PharmaSUG 2024 - Paper SI-362

SASBuddy: Enhancing SAS Programming with Large Language Model Integration

Karma Tarap, Bristol Myers Squibb, Boudry, Switzerland
Derek Morgan, Bristol Myers Squibb, Lawrenceville, NJ
Pooja Ghangare, Ephicacy, Bangalore, India
Nicole Thorne, Bristol Myers Squibb, Lawrenceville, NJ
Tamara Martin, Bristol Myers Squibb, Lawrenceville, NJ

ABSTRACT

SASBuddy, dubbed "your friendly SAS programming assistant," is a pioneering SAS macro designed to facilitate SAS programming through Large Language Models (LLMs). This tool empowers users, particularly those with limited SAS coding expertise, to generate precise SAS code efficiently by interpreting natural language inputs. The core of SASBuddy's functionality lies in its ability to provide contextually accurate SAS code, tailored to specific datasets based on user queries.

INTRODUCTION

In a realm where the intricacy of human language meets the computational prowess of machines, we find ourselves on the brink of significant advancements. Large Language Models (LLMs) like ChatGPT are making waves, promising a transformative impact across various sectors including the pharmaceutical industry. These models offer a glimpse into a future where tasks like code generation and data analysis could become notably streamlined. This paper delves into the world of LLMs, their journey from conception to the current state, and the prospective applications.

We introduce the SASBuddy, a SAS macro envisaged to tap into the potential of LLMs, aiming to provide a helping hand to Clinical SAS Programmers through a macro interface. Specifically, it addresses a critical nexus of challenges in clinical programming: the generation of relevant code without disclosing patient data to LLMs, the integration of standard macro usage where applicable, and the implementation of automated feedback loops for continual code refinement.

Through this discourse, we seek to offer a realistic insight into how LLMs could not only enhance clinical programming but also significantly alter our interaction with this emerging technology.

BACKGROUND

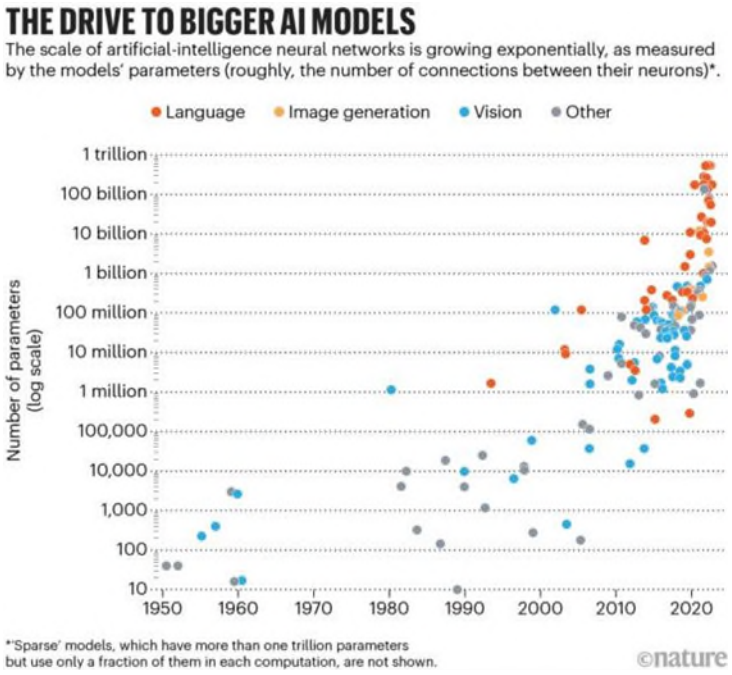
WHAT ARE LARGE LANGUAGE MODELS?

Language models, a subset of machine learning models, excel at understanding and generating human language. They're trained on vast amounts of text, leveraging statistical methods to predict subsequent words in a sequence based on the preceding ones. The journey from simple n-gram models and probabilistic context-free grammars to today's sophisticated architectures underscores the remarkable strides made in this field. The landmark paper "Attention Is All You Need" (Vaswani et al., 2017), marked a pivotal point by introducing the Transformer architecture, a significant shift from traditional RNNs and LSTMs. This novel approach to handling sequences through self-attention mechanisms spurred the development of advanced language models like BERT, GPT-2, and eventually GPT-3. The innovations brought about by the Transformer architecture, such as eliminating recurrent layers, enabled parallel processing and effective management of long-term dependencies, paving the way for the creation of models with billions of parameters – the cornerstone of contemporary LLMs.

RISE OF THE PARAMETERS

The advent of the Transformer architecture ignited the race for developing increasingly expansive language models. Parameter count became a crucial metric, with new models regularly showcasing billions of parameters. An augmented parameter count not only enhances a model's contextual understanding but also its capability to generate more nuanced and accurate responses (Chernyavskiy et al., 2021). However, this surge in parameters requires significant computational resources and extends training durations, presenting challenges in model training and deployment (A*nthaswamy, 2023*). Figure 1 shows this expansion:

Figure 1: The Drive to Bigger AI Models (Source: Ananthaswamy, 2023.)

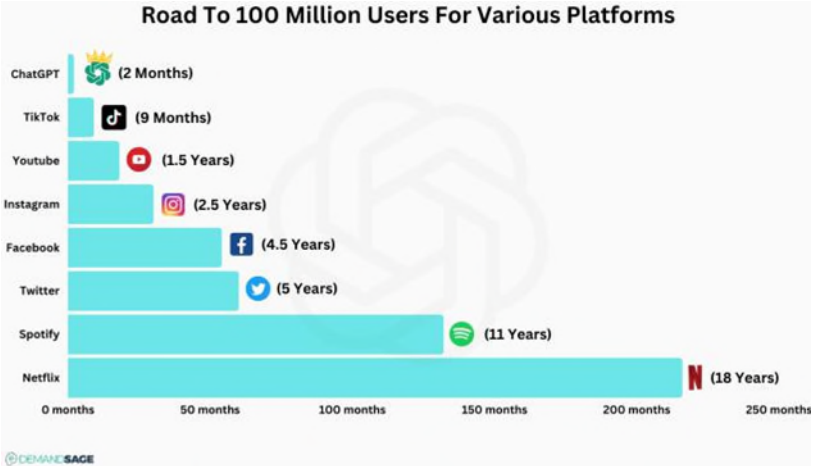


PUBLIC'S IMAGINATION: THE CHATGPT PHENOMENON

The graph in Figure 2 showcases the swift user adoption across various platforms; notably, ChatGPT amassed 100 million users within two months post-release). But what fueled this swift adoption?

Among LLMs, ChatGPT uniquely resonated with the public owing to its conversational proficiency, diverse knowledge base, and human-like reasoning. Although built on GPT-3's architecture, ChatGPT underwent a distinct fine-tuning process incorporating conversational data and user feedback. This resulted in a versatile model capable of managing a myriad of tasks and dialogues, thereby appealing to a broader user base.

Figure 2: Road to 100 Million Users for Various Platforms (Source: Demand Sage, n.d.)



CHALLENGES OF LLM'S IN CLINICAL SAS PROGRAMMING

Despite the potential benefits of LLMs, a significant hurdle remains in the proprietary nature of SAS, which results in less representation within LLM training datasets compared to open-source languages such as Python or Java. This discrepancy impacts the LLMs' capability to directly generate SAS code that is both relevant and precise. Moreover, the sensitive and voluminous nature of clinical data limits its availability for both LLM training and context providing. Additionally, accurately generating SAS code is challenging without detailed knowledge of the specific dataset structures or the standard macros available for use. These factors collectively underline the need for an innovative method to effectively apply LLMs within this niche.

SASBuddy attempts to address these challenges using the following approaches:

DATA-RELEVANT CODE GENERATION

SASBuddy's ability to generate data-relevant SAS code is intricately linked to its initial step of incorporating the metadata of datasets into the query process. This approach ensures that the generated code is not only syntactically correct but also contextually aligned with the specific data structures it is meant to analyze or manipulate.

By passing this rich metadata into the query, SASBuddy arms the LLM with the necessary context to understand where and how to look for specific data points within the datasets, such as identifying the appropriate variables or columns for a given analysis task.

STANDARD MACRO UTILIZATION

The utilization of standard macros within SASBuddy is facilitated by injecting a JSON function schema of the macros into the process that is parsed from the Doxygen macro headers.

This schema provides the LLM with a detailed blueprint of the available macros, including their functions, parameters, and usage conventions. With this information, SASBuddy can more reliably generate calls to these macros, ensuring that the produced code adheres to established coding practices and leverages the efficiencies that these macros are designed to provide.

AUTOMATED FEEDBACK LOOPS

SASBuddy's feedback loop mechanism is a critical component of its iterative improvement process. After generating and executing SAS code, the tool captures logs that include any errors encountered during execution. These logs are then sent back to the LLM, which analyzes the errors and adjusts the generated code in an attempt to fix the issues identified. This process not only enhances the accuracy of

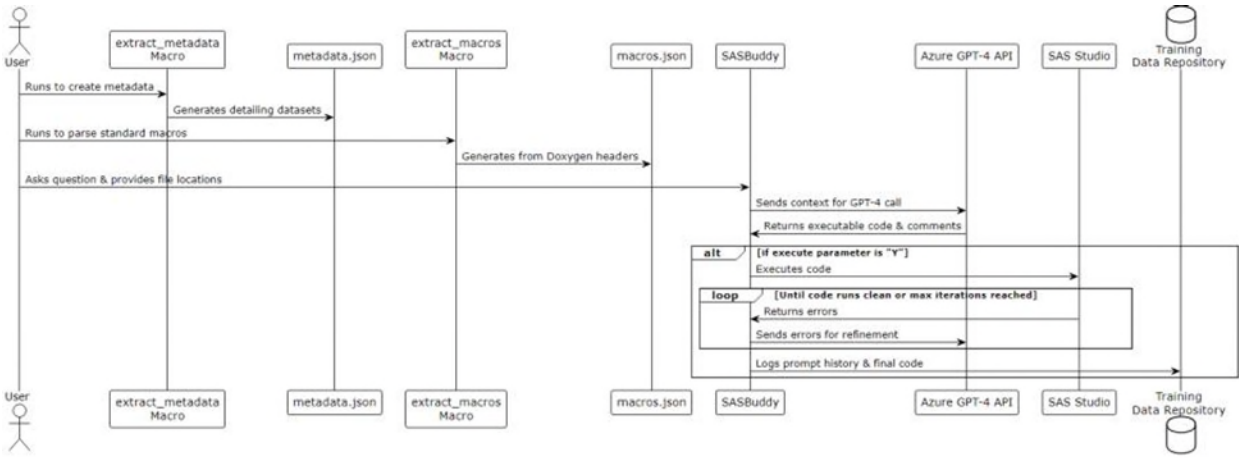
the code over time but also contributes to the LLM's learning, making it more adept at generating error-free SAS code in future queries.

VERSIONING AND FINE-TUNING

The versioning feature of SASBuddy involves recording prompts and the corresponding responses that were executed successfully. This historical record is logged into a file, creating a repository of effective queries and SAS code snippets. This repository serves as a valuable resource for fine-tuning future responses, allowing SASBuddy to draw on past successes to improve the relevance and accuracy of the code it generates. This versioning system thus plays a pivotal role in the tool's continuous learning and adaptation process.

SASBUDDY: OPERATIONALIZING THE SOLUTION

Figure 3: The SASBuddy Algorithm in PlantUML



SETUP PHASE:

The process begins when a user runs the extract_metadata macro. This macro scans the datasets of interest and creates a file named metadata.json, which details the metadata of these datasets. This metadata includes information about the structure and attributes of the data, but it abstracts away any confidential or patient-identifiable information to maintain privacy. Figure 4 provides an excerpt:

Figure 4: Excerpt of metadata.json, used to pass in data context for code generation

```
"context": [
  "Treatment group information and population flags (sometimes called sets) are on DM. Variables
],
"datasets": {
  "DM": {
    "description": [
      ""
    ],
    "keys": [
      "STUDYID, USUBJID"
    ],
    "variables": [
      {
        "name": "ACTARM",
        "label": "Description of Actual Arm",
        "type": "character",
        "unique_values": ["TRTA", "TRTB"]
      },
..
```

We are passing SDTM or ADaM specific rules as context here, as well as SDTM/ADaM metadata that are taken from our company metadata specifications. Variable labels help ensure the LLM understands the natural language to variable label mappings. The “keys” that determine what variables make the dataset unique are also passed as this is used to create join statements. Finally, the “unique_values” (codelists) fields, allow the LLM to provide information that could not be inferred from database schema alone. Eg. for BDS domains knowing what subset of data is necessary for a particular parameter of interest.

Concurrently, the user runs another macro named extract_macros. This macro analyzes the Doxygen headers of standard SAS macros used within the organization and compiles this information into a macros.json file. This file serves as a catalog of available macros, detailing their usage, parameters, and functionality.

An example of this is provided in Appendix A: Incorporating Standard Macros into SASBuddy.

QUESTIONING PHASE

With these setup steps completed, the user then poses a question to SASBuddy, specifically mentioning the location of the metadata.json and macros.json files. This question is framed in natural language, akin to asking a colleague for help with a SAS programming task.

SASBuddy, equipped with the question and the contextual information provided by the metadata and macros files, formulates a query to the Azure GPT-4 API. This query is crafted using reflexion and few-shot learning techniques, designed to prompt the API to generate executable SAS code. The prompt includes explanations and comments, ensuring that any generated code is accompanied by understandable guidance on its use and intention.

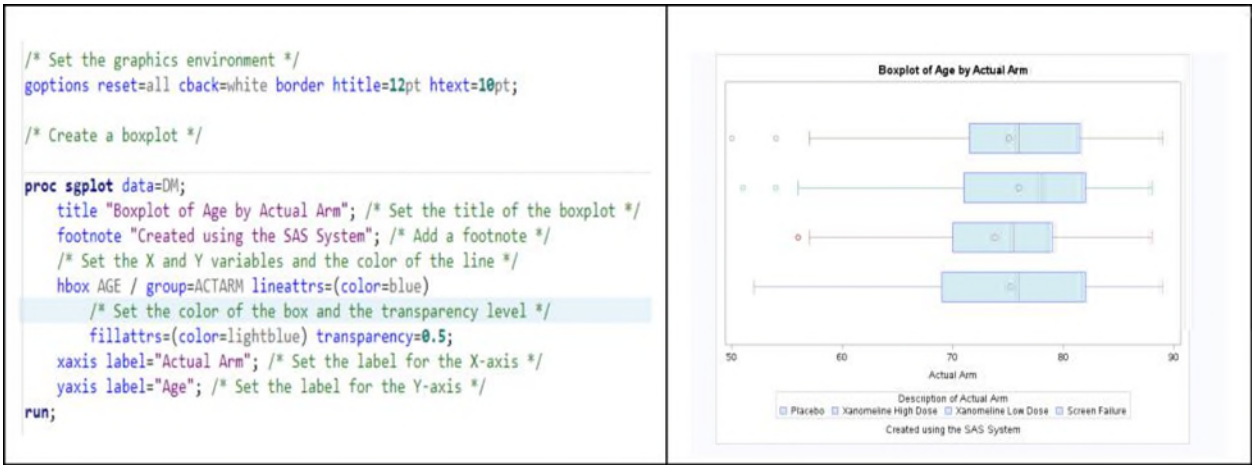
EXECUTION AND FEEDBACK LOOP

If the user has set the execute parameter to "Y", SASBuddy proceeds to execute the generated SAS code in an environment like SAS Studio. Should errors occur during this execution, SASBuddy captures these error messages and sends them back to the Azure API for further refinement of the code.

This loop of execution, error capture, and refinement continues until the code runs without errors or until a predefined maximum number of iterations is reached, ensuring an iterative approach to achieving clean, error-free code.

Once the code executes successfully without errors, and if the feedback option was selected, SASBuddy logs the successful prompt and response sequence into a Training Data Repository. This log serves as invaluable data for fine-tuning the system in the future, enhancing SASBuddy’s ability to generate accurate and useful SAS code. Results are in Figure 5:

Figure 5: The generated code and the executed output for the query “create boxplot of age by actual arm. Add a relevant title and footnote and aesthetically pleasing colors” to SASBuddy



```
/* Set the graphics environment */
goptions reset=all cback=white border htitle=12pt htext=10pt;

/* Create a boxplot */

proc sgplot data=DM;
    title "Boxplot of Age by Actual Arm"; /* Set the title of the boxplot */
    footnote "Created using the SAS System"; /* Add a footnote */
    /* Set the X and Y variables and the color of the line */
    hbox AGE / group=ACTARM lineattrs=(color=blue)
        /* Set the color of the box and the transparency level */
        fillattrs=(color=lightblue) transparency=0.5;
    xaxis label="Actual Arm"; /* Set the label for the X-axis */
    yaxis label="Age"; /* Set the label for the Y-axis */
run;
```

FUTURE WORK

As SASBuddy evolves, its development trajectory is guided by the need to enhance accuracy, introduce agentic workflows, and leverage a fine-tuned LLM model:

Accuracy Profiling: Future efforts will prioritize evaluating SASBuddy's code generation accuracy, focusing on how well it addresses clinical SAS programming needs across varied scenarios. Profiling activities will assess code correctness, efficiency, and adherence to programming standards, comparing SASBuddy's outputs against established benchmarks and real-world task outcomes.

Agentic Workflows Exploration: Investigating the potential for SASBuddy to adopt more autonomous operational capabilities, future development will explore enhancing the tool with features that allow for dynamic query refinement, strategic analysis suggestions, and proactive error detection. This direction aims to transition SASBuddy from a command-executing tool to a proactive coding partner.

Fine-Tuned Model Testing and Training Data Collection: Essential to refining SASBuddy's responsiveness to SAS programming nuances and clinical data analysis requirements is the testing on a model fine-tuned with a comprehensive dataset reflecting successful user interactions and code generation. Preparing for this involves the systematic collection of high-quality, real-world usage data to inform the fine-tuning process, ensuring SASBuddy's increased adeptness in clinical SAS programming contexts.

CONCLUSION

The journey of integrating LLMs into clinical SAS programming, as exemplified by SASBuddy, is only beginning. By addressing the method's current accuracy, exploring the potential of agentic workflows, and preparing for fine-tuning with a comprehensive training dataset, future work will seek to unlock the full potential of this innovative approach. These efforts will ensure that SASBuddy not only keeps pace with the evolving landscape of data analysis but also sets new standards for productivity and efficiency in clinical research programming.

REFERENCES

OpenAI, Chatgpt https://chat.openai.com, (2023), Accessed: 2023-07-30.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention Is All You Need. arXiv preprint arXiv:1706.03762. Available at: https://doi.org/10.48550/arXiv.1706.03762

Chernyavskiy, A., Ilvovsky, D. and Nakov, P., 2021. Transformers: "The End of History" for NLP? arXiv preprint arXiv:2105.00813. Available at: https://arxiv.org/abs/2105.00813.

Ananthaswamy, A. (2023). In AI, is bigger always better? Nature, 615, 202-205. https://doi.org/10.1038/d41586-023-00641-w

Demand Sage. (n.d.). Road to 100 Million Users For Various Platforms. [online] Available at: https://www.demandsage.com/chatgpt-statistics/ [Accessed on: 10 October 2023].

X. Wei, X. Cui, N. Cheng, X. Wang, X. Zhang, S. Huang, P. Xie, J. Xu, Y. Chen, M. Zhang et al., Zero-shot information extraction via chatting with chatgpt, arXiv preprint arXiv:2302.10205 (2023).

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. and Amodei, D., 2020. Language Models are Few-Shot Learners. arXiv. Available at: https://arxiv.org/abs/2005.14165

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K. and Yao, S., 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv preprint arXiv:2303.11366. Available at: https://arxiv.org/abs/2303.11366.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q. and Zhou, D., 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv preprint arXiv:2201.11903. Available at: https://arxiv.org/abs/2201.11903.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T.L., Cao, Y. and Narasimhan, K., 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv preprint arXiv:2305.10601. Available at: https://arxiv.org/abs/2305.10601.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P.S., Lachaux, M-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E.M., Subramanian, R., Tan, X.E., Tang, B., Taylor, R., Williams, A., Kuan, J.X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., Scialom, T., 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv preprint arXiv:2307.09288. Available at: https://arxiv.org/abs/2307.09288.

H. Face, Transformers language modeling, Available at: https://github.com/huggingface/transformers/tree/main/examples/pytorch/language-modeling, (2023).

L. Team, Llama.cpp https://github.com/ggerganov/llama.cpp, (2023).

Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y., and Wen, J.-R. (2023). A Survey of Large Language Models. arXiv. Available at: https://arxiv.org/abs/2303.18223.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Karma Tarap
Bristol Myers Squibb
Boudry / 2017
Email: karma.tarap@bms.com

APPENDIX A: INCORPORATING STANDARD MACROS INTO SASBUDDY

To demonstrate SASBuddy's of standard macros within its code generation process, we present a simplified example that outlines how these macros are selected and implemented based on the user's query and the contextual metadata provided.

Context: Let's consider a scenario where a user wants to calculate the mean age, but wants to use standard macros instead of raw SAS code. Let's also assume we have the following standard macros in our library.

```
/**
 * @brief Calculates the mean of a numeric variable in a SAS dataset.
 * @param ds_name Name of the SAS dataset.
 * @param var_name Name of the variable to calculate the mean for.
 */
%macro calc_mean(ds_name=, var_name=);
    proc means data=&ds_name noprint;
        var &var_name;
        output out=mean_result mean(&var_name)=mean_var;
    run;
%mend calc_mean;

/**
 * @brief Filters a SAS dataset based on a condition and creates a new dataset.
 * @param input_ds Name of the input SAS dataset.
 * @param output_ds Name of the output SAS dataset.
 * @param condition The condition to filter the dataset on (e.g., age > 18).
 */
%macro filter_dataset(input_ds=, output_ds=, condition=);
    data &output_ds;
        set &input_ds;
        if &condition;
    run;
%mend filter_dataset;
```

Through running *extract_macro* we get the following JSON in our *macros.json* file which describes the macro arguments available in our macro library.

```
{ "role": "assistant", "function_call": { "name": "calc_mean", "arguments": "{ds_name,var_name}" } }
{ "role": "assistant", "function_call": { "name": "filter_dataset", "arguments": "{input_ds,output_ds,condition}" } }
```

Now, when the user submits a query to SASBuddy and provides a macro location, SASBuddy has enough context to first understand which variables and domains to use as this is already provided in the metadata.json file.

```
%SASBuddy(question= calculate the mean age,
    output_file = &srcdir/gpt.sas,
    metadata_path=&srcdir/metadata.json,
    macros_path = &srcdir/macros.json);
```

It also has sufficient context to identify if a standard macro(s) exist that can answer the user's question and what the necessary parameters should be. Therefore, SASBuddy will generate the following code:

```
/* Call the calc_mean macro to compute the mean of the variable AGE in dataset DM */
%calc_mean(ds_name=DM, var_name=AGE);
```