# ModelBot System Design

System Design and User Journey
by Joshua Gottlieb

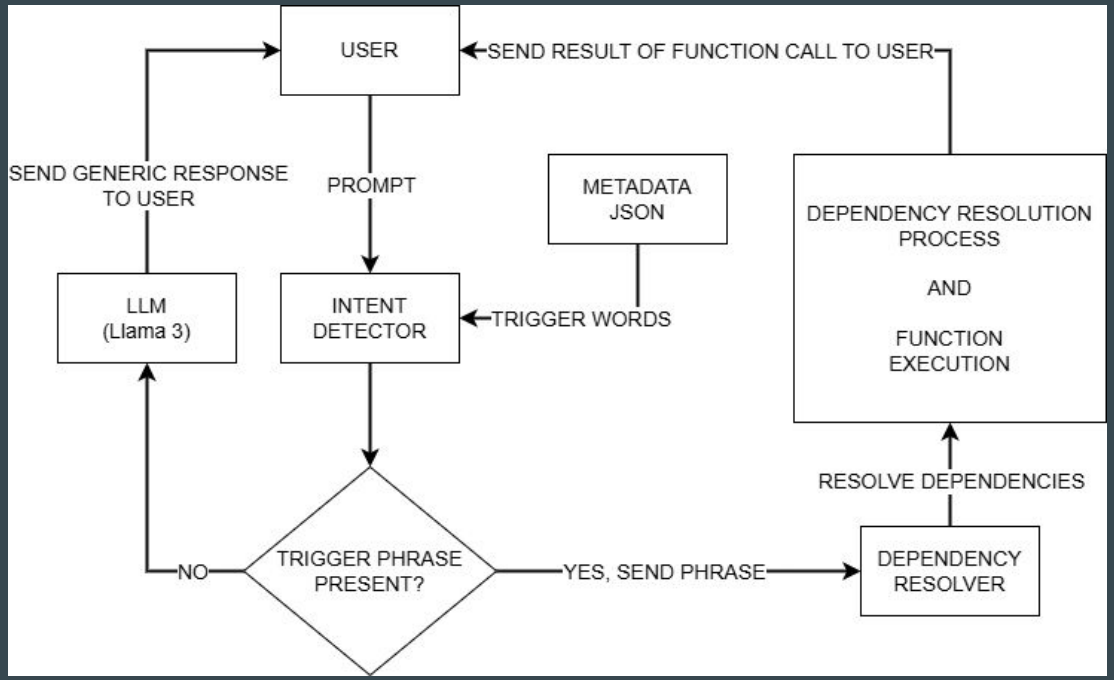
# Overview

- The purpose of ModelBot is to help non-technical users perform basic modeling tasks on data in a csv file and create reports. The end goal is to produce a PDF or inline report that explains the model performance to a non-technical stakeholder.

- The bot is designed to function such that the user need not know anything about data science or the modeling process. The bot engages in a conversation with the user, guiding the user in creating a classification or regression report. ModelBot examines the data to help determine whether the data is better suited for classification or regression.

- If the user tries to do the wrong task type (regression for classification data or vice versa), the bot automatically handles the procedure for either transforming the data or switching task types so that the user cannot take "wrong steps" or perform steps out of order.

# Basic Conversation Loop for ModelBot




- The user sends a prompt which is analyzed by the intent detector using trigger phrases in the metadata.
- If a trigger phrase is found, the phrase is passed to the dependency resolver, otherwise, the LLM generates a generic response to answer the user's prompt.

# Metadata JSON File for Processing Functions

| FUNCTION METADATA |
| --- |
| Function Name |
| Trigger Words |
| User Inputs |
| Prerequisites - Data |
| Prerequisites - Function |
| Alternative Function |

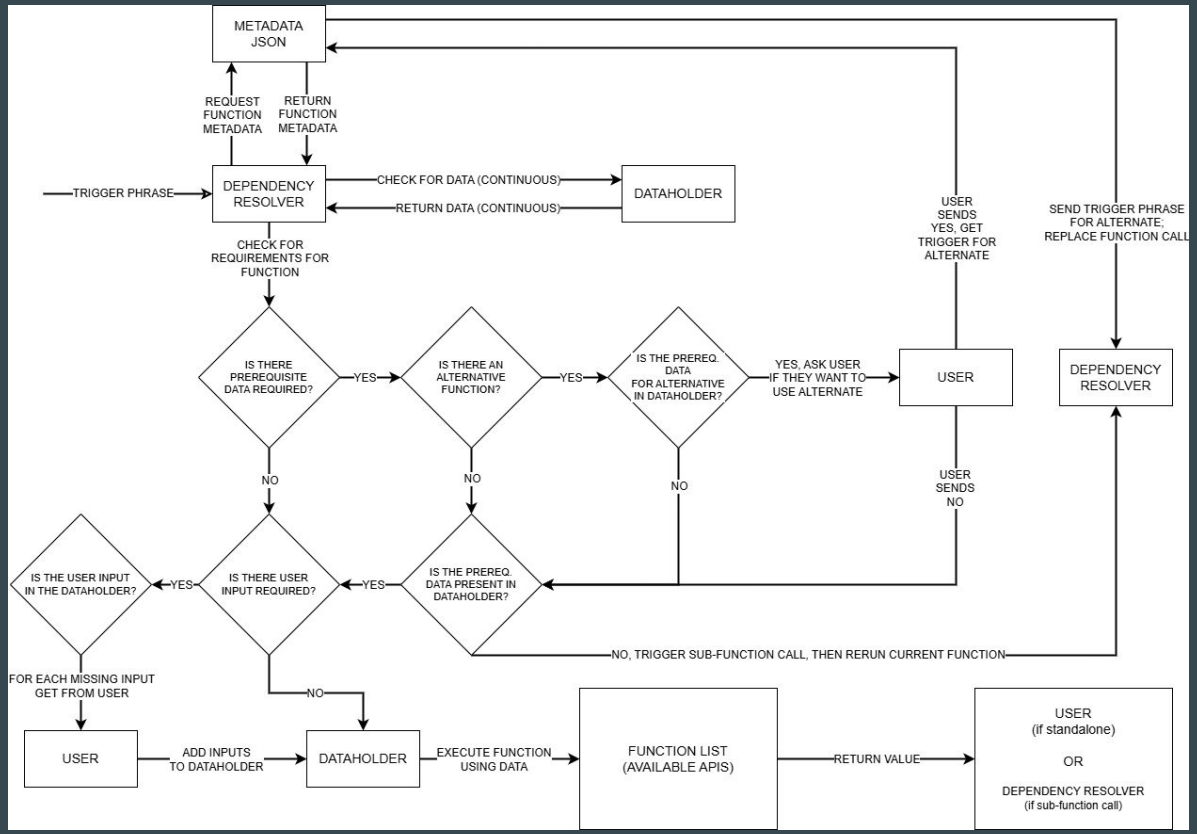
- Trigger Words: Phrases to check from user prompt to use for starting the function call process, typically 2 words long
- User Inputs: Inputs expected to be entered by the user prior to execution of function (can be null)
- Prerequisites - Data: Data expected to be present prior to execution of function that is a result of a prior function call (can be null)
- Prerequisites - Function: Prerequisite function to generate prerequisite data (null if prereq. data is null, required if prereq. data is not null)
- Alternative Function: Alternative function that can be invoked optionally, used to prevent errors from recursive calls (can be null)

# DataHolder Class for Storing Session Data

- The DataHolder class is used to hold all relevant data generated during a session, including all prerequisite data for invoking function calls. The DataHolder object itself is passed into each function so that it can be updated during execution.

- The reason to use this as a class rather than a pure event stream is to take advantage of object-oriented programming to define class methods for dynamically setting, getting, and clearing data from the DataHolder, as well as other helper class methods for other purposes, such as data normalization.

# Full Dependency Resolver Process - Diagram




Explanation on next slide.

Not shown, but at the end of any function call by the user, the user is prompted whether they wish to clear the DataHolder to remove stored data and to generate more reports within the same chat session.

Otherwise, previous data is held and functions will auto-execute upon call due to satisfying all prerequisites.

# Full Dependency Resolver Process - Explanation

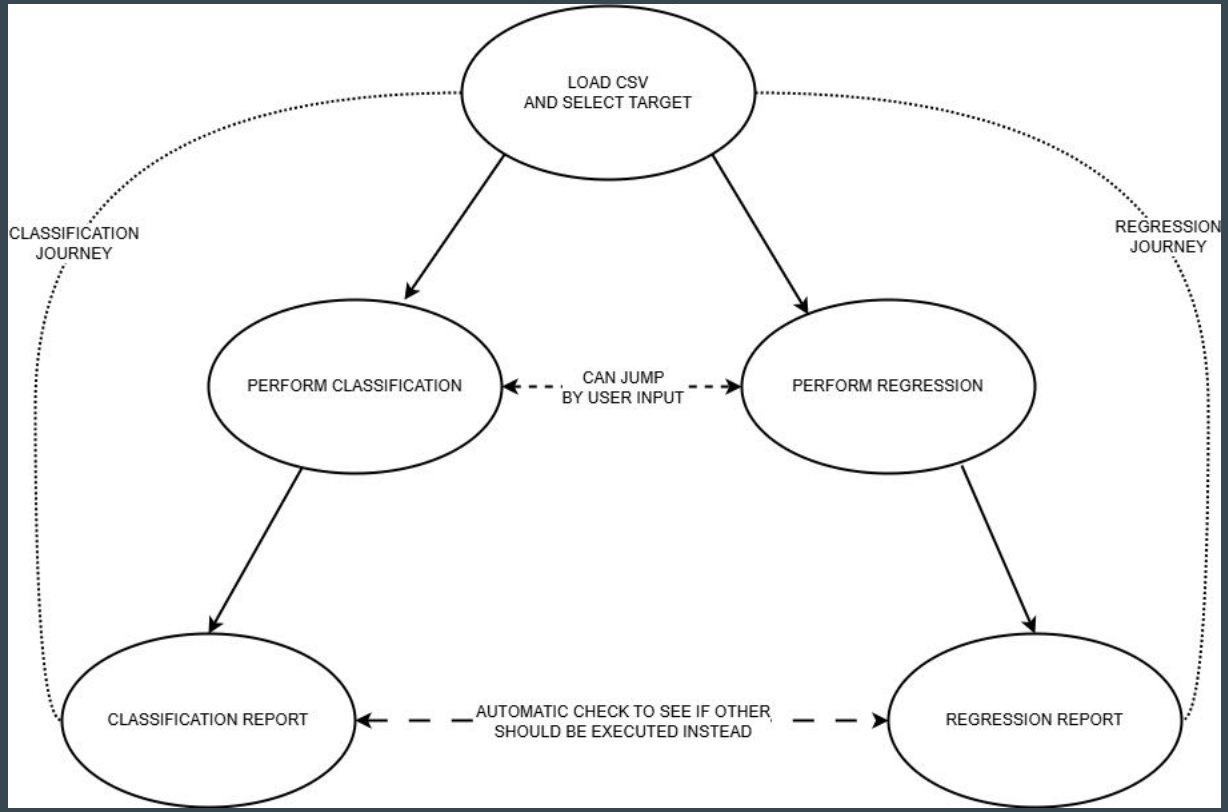
- First, the metadata is checked to see if the function requires prerequisite data. If it does, check for an alternative function and presence of prerequisite data in the DataHolder for the alternative function. If both exist, prompt user to switch to the alternative function and resolve dependencies for the alternate.

- If there is no alternative function, and the function requires prerequisite data, then, check if the data is present in the DataHolder. If the data is missing, execute the dependency resolution process on the prerequisite function as a sub-call, then re-run the dependency resolution process on the current function.

- Otherwise, check if user input is required. If required, get the input from the user, and then execute the called function. Return the result to the user if standalone or to the dependency resolver if run recursively.

- A benefit of this process is that functions can be called out of order and all of their prerequisites can be gracefully executed in order. The user does not need to know the underlying structure of the process.

# User Journey - Function Directed Dependency Graph




- Either journey starts by loading the csv and selecting a target. Then the appropriate models are fit, and finally a report is generated.
- Each function has the function above it as a prerequisite, meaning each function must be executed in order.

# Classification and Regression Validation Checks

- Both the perform_classification and perform_regression functions perform checks to ensure that the data corresponds to the correct task type. If the user tries to perform regression on categorical data, they are redirected to either reselect the target variable or automatically shifted to the perform_classification function. If the user tries to perform classification on continuous data, they are given the choice to reselect the target variable, automatically shift to perform_regression, or to use automatic discretization methods to bin the data and convert the continuous target variable into a categorical one.

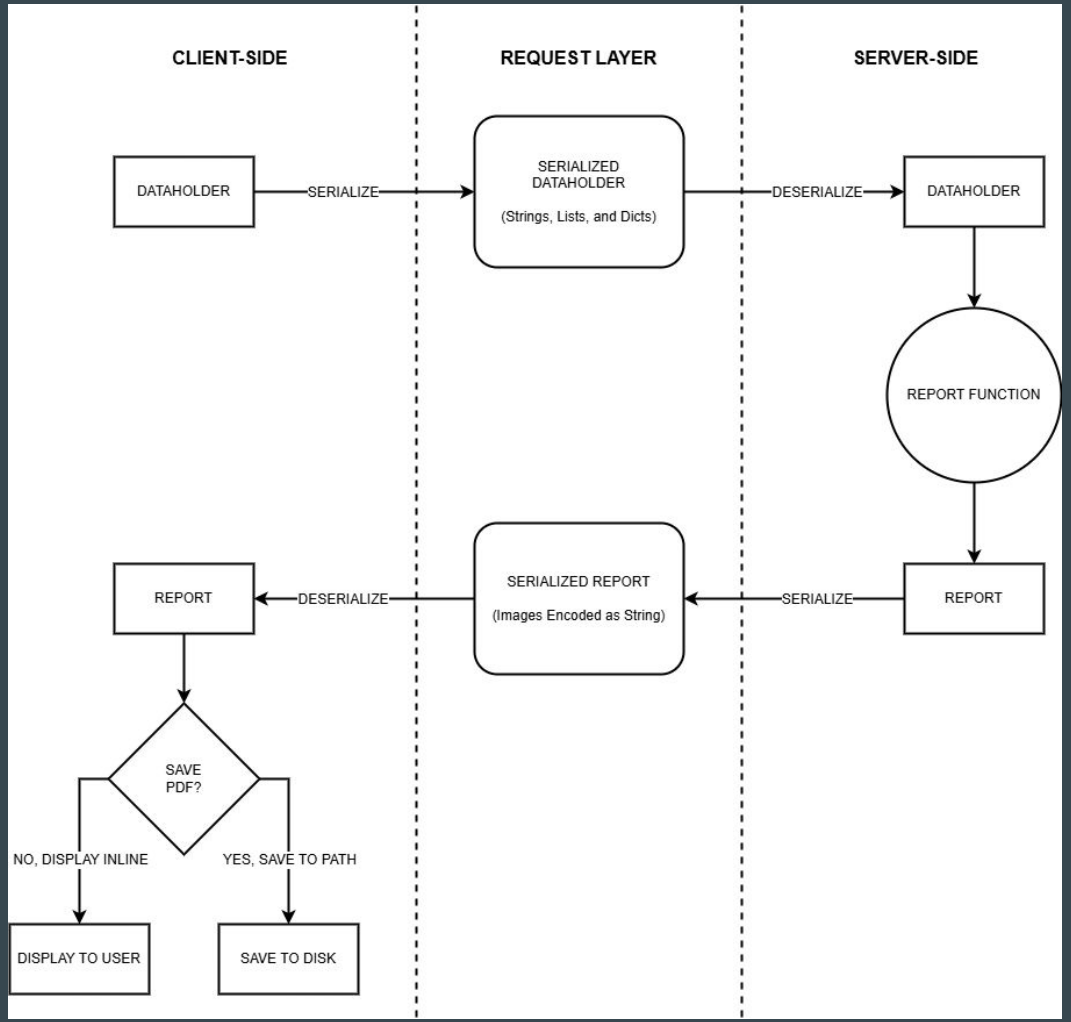
- Because the user may have initially started with one type of machine learning task and been redirected to the other task type, they may try to invoke the wrong type of report function, especially if executing recursively (see later slides). There is logic in the dependency resolver to automatically help the user execute the correct report type function and ensure the user does not try to create a classification report for a regression task and vice versa.

# FastAPI Requirements for the Report Functions

- Both of the report functions are set up to be invoked through the Requests Python package using FastAPI endpoints. Because these functions pass their data through HTTP protocols, data needs to be serialized and deserialized into basic Python types and the raw DataHolder object cannot be passed. A custom DataHolderInput class (inheriting from Pydantic's BaseModel) is used to hold the data as it is being sent through the request before being reconstructed as a DataHolder on the server side.

- The report functions require training and test data, predictions, a fitted model, and model scores.
  - The X data (training and testing) is converted from pandas DataFrames to dictionaries
  - The y data (training and testing) is converted from pandas Series to lists
  - The dictionary of prediction arrays is converted to a dictionary of lists
  - The fitted model is converted to a base64 encoded string using pickle
  - The dictionary of model score arrays is converted to a dictionary of lists

- The report functions return a dictionary of strings containing the report information, which is then rebuilt into a PDF or displayed to the end user. Text can be sent as is, but images are converted into base64 encoded strings using BytesIO and are regenerated client-side back into images.

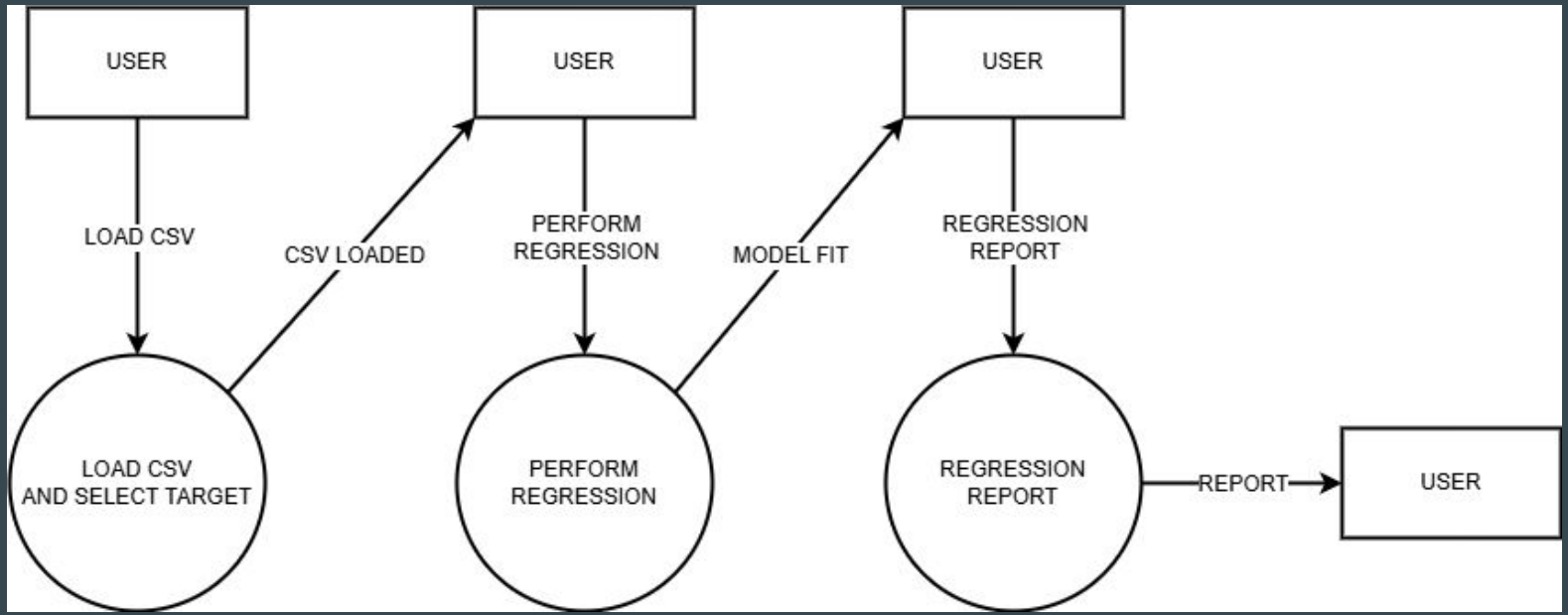



# Serialization Process

For each report function, the dataholder is serialized into a dictionary of basic JSON types on the client side, sent to the server side by an HTTP request, then deserialized back into Python objects.

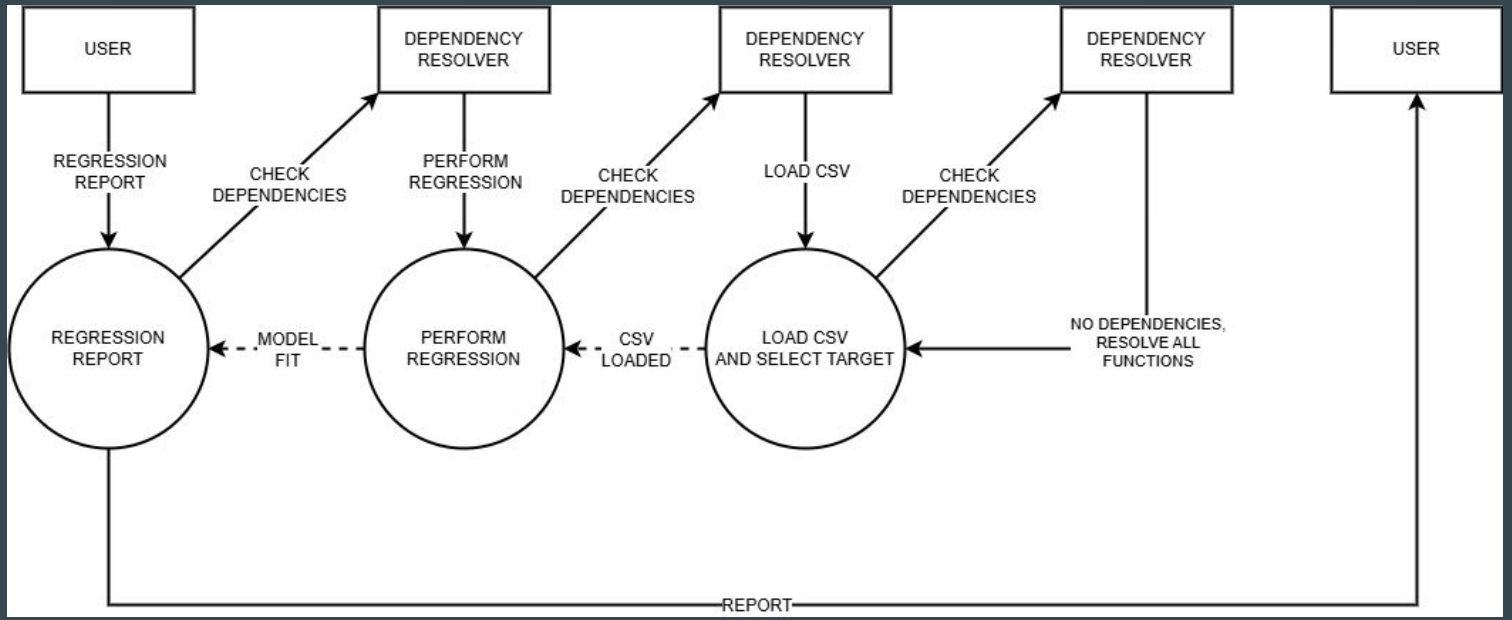
The report is then generated creating images and text on the server side. The report is serialized on the server side into a dictionary of strings, returned as part of the same HTTP request and regenerated into images, strings, and markdown HTML on the client side.

# User Journey - Sequential Execution



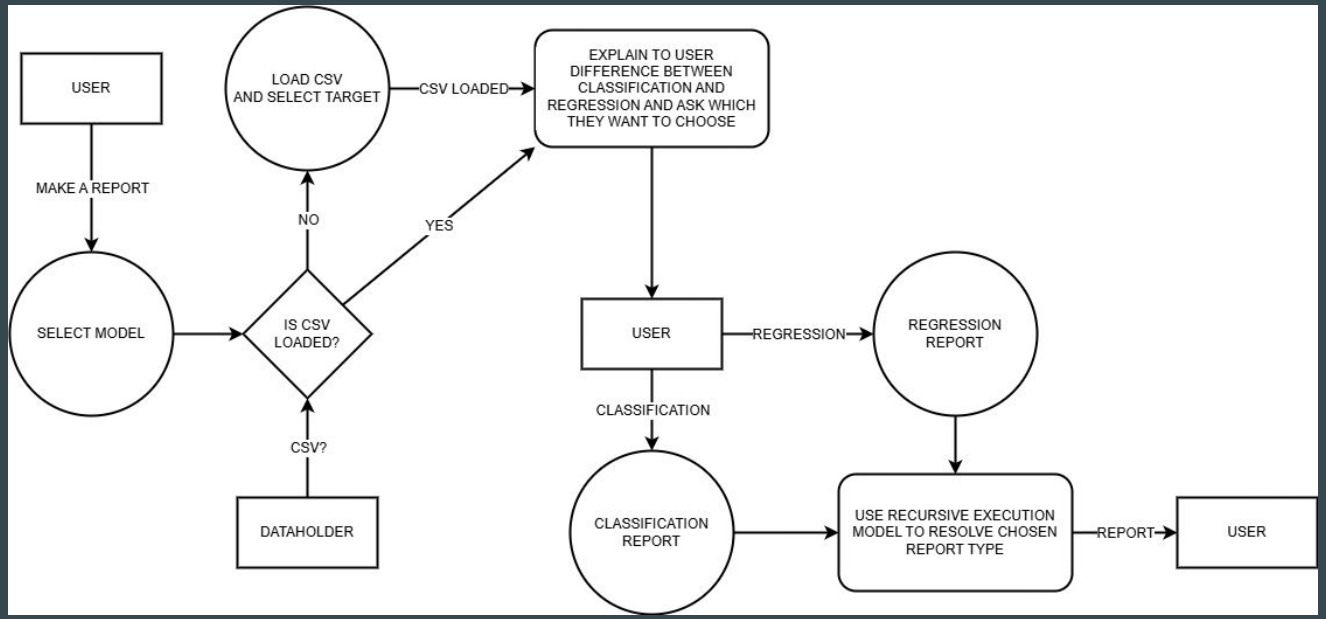

One possible option is for the user to execute each function in order. This requires 3 different prompts to invoke each function individually (intermediate processes omitted). This requires the user to have some knowledge of the underlying process.

# User Journey - Recursive Execution




Alternatively, the user can call functions at the end of the graph (the report functions) and allow the resolver to automatically invoke prerequisite functions in order. This requires the user to know which end result they want (regression or classification).

# User Journey - Guided Execution




The most user-friendly method is to call the SELECT MODEL function to automatically load the data and help the user understand which type of report to generate. This lets the user execute the appropriate function without prior knowledge.