

LLM-Powered Exploratory Data Analysis (EDA)

Welcome to **LLM-Powered Exploratory Data Analysis (EDA)**, an AI-augmented data exploration tool. This web app lets you upload a CSV dataset and automatically generates a comprehensive EDA report: from summary statistics and visualizations (think histograms, distributions, correlation heatmaps) to insightful natural-language commentary courtesy of a Large Language Model. In short, it's like having a data analyst and a storyteller in one convenient app!

1. Introduction




What this app does: It streamlines the first step of any data project – Exploratory Data Analysis. You drop in a CSV file, and the app does the rest: cleaning the data, computing key statistics, visualizing distributions, and even writing up a summary of insights using an AI model. No coding or manual plotting needed.


Problem it solves: Manual EDA can be time-consuming and repetitive. You often have to write code to inspect data, handle missing values, plot graphs, and interpret the results. For beginners, it might be overwhelming to decide what to look for. This app automates those tasks. It **quickly surfaces insights and patterns** in the data with minimal effort, so you can understand your dataset faster and with less hassle. It's especially useful when you have a new dataset and want a quick overview without writing a single line of code.

Why it's useful: By combining traditional data science tools with an LLM (Large Language Model), the app not only crunches numbers but also **explains** the data in plain English. You get the best of both worlds:

- **Speed:** Instantly generate plots and stats that would normally take several minutes of coding.
- **Insight:** The AI summary can highlight interesting patterns or anomalies that you might not notice at first glance.
- **Ease of use:** A simple web interface means anyone (students, beginner data scientists, or a developer curious about LLMs) can drag-and-drop a file and get results. It lowers the barrier to entry for exploratory analysis.

Key Features at a Glance:

-  **AI Insights:** Uses a local **Mistral-7B** LLM to generate human-like summaries of the dataset's characteristics, trends, and possible insights.
-  **Auto Visualizations:** Generates common EDA charts like histograms for each numeric column, distribution plots, and a correlation heatmap to show relationships between variables.
-  **Data Cleaning:** Automatically fills missing values (numeric columns get the median, categorical columns get the mode) to handle gaps in data without crashes.

-  **No-Code UI:** Provides a clean **Gradio** web interface, so you don't need to write or tweak code. Just upload and explore.

Technologies Used: *(the tech stack that makes it all possible)*

- **Gradio** – Web app framework for the UI (gives you the drag-drop interface in your browser).
- **Pandas** – Data loading and wrangling (reads the CSV, computes summaries, handles missing data).
- **Matplotlib & Seaborn** – Visualization libraries used to plot histograms and heatmaps for your data.
- **Ollama + Mistral-7B** – The LLM engine (Ollama) and model (Mistral-7B) that generate the natural language insights. **Mistral-7B** is a 7-billion-parameter large language model that runs locally via **Ollama**, which means all AI processing is done on your machine (no API calls to the cloud!).
- **Python** – The glue that ties everything together (Gradio app logic, data processing, and LLM integration are all written in Python).

2. How It Works (User Flow)

Using the app is straightforward and interactive. Here's a step-by-step of the user experience and what happens under the hood at each step:

1. **Upload a CSV file:** The user opens the app (which runs in a local browser tab via Gradio) and selects a CSV data file to analyze. For example, you might upload a `titanic.csv` containing passenger data. *(The app's UI has a file upload widget where you can drag and drop your file or click to select it.)*
2. **Data is loaded:** Once you upload, the app reads the CSV using **Pandas**. Under the hood, it's doing `pd.read_csv(file)` to load your data into a DataFrame. It then automatically performs **data cleaning**: any missing numeric values are filled with the median of that column, and missing categorical values are filled with the most frequent value (mode) of that column. This ensures subsequent analysis won't break due to NaNs and gives more complete visuals.
3. **Analysis & Visualization:** After cleaning, the app computes summary statistics using `pandas.DataFrame.describe()`. This produces counts, means, quartiles, etc., for numeric columns (and top values/frequencies for categorical columns). Next, it generates **visualizations**:
 - Histograms (with optional KDE curves) for each numeric column to show distributions (e.g., age distribution of Titanic passengers, fare distribution, etc.).
 - A correlation matrix heatmap for the numeric features, illustrating how strongly each pair of variables is correlated.
These plots are created with **Matplotlib/Seaborn** in the background. The code creates figures for each plot and saves them as images.
4. **LLM generates insights:** Here's the cool part – the app now takes the statistical summary from Pandas and sends it to the **Mistral-7B** language model via Ollama. Essentially, it prepares a prompt

that says something like: “*Analyze the dataset summary and provide insights.*” The summary stats (the output of `describe()`) are appended to this prompt. The **Ollama** engine loads the Mistral-7B model (if not already in memory) and feeds it the prompt. The LLM then responds with a written summary – for example, it might note “the dataset has 891 entries, with an average age of 29.7 years; there appears to be a higher survival rate for certain groups, etc.” The model is basically acting like a data analyst writing a quick report based on the numbers.

5. **Results displayed:** The app collects the AI’s text insights and the visualization images, and displays them in the browser for the user. You will see a text section containing the **EDA report** (with sub-sections for “Summary”, “Missing Values”, and “AI Insights”) and below that, a gallery of the generated charts. At this point, you can scroll through the insights or click on images to enlarge them. If you have multiple numeric columns, you’ll see multiple histograms – one for each column – plus the correlation heatmap.

All of these steps happen automatically in sequence when you upload the file – typically in just a few seconds for moderately sized data.

**(Diagram in words: Imagine a flowchart where `*[User] → [Gradio UI uploads CSV] → [Python EDA function (Pandas cleans & analyzes data)] + [Ollama (Mistral-7B) generates text] → [Gradio displays charts + summary]. It’s a smooth pipeline from raw data to insights.`*)

3. Tech Stack and Architecture

Stack Overview: This project uses a combination of a web UI library, data processing libraries, and an LLM engine – each component has a clear role:

- **Frontend/UI:** Gradio provides the user interface as a local web app.
- **Backend/Data Processing:** Python with Pandas/NumPy handles data reading, cleaning, and summary computations.
- **Visualization Engine:** Matplotlib and Seaborn generate the plots for visual EDA.
- **LLM Engine:** Ollama (an LLM runtime) running the Mistral-7B model for AI-generated text insights.
- **Orchestration:** The glue code ties these together in a Gradio interface function that coordinates the workflow (data -> analysis -> LLM -> output).
- **Architecture Breakdown:** When you run the app, you’re essentially running a small server on your machine that does the following in one go:
 - The **Gradio UI** (in the browser) is connected to a **Python function** on the backend (`eda_analysis()` in our code). Whenever a user uploads a file, Gradio passes the file path to this function.
 - Inside `eda_analysis()`, we sequentially call helper components:
 - **Data Cleaner & Summary (Pandas):** Loads the CSV into a DataFrame, fills missing values, then produces a summary (`df.describe()` and `df.isnull().sum()`).
 - **Visualization Generator:** A function uses **Matplotlib/Seaborn** to create plots. For each numeric column, it saves a histogram plot image. Then it computes a correlation matrix of all numeric columns and saves a heatmap image. These images are stored locally (in memory or disk) and their

paths are collected.

- **LLM Insight Generator:** Another helper function takes the textual summary of the data and calls **Ollama**'s API with the Mistral-7B model. This is done via the `ollama` Python library, which communicates with the Ollama backend (which must be installed on your system). The model processes the summary text and returns a nicely worded insight paragraph or two.
- Once those are done, `eda_analysis()` returns two things back to Gradio: (1) a big string containing the combined report (including the original summary stats, info on missing values, and the AI's insight text), and (2) the list of plot image paths.
- **Gradio** takes that returned data and displays it in its output components: a **Textbox** (for the report text) and a **Gallery** (for the images). Gradio handles embedding the images and formatting the text in the interface for the user to see.

Data & Control Flow (plain language): The flow is synchronous and simple. Think of it like a recipe:

1. **Input:** User gives data (CSV file) to the app.
2. **Process:** The app (Python backend) mixes ingredients – it “stirs” the data with Pandas to get stats, “bakes” some charts with Matplotlib, and then adds a special “AI seasoning” by asking Mistral-7B to interpret the stats.
3. **Output:** The finished “dish” is served back on the UI – a text report and a set of charts, ready for consumption. 🍽️

This architecture is lightweight. All components run on the user's local machine. **No external servers are involved** in processing the data or generating insights, which means your data stays private and the app can even work offline (after you've installed the necessary tools). The design is modular – for instance, the AI part (Mistral via Ollama) is decoupled from the rest; if needed, one could swap in a different model or even an API call with minimal changes.

4. Setup Instructions

To get the app running on your local machine, follow these steps. The setup involves installing Ollama (for the LLM), pulling the Mistral model, and setting up the Python environment for the Gradio app.

1. Install Ollama and Mistral-7B

Ollama is the engine that will run the Mistral-7B model on your computer. You can install Ollama by visiting the official website ollama.com and following instructions for your operating system. Ollama supports macOS (and also has Linux support via Docker; Windows users might use WSL or a VM since native support is limited at the moment). After installing the Ollama CLI/tool:

Download the Mistral-7B model: In a terminal, run:

```
bash
CopyEdit
ollama pull mistral
```

- This command will download the Mistral-7B model weights onto your machine. The model is a few gigabytes, so this may take some time on first download. Once pulled, Ollama will have the model available locally under the name “mistral”. *(Note: Ollama might have different variants like `mistral:7b` or an `instruct` version; in our code we use the default “mistral” which should correspond to the `instruct` version suitable for chat.)*

2. Set Up Python Environment

Make sure you have **Python 3.8+** installed. It’s recommended to use a virtual environment (venv or conda) to avoid conflicts with other packages on your system. Then install the required Python libraries. The main dependencies are: `gradio`, `pandas`, `matplotlib`, `seaborn`, and `ollama` (the Python package to interface with the Ollama engine). If you have a `requirements.txt`, simply do:

```
bash
CopyEdit
pip install -r requirements.txt
```

Otherwise, install manually:

```
bash
CopyEdit
pip install gradio pandas matplotlib seaborn ollama
```

This will fetch all necessary packages from PyPI. The `ollama` Python package allows our code to call the Ollama engine.

No API keys required: Everything is running locally, so you **do not need any OpenAI API key or internet access** for the app to work. As long as Ollama and the model are installed, the LLM will run offline. (The only time you needed internet was to download dependencies and the model itself.)

3. Run the App Locally

With Ollama installed and the Python environment ready, you can now launch the Gradio app. If you have the Python script (or notebook) for the app, run it. For example, if the main file is `app.py`:

```
bash
CopyEdit
python app.py
```

This will start a local web server and usually Gradio will output a local URL (like `http://127.0.0.1:7860`) and sometimes a public shareable URL if `share=True` is enabled. Open the local URL in your web browser. You should see the **LLM-Powered EDA** interface ready for input. (If running in a Jupyter notebook, you might see the Gradio interface inline or a link to open it in a new tab.)

Ensure Ollama is running: Ollama typically runs as a background service once installed. If for some reason the LLM part isn’t responding, make sure the Ollama service is active. On macOS, Ollama may run automatically. You can test by running a quick command in Python:

```
python
CopyEdit
```

```
import ollama
ollama.run("mistral", prompt="Hello")
```

or simply use the app – if the model is not found or not running, you'll get an error which indicates you might need to start the Ollama daemon (`ollama serve`) or ensure the model name is correct. Usually, though, just pulling the model is enough and the `ollama.chat` call in the app will trigger the model to load when needed.

Environment variables or configs: None required for basic use! The app doesn't require any secret keys. All configuration (like model name or parameters) is either hardcoded or uses defaults. Just ensure your environment has enough RAM to load the model (Mistral-7B is relatively lightweight for modern computers, often running on CPU). If you run into memory issues, you might need to close other programs or, if available, use a system with more RAM or a GPU. But many users can run 7B models on a decent laptop.

Tips for smooth local execution:

- Close unnecessary applications to free up memory, especially if your dataset is large or your RAM is limited.
- If the Gradio interface is not showing up, check your terminal for errors. You might need to disable the shareable link (`share=False`) for strictly local use or vice versa if you want to access it from another device.
- For large CSV files (tens of MBs), the analysis might take a bit longer; be patient during the first run when the model is loading. Subsequent runs will be faster as the model stays in memory.

Once the app is running, you can upload a CSV and enjoy the automated EDA. 🇬🇧

5. LLM Integration & AI Logic

One of the most intriguing parts of this app is how it integrates a language model to augment data analysis. Let's break down the AI side:

Mistral-7B via Ollama: *What is this?* Mistral-7B is an open-source large language model with 7 billion parameters, known for being efficient and fairly powerful for its size (it was a notable release in 2023 by Mistral AI). Ollama is the platform that allows us to run this model locally. Think of Ollama as a local "ChatGPT service" – it manages loading the model, running it, and returning the output, all on your machine. The app uses the **Ollama Python library** to communicate with the Ollama engine.

How the model is used in our app: We generate a text prompt that encapsulates the results of the data analysis and ask the model to provide insights. Specifically, after computing `df.describe()` (the summary statistics of the DataFrame), we convert that summary to a string and craft a prompt like:

"Analyze the dataset summary and provide insights:

<summary statistics tables>"

That prompt is sent to `ollama.chat()` with `model="mistral"`. In terms of roles, we send it as a user message (we could also have a system message for instructions, but in this simple approach we rely on the model's default behavior to interpret the request). Mistral-7B is presumably an instruction-tuned model (given by how Ollama uses it), so it will treat that prompt as a task to complete.

The LLM then returns a response, which the code captures as `response['message']['content']`. This content is a block of text – usually a few sentences or a paragraph – where the model discusses the data. For example, for the Titanic dataset, it might output something like:

“The dataset contains 891 entries. The average age of passengers is around 29.7 years (with a median of 28), indicating a relatively young passenger group. There are some missing values in the Age column, but they have been filled with the median. The survival rate (mean of the Survived column) is about 0.38, meaning roughly 38% survived. Notably, the average fare is quite high relative to the median, suggesting a skewed distribution (few passengers paid a lot more than most). These insights hint that passenger class and fare might be correlated with survival, which is worth further investigation.”

This is an example of the kind of summary **the AI produces** – it reads the stats and infers a mini-narrative. It might highlight the difference between mean and median (skewness), mention any striking min/max values (outliers), or relationships (like it noticed if one group has higher values).

Prompt crafting: In our current implementation, the prompt is relatively simple: we essentially dump the stats and say “provide insights.” There isn’t a complex prompt engineering or few-shot examples here, relying on the model’s ability to understand a data summary. In the future, one could enhance this by adding more context, for example: “You are a data analyst. I will give you summary statistics of a dataset. Provide a concise analysis of what these statistics mean, pointing out any notable patterns or anomalies.” For now, Mistral-7B does a decent job with just the direct approach.

Limitations and Quirks of the AI: It’s important to remember that the LLM is analyzing the *summary* of the data, not the raw data itself. This means:


- It can’t directly identify complex relationships or patterns beyond what the summary stats suggest. If something isn’t reflected in the stats, the model might miss it or even **guess/hallucinate** incorrectly. For instance, if two variables are perfectly correlated, that would show up in a correlation matrix (which we actually don’t feed to the model currently), so the AI might not mention it at all.
- The AI might sometimes speak in generalities. E.g., “there might be outliers” if it sees a big gap between mean and median. These are educated guesses based on stats, but not confirmed without deeper analysis. Use these insights as a starting point, not absolute truth.
- **Accuracy:** Mistral-7B is good but not infallible. It might misinterpret something from the summary table, especially if the table is wide or not formatted clearly. We try to mitigate this by providing the raw text of the `describe()` output. Still, for very large numbers of columns, the prompt could become long and the model might truncate or ignore some parts (depending on its context window, which for Mistral-7B is around 2048 tokens).
- **Tone and style:** The model’s writing style is generally formal-informative. It tends to produce a few sentences that read like an analyst’s report. We don’t heavily control the tone beyond what the model naturally does. You might find the phrasing sometimes a bit generic. That’s something that can be improved with prompt tuning or using a larger model for more nuance if needed.
- **Performance:** Running a 7B model locally on CPU is quite feasible, but each query might take a few seconds (anywhere from 2 to 15 seconds, depending on your hardware). For most datasets this is fine since plotting also takes a moment. However, if you notice the AI part is slow, it’s often the model doing its work. On an Apple Silicon chip or a decent GPU, it will be faster. The good news is the model only runs when you upload a file, not continuously.

In summary, the LLM integration adds a layer of interpretation to the raw numbers. It’s like getting a quick second opinion on your data from an AI assistant. Just remember it’s working off summaries, so it’s a

helpful guide, but you (as the human) should ultimately validate any critical insights, especially if you plan to make decisions based on them.

6. UI/UX and Visualizations

The app's user interface is designed to be simple and intuitive, leveraging Gradio's ready-made components. When you launch the app, you'll see a single-page web UI with a title, a short description, an upload button, and output sections. There's no complex navigation – everything happens in one view so you can focus on exploring the data.

Gradio Interface Layout: At the top, the app displays the title “ LLM-Powered Exploratory Data Analysis (EDA)” and a brief description instructing you to upload a CSV file. Below that:

- On the left (or top, depending on your screen size) you have the **file uploader** where you select your CSV.
- Once a file is uploaded and processed, the **outputs** appear. The outputs consist of:
 - An **EDA Report** textbox (scrollable) that contains text output. This includes messages like “Data Loaded Successfully!” followed by the **Summary** statistics table, the **Missing Values** count per column, and the **AI Insights** paragraph.
 - A **Data Visualizations** gallery that displays the generated charts. Each chart is shown as a thumbnail; you can click on any to enlarge it. The gallery is interactive – you can scroll through multiple images if there are many.
- The overall UX is “upload and observe.” There's a small delay as the analysis happens, after which the results populate automatically. Gradio handles showing a loading spinner while the backend works, so the user knows something's happening.
- *Screenshot Placeholder:* Imagine the app interface after uploading a dataset (e.g., the Titanic data). On the page, you would see an “**EDA Report**” section with text stating summary stats (count, mean, std for each column, etc.), and below, a section titled “**Data Visualizations**” showing several image thumbnails. For example, you might see small preview images of “Distribution of Age”, “Distribution of Fare”, and a “Correlation Heatmap.” The layout is clean: a gray background for the interface, with the text on the left and images on the right in a grid.
- **Visualizations Explained:** The types of plots the app generates automatically are chosen to cover the basics of exploratory analysis:
 - **Histogram for each numeric feature:** This gives a sense of the distribution (is it normal, skewed, bimodal?) and can reveal outliers. The app uses 30 bins by default and overlays a Kernel Density Estimate (smooth curve) to show the distribution shape. The x-axis is the value range of the feature, and the y-axis is the count of records. Each histogram is titled “Distribution of [ColumnName]” for clarity.
 - **Correlation Heatmap:** This is a grid showing pairwise correlations between numeric columns. It's color-coded (for example, red for positive correlation, blue for negative, with intensity indicating strength, and perhaps white or light color for near zero correlation). The diagonal is all 1.00 (each variable with itself). This plot is great for seeing if any two factors move together or inversely. For instance, in many datasets like Titanic, you might see that “Fare” is negatively correlated with “Pclass” (since higher-class tickets cost more, so Pclass (1st, 2nd, 3rd) inversely correlates with fare price).

Example of an auto-generated histogram (Age distribution from the Titanic dataset). The app produces such a chart for each numeric column, giving a quick view of the data's shape. In this histogram, we see the count of passengers (y-axis) across age bins (x-axis). Most passengers are in their 20s and 30s, with a peak around age twenty-eight. The blue curve (KDE) overlays the histogram to indicate the density. We can immediately spot that there were many young passengers and fewer older passengers, and possibly a small uptick in very young ages (children). Such visualizations help identify skewness (here, a tail toward older ages) and whether the distribution is unimodal or not.

Example of a correlation heatmap generated by the app (using Titanic dataset's numeric features). Each cell shows the correlation coefficient between a pair of features, with red meaning a strong positive correlation (closer to 1.0) and blue meaning a strong negative correlation (closer to -1.0). In this sample, notice the deep blue at the intersection of Fare and Pclass (-0.55) indicating that higher class numbers (which actually represent lower ticket class) strongly correlate with lower fares (since 1st class is Pclass=1 with high fare, 3rd class is Pclass=3 with low fare). Also, "Survived" has a moderate positive correlation with Fare (0.26, light orange) and a negative correlation with Pclass (-0.34), suggesting those who paid more or were in higher classes tended to survive more – an insight aligned with known Titanic outcomes. The app's heatmap makes such relationships immediately visible. You can quickly scan for any bright red or blue boxes to discover interesting variable relationships.

The UI/UX could certainly be enhanced (and we'll talk about future ideas soon), but even in its current form, it's quite user-friendly:

- **Clarity:** Each output component is labeled ("EDA Report", "Data Visualizations") so the user knows what they're looking at.
- **Scroll and Zoom:** Long text is in a scroll box, and images can be enlarged, preventing the interface from becoming overwhelming if the dataset is large or there are many charts.
- **Interactivity:** While the app doesn't have interactive plots (they are static images), simply being able to scroll through them or view full-size is useful. If a dataset has, say, 10 numeric columns, you'll get 10 histograms plus the heatmap – the gallery format keeps them organized.

Optional Enhancements for UX (ideas): In the future, the interface might add features like:

- Allowing the user to **select specific columns** to plot or analyze (to declutter if you don't need every single column's chart).
- **Downloadable report** button: compile the text and images into a PDF or HTML that the user can save.
- More dynamic visuals: e.g., interactive charts where you can hover to see values, though that would require using Plotly or another interactive library instead of static images.
- The ability for the user to ask *follow-up questions* to the LLM about the data via the interface ("chat with the data" scenario) – this would turn the UI into more of a two-way interactive experience, but would be a neat extension.
- Dark mode / theming options, since Gradio does support some customization – could be nice for prolonged use or preference.

As it stands, the UI is minimalistic but effective – you get your results quickly and can move on to deeper analysis if needed. It's meant to give a good first impression of the dataset without any tweaking.

7. Testing and Debugging

To ensure the app works correctly and handles various scenarios, it's important to test it with different datasets and conditions. Here are some suggestions for testing, common issues to watch out for, and debugging tips:

Functional Testing:

- *Basic functionality:* Start with a well-known small dataset (for example, the Titanic dataset or Iris dataset). Upload it and verify that you get a summary, some insights from the AI, and appropriate charts. Check that the numbers in the “Summary” section match what you’d expect (e.g., count of rows, mean values, etc., as cross-verified by manual calculation or another tool).
- *Different types of data:* Try a dataset with mostly numeric data (e.g., some scientific measurements) vs. one with many categorical columns. The app should handle both. With mostly numeric, you’ll see many histograms. With many categorical (and few numeric), you might only see a couple of histograms or maybe just the heatmap if at least two numeric columns exist. **Edge case:** If there are *no numeric columns*, the heatmap code will find `numeric_df.empty` true, so it won’t produce a heatmap, and no histograms either – effectively no images. The app should still show the text summary and AI insights, just with an empty gallery. Test this scenario (perhaps with a dataset of all strings/categories) to ensure it doesn’t crash and handles empty `plot_paths` gracefully.
- *Missing data handling:* Use a dataset with some missing values to ensure the filling logic works. For example, a column where some entries are blank – after running, check the summary to see if the count of that column equals the total rows (meaning missing were filled). Also, verify that the AI insight might mention fewer missing values (or none) since we fill them. If a column was entirely missing (all NaNs), our filling logic for mode might run into an issue (since `.mode()[0]` could fail if there’s no mode). That’s an edge case to test – in such a case, we might need to adjust the code to skip columns that are all NaNs. Currently it might throw an `IndexError`. Testing an all-NaN column will surface this; the fix could be adding a simple check or try/except (which we can implement if needed).
- *Large datasets:* Test with a larger CSV (e.g., 100k rows). Pandas can handle it, but the performance and memory use should be noted. The summary will still just show stats (not all data), so it should be fine. However, plotting 100k points in a histogram with KDE might be slower or use more memory – see if the app handles it (it might still be okay, but possibly a few seconds per plot). Also, the LLM will get the summary which is not size-dependent by rows, only by number of columns. But if you have a *very wide* dataset (hundreds of columns), the `describe()` output will be very wide text or very long text. We should test how the LLM handles a prompt that large. It might truncate or yield a very generic insight if overwhelmed. In such cases, consider limiting the prompt or summarizing the summary (like only sending certain stats or the top N columns by variance, etc.). For now, just be aware that extremely wide datasets could be challenging. As a tester, you might try a dataset with, say, 50 columns to see how it goes.
- *Non-CSV input:* The interface restricts to file type, but it doesn’t strictly check extension. Try uploading a text file that isn’t CSV or a malformed CSV. Pandas will likely throw an error. Right now, our code does not have a try/except around `pd.read_csv`, so if it fails, the Gradio interface might show an error (possibly a red error trace in place of the outputs). This is something to test – ensure the user sees a useful message. In the current state, they might get a somewhat ugly Python error. We could improve it by catching exceptions and returning a user-friendly message (“Failed to read the file. Please make sure it’s a valid CSV.”). As a workaround, testers should stick to actual CSV

files.

Common Issues & Debugging:

- **“Model not found” or Ollama errors:** If the AI insights part isn’t working, the first thing to check is the console/log where you ran the app. If you see an error like *“model ‘mistral’ not found”*, it means the model isn’t installed or not accessible. Ensure you ran `ollama pull mistral` and that the Ollama service knows about it. If you see something like *“No module named ‘ollama’”*, it means the Python Ollama package isn’t installed in your environment (run `pip install ollama`). If Ollama isn’t running at all, you might see a connection error. Running `ollama serve` in a terminal can manually start the Ollama backend service – after that, try the app again.
- **Gradio launch issues:** Sometimes Gradio might not launch a public share URL due to firewall or network issues. If `demo.launch(share=True)` hangs or fails, try `share=False` to stick to local. If it still doesn’t open, verify you’re not running on a port that’s blocked. By default it’s 7860; you can specify another port in launch if needed (`demo.launch(server_port=7861)` for example).
- **Slow performance or freeze:** If you upload a very large file and the app seems stuck, it might be busy computing or possibly ran out of memory. Check your system resources. The combination of loading data + generating multiple plots + running a model can tax memory. For extremely large data, consider downsampling for EDA or increasing system memory. Also, the first time Mistral-7B runs, it might take some time to load into RAM (you’ll notice this as a delay before the first insight appears). Subsequent runs are faster. If it’s truly stuck, you can interrupt and see logs for where it got hung.
- **Incorrect or odd AI output:** If the AI insight seems off or contains an error (e.g., mentions a column that doesn’t exist, or says “no missing values” when you know there were missing), this might be a hallucination or misinterpretation by the model. While not a “bug” in code, it’s something to be aware of. Debugging this can involve printing the prompt you sent to the model to ensure it was formatted correctly. You might find that the formatting of the summary (a big table in text form) could confuse the model. We could experiment with a different prompt format (maybe a bullet list of findings passed in instead of a raw table). If needed, log the `prompt` string to see how it looks. This can be done in the code for development, but you wouldn’t want to expose that in production because it’s a lot of text. For debugging, though, it’s useful.

Logging: By default, our app doesn’t have a custom logging mechanism (aside from what Gradio and Ollama output). If you run it in a terminal, you’ll see prints or errors there. If you want to add logging, you could insert some `print()` statements in the `eda_analysis` function (like printing “Data loaded, shape: ...”, “Plots generated: n plots”, etc.) to trace the execution. Gradio will show these prints in the terminal where you launched the app. That can help pinpoint if it crashes at a specific step. For instance, if it prints the summary but doesn’t print “Plots generated” then you know it likely failed during plotting.

Testing edge cases:

- Extremely small dataset (e.g., 1 row or 1 column) – with 1 row, stats like std dev might be NaN; see if the app handles that (filling NaNs for numeric might handle some, but a single-row describe might produce NaNs for std which remain since median fill wouldn’t change a single value). The LLM might say very little in this case, but it should still run.
- Non-English characters or unusual text in data – if your CSV has column names or values with special characters, just ensure nothing breaks. Pandas can handle unicode, and Python’s default encoding is UTF-8 so it should be fine, but always good to test. The model might not mention them unless

relevant.




- Ensure the **Missing Values count** in the text is correct after filling. We call `df.isnull().sum()`, which should ideally be all zeros after filling (except perhaps if any columns couldn't be filled like all-NaN scenario). If it's not all zeros, that means some columns still had NaNs (e.g., non-numeric columns that are all NaN, as mentioned). The insight might still say "X missing values in column Y" based on that. We should interpret that carefully – perhaps we'd clarify that we filled what we could. For now, it's just reporting what's left missing (if anything).




By methodically testing these scenarios, you can gain confidence in the app's robustness. Each time you test, try to break something – if you succeed, that's an opportunity to improve the code. Overall, debugging is fairly straightforward since the app's pipeline is linear. Most issues will either be with reading data, plotting, or the LLM responding. And as always, check the console logs for tracebacks; they will usually pinpoint the line of code where an error occurred, making it easier to fix.

8. Future Work and Improvements

This project is a solid starting point for automated EDA, but there are many ways it could be extended and improved. Here are some ideas and potential features for future development:

New Features and Enhancements:

-  **Column Selection & Customization:** Enable users to pick which columns to include in the analysis. Perhaps the UI could list column names with checkboxes. This way, if a dataset has 50 columns but the user only cares about 5, they could focus the EDA on those. Additionally, let the user choose what analysis to run (maybe skip correlation matrix or choose between histogram vs boxplot, etc.).
-  **More Visualization Types:** Currently we have histograms and a heatmap. We could add:
 - **Box plots or violin plots** for numeric columns to show distribution and outliers in another way.
 - **Bar charts** for categorical columns (showing counts of each category) – since right now, purely categorical columns don't get a visualization in our app.
 - **Pair plots or scatter plots** for pairs of variables that seem interesting (e.g., if the heatmap flags a strong correlation, automatically plot those two variables on a scatter plot).
 - **Time-series plots** if a date/time column is present (detect temporal data and maybe line-plot it if it makes sense).
-  **Enhanced AI analysis:** We can make the LLM even more involved:
 - Instead of a single prompt, possibly do a multi-step query. For example, first ask the LLM to list interesting aspects from summary, then maybe ask it a follow-up question like "which column seems most important and why?" to get deeper insight. This could then be combined in the report.
 - Use a larger or different model for more complex analysis if running on a machine with better hardware. For instance, if someone has a GPU, they might load a 13B or 70B model in Ollama for more detailed insights. (The app could allow specifying the model name to use via a config.)

- **LLM-based data cleaning or feature engineering:** A cutting-edge idea is to have the LLM suggest how to preprocess or augment the data. It could look at the distributions and say “maybe you should bucket this continuous variable” or “there are 5 categories in Column X, perhaps combine two of the smaller ones.” This moves towards automated data prep, not just analysis.
-  **Downloadable Report:** Implement a feature to export the entire analysis (text + images) as a nicely formatted report. This could be a PDF or an HTML file. It would allow users to save the results or share them. We could use libraries like `pdfkit` or ReportLab for PDFs, or even just generate markdown/HTML and convert.
-  **Interactive Exploration:** Integrate a conversational interface where after the initial output, the user can ask the LLM follow-up questions about the data. This turns the app into more of a chat-with-your-data experience, powered by the same model. For example, user could ask “Which factor influenced survival the most?” and the LLM (grounded on the data it has) could attempt an answer. This would blur the line between static EDA and dynamic analysis, and would require keeping the data or its summary in context for the LLM.
-  **Performance improvements:** For larger data, consider lazy loading or sampling:
 - We could sample a subset of the data for plotting to speed it up (especially if millions of rows – plotting every point might be overkill for EDA).
 - Use streaming or asynchronous calls for the LLM so the UI could update as soon as charts are ready, and maybe show the AI text a moment later when it arrives, rather than waiting for everything.
 - Caching results for identical datasets – if you upload the same file twice, it could reuse the previous analysis.

Scalability and Deployment:

-  **Deploying on Hugging Face Spaces:** Since the app is Gradio-based, it’s straightforward to deploy on HF Spaces (which provides free hosting for Gradio apps). This would make the app accessible to anyone via a URL, without needing them to install anything. The challenge here is that Hugging Face Spaces typically run on limited CPU and memory, so running Ollama + Mistral there might not be possible (especially since Ollama would need to be installed and models downloaded). One way around this is to switch to an online model (like use an API to OpenAI or others) when deploying in such an environment, but that sacrifices the local/offline aspect. Alternatively, Spaces now support containers – one could containerize Ollama with the model and the app, but that might be heavy. Still, the idea is to make a shareable demo.
-  **Dockerizing the App:** Containerization would help in reproducibility and deployment. A Docker image could contain the Python environment, the code, and even the Ollama binary and possibly the model. Users with Docker could run the whole thing with one command, no manual setup. (The image might be large due to the model, but it’s convenient.) For scaling on a server, Docker would make it easier to deploy on cloud VM or a local server for a team to use.
-  **Multi-user or Web Service Mode:** Right now, it’s a local app for one user at a time. If we wanted to make it multi-user (e.g., running on an internal server where many people upload their files), we’d have to consider concurrency. Gradio can handle multiple sessions, but running multiple LLM calls in parallel might strain resources. We might queue requests or limit to one at a time if using a single model. Scaling horizontally (multiple instances of the app behind a load balancer)

could be considered if there was high demand.

Collaboration and Extensibility:

- For open-source collaboration, a **contributor guide** could be added. This would include how to set up the dev environment, how to run tests, code style guidelines, etc. Encouraging contributions for new chart types or support for other data formats (like JSON or Excel input) could be valuable.
- Documenting the code with comments and perhaps docstrings, so others can understand the functions easily. It's already a relatively short script, but clear comments help if someone new wants to modify the prompt or change the model.
- **Plugin system or config file:** An improvement could be to allow configuration of analysis steps via a config file. For instance, a YAML or JSON config where a user can turn on/off certain analyses or set the number of bins for histograms, etc., without editing code. This way, collaborators or users can tweak behavior easily.

Learning and Next Steps:

- This project opens the door to experimenting with LLMs in data science workflows. Next steps could include trying different models (like Llama-2 7B, or if one has the resources, a 13B model) to see if insights improve.
- One could also integrate this with a Jupyter Notebook environment – for example, have a notebook version where after the analysis, the notebook is populated with the charts and text, bridging the gap between an automated tool and a report notebook.
- Another idea: **real-time updates** – if you had a streaming data source (though CSV upload is static), how would you keep updating the analysis? That's a broader scope, but interesting for future exploration (like monitoring a dataset over time with AI commentary on what's changing).

In conclusion, there's a lot of potential to make this app even more powerful. The current version provides a strong foundation by demonstrating that combining an LLM with traditional EDA can yield a user-friendly and insightful tool. As we iterate, we can add flexibility, handle more edge cases, and broaden the scope beyond just static CSV exploration. We invite collaborators to try it out, suggest improvements, and help build the next version – one where exploring data is as easy as having a conversation about it! 🙌👏