Paper SM06

Why I can't see myself coding that much anymore

Angelo Wibbeler, Daiichi-Sankyo, Munich, Germany

Abstract

Various presentations/papers have shown that simple standardizations of specification documents can automate many programming tasks, including deriving simple variables. This reduces duplication of information, enhances consistency, and speeds up development.

With the advent of Large Language Models (LLMs), it is now possible to use less standardized information from specification files and generate sample code for the remainder of programming tasks that can't be automated yet. This can be used as a starting point of the programming process.

This presentation, using the ADaM programming process as an example, will discuss:

- Which programming tasks can be automated and which code should not be written
- How to leverage LLMs and some simple scripts to write entire ADAM programs without manual coding
- How to stay flexible despite adding standardizations.
- A real-world comparison of this approach versus conventional programming
- Current automation limits, considerations when using LLMs, and future prospects

Introduction

In recent years, the landscape of statistical programming has undergone significant transformation, driven by advances in automation, standardisation, and artificial intelligence. This paper explores the shifting role of the programmer in light of these developments, focusing particularly on the ADaM programming process within the pharmaceutical industry. With the increasing prevalence of Large Language Models (LLMs) and metadata-driven tools, tasks that once required intensive manual coding are now becoming candidates for automation or semi-automation. This evolution not only promises improvements in efficiency and consistency but also prompts a fundamental reconsideration of what programming should entail in a modern workflow.

Drawing on practical examples and a real-world comparison between traditional and AI-driven approaches, this paper discusses which programming tasks can and should be automated, how specification documents and LLMs can be harnessed to generate entire ADaM dataset programs, and the limitations and future prospects of these methods. By examining both the technical and organisational implications of a more automated future, the paper aims to offer insights for programmers and decision-makers striving to strike the right balance between standardisation, flexibility, and human oversight.

Software and tools

The approach outlined in this paper for producing statistical programming code was implemented solely using Python version 3.12. For the automated generation of code, GPT-5 by OpenAI was used as LLM. Statistical programming code shown is written in SAS®.

Don't Repeat Yourself — Build Once, Trust Everywhere

One of the most fundamental and widely endorsed best practices in programming is the DRY (Don't Repeat Yourself) principle. DRY serves as a cornerstone for producing maintainable, reliable, and scalable code. But rather than merely warning against the perils of copying and pasting code, DRY advocates for capturing every piece of knowledge or logic in a single, authoritative place within the system. This approach ensures that all references to specific information or logic consistently point to the original source, significantly reducing the likelihood of errors or inconsistencies caused by duplication of information.

In statistical programming, this means if a concept or calculation is clearly defined in a specification, it should be directly reused by the code, instead of re-implemented it. This approach preserves consistency and reduces the risk of errors from manual rewrites. When updates are necessary, changes need only be made once, allowing all dependent programs to align automatically.

Fully embracing DRY means treating specification documents as authoritative sources and directly integrating their content into programming workflows. This not only minimises redundancy but also supports clearer, more maintainable, and more robust systems—far beyond the simple avoidance of copy-paste coding.

Code that should not be written

As Robert C. Martin aptly stated, “The best code is no code at all.” This principle resonates deeply when considering the DRY philosophy. Rather than striving to write clever or extensive code, the true goal should be to eliminate unnecessary implementation wherever possible, especially when the logic or information already exists in a clear, authoritative form.

The following is a simplified example of code used to finalize a dataset by sorting and assigning attribute statements—an archetypal case of code that should not be written manually, and is already automated in nearly every organization:

```
data adam.adsl;
    retain USUBJID ...;
    keep USUBJID ...;
    attrib
        USUBJID length=$20 label="Unique Subject Identifier"
        ...
    ;
proc sort; by usubjid;
run;
```

The handling of this and similar tasks is automated in most companies and commonly abstracted into extensive macros systems but should be called by simple generic macros like:

```
%load_input_datasets(program = adsl);
%autocall_vars(program = adsl, output=autocall_0, vars = STUDYID SUBJID SITEID ...);

/* Non-automated variables code START */
...
/* Non-automated variables code END */
%finalize_adam(program = adsl);
```

For a more detailed exploration of metadata driven automation, see [1].
There is no definitive boundary for how far automation should be taken. As automation becomes more complex, the anticipated return on investment may never materialise. Ultimately, each organisation must determine for itself what level of automation is appropriate. Nonetheless, the examples provided above represent a sensible minimum standard.

How to write entire programs using AI and a specification

Even though full automation is likely not achievable or unlikely to pay off, the remainder of the specification still holds valuable instructions for LLM’s to prepare programs that either are ready to use or can be used as a starting point for the programmer. Some preprocessing and standardisations greatly improve the quality of AI written programs. But before we start, a minor addition to the specification needs to be done. Like many other companies, an additional column to indicate how variables are derived has been added, called [Derivation type]. This column is used to indicate for each variable if it will be generated automatically (without AI code) or the LLM shall be prompted (CodeSuggestion) or left empty if programmers just want to write the code themselves. A generic example of the column is shown below:

Derivation type
source
coded
...
CodeSuggestion

Step 1: Finding dependencies within the dataset

As variable dependencies can be complex, LLM's tend to get confused when handed over an unstructured specification and the simple instruction to write a complete program. To reduce complexity a blueprint for the program needs to be created. To achieve this, each variable is initially assigned a dependency level, which determines the sequence in which it need to be derived.

The dependency level of variables builds up by sequentially referencing upon another, starting at 0 for variables that are independent on any other variable within the same dataset (outside references are allowed). The dependency level is then determined by the highest dependency level of all referenced variables in the derivation plus 1.

In the below example, variable SITESET only references to an outside dataset and gets the dependency level 0 assigned. SITESETN references SITESET and thus gets the dependency level of SITESET assigned plus 1, which adds up to dependency level 1. In the same way, SITEGRP gets the dependency level 2 assigned.

Variable	Source / Derivation	Order	Dependency Level
SITESET	ZS.ZSSTRESC where ZS.ZSTESTCD = "SITESET".	5	0
SITESETN	Assign in accordance with the codelist and SITESET value.	6	1
SITEGRP	If SITESETN in (1 2 3) then set to 1, else 2	7	2

These dependency levels determine the order in which variables must be derived. The final derivation order can be dependent on further sorting keys, however, like the variable order in the dataset, but these keys are only applied after the dependency level.

Even though looping references should be avoided in any specification, these will be handled as dependency level 999 in following examples.

Step 2: Creating a template blueprint

Next, we will create a blueprint for the program, which will predefine the structure of our program. A blueprint data frame will be derived in an intermediate step holding the following columns:

Column	Purpose
section_name	Identifier for a section
order	Section order for the program
content	Holds the respective sections programming code that will be pasted into the programming files
Source / Derivation	Original derivation. Represented as dictionary for variable block or empty for autocall. Will be provided in the prompts to the agent
variables	Variables to be derived in the respective section
output_dataset	final dataset to hold all variables to be derived within the section
referenced_vars_datasets	A dictionary to describe all the variables referenced from the same program and the datasets they are found in, used for prompting the AI agent

And blueprint will contain the following types of sections:

- One __HEADER__ section
- One __END__ section
- Zero or multiple autocall sections
- Zero or multiple single variable sections
- Zero or multiple variable block sections

Below is shown an arbitrary example of a program blueprint with a yet to fill [content] column, where variables that can be generated through metadata-driven automation are condensed into *autocall* sections for each dependency level. Most other variables received single variable sections. Only variables that have looping references were condensed into a variable block section in this example.

section_name	order	content	Source / Derivation	variables	output_datase	referenced_vars_datasets
__HEADER__	0					{}
autocall_0	1			STUDYID SUBJID SITEID SITESET RFICDT WICDT	autocall_0	{}
IN001FL	3		If IE.IESTRESC = "N" where	IN001FL	IN001FL	{}
IN002FL	4		If IE.IESTRESC = "N" where	IN002FL	IN002FL	{}
SOSDT	10		Set to numeric SV.SVSTD1	SOSDT	SOSDT	{}
FU1VSDT	12		Set to SV.SVSTDTC as nun	FU1VSDT	FU1VSDT	{}
FU2VSDT	13		Set to SV.SVSTDTC as nun	FU2VSDT	FU2VSDT	{}
EOSDT	14		Set to numeric max(SV.S\	EOSDT	EOSDT	{}
DTHDT	15		Set to numeric DD.DDDTC	DTHDT	DTHDT	{}
autocall_1	17			SITESETN IN001FN IN002FN IN003FN IN004FN	autocall_1	{}
SITESPEC	18		Set to ZS.ZSSTRESC where	SITESPEC	SITESPEC	{'autocall_0': ['SITEID']}
AGEGR1N	22		if . < AGE < 55 then set to :	AGEGR1N	AGEGR1N	{'autocall_0': ['AGE']}
autocall_2	32			AGEGR1 AGEGR2 AGEGR3 ASEX	autocall_2	{}
IEFN	33		Set to 1 if all of the inclus	IEFN	IEFN	{'autocall_1': ['IN001FN', 'IN002FN', 'IN003FN', 'IN004FN', 'IN005FN', 'IN006FN'], 'autocall_0': ['RFICDT']}
autocall_3	34			IEFL	autocall_3	{}
ENRLFL	35		Set to "Y" if part of USUBJI	ENRLFL	ENRLFL	{'autocall_3': ['IEFL'], 'autocall_1': ['COUNTRYN']}
FAS1FL	37		If BASFL = 'Y' and not mis:	FAS1FL	FAS1FL	{'BASFL': ['BASFL'], 'FU1VSDT': ['FU1VSDT']}
FAS2FL	38		If BASFL = 'Y' and not mis:	FAS2FL	FAS2FL	{'BASFL': ['BASFL'], 'FU2VSDT': ['FU2VSDT']}
loopref_999	42		{'EOSSTT': 'Part 1:\nIf DS.I	EOSSTT DCSREAS DCSREASN DCSREASP	loopref_999	{'autocall_0': ['COUNTRY', 'ACTARM'], 'FU2VSDT': ['FU2VSDT'], 'FU1VSDT': ['FU1VSDT']}
__END__	43					{}

Step 3: Creating an AI agent

Creating an agent AI coding agent in Python became straight forward with OpenAI's new agent SDK, taking only 3 arguments (given the API key is loaded via dotenv): a name, a general instruction for the agent, and the model to be used. Despite being developed by OpenAI, the SDK is not restricted to OpenAI LLM's but can interact with most LLM's on the market.

```python
from agents import Agent
from dotenv import load_dotenv

load_dotenv() # Load environment variables (API key) from .env file

agent = Agent(
    name="SAS_Programming_Agent",
    instructions=context,
    model="gpt-5"
)
```

The context given as instruction is simply a string that can be build dynamically dependent on some general information about the dataset. Below is a generic instruction for a programming agent with some dynamic variable resolution to abstract the programming language and actual programming task. As each dataset will be getting its own agent, this information will be retrieved from the specification for the respective dataset.

```python
context = (
    f"You are a {programming_language} programming agent for {programming_task}. We are programming {dataset_name}."
    f"You will be asked to generate {programming_language} code based on a specification."
    "Referenced variables in the derivation instruction are written in uppercase and unquoted."
    "Two upcase words connected by a dot indicate a variable and dataset reference, "
    "whereby the word before the dot represents the dataset name."
    "For variables that have no dataset prefix, a list of datasets where to find the variable is provided."
    f"Key-variables to link datasets together are: {keyvars} "
    "All datasets you need to load are already sorted by USUBJID."
    f"{dataset_structure} per {keyvars} is allowed for each dataset to be created in the 'dataset' column. "
    "Intermediate datasets can be created as needed and do not need to be deleted."
    "Treat all string instructions literally and avoid regex if possible required."
    "Only return code, no explanation.\n"
)
```

It is important to note that this instruction is not the final prompt for the agent, but a general instruction that will be considered by the agent for each prompt given to the agent later.

The agent described in this example represents a very basic implementation. The agent SDK supports both complex multi-agent interaction and straightforward conversation and knowledge retention for each agent. Additional information is available in OpenAI's documentation and highly recommended for interested readers.

Step 4: Prompting the Agent and finalizing the blueprint

Next, the previously generated program blueprint will be iterated and either fixed content will be written to the content column (for autocall, __HEADER__ and __END__ sections) or the agent will be prompted to give a code suggestion when requested.

To prompt the agent for a response, the Runner class is to be used in an asynchronous manner and needs as input the respective agent object to be prompted and an additional prompt, which will be the specific task for the agent, who will always consider the previously given instruction provided at his creation

```
response = asyncio.run(Runner.run(agent, prompt))
```

The specific section prompts are build utilizing the blueprints information about the variables to be created, the datasets to find referenced variables in and output dataset naming

```
prompt = (
    f"Please write me concise code for the variable/s '{row['variables']}'.\n"
    f"The resulting dataset should be called {row['output_dataset']}. "
    f"If intermediate datasets are needed, use the same name with a numeric suffix like _xx.\n"
)
if not pd.isna(row['Referenced Vars']):
    prompt += "Referenced variables can be found as follows:\n"
    # Group variables by dataset (section_name)
    ref_dict = row['referenced_vars_datasets']
    dataset_vars = {}
    for dataset, var in ref_dict.items():
        dataset_vars.setdefault(dataset, []).append(var)
    # Format as requested
    for dataset, vars_list in dataset_vars.items():
        prompt += f"The dataset {dataset} contains the variable/s {' '.join(vars_list[0])}.\n"
prompt += f"Derivation instruction:\n{row['Source / Derivation']}\n\n"
```

The prompts can (and should) be written to a log file for traceability.
As an example, the following blueprint entry:

section_name	order	content	Source / Derivation	variables	output_datase	referenced_vars_datasets
			if .<VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 <18.5 set to "< 18.5" else if 18.5<=VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 <25 set to "[18.5, 25)" else if 25<=VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 <30 set to "[25, 30)" else if 30<=VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 <35 set to "[30, 35)" else if 35<=VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 <40 set to "[35, 40)" else if 40<=VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 set to ">= 40"			
BMICATBL	29		Only populate if SOSDT is not missing.	SOSDT	BMICATBL	{'SOSDT': ['SOSDT']}

generates the following prompt for the agent:

```
Please write me concise code for the variable/s 'BMICATBL'.
The resulting dataset should be called BMICATBL. If intermediate datasets are needed, use the same name with a numeric suffix like _xx.
Referenced variables can be found as follows:
The dataset SOSDT contains the variable/s SOSDT.
Derivation instruction:
if .<VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 <18.5 set to "< 18.5"
else if 18.5<=VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 <25 set to "[18.5, 25)"
else if 25<=VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 <30 set to "[25, 30)"
else if 30<=VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 <35 set to "[30, 35)"
else if 35<=VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1 <40 set to "[35, 40)"
else if 40<=VS.VSSTRESN where VS.VSTESTCD eq BMI and VS.VISITNUM eq 1  set to ">= 40"

Only populate if SOSDT is not missing.
```

And returned the following SAS ® code:

```
proc sort data=VS(where=(VSTESTCD='BMI' and VISITNUM=1)) out=BMICATBL_01(keep=USUBJID VSSTRESN) nodupkey;
  by USUBJID;
run;

data BMICATBL;
  length BMICATBL $12;
  merge SOSDT(in=a) BMICATBL_01(in=b);
  by USUBJID;
  if a;
  if not missing(SOSDT) then do;
    if not missing(VSSTRESN) and VSSTRESN<18.5 then BMICATBL="< 18.5";
    else if 18.5<=VSSTRESN<25 then BMICATBL="[18.5, 25)";
    else if 25<=VSSTRESN<30 then BMICATBL="[25, 30)";
    else if 30<=VSSTRESN<35 then BMICATBL="[30, 35)";
    else if 35<=VSSTRESN<40 then BMICATBL="[35, 40)";
    else if 40<=VSSTRESN then BMICATBL=">= 40";
  end;
  keep USUBJID BMICATBL;
run;
```

This code is equivalent to the final response of the model which is fully functional and did not require any subsequent formatting. The final response needs to be extracted from the response object and is then saved to the [content] column using the following code

```
blueprint_df.at[idx, 'content'] = response.final_output
```

The response object itself and its additional content are beyond the scope of this paper. The full wrapping function and final blueprint, which is equivalent to the one shown before but including the filled [content] column are omitted for the sake of brevity.

Step 5: Writing the content to a programming file

The final step is rather straightforward. The programming file to be processed will be opened and each blueprint sections content will be written to it. The only conceptual addition is a start and end indicator for each section. These indicators are in theory arbitrary and can designed any way if they do not interfere with the program. An example of their design is:

```
f"{comment_symbol} --- {section_name} {start_or_end_of_section} ---{comment_symbol_end}"
```

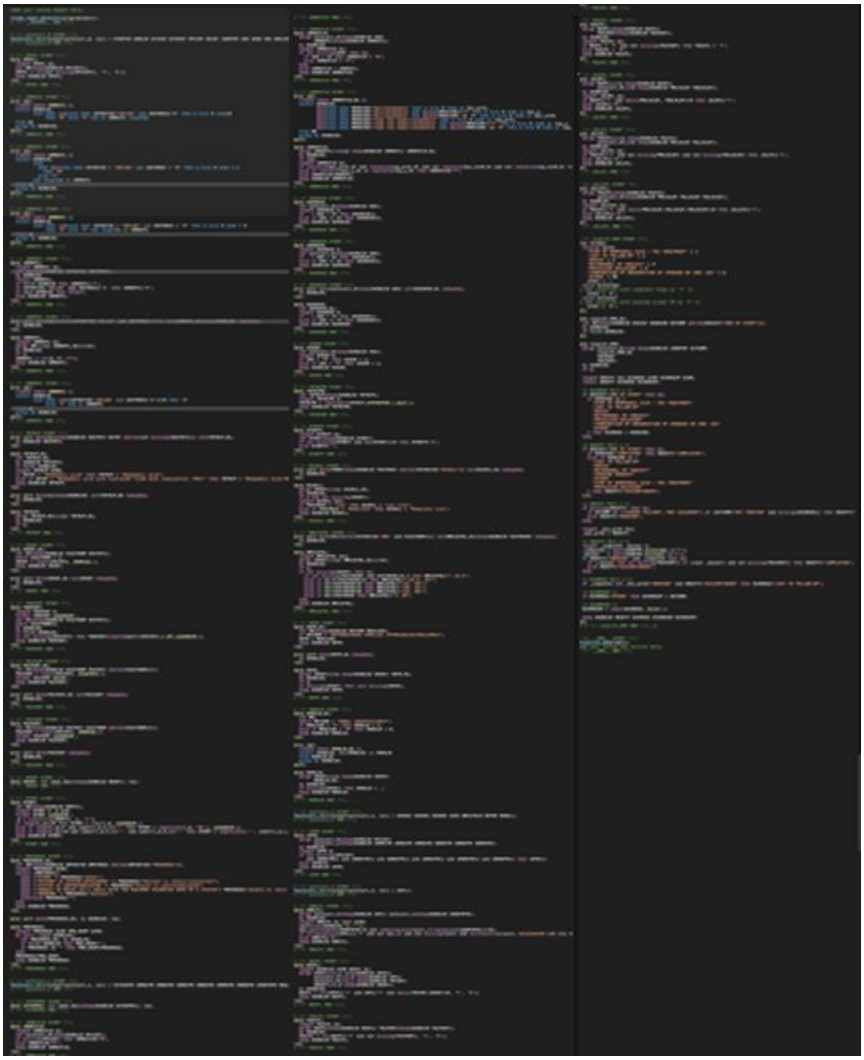
Whereby {comment_symbol:} variables only serve to facilitate a language agnostic approach to comment out the indicators.

The reason to use indicator is simply to facilitate a backwards compatibility (creating a blueprint from the program instead of creating a program from the blueprint) and clear visual barrier between automatically created program sections, but they are not required to write functional programs.
A conceptual example of the final programming file is shown below, including the complete and functional section for the BMICATBL variable from the previous example. Note that the header section has no start indicator, as the start of the file is deemed as the start of the __HEADER__ section.

```
*Add your custom header here;
%load_input_datasets(program = adsl);
* --- __HEADER__ END ---;

* --- autocall_0 START ---;
%autocall_vars(program = adsl, output=autocall_0, vars = STUDYID SUBJID SITEID ...);
* --- autocall_0 END ---;

*...
Placeholder for X additional section
...;

* --- BMICATBL START ---;
proc sort data=VS(where=(VSTESTCD='BMI' and VISITNUM=1)) out=BMICATBL_01(keep=USUBJID VSSTRESN) nodupkey;
  by USUBJID;
run;

data BMICATBL;
  length BMICATBL $12;
  merge SOSDT(in=a) BMICATBL_01(in=b);
  by USUBJID;
  if a;
  if not missing(SOSDT) then do;
    if not missing(VSSTRESN) and VSSTRESN<18.5 then BMICATBL="< 18.5";
    else if 18.5<=VSSTRESN<25 then BMICATBL="[18.5, 25)";
    else if 25<=VSSTRESN<30 then BMICATBL="[25, 30)";
    else if 30<=VSSTRESN<35 then BMICATBL="[30, 35)";
    else if 35<=VSSTRESN<40 then BMICATBL="[35, 40)";
    else if 40<=VSSTRESN then BMICATBL=">= 40";
  end;
  keep USUBJID BMICATBL;
run;
* --- BMICATBL END ---;

*...
Placeholder for X additional section
...;

* --- __END__ START ---;
%finalize_adam(program = adsl);
*Add your custom end section here;
* --- __END__ END ---;
```

How it compares to conventional programming

To assess the approach in a real-world setting, a legacy study specification for ADSL was utilized and slightly updated into a usable shape for this approach. This extra effort is expected to be obsolete for newly written specifications if directly adhering to the new standard. Using the described approach, a double programming task was performed where 39 non-automated variables had to be matched. An AI generated script was provided to two programmers as a starting point. To gain an abstract idea of the generated program, a zoomed-out view is shown below:



Some notable endpoints discovered in this short experiment are listed below.

	Conventional approach	AI driven approach
Time to match	Only estimated on spec complexity to be around 4-6 hours	Approximately 1-1.5 hours. Tested independently by two programmers
Code length	Approximately 400 (original program after removing extraneous content like headers)	668 for AI template. Minor change after matching the outputs.
Code structure	Related variables are derived together, but derivations are often dependent on one another	Dependency structure that often leads to separation of related variables. However, single variables are derived independently, avoiding accidental side-effects when implementing changes

Handling Standardisation

It is evident that the approach outlined in this paper necessitates a degree of standardisation. However, it is important to acknowledge that excessive standardisation can sometimes result in systems becoming overly rigid, which may ultimately reduce flexibility and increase the time required to complete tasks. It should never be forgotten that the goal of standardization is always to reduce effort.

Some key heuristics for effective standardisation should be considered for this approach

- **Standardize the interfaces, not the implementations**. → Define how components interact but let teams optimize internals. Examples: The specifications shape should not contaminate the codebase for the presented approach. Adapter patterns should separate the actual specification shape from its conceptual content.
- **Bias toward defaults, not mandates.** → Provide recommended defaults that cover 80% of cases, but always allow for opt-outs where possible.
- **Flexibility at the edges, stability at the core.** → Core principles rarely change; peripheral details can evolve. Example: The conceptual programming structure with dependency-based order is a core requirement for this approach, the sections start and end indicator, however, are a mere detail and should be flexible in design.

As a final note on standardization: if adherence and feedback are poor, the issue often lies with the standard itself. A well-designed standard should provide enough value that users choose to follow it voluntarily. When compliance must be enforced, it is usually a sign of a weak or poorly conceived standard.

Limitations and prospects

While the approach presented in this paper offers considerable promise for reducing programming effort, several limitations remain. Firstly, the current implementation is static: it does not support partial programming requests and lacks integrated system version control. This restricts the flexibility and traceability of the programming process.

Secondly, the creation and handling of complex derivations continue to challenge the AI. In certain cases, understanding and updating the AI-generated code for such derivations may prove more time-consuming than authoring the code manually, potentially negating the expected efficiency gains.

Thirdly, the agent currently operates with low contextual awareness. It does not receive clarifying content from SAP sections or similar sources, which could otherwise enhance its accuracy and decision-making. Furthermore, the AI is unable to evaluate the correctness of the underlying specification—if the specification itself contains errors, these are likely to be perpetuated in the output.

Finally, this approach is presently more effective for validation tasks, such as facilitating rapid sponsor double programming when outsourcing, than for production programming. How best to adapt and apply this methodology to both the production and validation stages remains an open question and warrants further exploration and refinement.

On the other hand, the following future enhancements hold great potential for improvement:

- Exploration and refinement of more effective prompts and context for the Agent. LLMs have proven to increase in accuracy of programming task by increased context (if task related). One example is the inclusion of relevant information from the SAP.
- A sophisticated Agent orchestration to handle more specific tasks with a more general approach. This is done by creating multiple agent AI interacting with one another, which it typically done for common AI tools like coding Copilots.
- Even more sophisticated LLMs. Though GPT-5 has proven to be a significant leap toward accuracy, future models promise even lower error rates for programming tasks
- Bidirectional and session-based interaction. The paper was written based on a static script approach. However, in a more sophisticated approach, it would be possible to create a session-based tool with an interactive AI that also helps to write specification and searches for inconsistencies.
- Though shown to be effective for dataset programming, the basic principle is transferable to TFL programming, holding even greater potential in time reduction there

Conclusion

The landscape of statistical programming, particularly in clinical data environments, presents unique automation opportunities thanks to the prevalence of comprehensive specifications. Large Language Models (LLMs) are poised to revolutionise this space, offering even greater automation capabilities than those currently seen in mainstream software development. As demonstrated, the adoption of straightforward scripting techniques and the strategic restructuring of programmes enable AI agents to efficiently generate entire ADaM datasets with minimal intervention.

The approach outlined in this paper holds significant promise for reducing programming effort, particularly on the validation side, where rapid double programming of datasets can be achieved. Nonetheless, there are limitations—namely, the reliance on existing specifications and the need for programmers to adjust their workflow towards specification writing to fully leverage these tools. This transition may feel unnatural to some, highlighting an area for further refinement.

Looking ahead, several enhancements offer exciting prospects. These include the exploration of more effective prompts and richer contextual inputs for agents, improved orchestration through the deployment of multiple interacting AI entities, and the continued evolution of LLMs towards greater accuracy and reliability. Furthermore, shifting from static script-based solutions to interactive, session-driven tools could enable bidirectional communication between AI and programmers, facilitating both specification creation and inconsistency detection.

In summary, while current methodologies provide a robust foundation for automated programming in statistical contexts, continuous innovation in agent design, prompt engineering, and LLM capabilities will be essential to maximise efficiency and user experience in the years to come.

References

[1] Sushchenko, Rie (Sanofi). Automating SDTM and ADaM Creation in Clinical Trials: https://www.listendata.com/2023/03/run-chatgpt-inside-sas.html

Acknowledgements

I would like to express my sincere gratitude to Daiichi-Sankyo for providing the opportunity to attend this conference and present my work. I also want to extend my thanks to Marcin Pracz, helping in review processes and providing valuable feedback on code structure and AI generated programs.

Recommended Reading

- Hunt, A., & Thomas, D.: The Pragmatic Programmer.
  Practical advice on coding habits and mindset, highlighting the DRY principle and highly accessible for practitioners.

Contact Information

Angelo Wibbeler

Daiichi-Sankyo Europe

Registered Office: Daiichi Sankyo Europe GmbH

Zielstattstr. 48

81379 Munich, Germany

Email: angelo.wibbeler@daiichisankyo.com

Web: www.daiichi-sankyo.eu