklearn gives you an ensemble or best pipeline; H2O gives a leaderboard of models). Extract the best model's metrics. If doing manually, compare the metrics you computed for each model and decide the best. This decision could also be made by the LLM – for example, present the LLM a small table of model scores and ask it which is best and to provide reasoning. However, that might be overkill; selecting the max accuracy is straightforward for code. Perhaps more useful is having the LLM **explain** why that model might have performed best (linking it to data characteristics, if possible). For instance, *"The Random Forest outperformed Logistic Regression by 10%, likely due to its ability to capture nonlinear relationships in the data that a linear model could not."* Such an insight in the report shows a deeper understanding and is exactly the kind of narrative a human would write – the LLM can draft this if you prompt it with the model results and some context.

At this stage, your agent will have a chosen model and associated performance metrics. It should also have enough information from earlier steps (like what transformations were applied, key data insights) to feed into the final reporting step.

8. Generate the Final Report with AI Assistance

Now, focus on producing the output that a hiring manager or interviewer would see: the summary of what the agent found and did. This is a great opportunity to use the LLM for what it excels at – generating human-like explanations and summarizing information. Your report can be structured as follows (feel free to adjust, but consistency and clarity are important):

- **Introduction:** A brief description of the dataset and the analysis goal (e.g., "Analyzing the Titanic passenger dataset to predict survival"). This you can write yourself or have the agent draft from the input parameters.
- **Data Overview:** Mention how many samples, how many features, types of features (numeric, categorical, etc.), and any initial observations (like "there are some missing values in Age and Fare" or "the classes are imbalanced" if applicable). You can have the LLM generate sentences from the data overview stats you gathered.
- **Exploratory Analysis Findings:** For example, *"Feature Distributions: Age is right-skewed (most passengers are young, with a few outliers up to 80). Fare has a long tail with a few very high values. Gender is binary (65% male, 35% female). We observed that females had a much higher survival rate than males (75% vs 19%). There were ~20% missing values in Age, which we imputed with the median age (28 years)."* – These kind of insights can be prepared as bullet points and then fed to the LLM to turn into a narrative paragraph. If you have charts, refer to them in the text (e.g., "(see Figure 1)" assuming you label the histograms or include them in an appendix). The LLM can also help interpret a correlation: *"We noted that Fare and PassengerClass are correlated; higher class passengers paid more on average, and also had higher survival rates, indicating socio-economic status might play a role in survival."* The richness of the summary will depend on the data – you don't have to overstate things, just a few interesting observations.
- **Modeling Approach:** Explain what models or AutoML was tried. For instance, *"We applied automated machine learning to try a range of models. The AutoML process evaluated multiple algorithms (including ensemble methods and logistic regression) with 5-fold cross-validation to find the best performer. It also handled hyperparameter tuning and feature preprocessing automatically."* If you manually tried specific models, mention them: *"We trained and compared a Random Forest, an XGBoost decision tree ensemble, and a Neural Network. Each model was evaluated with 5-fold CV on the training set."* This section is about demonstrating the breadth of techniques considered

without going into too many technical details – keep it high-level and LLMs are usually good at phrasing this succinctly.

- **Results:** Present the outcome – *which model was selected as best* and *what its performance was*. If it's classification, report accuracy and maybe other metrics like F1 or AUC if relevant. If regression, report RMSE or R^2. Also mention if the improvement over baseline or other models is significant: *"The best model was a Random Forest with an accuracy of 82% on the test set, outperforming the next best (XGBoost at 79%). The model's confusion matrix (Figure 2) shows it reduced false negatives by capturing nonlinear relationships."* You can include a small table of metrics or just prose. For feature importance, you could list the top 3 important features the model used. E.g., *"Feature importance analysis indicates that 'Gender', 'Fare', and 'PassengerClass' were the most influential factors in predicting survival."* This ties back to the domain insight. An LLM can definitely help articulate why those features matter (it might say "Gender being important aligns with the known fact that women and children were given priority during evacuation" – which is context beyond the data, a nice human-like touch). Ensure that any claims are reasonable and come either from data or known domain info; you might fact-check the LLM's more imaginative statements.
- **Conclusion:** End with a short statement about the success of the autonomous agent: *"In summary, the AI assistant autonomously explored the Titanic dataset and built a model that achieves over 80% accuracy in predicting survival. This demonstrates how automated data science workflows can quickly yield insights and effective models without intensive human effort."* You can also nod to the trend: *"This kind of AI-driven analysis showcases the potential of autonomous AI agents to augment data scientists, a cutting-edge development in 2025* [2] *."* This ties your project to the broader context and impresses the reader that you are aware of industry trends (Deloitte's 2025 report, for example, highlights that 25% of enterprises using AI are expected to deploy such autonomous AI agents by 2025 [2]).

The report generation can be done in a Jupyter Notebook where each section's text is printed or displayed in sequence, or you could have the agent write to a markdown file. If you want to get fancy, you can use a library like `markdown2` to convert to HTML or `weasyprint` to make a PDF. But a well-organized notebook with Markdown and outputs is usually sufficient for showcasing the project.

Make sure any **embedded visuals or tables** are properly labeled and referenced in the text. For example, if you show a feature importance plot, have a caption like "Figure 3: Feature importances from the Random Forest model." This level of detail shows professionalism in reporting results, as you would in a real data science project.

9. Test the Entire Pipeline on Example Datasets

With everything implemented, it's crucial to validate that the autonomous pipeline works as expected **from end to end**. Treat your example dataset as a real use-case: feed it into the agent and observe the process and outputs. It's likely you will need to iterate on this, because integrating all parts often reveals new bugs or needed tweaks (especially with LLM in the loop).

- **Dry Run:** Run the agent with the LLM in a controlled way, possibly step by step, to see that each part triggers correctly. For instance, ensure the LLM's plan is reasonable and that your code can follow it. If using the agent approach with tools, run in debug/verbose mode so you can watch the LLM's reasoning (LangChain agents, for example, allow you to see each thought and action). This is fascinating to include in your documentation: showing how the AI "thinks" (*e.g.*, *"Thought: I should check missing values. Action: call tool CheckMissing"* etc.).
- **Verify Outputs:** Check the content of the generated report for accuracy. Are the numbers in the narrative correct (e.g., if it says 20% missing, is that actually true)? The LLM might occasionally

hallucinate or misstate something, so you may need to enforce correctness by injecting actual values into the prompt. For example, instead of letting the LLM guess the percentage of missing data, calculate it in code and include that in the prompt: *"There are 177 missing values out of 891 in Age (~19.8%)."* so the LLM will likely use those numbers. Similarly, ensure it reports the actual accuracy that the model achieved. Minor inaccuracies in the report can undermine the credibility of the project, so it's worth spending time to get this right (or manually editing the final output if needed).

- **Time and Resource Check:** Note how long the process takes. If the EDA with plots plus AutoML training and LLM calls ends up taking a long time, you might want to trim or optimize some steps for the demo. Perhaps reduce the number of models or the time given to AutoML for the demo run. You can explain in your write-up that more thorough searches were truncated for the sake of time. Also monitor memory usage if using a big local LLM. If things are too slow, one trick is to use a smaller subset of the data just for showcasing the pipeline (e.g., use 500 rows instead of 50000, so that training is faster).
- **Iterate:** It's unlikely to be perfect on first run. Use the tests to refine prompts, fix logic bugs, and possibly add fallback rules. For instance, if the LLM agent gets stuck in a loop or makes a poor decision (like trying an irrelevant analysis), you might hard-code a condition or adjust the prompt to avoid that. It's all about balancing autonomy with reliability. In a pinch, you can always *constrain the agent's freedom* – for example, instead of open-ended "plan anything", you might restrict the plan to a set of known steps. This still shows autonomy but within safe bounds.

As a final test, run the entire pipeline on at least two different datasets (one classification, one regression). This will show the agent can generalize to different tasks. Use the Titanic dataset for classification and perhaps a regression dataset like the **California Housing** dataset (which is straightforward and small). Examine the outputs: does it correctly identify one as classification and the other as regression? Does it change its model selection accordingly? Ideally, yes – this adaptability is a key selling point. If it treats everything the same, then the agent isn't really "thinking"; you want it to demonstrate conditional behavior (even if underneath, you guided it).

10. Polish the Project and Documentation

Now that the technical part is working, make sure you present the project in the best possible way for your resume/portfolio:

- **Clean Up the Notebook/Code:** Remove or consolidate extraneous prints or debugging info (unless you plan to show the agent's thought process, which you can in an appendix). Organize the notebook logically: introduction, then maybe a section showing the agent's intermediate decisions (if you want), then the final report output. Comment your code where needed, especially parts that might be non-obvious (like prompt construction or parsing LLM outputs). This helps demonstrate your ability to write maintainable code around advanced techniques.
- **Add Context and Reflections:** In your project README or notebook markdown, include a discussion about the challenges and how you addressed them. For instance, note that *"LLM-generated code sometimes failed; we implemented an iterative retry mechanism to handle this* [8] *."* Or *"AutoML can be a black box; to ensure transparency, the agent retrieves and logs the top models and their parameters."* Acknowledge limitations, such as *"The LLM's suggestions are not guaranteed to be optimal; in some cases a human expert might do feature engineering that the agent misses. However, the agent succeeded in automating the basics."* This kind of analysis shows maturity and understanding of the tools' constraints [14] .
- **Highlight the Cutting-Edge Aspect:** Make sure to explicitly state why this project is impressive in 2025 terms. You can reference how autonomous agents are an emerging trend in AI that **augment knowledge workers' productivity** [1] . Mention that by building this, you've gained

experience in orchestrating LLMs with tools, which is a highly sought-after skill (few data science candidates will have done this!). You can even quote that *"Deloitte predicts that 25% of enterprises using AI will deploy agentic AI by 2025"*, and position your project as proof that you are ready for that wave [2].

- **Optional – Interactive Demo:** If you have time, you could deploy the Streamlit app (or Jupyter Notebook) on a platform like Heroku, Streamlit Cloud, or GitHub Pages (for static outputs) so that recruiters can see it in action. Even a short video demo or animated GIF of your agent running through a task could be effective in a portfolio. But these are bonus enhancements – the core is having a well-documented, well-structured project that demonstrates the autonomous workflow.

Finally, approach this project with an iterative mindset. Start simple and gradually add complexity. For instance, you might first build a version that *doesn't* use an LLM at all – it just runs AutoML and prints results (that's already useful). Then add the LLM to do the narrative/report part. Then add the LLM to do the planning part. Incremental progress will ensure you always have something working, even if a more advanced component gives you trouble.

By following these steps, you'll implement a cutting-edge **Autonomous Data Science Assistant** that not only functions end-to-end but also highlights your ability to integrate state-of-the-art AI (LLMs and AutoML) into practical data science tasks. This kind of project is sure to stand out on your resume, showing prospective employers that you are forward-thinking and capable of leveraging new technologies to automate and enhance the work of a data scientist – a valuable skill in the AI-driven job market of 2025. Good luck, and enjoy the process of building your AI agent!

Sources:

1. Deloitte – *Autonomous Generative AI Agents* (2024). Prediction that by 2025, a quarter of companies using AI will pilot autonomous "agentic AI" solutions, which automate multi-step knowledge work tasks [15]. This underscores the relevance of autonomous AI workflows in industry.
2. Hossain, M. (2024). *Automating CSV Data Analysis with LLMs: A Comprehensive Workflow*. Medium. Describes how LLMs like OpenAI's GPT can generate and refine Python code for data analysis tasks, enabling automation of EDA and visualization with error-handling loops [10] [8]. Emphasizes combining LLMs with tools (e.g., Python execution, databases) to streamline analysis.
3. GitHub – *Auto-sklearn Documentation*. Auto-sklearn is an open-source AutoML toolkit that frees users from manual model selection and hyperparameter tuning, using Bayesian optimization and ensemble techniques [3]. It can serve as a drop-in replacement for scikit-learn classifiers/regressors, automating preprocessing (like one-hot encoding) and model training [5] [4].
4. Holla, K. (2023). *Building a Data Analyst Agent using LangChain and Open-Source LLM*. AI Advances blog. Demonstrates an LLM-driven agent that breaks down a data query into subtasks, generates pandas code for each subtask, executes it, and aggregates results into a summary [11] [12]. This showcases the use of an LLM with a Python tool to autonomously perform data analysis steps and then compile a human-readable answer, highlighting the potential of combining LLM reasoning with programmatic actions.

---

[1] [2] [15] Autonomous generative AI agents | Deloitte Insights
https://www.deloitte.com/us/en/insights/industry/technology/technology-media-and-telecom-predictions/2025/autonomous-generative-ai-agents-still-under-development.html

3 4 5 GitHub - yu-iskw/auto-sklearn-examples: auto-sklearn examples on Jupyter notebooks
https://github.com/yu-iskw/auto-sklearn-examples

6 7 Auto-Sklearn for Automated Machine Learning in Python - MachineLearningMastery.com
https://machinelearningmastery.com/auto-sklearn-for-automated-machine-learning-in-python/

8 9 10 14 AutoAnalyst - Automating CSV Data Analysis with LLMs: A Comprehensive Workflow | by Mosharraf Hossain | Medium
https://medium.com/@mail2mhossain/automating-csv-data-analysis-with-llms-a-comprehensive-workflow-4f6d613f1dd3

11 12 Building Data Analyst with AI Agent using LangChain and Opensource LLM | by Kaushik Holla | AI Advances
https://ai.gopubby.com/building-data-analyst-with-ai-agent-using-langchain-and-opensource-llm-37f160939b84?gi=59f379d2f3a5

13 LangChain: Data Analysis with REPL-Tool and LLM - Kaggle
https://www.kaggle.com/code/ksmooi/langchain-data-analysis-with-repl-tool-and-llm