PHUSE SDE 2025 - Paper

ADaM AI: PoC on how to interact with CDISC Library API using Natural language

Saikrishnareddy Yengannagari, Bristol Myers Squibb

ABSTRACT

Is it possible to interact with the CDISC Library API using natural language in plain English? The answer is yes. This paper introduces a web application enabling users to query ADaM variable information (metadata and codelist) through natural language by leveraging AI-driven Natural Language Processing (NLP).

The underlying logic involves initially creating a SAS macro to extract ADaM variable and codelist information from the CDISC Library API, followed by converting this SAS code into Python while preserving its original functionality. This Python script serves as the application's core logic, interfaced with AI to manage both input and output. When a user submits a query in natural language, AI identifies the requested variables, standards (ADaM), and Implementation Guide (IG) versions (selecting the latest by default if not specified), and then passes these parameters to the Python script. The Python output is processed again by AI to deliver a natural language response.

INTRODUCTION

As clinical SAS programmers, we rely heavily on the CDISC Library as the central, authoritative source for standards metadata. Getting the details right for Analysis Data Model (ADaM) variables and their codelists is a crucial part of our daily work. Traditionally, finding this information meant either navigating the CDISC website or using its Application Programming Interface (API) programmatically. While powerful, using the API directly often requires specific technical skills – knowing the right endpoints, crafting the calls correctly, and parsing the structured JSON responses. I found this could be cumbersome, especially when I just needed a quick answer about a specific variable or codelist, rather than building a full integration.

This got me thinking about recent advancements in Artificial Intelligence (AI), particularly in Natural Language Processing (NLP). Could we use AI to create a more intuitive, conversational way to interact with the CDISC Library API? This paper explores that question through a Proof of Concept (PoC) project I developed, called “ADaM AI”.

My goal with ADaM AI was to build a web application allowing users, especially fellow clinical SAS programmers and our managers, to ask for ADaM variable information using plain English. The core idea wasn’t to have the AI generate the information itself – that carries the risk of making things up, or “hallucinating.” Instead, I wanted the AI to act as an intelligent interface or layer. In this setup, the AI interprets the user’s natural language question, figures out the necessary details (like the variable name, standard, and IG version), and then triggers a reliable backend script I wrote (first in SAS, then converted to Python) to fetch the actual data directly from the CDISC Library API. Finally, the AI takes that authoritative information and presents it back to the user in a clear, easy-to-understand, natural language format.

This approach aims to give us the best of both worlds: the ease of asking a question in plain English, combined with the accuracy and reliability of getting data straight from the source. This way, we can minimize the risk of the AI providing incorrect information. In this paper, I'll walk you through the process – starting with the initial SAS macro I created for API interaction, moving to the Python conversion (using a technique I call "vibe coding"), integrating the AI for NLP, developing the user-friendly Streamlit web app, and discussing how this method helps avoid AI hallucinations.

METHODS

Developing the ADaM AI Proof of Concept was a multi-stage process for me. I started with familiar methods for interacting with the CDISC Library API and gradually brought in modern AI and web application technologies. My core approach was always to use AI as an intelligent interface layer, not as the source of the data itself, ensuring I relied on the authoritative CDISC Library API for the actual facts.

Stage 1: Building the Foundation with SAS

My first step was to create a solid SAS macro, which I named adamgenius.sas, to talk to the CDISC Library API programmatically. This felt natural, given that SAS is such a common tool in our clinical programming world for handling data and interacting with external systems. I designed the macro with several key capabilities:

a. **Taking Input**: It needed to accept an ADaM variable name (adamvar) and could optionally take specific ADaM Implementation Guide (adamigversion) and Controlled Terminology (adamctversion) versions.
b. **Handling Defaults**: If someone didn't provide the versions, I wanted the macro to be smart enough to figure out the latest ones. So, I added logic to query the CDISC Library API's /mdr/products endpoint to find the newest ADaM IG version, and similarly used /mdr/products/Terminology to find the relevant CT package version (usually ADaM CT, but falling back to SDTM CT if needed).
c. **Getting the ADaM Structure**: It then fetched the whole structure for the specified (or latest) ADaM IG using an HTTP request to the /mdr/adam/adamig-{adamigversion} endpoint.
d. **Finding the Variable's Home**: I needed to know which dataset (like ADSL or BDS) the variable belonged to, so I included steps to parse the IG structure and find this context.
e. **Fetching the Metadata**: With the context known, the macro made a specific API call to /mdr/adam/adamig-{adamigversion}/datastructures/{dataset}/variables/{adamvar}. This retrieved the detailed metadata I was after – things like the variable's label, data type, core status, and description.
f. **Checking for Codelists**: I parsed the variable metadata's _links section to see if any codelists were linked.
g. **Getting Codelist Terms**: If codelists were present, I had the macro call a sub-macro I wrote (%GetCDISCCodelist). This sub-macro's job was to fetch the right Controlled Terminology package (like /mdr/ct/packages/adamct-{adamctversion} or the SDTM fallback) and pull out the submission values and decoded terms for those specific codelists.
h. **Showing the Results**: Finally, the macro printed the retrieved variable metadata and any codelist terms neatly in the SAS log.

Proc http call to fetch adam variable metadata using SAS:

```
/*-------------------------------------------------------------------------------------------------------/
    Get details of the variable from the API
/-------------------------------------------------------------------------------------------------------*/
filename varcheck TEMP;
proc http
    url="https://library.cdisc.org/api/mdr/adam/adamig-&adamigversion/datastructures/&dataset/variables/&adamvar"
    method="GET"
    out=varcheck;
    headers
        "api-key"="&cdiscapikey"
        "Accept"="application/json";
run;

libname varcheck JSON fileref=varcheck;
```

This SAS macro became my functional baseline. It confirmed that I could indeed extract all the necessary information directly from the CDISC Library API using code.

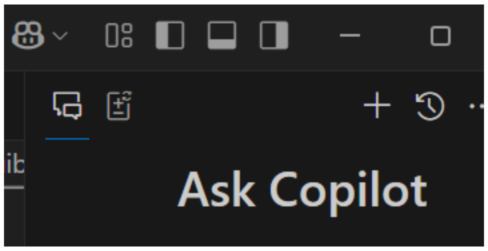
Stage 2: Moving to Python with "Vibe Coding"

Recognizing that Python is increasingly popular in data science and AI, and wanting to build a web-based interface, my next step was to convert the SAS macro's logic into a Python script (adam_genius.py). Since my SAS skills were much stronger than my Python skills at the time, I used a process called "vibe coding" coined by Andrej Karpathy.

"Vibe coding" is basically using AI-powered code assistance tools, like GitHub Copilot within VS Code, to help with the translation. My process looked like this:

a. **Giving SAS Context**: I fed my SAS macro code, piece by piece, to the AI assistant.
b. **Stating My Goal**: I clearly told it what I wanted: to replicate the SAS logic in Python, hitting the same CDISC Library API endpoints.
c. **Refining Iteratively**: I carefully reviewed the Python code the AI generated. I tested it using inputs where I knew the expected output from my SAS macro. If something wasn't right, I'd go back to the AI, explain the problem, and ask for corrections or different ways to approach it, repeating this until the Python script behaved just like the SAS macro.
d. **Using Python Best Practices**: While keeping the core logic the same, I made sure the Python script (adam_genius.py) used standard Python tools and practices. I used the requests library for making the HTTP calls, json for handling the API responses, argparse for command-line arguments, and dotenv to manage the API keys securely.

Github Copilot sample:



Python version of fetching adam variable metadata:

```python
class ADaMMetadataRetriever:
    def get_variable_details(self, adam_variable, adamig_version):

        print(f"Fetching details for {dataset}.{adam_variable} (ADaMIG {adamig_version_hyphen})...")
        url = f"{self.BASE_URL}/mdr/adam/adamig-{adamig_version_hyphen}/datastructures/{dataset}/variables/{adam_variable}
        data = self._make_request(url)

        if not data:
            print(f"ERROR: Could not fetch details for variable {adam_variable} in dataset {dataset}.")
            return None

        # Extract key details (similar to SAS macro output)
        details = {
            "Variable": data.get("name"),
            "Label": data.get("label"),
            "DataType": data.get("simpleDatatype"),
            "Core": data.get("core"),
            "CDISCNotes": data.get("description"), # Or use 'definition' if more appropriate
            "Dataset": dataset, # Add the dataset context
            "ADaMIGVersion": adamig_version_hyphen, # Store the version used
            "CodelistLinks": [],
            "Codelists": [] # To store fetched codelist details later
        }
```

Stage 3: Integrating AI for Natural Language Interaction

With a reliable Python script (adam_genius.py) ready to fetch the data, I then focused on building the AI layer (adamai.py) to allow users to interact using natural language. I used OpenAI's GPT models (specifically gpt-4o-mini because it offered a good balance of cost-effective and efficiency) for two distinct NLP jobs:

a. **Understanding the Query and Extracting Parameters(Input)**: I carefully crafted an AI prompt to make the model act like an expert in ADaM terminology. When a user asked something like, "Tell me about the variable for subject age," the AI's only job was to figure out the most likely ADaM variable name (in this case, "AGE") and return just that name, in uppercase. Giving it such a focused task helped minimize the chance of the AI making stuff up or getting sidetracked at this crucial first step.

b. **Generating the Response(Output)**: After the Python script (adam_genius.py) ran with the extracted variable name (and the right IG/CT versions), it produced structured metadata and codelist information. I then passed this structured output to a second AI prompt. This prompt asked the AI to act as an expert explainer. Its task was to take the factual, structured metadata and rephrase it into a friendly, conversational explanation that would make sense to my target audience (clinical SAS programmers and managers).

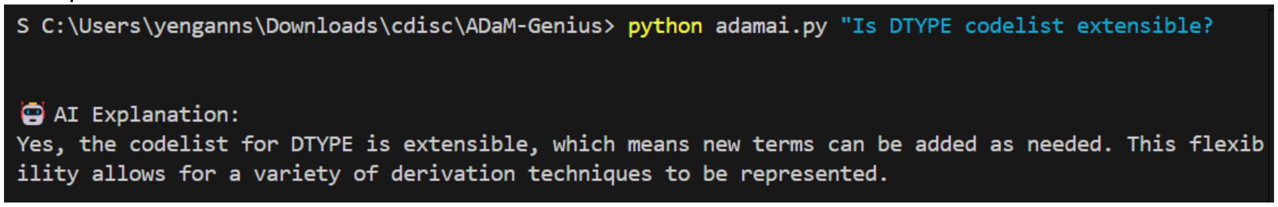
This two-step AI process became the heart of the interaction logic: the AI understands the request, triggers my deterministic script to get the real data, and then the AI makes the factual results easy to understand.

Stage 4: Building the User Interfaces (CLI and Web)

Finally, I developed two ways for people to use this AI-driven workflow:

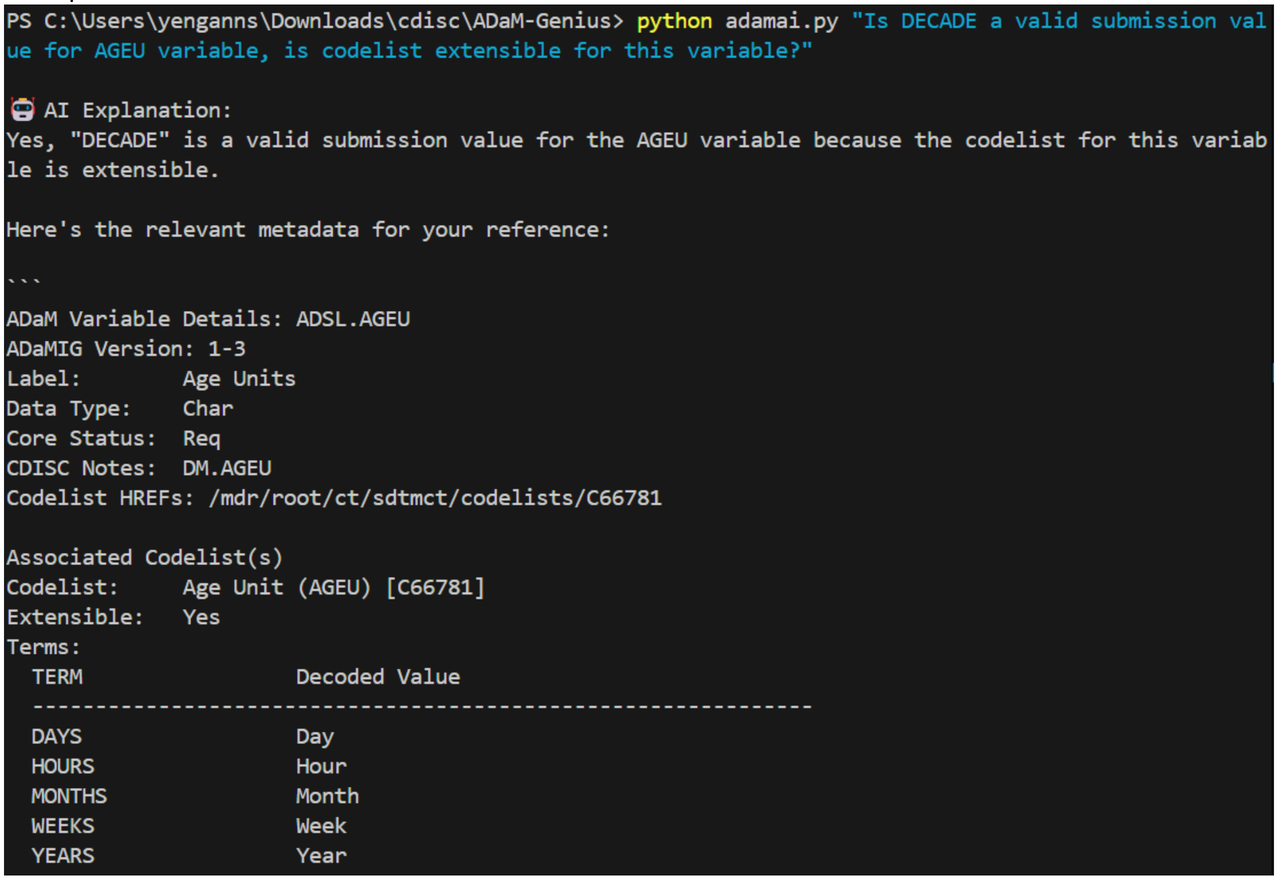
a. **Command-Line Interface (CLI):** I structured the adamai.py script so it could be run directly from the command line. You just provide the natural language query as an argument. The script handles the variable extraction, calls adam_genius.py using Python's subprocess module, and then prints the final AI-generated explanation to the console.

Example 1:



```
S C:\Users\yenganns\Downloads\cdisc\ADaM-Genius> python adamai.py "Is DTYPE codelist extensible?


🤖 AI Explanation:
Yes, the codelist for DTYPE is extensible, which means new terms can be added as needed. This flexib
ility allows for a variety of derivation techniques to be represented.
```

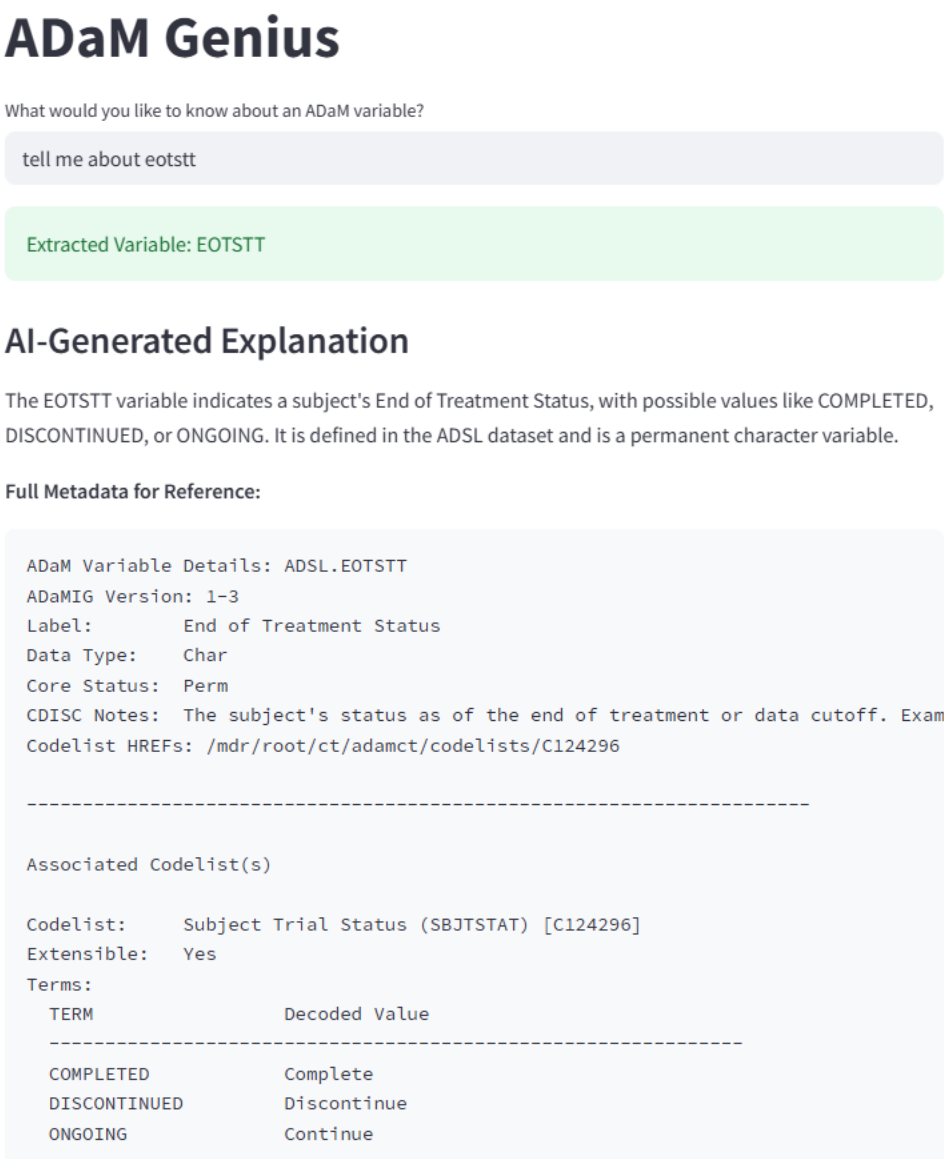
Example 2:



```
PS C:\Users\yenganns\Downloads\cdisc\ADaM-Genius> python adamai.py "Is DECADE a valid submission val
ue for AGEU variable, is codelist extensible for this variable?"

🤖 AI Explanation:
Yes, "DECADE" is a valid submission value for the AGEU variable because the codelist for this variab
le is extensible.

Here's the relevant metadata for your reference:

```
ADaM Variable Details: ADSL.AGEU
ADaMIG Version: 1-3
Label:         Age Units
Data Type:     Char
Core Status:   Req
CDISC Notes:   DM.AGEU
Codelist HREFs: /mdr/root/ct/sdtmct/codelists/C66781

Associated Codelist(s)
Codelist:      Age Unit (AGEU) [C66781]
Extensible:    Yes
Terms:
  TERM                    Decoded Value
  --------------------------------------------------------------
  DAYS                    Day
  HOURS                   Hour
  MONTHS                  Month
  WEEKS                   Week
  YEARS                   Year
```

b. **Web Application**: To provide a more visual and perhaps user-friendly option, I created a Streamlit application (streamlit_app.py). This gives a simple web page with a text box where the user can type their question. When they submit it, the app uses the same functions from adamai.py to extract the variable, runs adam_genius.py (again, via subprocess) to get the metadata (which it shows in a code block for reference), and then displays the nice, conversational explanation generated by the AI.
```

Example of streamlit app response:



ADaM Genius

What would you like to know about an ADaM variable?

tell me about eotstt

Extracted Variable: EOTSTT

AI-Generated Explanation

The EOTSTT variable indicates a subject's End of Treatment Status, with possible values like COMPLETED, DISCONTINUED, or ONGOING. It is defined in the ADSL dataset and is a permanent character variable.

Full Metadata for Reference:

```
ADaM Variable Details: ADSL.EOTSTT
ADaMIG Version: 1-3
Label:        End of Treatment Status
Data Type:    Char
Core Status:  Perm
CDISC Notes:  The subject's status as of the end of treatment or data cutoff. Exam
Codelist HREFs: /mdr/root/ct/adamct/codelists/C124296

----------------------------------------------------------------------


Associated Codelist(s)

Codelist:     Subject Trial Status (SBJTSTAT) [C124296]
Extensible:   Yes
Terms:
  TERM                 Decoded Value
  ----------------------------------------------------------------
  COMPLETED            Complete
  DISCONTINUED         Discontinue
  ONGOING              Continue
```

My Architecture and How I Addressed Hallucination

My overall design emphasizes using the AI for what it's good at – understanding language – while making sure the actual data comes solely from my Python script's direct calls to the CDISC Library API. The flow looks like this:

User Query (Natural Language) -> AI (Variable Extraction) -> Python Script (API Call to CDISC Library) -> Structured Metadata -> AI (Explanation Generation) -> User Response (Natural Language)

This layered approach was my key strategy for minimizing the risk of AI hallucination. I never ask the AI to know the ADaM metadata itself. I only ask it to process the user's request and then to format the actual data that my adam_genius.py script retrieved from the official source. This way, the factual basis of the answer is always grounded in that reliable CDISC Library API call.

Success in Avoiding Hallucination

Most importantly for me, the architectural approach I took successfully avoided the risk of the AI making up ADaM metadata. By limiting the AI's role to understanding the query and formatting the final response, and relying completely on my adam_genius.py script for getting the actual data from the CDISC Library API, I ensured the core information presented was 100% accurate and came directly from the official source. I didn't see any cases where the AI fabricated metadata details; its job was strictly translation and presentation.

DISCUSSION

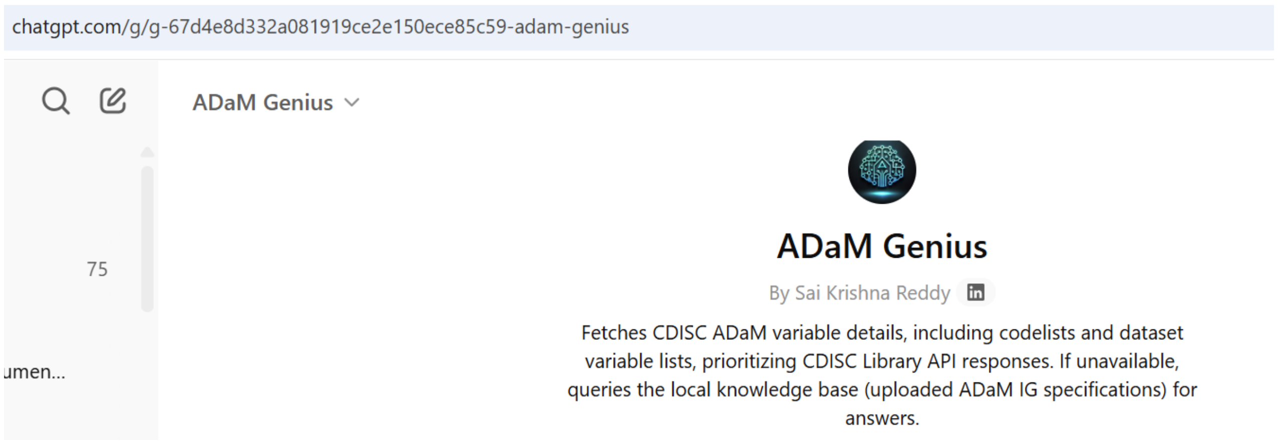
Looking back at the ADaM AI PoC, I think it offers some valuable insights into how we might blend natural language interfaces with authoritative technical resources like the CDISC Library API. My results suggest that AI can genuinely make these resources more usable and accessible, especially for those of us who aren't focused on direct API programming day-to-day, and importantly, we can do this without losing the accuracy of the underlying data.

The architecture I chose – strictly separating the AI's job (understanding language, formatting responses) from the data retrieval job (my deterministic Python script calling the CDISC API) – really seemed to work well for preventing AI hallucination. This feels critical when we're dealing with regulated standards information where getting it right is non-negotiable. By making sure the AI only ever worked with factual data pulled directly from the source of truth, the system felt trustworthy and reliable. This addresses one of the biggest worries people often have about using generative AI. I think this layered approach could be a useful model for other situations where we want a user-friendly AI front-end for a complex but reliable backend system or API.

My experience with "vibe coding" was interesting. It showed me a practical way to use AI code assistants for translating between languages, especially when, like me, you know the source language (SAS) much better than the target (Python). It speeds up the conversion process, but it wasn't magic – it required a lot of careful checking and testing on my part to make sure the AI-generated Python code perfectly matched the original SAS logic and handled all the edge cases correctly. It's a useful technique, but validation is key.

I believe my target audience – clinical SAS programmers and managers – could really benefit from a tool like this. The natural language interface just makes the CDISC Library's information feel more readily available.

It's also worth mentioning that I took the principles from this PoC and built them into an OpenAI Custom GPT called "Adam Genius". I made it publicly available for ChatGPT Plus subscribers. It uses the same core idea, interacting with the CDISC Library API via predefined actions (based on an OpenAPI spec I adapted) to answer natural language questions about ADaM. This further shows the practical potential of using AI as a friendly interface to standards information. You can find it here: https://chatgpt.com/g/g-67d4e8d332a081919ce2e150ece85c59-adam-genius.


chatgpt.com/g/g-67d4e8d332a081919ce2e150ece85c59-adam-genius

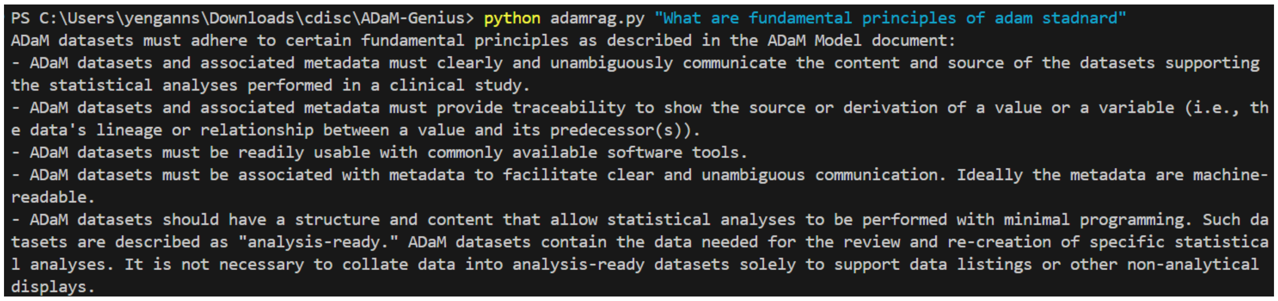
ADaM Genius

By Sai Krishna Reddy

Fetches CDISC ADaM variable details, including codelists and dataset variable lists, prioritizing CDISC Library API responses. If unavailable, queries the local knowledge base (uploaded ADaM IG specifications) for answers.

Further extending this concept, I also developed a complementary component called adamrag.py. While the core ADaM AI focuses on precise metadata and codelist retrieval via the CDISC Library API, I realized many questions relate to the broader context and guidance within the ADaM Implementation Guide itself. The adamrag script addresses this by using Retrieval-Augmented Generation (RAG). It leverages the LlamaIndex library to build a vector index from a local copy of the ADaMIG document (like v1.3). Users can then ask questions in natural language, and the script queries the indexed document to synthesize answers based directly on the IG text. This RAG approach perfectly complements the API method; the API provides live, structured data for specific variables, while RAG provides contextual answers based on the implementation guide's narrative, examples, and principles. Integrating both could offer an even more comprehensive solution for ADaM-related queries. Please note this RAG code can be used on any document; in this case I use adamigv1.3.

Sample RAG query:


```
PS C:\Users\yenganns\Downloads\cdisc\ADaM-Genius> python adamrag.py "What are fundamental principles of adam stadnard"
ADaM datasets must adhere to certain fundamental principles as described in the ADaM Model document:
- ADaM datasets and associated metadata must clearly and unambiguously communicate the content and source of the datasets supporting
the statistical analyses performed in a clinical study.
- ADaM datasets and associated metadata must provide traceability to show the source or derivation of a value or a variable (i.e., th
e data's lineage or relationship between a value and its predecessor(s)).
- ADaM datasets must be readily usable with commonly available software tools.
- ADaM datasets must be associated with metadata to facilitate clear and unambiguous communication. Ideally the metadata are machine-
readable.
- ADaM datasets should have a structure and content that allow statistical analyses to be performed with minimal programming. Such da
tasets are described as "analysis-ready." ADaM datasets contain the data needed for the review and re-creation of specific statistica
l analyses. It is not necessary to collate data into analysis-ready datasets solely to support data listings or other non-analytical
displays.
```

CONCLUSION

In conclusion, this Proof of Concept, ADaM AI, has shown me that using AI as an intelligent interface layer really can make technical resources like the CDISC Library API much easier to use for clinical programmers and managers like myself. By letting NLP handle the user queries and format the responses, while ensuring a deterministic backend script fetches the actual data directly from the authoritative API source, I was able to build a user-friendly tool that maintains a high level of accuracy and avoids the pitfalls of AI hallucination.

My journey from writing a traditional SAS macro to building a Python application integrated with AI, using "vibe coding" along the way, demonstrates a practical path we can take to modernize some of our existing tools and workflows in the clinical programming field. The command-line tool and the Streamlit app I created offer accessible ways to query ADaM variable metadata and codelists without needing deep API knowledge.

Looking ahead, I can see several ways to build on this. We could expand the scope to cover other CDISC standards like SDTM or SEND. We could enhance the NLP to handle more complex questions. Performance could be optimized, perhaps with caching. And maybe this kind of functionality could even be integrated directly into the development environments or data review platforms we use every day. But for me, the core principle remains the most important takeaway: using AI as a facilitator for accessing authoritative data, rather than a generator of it, is key to developing AI-powered tools we can trust in our regulated environment.

REFERENCES

1. CDISC Library API https://api.developer.library.cdisc.org
2. OPENAI API https://platform.openai.com/docs/overview
3. ADaM Genius custom GPT can be accessed at https://chatgpt.com/g/g-67d4e8d332a081919ce2e150ece85c59-adam-genius
4. My GitHub open source code can be found at https://github.com/kusy2009/ADaM-Genius.git
5. Working with dataset JSON using SAS by Lex Jansen https://www.lexjansen.com/pharmasug/2022/AD/PharmaSUG-2022-AD-150.pdf https://github.com/lexjansen/dataset-json-sas.git
6. Enhancing SAS Programming with LLM Integration by Karma Tarap https://pharmasug.org/proceedings/2024/SI/PharmaSUG-2024-SI-362.pdf
7. Deep research on Vibe Coding by Robert Lavigne https://www.linkedin.com/pulse/deep-research-vibe-coding-using-o3-mini-high-across-24-robert-lavigne-axhfc/
8. Enhancing RAG: A study of best practices https://arxiv.org/abs/2501.07391
9. A Streamlit Web Application for the Analysis of Olympic Dataset https://www.jetir.org/view?paper=JETIR2311414

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Rajprakash Chennamaneni for inspiring me to dive into the worlds of Python and AI. I also want to thank Derek Morgan & Nicole Thorne for their constant encouragement, which motivated me to present my work at conferences like this one.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact me at

Saikrishnareddy Yengannagari
Global Macro Programmer, BMS
Email: Saikrishnareddy.y@gmail.com or Saikrishnareddy.yengannagari@bms.com

All code used in this paper can be found at Github: https://github.com/kusy2009/ADaM-Genius.git