PharmaSUG 2026 – Paper PO-395

Implementing AI Agent-Driven Tools to Accelerate Clinical Research Workflows

Hohyun Lee, Clinvia

Abstract

With the rapid emergence of large language models (LLMs), generative AI has evolved from a conversational tool into a powerful platform that has accelerated and changed the ways industries use automation. Beyond just a prompt-level interaction, LLMs now enable the development of Agentic AI tools and workflows to augment statistical programming and increase automation within clinical research. This paper explores the usage of these AI agent-driven tools and the design/workflow of such systems.

Specifically, the primary systems this paper focuses on are an SAP (Statistical Analysis Plans) generator, a Clinical Trial Protocol generator, and a TFL (Tables, Figures, and Listings) Validator. This paper first outlines how to design workflows for building such AI tools. Then, this paper explains the internal logic of each application. Next, this paper describes the steps it takes to communicate effectively with both the LLMs and the user. Finally, this paper demonstrates how to test and analyze these applications to generate consistent, regulatory-aligned SAP and protocol content, as well as validating TFLs through generated python code.

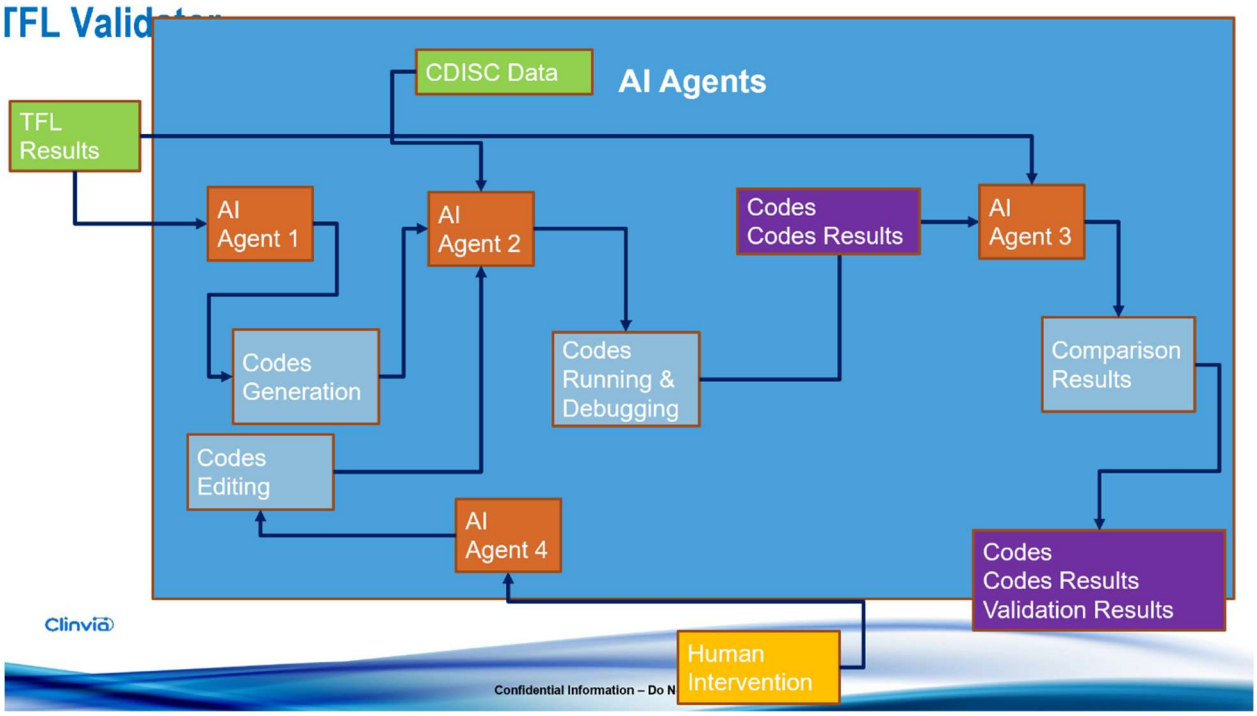
In addition, the development and iteration of these AI agent systems are accelerated through the use of Cursor, an AI-assisted coding environment. This paper also explains how Cursor can be used to generate and refactor code, explain functions and workflows, and even reflect on and resolve errors the applications face.

Altogether, this work demonstrates how AI agents, combined with AI-augmented development tools, can help modernize, accelerate, and transform the statistical field by reducing manual effort, improving consistency, and enhancing autonomy.

AI Agent Workflow

Like any well-designed software system, there should be a clearly defined workflow to outline how your system works. In an AI Agent system, it is essential to clearly indicate and define which LLM/agent is responsible for processing specific types of information. Tasks should be intentionally delegated to their most appropriate AI Agent for efficiency and modularity. Additionally, it should be explicit what input each AI Agent receives and what output it produces. Moreover, it should specify how AI Agents interact sequentially (outputs from one AI Agent as inputs to another).

Below is an example of an AI Agent Workflow:



In the example above, it can be seen how the process of this workflow is split up. For context, this is our TFL Validator Application that takes in a TFL and generates code to produce the values found in the TFL result and validates whether the TFL result is accurate or not. In this workflow, each AI Agent is shown with a specific purpose.

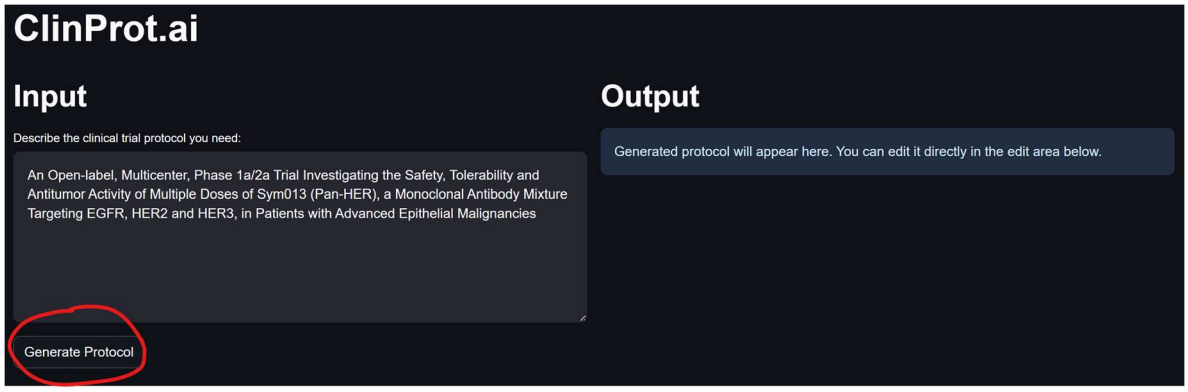
- AI Agent 1 taking in the TFL results to begin generating code
- AI Agent 2 then checks if the code is running properly and debugs if needed
- AI Agent 3 checks over the code results and produces a comparison table that shows which values the TFL is consistent with our code and add on to our code and code result output.
- AI Agent 4 is also used if the user would like to modify or add on to the code and comparison generated.

As shown, this structured design keeps the workflow easy to maintain and straightforward with how the system should run, improving interpretability, maintainability, and scalability of the overall system.

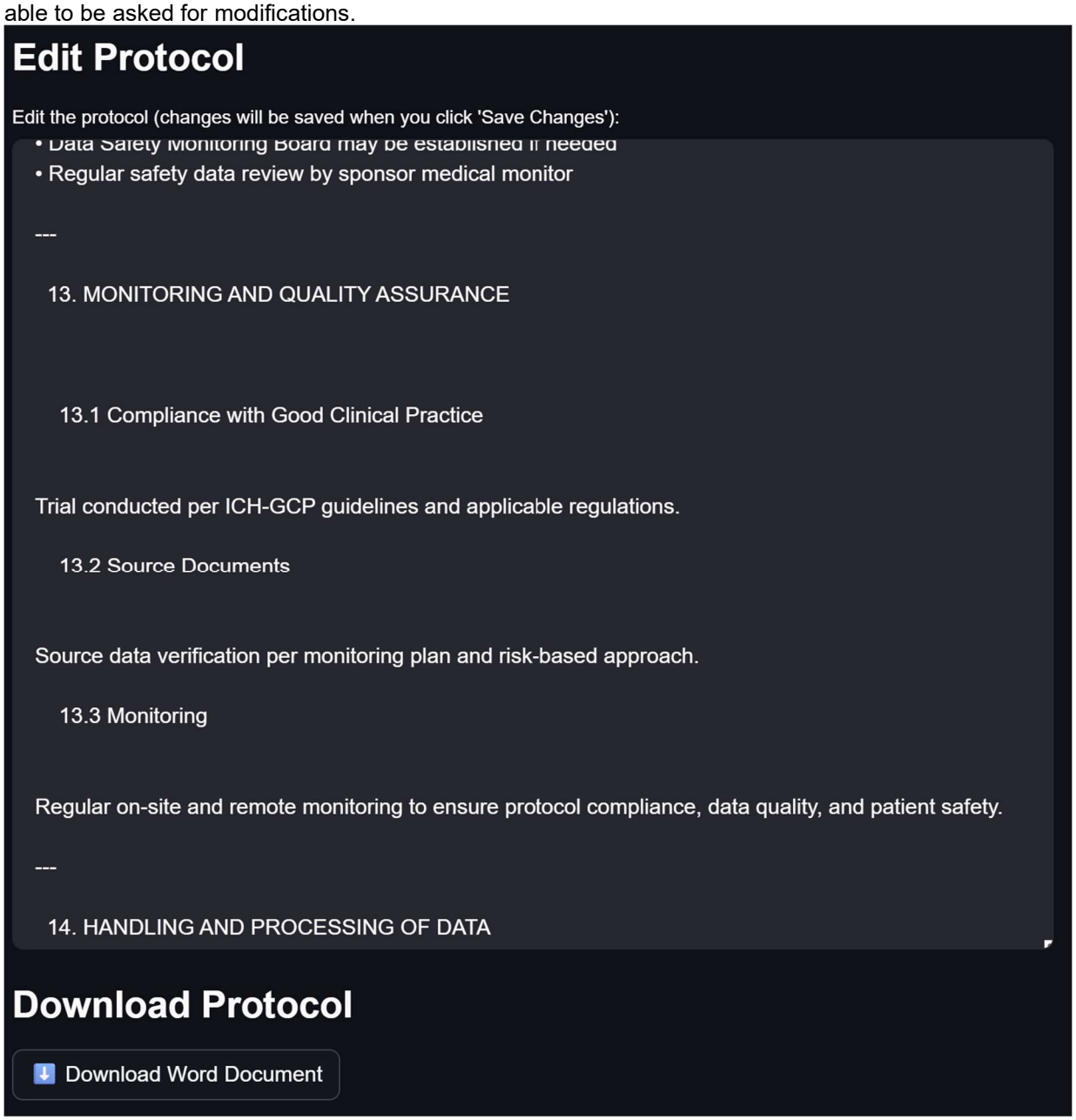
Document Generators

Clinical Trial Protocol Generator

The Clinical Trial Protocol Generator, titled ClinProt.ai, takes a simple trial title input and generates an 18 section length protocol word document with information automatically filled in based on the input and following regulations. The system requires only to simply input the title in the textbox and click generate protocol.



The word document is then able to be downloaded and/or edited for the user's own discretion or is also able to be asked for modifications.



Now, how the application works is that with the information that the user inputs, it is sent to the LLM (implemented using Anthropic) and all other AI agents, along with a user prompt and a system prompt. What a user prompt does is it explains to the LLM what their task is and the instructions and information it needs to know to specifically fulfill the request and requirements needed. A system prompt is what tells the LLM what their role is and guidelines/requirements they need to follow whenever they are generating a request. Typically, the system prompt is sent to the LLM every time it is run in the application and is kept the same every time it is sent.

Example system prompt:

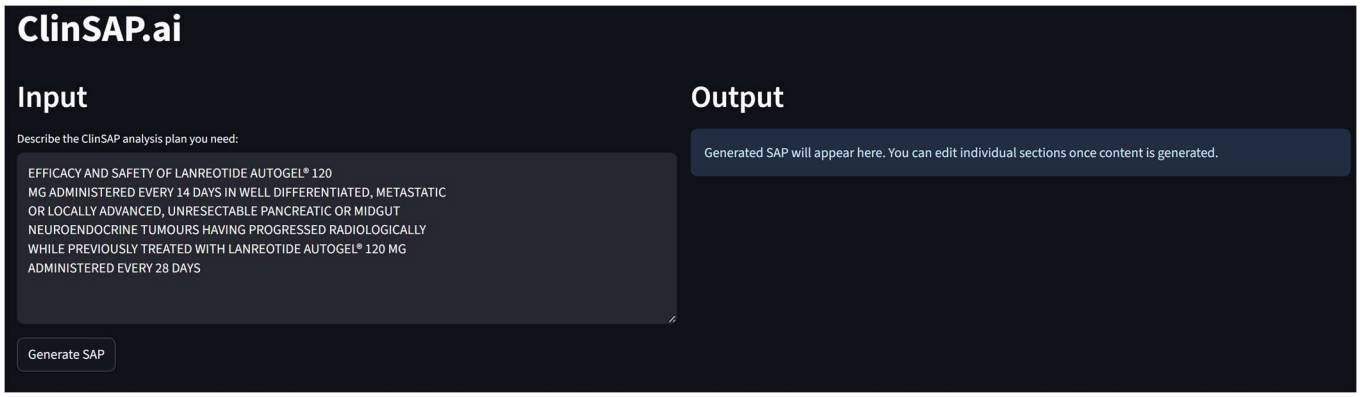
```
## Role

You are an expert medical protocol writer. Generate a concise clinical trial
protocol that covers all essential sections while keeping each section brief
and focused.


## Guidelines

1. Keep each section brief but complete

2. Use bullet points for lists

3. Focus on essential information only

4. Use clear, concise language

5. Maintain ICH-GCP compliance

6. Include all required sections

7. Use standard medical terminology

8. Keep descriptions brief but clear

9. Use tables where appropriate

10. Focus on key requirements only
```

Using the prompts and information given, the LLM then generates the protocol information, and the application processes the information into a word document for the user to be able to download and use. Once generated, if the user decides to send a modification request for the document, that request

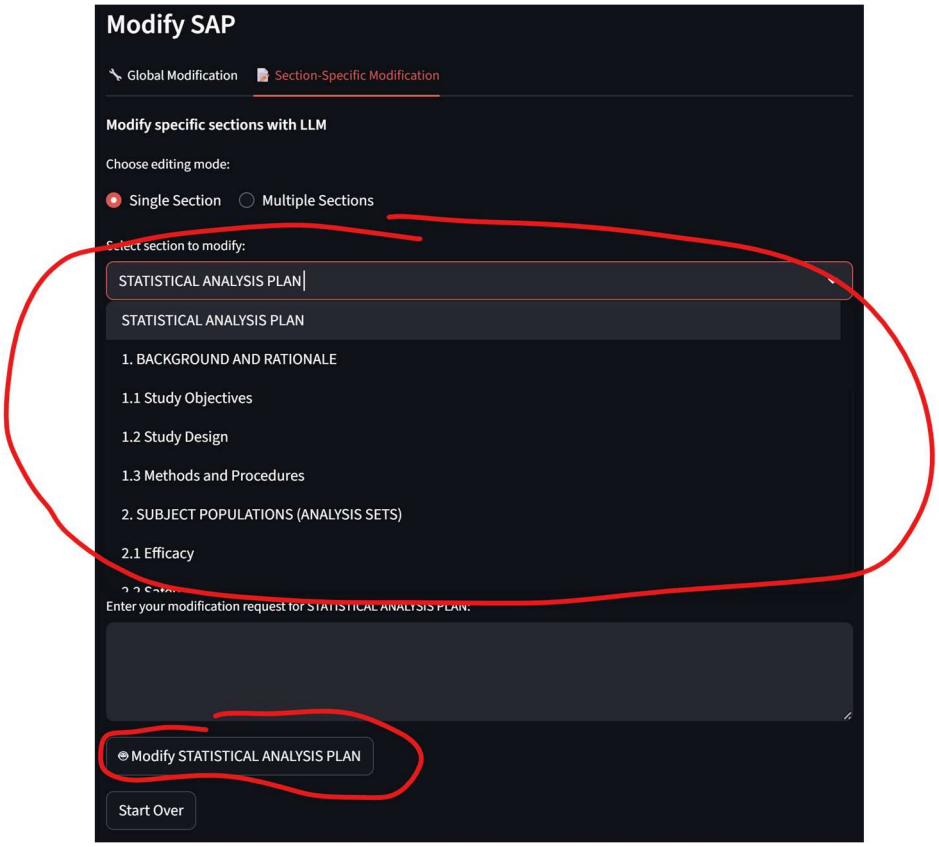
information and a different user prompt + the original system prompt is then sent back to the LLM to modify the document accordingly.

Statistical Analysis Plan Generator

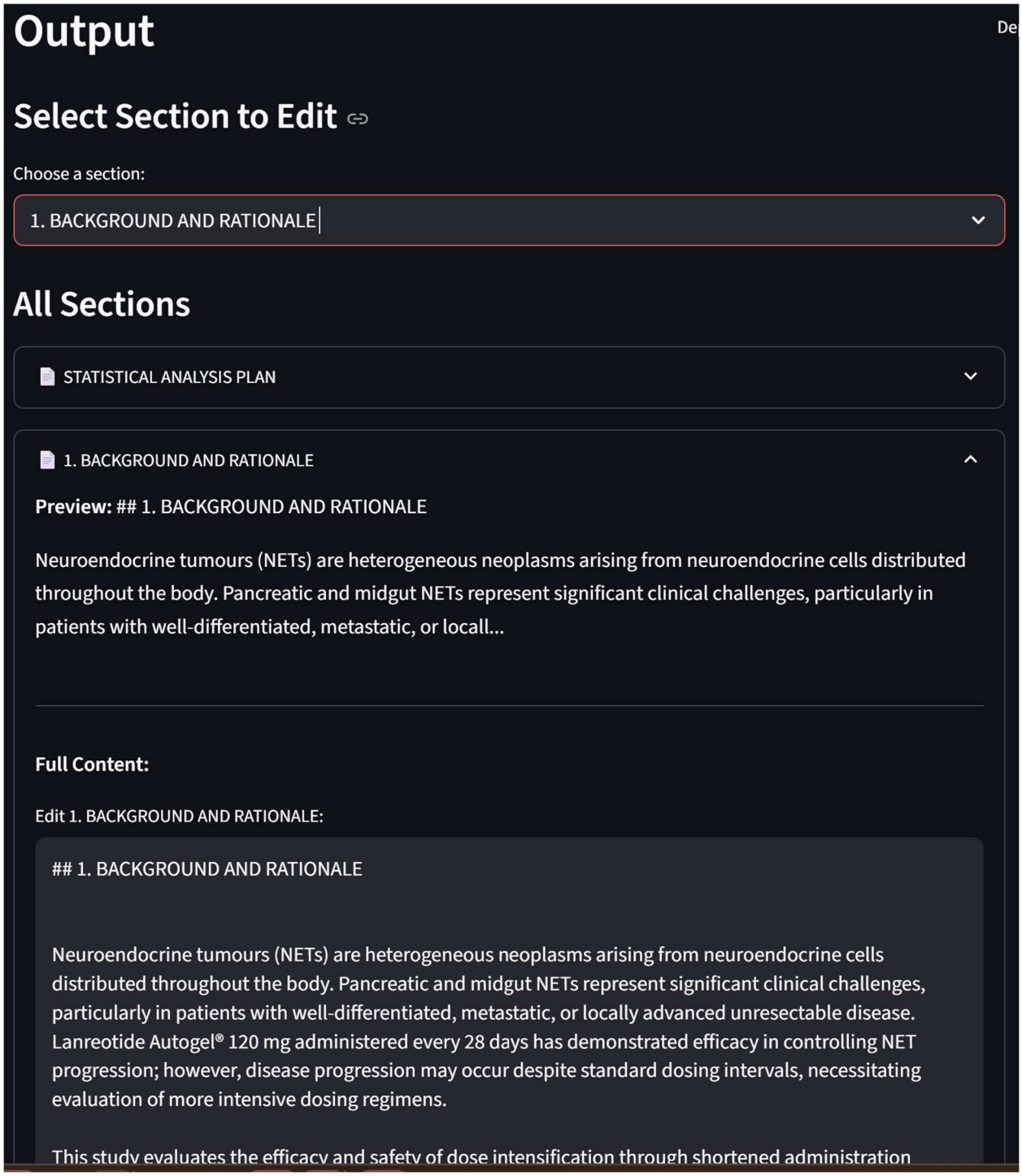
The SAP Generator is very similar to the Protocol Generator but has additionally functionality to the application. It takes a simple protocol title and then generates a 6 section length Statistical Analysis Plan.



The SAP can then also be downloaded into a word document and/or edited for the user's discretion, along with allowing for modification requests. However, this application also allows users to modify specific sections so that if the user only wants to make a change to a specific section with a quick fix, they're able to make that request. The user simply needs to select the section(s) they want to modify and the user can then ask what they want to modify in that specific section(s).



They can also choose to edit directly and save their changes if they would like to update directly on the application for future use.



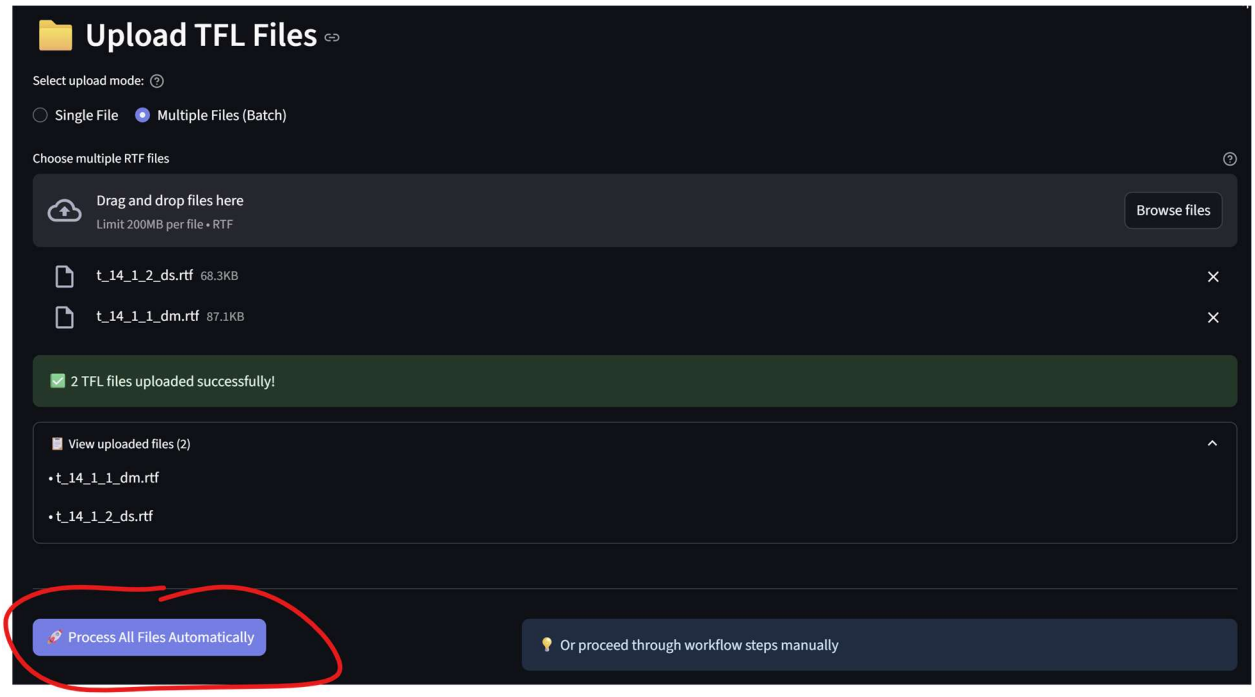
Again, SAP Generator follows a very similar AI Agent process to the Protocol Generator, with additional functionality for section-level modifications. The LLM first takes the title input that the user enters and sends it to the first AI Agent with a user prompt and system prompt. This then generates our SAP document to be downloaded or modified based on the user's decision.

The user then has the option to modify the document as a whole or a specific section. If they choose to modify as a whole, then the same process happens as it did in the Protocol Generator, the generated document, modification request, and the modification user and system prompt are sent back to the LLM to modify accordingly. If the user chooses to do a specific section, only that section of the generated document will be sent to the LLM, along with the modification request and the same user + system prompts as the other modification option. The specific section option is important as it allows the AI Agent to take less time focusing on aspects of the document it doesn't need to and allows for a quick change that takes seconds to edit.

Both these generators allow for documents that may take hours or even days to create, to be done easily for the user to work with and edit in less than a minute. Additionally, with the modification feature, it can even shorten time to finalize documents which can take weeks or even months to fully finalize!

TFL Validator

For the TFL Validator, the process and workflow are a bit more complex. To begin, the user will have the option of uploading one or more TFL files that they would like to validate. Then to begin the automated process, the user will click process all files automatically.



For context, the data that is being used is from CDISC and this is one of the example TFLs we will be using:

Table 14.x.x.x
Demographic Table
All Patients

Characteristic Category/Statistic	Placebo (N=86)	Xanomeline Low Dose (N=84)	Xanomeline High Dose (N=84)	TOTAL (N=254)
Age at Informed Consent				
n	86	84	84	254
Mean (SD)	75.2 (8.59)	75.7 (8.29)	74.4 (7.89)	75.1 (8.25)
Median	76.0	77.5	76.0	77.0
Q1, Q3	69.0, 82.0	71.0, 82.0	71.0, 80.0	70.0, 81.0
Min, Max	52.0, 89.0	51.0, 88.0	56.0, 88.0	51.0, 89.0
Sex, n (%)				
Female	53 (61.6)	50 (59.5)	40 (47.6)	143 (56.3)
Male	33 (38.4)	34 (40.5)	44 (52.4)	111 (43.7)
Ethnicity, n (%)				
Hispanic or Latino	3 (3.5)	6 (7.1)	3 (3.6)	12 (4.7)
Not Hispanic or Latino	83 (96.5)	78 (92.9)	81 (96.4)	242 (95.3)
Race, n (%)				
White	78 (90.7)	78 (92.9)	74 (88.1)	230 (90.6)
Black or African American	8 (9.3)	6 (7.1)	9 (10.7)	23 (9.1)
Asian	0	0	0	0
American Indian or Alaska Native	0	0	1 (1.2)	1 (0.4)
Native Hawaiian or Other Pacific Islander	0	0	0	0
Multiple	0	0	0	0

There is then python code generated to find the same data using data sets that were already integrated in the application and for the user to have and test on their own.

Snippet of a section of code:

```python
import pandas as pd
import numpy as np
import os
from pathlib import Path

def load_dataset_case_insensitive(filename, cdisc_data_path):
    """Load dataset with case-insensitive filename matching"""
    try:
        # List all files in directory
        available_files = os.listdir(cdisc_data_path)

        # Find matching file (case-insensitive)
        matching_file = None
        for file in available_files:
            if file.lower() == filename.lower():
                matching_file = file
                break

        if matching_file:
            file_path = os.path.join(cdisc_data_path, matching_file)
            df = pd.read_sas(file_path, encoding='utf-8')
            # Convert column names to uppercase for consistency
            df.columns = df.columns.str.upper()
            return df
        else:
            print(f"File {filename} not found in {cdisc_data_path}")
            return None
    except Exception as e:
        print(f"Error loading {filename}: {str(e)}")
        return None

def list_available_datasets(cdisc_data_path):
    """List all available datasets in the CDISC data path"""
    try:
        return [f for f in os.listdir(cdisc_data_path) if f.endswith('.xpt')]
    except Exception as e:
```

The code is then automatically run through a virtual python environment to check for any errors and debug if necessary. Finally, the application sends out a comparison table to the user for values that the generated code found to be the same as the TFL inputs and values that were found to be different.

Validation Comparison Table

Section	Subcategory	Placebo (N=86)	Xanomeline Low Dose (N=84)	Xanomeline High Dose (N=84)	TOTAL (N=254)	Notes
Age at Informed Consent						
	n	✅ 86	✅ 84	✅ 84	✅ 254	All match exactly (TFL: 86, 84, 84, 254)
	Mean (SD)	✅ 75.2 (8.59)	✅ 75.7 (8.29)	✅ 74.4 (7.89)	✅ 75.1 (8.25)	All match exactly (TFL: 75.2 (8.59), 75.7 (8.29), 74.4 (7.89), 75.1 (8.25))
	Median	✅ 76.0	✅ 77.5	✅ 76.0	✅ 77.0	All match exactly (TFL: 76.0, 77.5, 76.0, 77.0)
	Q1, Q3	❌ 69.25, 81.75	✅ 71.0, 82.0	❌ 70.75, 80.0	✅ 70.0, 81.0	Differences in quartiles. Code output: 69.25, 81.75 vs TFL: 69.0, 82.0 (Placebo); Code output: 70.75, 80.0 vs TFL: 71.0, 80.0 (High Dose)
	Min, Max	✅ 52.0, 89.0	✅ 51.0, 88.0	✅ 56.0, 88.0	✅ 51.0, 89.0	All match exactly (TFL: 52.0, 89.0; 51.0, 88.0; 56.0, 88.0; 51.0, 89.0)
Sex, n (%)						
	Female	✅ 53 (61.6)	✅ 50 (59.5)	✅ 40 (47.6)	✅ 143 (56.3)	All match exactly (TFL: 53 (61.6), 50 (59.5), 40 (47.6), 143 (56.3))
	Male	✅ 33 (38.4)	✅ 34 (40.5)	✅ 44 (52.4)	✅ 111 (43.7)	All match exactly (TFL: 33 (38.4), 34 (40.5), 44 (52.4), 111 (43.7))
Ethnicity, n (%)						
	Hispanic or Latino	✅ 3 (3.5)	✅ 6 (7.1)	✅ 3 (3.6)	✅ 12 (4.7)	All match exactly (TFL: 3 (3.5), 6 (7.1), 3 (3.6), 12 (4.7))

This can then be downloaded as a word document as well in any case where the user may need a report of the validation. Additionally, the TFL validator also has the option for the user to make modification requests for the code and/or edit the code directly to test in the virtual python environment.

This system is implemented as a multi-agent architecture consisting of distinct components responsible for generation, execution, validation, and modification.

1. The files that the user uploads are sent to the LLM along with our prompts and data set to which the LLM then decides which data set(s) is most appropriate to use.
2. It then analyzes what values are needed and generates python code that matches the type of data in the TFL file.
3. The outputted code proceeds to then run through the virtual python environment and resolves any errors found by sending the errors back to the LLM again, along with the generated code, for it to debug and fix until solved.
4. The output of the final generated python code is then sent to the LLM one last time along with the original TFL input values and is asked to create a comparison table to output to the user and provided as a downloadable word document along with the final generated code for the user to keep and test themselves.
5. If at the end, the user would like to make changes to the code, they can then ask for modifications to be sent back to the LLM or edit directly to the code to be run through the python virtual environment and shown a new comparison table.

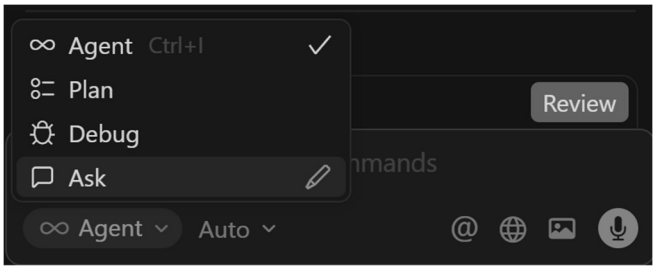
Each time the LLM is run, it uses the corresponding prompt for its task when needed including: code generation, code debugging, comparison, and modifications. This is also what is shown in our AI Agent Workflow example.

Although a complex process, the user will now be able to simply input the TFL files they would like to validate and see a visible comparison table, as well as the code used to validate so that they may test it themselves.

Testing and Fixing the Applications Using Cursor

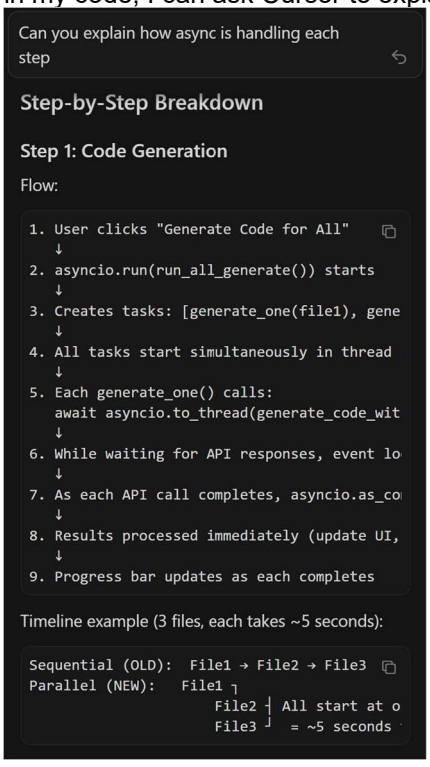
Of course, all applications are never perfect. Therefore, it's important to understand how to test and fix these applications so that they can run as well as possible. One method that allows this process to be streamlined is by using Cursor, an AI-assisted coding environment.

By opening Cursor with your coding platform (I'm using VS Code with Cursor), it can have access to your code and begin to analyze and understand your entire application so that you can ask it any questions or changes to make.

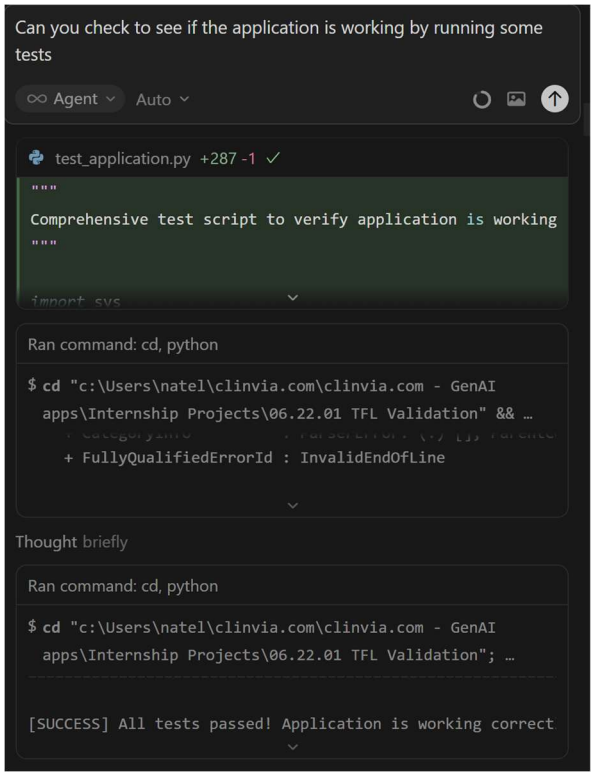


You can either use agent to allow Cursor to automatically make changes to your code based on your request, plan to process what you want Cursor to do and then make the changes when ready, debug to fix errors in your code, or ask to understand aspects of the code or application.

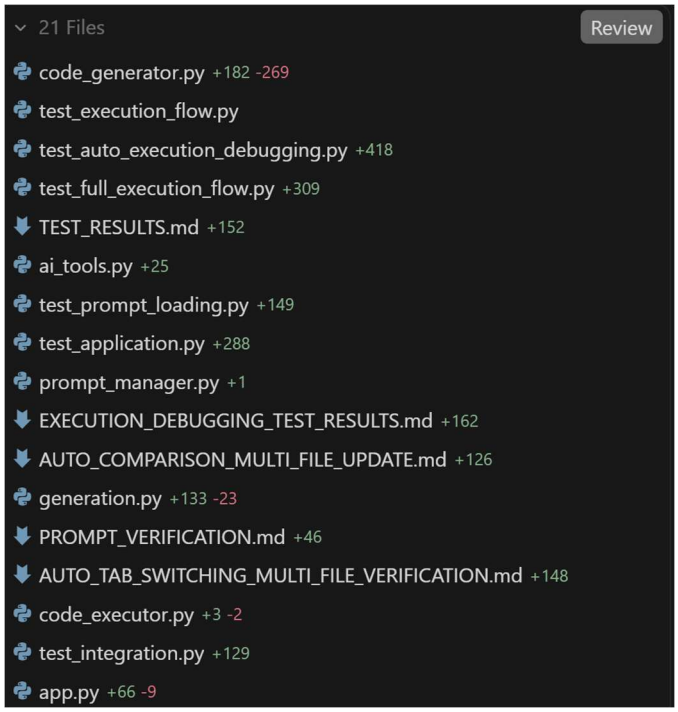
An example is for the TFL validation tool, it uses async.io and if I were curious how it's exactly being used in my code, I can ask Cursor to explain it step by step for me. This being a snippet of the output:

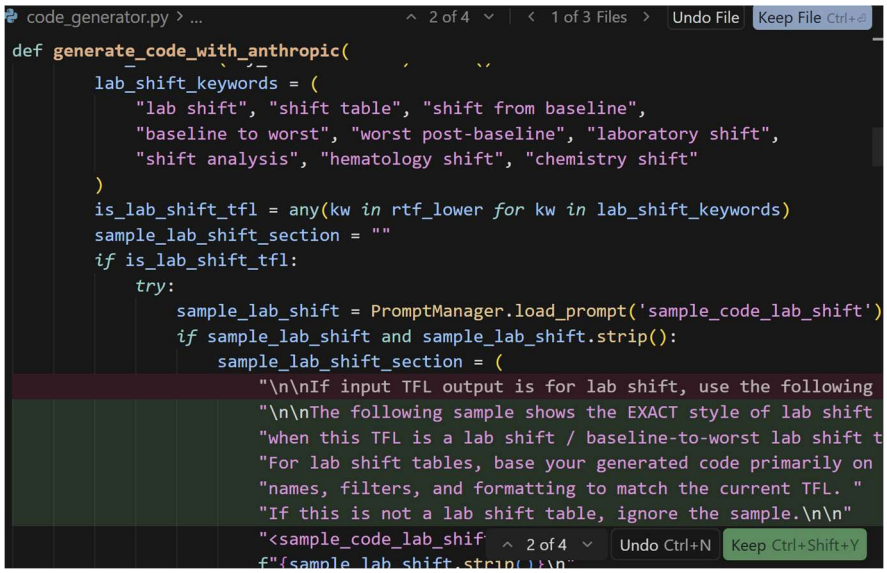


It can also detect errors that the application faces while it's running and is able to debug when prompted to after the application is done running. This enables easy understanding of what to fix and can help with future testing. In addition to testing, since Cursor understands well on how the application runs, it can also be used to create test cases for the applications and even run them on its own to find any errors that may need fixing. After all, AI understands AI the best.



As shown, testing can be done easily within the program without having to run the application each time and if during testing Cursor finds errors in the program, then it can automatically fix your code until it is properly functioning. You are also able to check on the changes made by Cursor and see exactly what it added or deleted and choose whether or not you would like to keep those changes as seen below.





Conclusion

In conclusion, this paper demonstrates how AI agent–driven tools, combined with AI-augmented environments, can significantly accelerate and enhance clinical research workflows. By integrating structured user inputs, LLM-based reasoning, and deterministic validation mechanisms, these types of systems can strengthen automation while maintaining transparency and control in regulated environments. As these tools continue to evolve, manual effort will be significantly reduced and, thus, improve reproducibility, accelerate developments, and faster usage and results for not only statisticians, but for all types of fields in the industry.

Contact Information

Questions and comments are welcomed. Please contact the author at

Nathan Lee

267-736-4477

hnlee1494@gmail.com