PharmaSUG 2025 - Paper AI-286

Get Started with ChatGPT and DeepSeek in SAS

James Austrow, Cleveland Clinic

ABSTRACT

If you've been feeling like a wallflower at the LLM party, this paper is for you. Whether you're intimidated by the jargon, a SAS programmer marooned in a sea of Python tutorials, or simply don't know where to begin, start here.

Follow along with this walkthrough to build, from scratch, the foundations of an AI-powered SAS application. You'll get properly acquainted with the language model lingo: prompt engineering, zero-shot, few-shot, chain-of-thought, and prompt chaining; all receive practical treatment. Don't just have them explained, see how you can apply them to solve a real data processing problem.

The AI landscape is constantly evolving. Two of the most well-known language model services, ChatGPT and DeepSeek, receive specific focus here, but the concepts are generalizable and apply to all the major providers.

Advanced SAS users stand to get the most out of this paper, but anyone, even non-programmers, with an interest in prompt engineering can benefit as well.

INTRODUCTION

It's now 2025, and large language models like ChatGPT are no longer the new kid on the block. Though they still seem magical in many ways, their use cases and limitations are much better understood than just a few years ago. The market has also matured significantly, with no shortage of competing platforms surfacing since OpenAI publicly debuted the core technology in late 2022. That's not even to mention the homebrew possibilities enabled by self-hosting your own model, provided you have a beefy enough GPU.

For a software developer or programmer looking to build applications on top of language models, there has never been a better time to get started. Prompt engineering is becoming a distinct skillset with its own principles, concepts, and complexities. There are generalizable techniques and a plethora of resources to learn from.

Despite these advances in the ecosystem, relatively little guidance oriented towards the SAS user community seems to exist. This paper aims to address that gap. Consider it your starter kit for working with language model APIs in SAS.

There is first the technical matter of actually getting a prompt round-trip to and from a model. You'll work out the basic mechanics in the "Fundamentals" section. Next, "Building Your Own API Client" will walk through how to abstract those details behind a friendly user (programmer) interface. It will be helpful to have an advanced understanding of SAS for these sections.

In the second half, you'll get a crash course in prompt engineering as you try to get the language model to perform date imputation for you. This will provide a survey of the basic techniques and vocabulary to help you later navigate advanced prompt engineering resources on your own. No particular prerequisites are assumed for this section, and anyone (even non-programmers) should be able to benefit from it.

A WORD ON DATA PRIVACY

It would be remiss to fail to remind you of the inherent danger of interacting with language model platforms. You can be quite certain that any material you submit as part of a prompt will be used to help train an AI product or be otherwise mined for information. As such, remember to **only send text that you wouldn't care if it became public**. This is doubly important when using a platform controlled by a foreign country.

That said, language models are very powerful and quite rewarding to experiment with. They are absolutely worth exploring provided you keep the above warning in mind.

FUNDAMENTALS

OBTAINING AN API KEY

Before anything else, you must obtain an API key. You can do so at these locations after logging in to the platform:

- For ChatGPT: https://platform.openai.com/api-keys
- For DeepSeek: https://platform.deekseek.com/api_keys

Regardless of which platform you use, you will also need to fund your account. Neither service has a free tier for its API but fortunately, both have extremely cheap models available. As little as a single U.S dollar will last you a long time.

Keep in mind that the API key is the only information used by the platform to authenticate your requests. If someone else gets ahold of it, they can impersonate you. Hence, it's important to avoid accidentally leaking your key. The most common way this happens is by writing the key directly into your program. A better practice is to read the key from an external file instead.

Create a file with your API key as its only contents, then read the key into SAS like this:

```
data _null_;
    infile "sources/api_key_chatgpt.txt" length=len;
    input contents $varying2000. len;
    call symputx("api_key", contents);
run;
```

Program 1: Loading an API key from an external file.

A macro variable is a convenient place to store the key, but watch out! If you have the MPRINT option enabled, your key will be printed to the program log in plaintext. You may prefer to use a different technique if this will be a problem for you.

The other common way that keys are leaked is by committing the file that contains them to version control. This is potentially much worse if your repository is somewhere on the public internet, such as Github. If you ever leak your key, you can always just delete it using the OpenAI or DeepSeek dashboard, so staying vigilant will usually prevent any serious consequences.

SENDING YOUR FIRST PROMPT

It's time to introduce yourself! This example uses the OpenAI API, but the DeepSeek API is fully compatible and shares the same request format (DeepSeek 2025).

You can construct and send a request manually in SAS using PROC HTTP (Henry 2019). The most complex piece of the request is the JSON body. Create that separately by using the FILE statement:

```
filename req_json TEMP; /* Name has to be 8 characters or less. */
data _null_;
    file req_json recfm=f lrecl=1;

    put '{';
    put     '"model": "gpt-4o-mini",';
    put     '"messages": [';
    put         '{';
    put             '"role": "user",';
    put             '"content": [';
    put                 '{ "type": "text", "text": "Hi ChatGPT! This is just a test." }';
    put             ']';
    put         '}';
    put     ']';
    put '}';
run;
```

Program 2: Crafting a hard-coded request payload.

First, the "model" parameter specifies which model the prompt will be sent to. There are many different models and they differ primarily in price, capabilities, quality of response, and speed. The "4o-mini" model from ChatGPT is very cheap and quite good for most purposes. As of writing, you can see the current list of OpenAI models at https://platform.openai.com/docs/models

The other parameter here, "messages", is where you'll perform the bulk of prompt customization. This prompt is very simple; it's the equivalent of going to the ChatGPT website and typing a single line. Later on, you'll construct some more interesting prompts.

With the request prepared, you can now send it over the web:

```
filename response TEMP;

/* Make the request... */
proc http
    url = "https://api.openai.com/v1/chat/completions"
    method = "POST"
    ct = "application/json"
    in = req_json
    out = response;

    headers "Authorization" = "Bearer &api_key.";
run;

/* ... and then view the response! */
data _null_;
    infile response length=len;
    input line $varying2000. len;
    put line;
run;
```

```
"message": {
"role": "assistant",
"content": "Hello World! 😊 How can I assist you today?",
"refusal": null,
"annotations": []
},
```

Figure 1: Sending a request to ChatGPT and an excerpt from the full response object.

Much of the request is boilerplate, but make note of the following parameters:

- "in" is where to put the request body you constructed
- "out" specifies the file that the response will be written to
- "headers" contains the API key

The full response JSON object contains many other interesting items. Feel free to explore them by consulting your chosen platform's documentation.

BUILDING YOUR OWN API CLIENT

You'll be repeating the above steps many times. Now that you've tried out the basic mechanics, it's worth investing the effort to develop them into reusable program structures. The goal is to not have to think about the plumbing required to send your prompts over the web. To do that, you'll build a simple client compatible with both the ChatGPT and DeepSeek APIs.

First, create some small helper macros. They don't do a lot individually, but giving these processes names will help avoid distractions later.

```
/* Create a file handle and ensure it is fresh. */
%macro FileHandle(handle, location=TEMP);
    filename &handle. clear; /* Will give warning on first invocation. */
```

```
    filename &handle. &location.;
%mend;

/* Read the contents of a file into a macro variable. */
%macro FileContents(variable_name, file_location);
    data _null_;
        infile &file_location. length=len;
        input contents $varying2000. len;
        call symputx(&variable_name., contents);
    run;
%mend;
```

Program 3: Small helper macros for file input and output.

MAKING AND SENDING REQUESTS

The first major task to tackle is prompt construction. This is a good time to discuss some of the basic prompt elements in more detail.

Most prompts consist of a series of messages. Each message consists of some text and a "role". The role defines the level of priority the model should give to the message. Some common examples of roles:

- "system" - A high-priority background message from the system developer (that's you!)
- "user" - A message from the end user of the application
- "assistant" - A message from model itself

You can get creative with how you use the roles. For instance, "assistant" is nominally used to store response history when designing a chatbot, but it can be repurposed for few-shot learning. You will see more on that later.

Right now, the concern is making it easy to construct prompts from within a SAS program. One simple design is to create a sequence of roles and texts using a data step:

```
/* Hardcode the prompt as a dataset. */
data my_1st_prompt;
    length role $10 text $2000;
    input role $ text &:$2000.;
    datalines;
system You are ChatGPT, a helpful and friendly AI.
user Hi ChatGPT!
assistant Hello World! What can I do for you?
user Can you summarize the significance of the "Hello world" example in two sentences?
;
run;
proc print data=my_1st_prompt; run;
```

The SAS System

Obs	role	text
1	system	You are ChatGPT, a helpful and friendly AI.
2	user	Hi ChatGPT!
3	assistant	Hello World! What can I do for you?
4	user	Can you summarize the significance of the "Hello world" example in two sentences?

Figure 2: Demonstrating simple prompt construction using a data step.

You'll then need a method for turning the prompt data set into a valid JSON payload. This macro is a bit clunky but it gets the job done:

```
%macro build_request(
    /* Required parameters: */
    data,
```

```
    request_handle,

    /* Optional parameters: */
    model="gpt-4o-mini",
    request_location=TEMP
);
    %FileHandle(&request_handle., location=&request_location.);
    data _null_;
        file &request_handle.;

        /* Write the opening of the JSON payload. */
        put '{';
        put     '"model": "' &model. '",';
        put     '"messages": [';

        /* Loop through the dataset and construct each message block. */
        do i = 1 by 1 until (eof);
            set &data end=eof;

            /* Escape quotation marks. */
            length esc_text $4000;
            esc_text = tranwrd(text, '"', '\"');

            /* Begin the message object */
            put '{';
            put     '"role": "' role +(-1) '",';
            put     '"content": [';
            put         '{ "type": "text", "text": "' esc_text +(-1) '" }';
            put     ']';

            /* End the message object */
            if eof then put '}';
            else put '},';
        end;

        /* Close the messages array and the JSON object */
        put     ']';
        put '}';
        stop;
    run;
%mend;
```

Program 4: Encapsulating the process of building a request object.

Despite the length of this macro, there's nothing too remarkable about it. Do note that the model name is now an optional parameter, which you can use to try out the various models easily and compare their responses.

Having a request object in hand is not very useful without the ability to send it over the web. Take care of that now:

```
%macro send_request(
    /* Required parameters */
    request_handle,
    response_handle,
    api_key_location,

    /* Optional parameters */
    service=openai, /* Can pass "deepseek" instead with no other changes. */
    response_location=TEMP
);
    %local api_key;
    %FileContents("api_key", &api_key_location.);
    %FileHandle(&response_handle., location=&response_location.);

    /* Make the request. This may take a few seconds to complete. */
```

```
    proc http
        url = "https://api.&service..com/v1/chat/completions"
        method = "POST"
        ct = "application/json"
        in = &request_handle.
        out = &response_handle.;

        headers "Authorization" = "Bearer &api_key.";
    run;

    /* Results are now stored in response_handle! */

%mend;
```

Program 5: Encapsulating sending the request.

This code should look very familiar. Indeed, it requires virtually no changes from the opening example. The API key now gets freshly loaded each time.

Time to put these macros to work! Using the example prompt you constructed earlier, see that the process of building, sending, and receiving data has been made vastly simpler:

```
/* Send our prompt to ChatGPT over the network. */
%build_request(my_1st_prompt, req_json);
%send_request(req_json, response, "sources/api_key_chatgpt.txt");

/* Take a peek at the results using the program log.
 * You could also use a file location on disk instead of TEMP.
 */
data _null_;
   rc = jsonpp('response', 'log');
run;
```

```
      "message": {
        "role": "assistant",
        "content": "The \"Hello, World!\" example serves as a classic introductory program in computer programming, demonstrating
the basic syntax of a programming language and the process of compiling or running code. Its simplicity makes it an ideal first
step for beginners to understand the fundamental concepts of programming and establish a foundation for more complex projects.",
        "refusal": null,
        "annotations": [

        ]
      },
```

Figure 3: Demonstrating the API macro suite, with excerpted response.

Due to their APIs being fully compatible, it's quite simple to call DeepSeek instead of ChatGPT. You need only change the model name, service name, and API key. Here's what that looks like:

```
%build_request(my_1st_prompt, req_json, model="deepseek-chat");
%send_request(req_json, response, "sources/api_key_deepseek.txt", service=deepseek);
```

Program 6: Calling DeepSeek instead of ChatGPT.

REFINING THE INTERFACE

You've now got the web plumbing taken care of, but the ergonomics still have room for improvement. First, the prompt construction data step requires enough boilerplate to be annoying. Wrap that up so that prompts can be specified with a macro variable:

```
%macro Prompt(name, contents);
    data &name.;
        length role $20 text $2000;
        retain i 1;
        do while(scan("&contents.", i, '~') ne '');
            line = scan("&contents.", i, '~');
            role = scan(line, 1, '|');
            text = scan(line, 2, '|');
            output;
            i + 1;
        end;
```

```
        drop i line;
    run;
%mend;
```

Program 7: A small utility macro for constructing prompts.

Now you can write a prompt this way instead!

```
%Prompt(my_prompt, %quote(
    system|
    You are a master Japanese poet specializing in haikus.
    Your style is to incorporate mathematical imagery.~
    user|Please write a poem about water.
));
proc print data=my_prompt; run;
```

The SAS System

Obs	role	text
1	system	You are a master Japanese poet specializing in haikus. Your style is to incorporate mathematical imagery.
2	user	Please write a poem about water.

Figure 4: Constructing a prompt from a macro variable.

You're almost there! One last piece of missing functionality is actually extracting the messages from the response object. This code is adapted directly from (Mc Cawille 2024), which is also an excellent resource. For even more detail on dealing with JSON objects in SAS, you can also consult (Linker 2019).

```
/* Thanks to Stephen Mc Cawille ("Getting a Prompt Response Using ChatGPT and SAS") for
this code. */
%macro extract_response(dataset_name, response_handle, temp_library_handle);
    libname &temp_library_handle. JSON fileref = &response_handle.;

    data &dataset_name.;
        set &temp_library_handle..choices_message;
        output;
        /*
        do row = 1 to max(1, countw(content, '0A'x));
            outvar = scan(content, row, '0A'x);
            output;
        end;
        */
    run;
%mend;
```

Program 8: Pulling the message contents from an API response into a data set.

COMPLETING THE CLIENT

Now, you're in position to bring everything together into a single macro:

```
/* Fully process a raw prompt into a response message. */
%macro send_prompt(
    /* Required Parameters: */
    message_handle,
    prompt,

    /* Optional Parameters: */
    model="gpt-4o-mini",
    api_key_location="sources/api_key_chatgpt.txt",
    service=openai
);
    %Prompt(temp_user_prompts, &prompt.);
    %build_request(temp_user_prompts, tempreq, model=&model.);
    %send_request(tempreq, tempresp, &api_key_location., service=&service.);
```

```
    %extract_response(&message_handle., tempresp, templib);
%mend;
```

Program 9: Implementing an end-to-end API client as a macro.

As a user, all it takes is to specify your prompt:

```
/* Demonstrate simple usage. */
%send_prompt(haiku, %quote(
    system|
    You are a master Japanese poet specializing in haikus, but you do not use
    nature references.
    Instead, your style is to incorporate mathematical imagery.~
    user|Please write a poem about housing policy.
));
proc print data=haiku; run;
```

The SAS System

Obs	ordinal_choices	ordinal_message	role	content	refusal
1	1	1	assistant	Blueprints of promise, Equations of equity, Hope housed in numbers.	.

Figure 5: Demonstrating use of the end-to-end client.

You are now in possession of a fully-functional, if basic, ChatGPT/DeepSeek API client built from scratch in SAS! You'll get a chance to put it through its paces in the next section.

PROMPT ENGINEERING

Time for the fun stuff! In this second half of the paper, you'll learn how to apply a variety of prompt engineering techniques in the context of a real problem.

Despite the fact that language models are inherently nondeterministic black boxes, prompt engineering still has many aspects in common with programming. It's both possible and effective to test, iterate, debug, and sometimes think outside the box as you develop a prompt-based solution.

USE CASE: DATE IMPUTATION

Dates and times seem to always cause for trouble, especially so when it's necessary to impute parts of them. Imputation rules can grow complicated very quickly as corner cases and ambiguities start cropping up. Not to mention that incoming dates and times can vary almost arbitrarily in format.

The goal of this exercise is to get a working date imputation prompt suitable for integrating into a data pipeline. That means that not only should the output be accurate and fairly reliable, but also in a standard format conducive for reporting or further processing.

Do note, however, that actually integrating the code into a production system is explicitly **not** an aim of this exercise. The goal is to demonstrate a few techniques and to satisfy an intellectual curiosity, not to recommend an approach for any setting with real stakes.

As with any complex task, the key is to start small and focus on one problem element at a time.

FIRST ATTEMPT: PROMPT BASICS AND ZERO-SHOT PROMPTING

What's the simplest possible thing that might actually work? Try just asking the model directly!

```
%send_prompt(result, %nrstr(
    user|
    Can you impute the following date for me?
    Use the last day of the month if none is specified: February 2024.
));
proc print data=result; run;
```

The SAS System

Obs	ordinal_choices	ordinal_message	role	content	refusal
1	1	1	assistant	The specified date is February 2024. Since no specific day is mentioned, we will use the last day of the month, which is February 29, 2024, because 2024 is a leap year.	.

Figure 6: A first attempt at date imputation.

This is an example of what is called "zero-shot" prompting, which relies entirely on the model's pretraining (DAIR.AI 2025). Many of the models are already very powerful and you can often get great results with no more effort than this. On the other hand, it can be misleading if it doesn't result in exactly what you wanted right away. It's not necessarily obvious that other techniques are available, but that is what prompt engineering is all about.

In this case, it appears that ChatGPT "understands" the task correctly and even knows about leap years. However, it spends too many words talking about the result. The goal is to get this integrated into a data processing pipeline and that extra text will just get in the way.

This is a good opportunity to exploit some of the other prompt roles that are available. The "system" role can often be used to provide general instructions or guidance to the model. Try using a system prompt to control the model's output:

```
%send_prompt(result, %nrstr(
    system|
    The user will ask you for several dates.
    You are to apply the specified imputation rules and respond with only the result
    of the imputation.~
    user|
    Can you impute the following date for me?
    Use the last day of the month if none is specified: February 2024.
));
proc print data=result; run;
```

The SAS System

Obs	ordinal_choices	ordinal_message	role	content	refusal
1	1	1	assistant	February 29, 2024.	.

Figure 7: Attempting to use a system prompt to impute dates.

That actually seems to work great! It's also a good demonstration of a general prompt engineering guideline: prompts should be clear and direct (Anthropic 2025). This system prompt is better than the first attempt in part because it clearly and directly states the desired output format.

Of course, it does not suffice to test just one input case. Try another:

```
%send_prompt(result, %nrstr(
    system|
    The user will ask you for several dates.
    You are to apply the specified imputation rules and respond with only the result
    of the imputation.~
    user|
    Can you impute the following date for me?
    Use the last day of the month if none is specified: May 11 2024.
));
proc print data=result; run;
```

The SAS System

Obs	ordinal_choices	ordinal_message	role	content	refusal
1	1	1	assistant	May 31, 2024	.

Figure 8: Attempting to use a system prompt to impute dates, part two.

Here, the model incorrectly imputes the day even though a full date was specified. It's possible that iterating on the system prompt would be enough to solve this, but there are also many other techniques to try out that might be more effective.

As an aside, it will be tedious to continue testing one user input at a time. The remainder of these examples will make use of a testing utility that sends multiple prompts and compiles the results. The full code for the macro is too long and distracting to print here, but it is included in the appendix. Here's how to use it:

```
%send_multiple_prompts(
    result,
    system_prompt=%nrstr(
        system|
        The user will ask you for several dates.
        You are to apply the specified imputation rules and respond with only the result
        of the imputation.~
        user|
        Can you impute the following date for me?
        Use the last day of the month if none is specified: ),

    user_prompts=%nrstr(February 2024|May 11, 2024|July of 1999|Christmas 2011)
);
proc print data=result; run;
```

The SAS System

Obs	user_prompt	content
1	February 2024	February 29, 2024
2	May 11, 2024	May 31, 2024
3	July of 1999	July 31, 1999
4	Christmas 2011	December 31, 2011

Figure 9: Demonstrating multiple prompts.

TEACH BY EXAMPLE: FEW-SHOT PROMPTING

Sometimes, the easiest way to get what you want is to demonstrate it. Supplying even a single example can yield a suprising amount of improvement in model performance. Providing examples as part of the prompt is a way to induce contextual learning by the model and is what's referred to as "few-shot" prompting (DAIR.AI 2025).

In practice, the "assistant" role can be used to create example responses. These "previous" interactions steer the model towards the desired behaviour. Try an example of this now:

```
%send_multiple_prompts(result, system_prompt=%nrstr(
    system|
    The user will ask you for several dates.
    Apply the following rules and respond with only the result of the imputation.
    Imputation rules:
    1.  If no day is supplied, impute the day to the last day of the month.
        Otherwise, use the supplied day. ~
    user|February 23, 2024.~
    assistant|February 23, 2024~
    user|May 2024~
```

```
    assistant|May 31, 2024~
    user|
), user_prompts=%nrstr(May 11, 2024|Feb 2023|Independence day 1990|2012 June)
);
proc print data=result; run;
```

The SAS System

Obs	user_prompt	content
1	May 11, 2024	May 11, 2024
2	Feb 2023	February 28, 2023
3	Independence day 1990	July 4, 1990
4	2012 June	June 30, 2012

Figure 10: Trying a few-shot prompt.

As you can see, with only a few examples the model's performance improves dramatically. It even handles the holiday curveball correctly.

That's enough toy examples. It's time to get thrown into the deep end. Can you get the model to correctly process significantly more complicated imputation rules? Here is a real set of instructions from a paper about date imputation, adapted into a few-shot prompt (Akinyi 2021):

```
%send_multiple_prompts(results, system_prompt=%nrstr(
    system|
    The user will supply you with several dates regarding event start and end dates.
    Apply the following rules to impute the end date of the event.
    Respond with only the result of the imputation.
    Imputation rules:
    1.  For start date: if day is missing then impute to the first day of the month.
        If both day and month are missing the impute to 01-Jan.
    2.  For end date: if day is missing then impute with last day of that month and
        if day and month both are missing then impute with 31-Dec.
        If the imputed end date is after the date of death or last known date alive
        then set end date to death date if not missing, else set to last known date
        alive.
    3.  If the year is missing for both start and end dates, then keep as is, no
        imputation is required if complete date is missing.~
    user|start:2020-03-22, end:2020-03-25, death:2020-08-03, last known alive:2020-05-02~
    assistant|2020-03-25~
    user|start:2020-03, end: 2020-03, death:2020-08-03, last known alive:2020-05-02~
    assistant|2020-03-31~
    user|start:2018-12, end: 2019-02, death:2020-08-03, last known alive:2020-05-02~
    assistant|2019-02-28~
    user|start:2020, end: 2020, death:2020-08-03, last known alive:2020-05-02~
    assistant|2020-12-31~
    user|),
    user_prompts=%nrstr(
        start:2020-06,    end: 2020-06,    death:2020-08-03, last known alive:2020-05-02|
        start:2020-02,    end: 2020-02,    death:2020-08-03, last known alive:2020-05-02|
        start:2020-02,    end: 2020-02,    death:2020-01-13, last known alive:2019-11-27|
        start:2018,       end: 2019,       death:2020-01-13, last known alive:2019-11-27|
        start:2019-10-30, end: 2019-11,    death:2020-01-13, last known alive:2019-11-27|
        start:2019-10-30, end: 2019-11-12, death:2020-01-13, last known alive:2019-11-27
    )
);
proc print data=results; run;
```

The SAS System

Obs	user_prompt	content
1	start:2020-06, end: 2020-06, death:2020-08-03, last known alive:2020-05-02	2020-06-30
2	start:2020-02, end: 2020-02, death:2020-08-03, last known alive:2020-05-02	2020-02-29
3	start:2020-02, end: 2020-02, death:2020-01-13, last known alive:2019-11-27	2020-01-13
4	start:2018, end: 2019, death:2020-01-13, last known alive:2019-11-27	2019-12-31
5	start:2019-10-30, end: 2019-11, death:2020-01-13, last known alive:2019-11-27	2019-11-30
6	start:2019-10-30, end: 2019-11-12, death:2020-01-13, last known alive:2019-11-27	2019-11-27

Figure 11: Trying a real-world date imputation scenario.

All of the "training" cases and the first two user cases are from the source paper. The model correctly imputes the source paper's user cases and manages to adapt to a different set of death and last-known-alive dates. However, it starts to fall down for the more complex cases where the imputed event date is after last-known-alive but before death. Also, it is again incorrectly imputing complete dates.

You can (and should!) try a few approaches based on the topics covered so far:

- Make the system prompt more clear and direct about what to do in the ambiguous cases.
- Provide more examples that cover all of the possibilities.

However, sometimes these techniques won't suffice, or the resulting system will be unreliable. It's also not ideal to just guess-and-check your way through refining the prompt. The next two techniques can help with both seeing what the model is doing and ensuring that it follows complex instructions without dropping the ball.

THINK IT THROUGH: CHAIN-OF-THOUGHT AND PROMPT CHAINING

As you've just seen, it can be difficult to get the model to follow all of the rules correctly when working on more complicated tasks. If the model could explain what it is thinking, that would likely help significantly in diagnosing the issue.

As a matter of fact, you can do just that! The concept of chain-of-thought prompting is to have the model explain its reasoning to itself. Somehow, this seems to help, though exactly *why* it helps is not well understood. It's thought that this ability is an emergent property of models with sufficiently many parameters (DAIR.AI 2025).

Try it with a simpler version of the last prompt. Notice how the system prompt has changed to emphasize the thinking process rather than the final output:

```
%send_multiple_prompts(results, system_prompt=%nrstr(
    system|
    The user will supply you with several dates regarding event start and end dates.
    First, think carefully about which rules apply for the input.
    In particular, decide if any imputation is required at all.
    Then, if imputation is needed, note where the end date falls relative to the death
    and last known alive dates.
    State how the end date should be imputed based on this information.
    Finally, state the correct result of the imputation.
    Imputation rules:
    1.  For start date: if day is missing then impute to the first day of the month.
        If both day and month are missing the impute to 01-Jan.
    2.  For end date: if day is missing then impute with last day of that month and
        if day and month both are missing then impute with 31-Dec.
        If the imputed end date is after the date of death or last known date alive
        then set end date to death date if not missing, else set to last known date
        alive.
    3.  If the year is missing for both start and end dates, then keep as is, no
        imputation is required if complete date is missing.~
    user|),
```

```
    user_prompts=%nrstr(
        start:2020-06,    end: 2020-06,    death:2020-08-03, last known alive:2020-05-02|
        start:2018,       end: 2019,       death:2020-01-13, last known alive:2019-11-27|
        start:2019-10-30, end: 2019-11,    death:2020-01-13, last known alive:2019-11-27|
        start:2019-10-30, end: 2019-11-12, death:2020-01-13, last known alive:2019-11-27
    )
);
proc print data=results; run;
```

The SAS System

Obs	user_prompt	content
1	start:2020-06, end: 2020-06, death:2020-08-03, last known alive:2020-05-02	In this case, we have the following information: - Start date: 2020-06 (month is provided, but the day is missing) - End date: 2020-06 (month is provided, but the day is missing) - Death date: 2020-08-03 - Last known alive date: 2020-05-02 ### Start Date Imputation: The start date is 2020-06 with a missing day. According to the imputation rules, if the day is missing, we impute to the first day of the month. Thus, the imputed start date will be: **Start Date: 2020-06-01** ### End Date Imputation: The end date is 2020-06, also with a missing day. Following the same imputation rule, we will impute this to the last day of June. The last day of June is June 30, so the imputed end date will be: **End Date: 2020-06-30** Now we need to check if the imputed end date (2020-06-30) falls after the date of death (2020-08-03) or the last known alive date (2020-05-02). The imputed end date (2020-06-30) is before the death date (2020-08-03) and after the last known alive date (2020-05-02). Therefore, no further changes are needed for the end date. ### Summary of Results: - Imputed Start Date: 2020-06-01 - Imputed End Date: 2020-06-30 (which remains unchanged as it's before death date) **Final Result:** - Start Date: 2020-06-01 - End Date: 2020-06-30
2	start:2018, end: 2019, death:2020-01-13, last known alive:2019-11-27	In this case, we have the following dates: - Start date: 2018 (year is provided, but day and month are missing) - End date: 2019 (year is provided, but day and month are missing) - Death date: 2020-01-13 - Last known alive date: 2019-11-27 Following the imputation rules: 1. **Start Date Imputation**: Since both the day and month are missing, we impute to the first day of the year. So, the start date becomes 2018-01-01. 2. **End Date Imputation**: Similarly, since both the day and month are missing, we impute to the last day of the year. So, the end date becomes 2019-12-31. Now, we need to check where the imputed end date falls relative to the death and last known alive dates: - The imputed end date (2019-12-31) is after the last known alive date (2019-11-27) but before the death date (2020-01-13). Since the end date is after the last known alive date, we have to set the end date to the last known alive date of 2019-11-27. So, the final results are: - Imputed start date: 2018-01-01 - Imputed end date: 2019-11-27 Final result: - Start: 2018-01-01, End: 2019-11-27
3	start:2019-10-30, end: 2019-11, death:2020-01-13, last known alive:2019-11-27	Let's analyze the given dates based on the provided imputation rules. 1. **Start date:** The start date provided is complete: 2019-10-30 (October 30, 2019). Therefore, there is no need for imputation for the start date. 2. **End date:** The end date provided is incomplete: 2019-11. It is missing the day. According to the rules, we need to impute the day to the last day of that month. November has 30 days, so we impute the end date as 2019-11-30 (November 30, 2019). Next, we need to check how the imputed end date (2019-11-30) relates to the death date and last known alive date: - Death date: 2020-01-13 - Last known alive date: 2019-11-27 The imputed end date (2019-11-30) is after the last known alive date (2019-11-27) but before the death date (2020-01-13). Since the end date is after the last known alive date, we must adjust it to the last known alive date since it is the last recorded date when the individual was known to be alive. So, following the imputation rules: **Result of imputation:** - The final end date should be set to the last known alive date, which is **2019-11-27**. Therefore, the imputed dates are: - Start: **2019-10-30** - End: **2019-11-27**
4	start:2019-10-30, end: 2019-11-12, death:2020-01-13, last known alive:2019-11-27	In this case, both the start and end dates are complete and provided. - Start date: 2019-10-30 - End date: 2019-11-12 - Death date: 2020-01-13 - Last known alive date: 2019-11-27 Since the end date (2019-11-12) falls before the last known alive date (2019-11-27), no imputation is necessary for the end date. It is valid as it is. Therefore, the correct result is: - Start date: 2019-10-30 - End date: 2019-11-12 No changes made, thus the imputed end date remains 2019-11-12.

Figure 12: Demonstrating chain of thought reasoning.

Aha! This output is very revealing. The model clearly states its reasoning step by step and its final answers are coherent with their respecitve explanations.

This is great feedback for adjusting the prompt. For instance, if, say, case 3 should have been imputed to the date of death, it's clear how to adjust the prompt to get this outcome. You can now go back to the original prompt and revise the rules to be more direct and/or supply additional examples.

This prompt seems to also fix the problem with imputing complete dates even though it doesn't provide any examples. Different prompting techniques can be somewhat interchangeable.

Now what if, rather than going back and revising the earlier prompt, you want to use this style directly in the application? Chain-of-thought is often highly effective but it does result in a lot of output clutter. This is where prompt chaining comes in.

The idea behind prompt chaining is to break down a task into multiple prompts specialized for a distinct step in the process (Anthropic 2025). You do not try to force the model to handle the entire job in one go. Instead, you design a logical sequence of prompts such that the output of one can be fed directly to another. For instance, after applying the reasoning step, you can do something like this:

```
%send_multiple_prompts(results, system_prompt=%nrstr(
    system|
    The user will supply you with a long chain of reasoning about a date imputation.
    Your job is to extract _only_ the final result for the "end date" and print it by
    itself. It is very likely that the target date will be at the very end of the text.
    Use ISO 8601 formatting.~
    user|),
    user_prompts=%nrstr(<<<FULL TEXT OMITTED FOR BREVITY>>>)
);
proc print data=results; run;
```

The SAS System

Obs	user_prompt	content
1	Lets analyze the given dates based on the provided imputation rules. 1. **Start date:** The start date provided is complete: 2019-10-30 [October 30, 2019]. Therefore, there is no need for imputation for the start date. 2. **End date:** The end date provided is incomplete: 2019-11. It is missing the day. According to the rules, we need to impute the day to the last day of that month. November has 30 days, so we impute the end date as 2019-11-30 [November 30, 2019]. Next, we need to check how the imputed end date [2019-11-30] relates to the death date and last known alive date: - Death date: 2020-01-13 - Last known alive date: 2019-11-27 The imputed end date [2019-11-30] is after the last known alive date [2019-11-27] but before the death date [2020-01-13]. Since the end date is after the last known alive date, we must adjust it to the last known alive date since it is the last recorded date when the individual was known to be alive. So, following the imputation rules: **Result of imputation:** - The final end date should be set to the last known alive date, which is **2019-11-27**. Therefore, the imputed dates are: - Start: **2019-10-30** - End: **2019-11-27**	2019-11-27

Figure 13: Using a prompt chain to extract the final answer.

That demonstrates the idea. The next step would be to automatically extract the response from the first prompt and feed it directly into the second prompt without having to copy and paste. This is left as an exercise to the reader.

WHERE TO GO NEXT

Hopefully, this introduction has been sufficient to whet your appetite! There are many details that were glossed over which are worth your time to investigate further, including:

- Other prompt parameters: temperature, "top p", stop sequences, etc.
- Using a prompt generator
- Comparing performance and speed between models

There are also many advanced topics in prompt engineering. To provide a sample:

- Use of images and audio in prompts
- Structured outputs (automatic JSON formatting of responses)
- Tree-of-thoughts prompting
- Meta prompting
- Model fine-tuning

There is plenty to explore! The user and prompting guides in the references are good places to get started, in addition to the API documentation itself.

CONCLUSION

You have now built a working SAS API client! You are now also familiar with the basics of prompt engineering and have a sense of how to apply its principles. The world of language models is your oyster.

There is no better way to learn about both the capabilities and limitations of language models than to try to solve problems with them. There is a lot of hype that grows unimpeded in the imagination but collapses entirely after a moment's experimentation. At the same time, these models continue to surprise even their most practiced users with unanticipated abilities. It is a magical time to be learning about and with this technology. Make the most of it!

ADDITIONAL READING

GITHUB REPOSITORY

This paper is available as a Jupyter notebook. Clone or fork the repo on github at https://github.com/austrowj/psug2025-chatgpt-in-sas (don't forget to star! 😊)

TECHNICAL RESOURCES

OpenAI API Documentation: https://platform.openai.com/docs/api-reference/introduction

DeepSeek API Documentation: https://api-docs.deepseek.com/

Tidyverse ellmer - Call LLM APIs from R: https://github.com/tidyverse/ellmer

GENERAL LLM DISCUSSION

Skliar, Illia. 2023. "Boosting SAS Programming Efficiency with ChatGPT: A Clinical Trials Perspective." PHUSE Connect EU 2023. Available at https://www.lexjansen.com/phuse/2023/cm/PAP_CM04.pdf

Sturdy, Ian. 2024. "Leveraging ChatGPT in Statistical Programming in the Pharmaceutical Industry." PharmaSUG 2024. Available at https://www.lexjansen.com/pharmasug/2024/AP/PharmaSUG-2024-AP-256.pdf

Tarap, Karma. 2023. "LLMs Unleashed: A New Chapter in Clinical Programming?" PHUSE Connect 2023. Available at https://www.lexjansen.com/phuse/2023/cm/PAP_CM07.pdf

Vysotskyi, Mykyta. 2024. "Navigating the Safe Integration of AI (ChatGPT) into Clinical Trials Programming Workflow." PHUSE Connect US 2024. Available at https://www.lexjansen.com/phuse-us/2024/et/PAP_ET09.pdf

REFERENCES

Akinyi, Teckla. 2021. "Common Dating in R: With an example of partial date imputation." PharmaSUG 2021. Available at https://pharmasug.org/proceedings/2021/QT/PharmaSUG-2021-QT-139.pdf

Henry, Joseph. 2019. "The ABCs of PROC HTTP." SAS Global Forum 2019. Available at https://support.sas.com/resources/papers/proceedings19/3232-2019.pdf

Linker, Adam. 2019. "Creating and Controlling JSON Output with the JSON Procedure." SAS Global Forum 2019. Available at https://support.sas.com/resources/papers/proceedings19/3506-2019.pdf

Mc Cawille, Stephen. 2024. "Getting a Prompt Response Using ChatGPT and SAS." PHUSE EU Connect 2024. Available at https://phuse.s3.eu-central-1.amazonaws.com/Archive/2024/Connect/EU/Strasbourg/PAP_CT02.pdf

Anthropic. 2025. "Claude User Guide." Accessed 30 March 2025. https://docs.anthropic.com/en/docs/welcome

DAIR.AI. 2025. "Prompting Guide." Accessed 30 March 2025. https://www.promptingguide.ai/

CONTACT INFORMATION

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

James Austrow
Cleveland Clinic
austroj AT ccf DOT org

APPENDIX: MULTIPLE PROMPTS UTILITY

```
%macro send_multiple_prompts(output_ds, system_prompt=, user_prompts=);

    /* Clean up the output dataset if it exists */
    proc datasets lib=work nolist;
        delete &output_ds;
    quit;

    %let i = 1;
    %let single_prompt = %scan(&user_prompts, &i, |);

    %do %while(%length(&single_prompt));

        /* Build the full prompt for this iteration */
        %let full_prompt = &system_prompt.&single_prompt;

        /* Temporary dataset for each iteration */
        %let temp_ds = _tmp_resp_&i;

        %put &full_prompt.;

        /* Call the existing %send_prompt macro */
        %send_prompt(&temp_ds, %superq(full_prompt));

        /* Add a variable for the user prompt */
        data &temp_ds;
            length user_prompt $2000;
            set &temp_ds;
            user_prompt = "&single_prompt";
        run;

        /* Append this result to the compilation output dataset */
        %if &i = 1 %then %do;
            data &output_ds;
                length user_prompt $2000 content $2000;
                set &temp_ds;
                keep user_prompt content;
            run;
        %end;
        %else %do;
            proc append base=&output_ds data=&temp_ds force; run;
        %end;

        /* Move to the next prompt */
        %let i = %eval(&i + 1);
        %let single_prompt = %scan(&user_prompts, &i, |);

    %end;

    /* Reorder for clarity: user prompt first, then response */
    data &output_ds;
        retain user_prompt;
        set &output_ds;
    run;

%mend;
```