Prompt engineering

GPT-5:

[1] https://platform.openai.com/docs/guides/prompt-engineering

Claude Sonnet 4:

[2] https://docs.claude.com/en/docs/build-with-claude/prompt-engineering/overview

In order to prevent bias in the prompt, we want to come up with a prompt that use best practices of both LLMs. First, we will look at the best practices of GPT-5 and Sonnet 4. After which we will analyse the similarities to create a combined prompt. This combined prompt will then be evolved into the final prompt by looking at the results on test dataset outputs.

ChatGPT

ChatGPT documents the following best practices [1]:

Use model snapshots

Message roles and instruction following

- Can be done via instructions parameter in the api
- By giving it in the instruction's parameter, it prioritizes the instructions parameter text over the input parameter
-

Reusable prompts

- Helps you with building and evaluating prompts

Message formatting

- Can use markdown and xml in order to understand logical boundaries
- Input format
    - Purpose
    - Instructions
    - Examples
    - Context
- Can save cost and latency with prompt cashing

Few-shot learning

- Show diversity of desired outputs
- Show both the input and output

Include relevant context

- Reasons to give context
    - Give model access to proprietary data
    - To constrain the model's response to specific set of resources
- Called RAG

Prompt GPT-5 models

- Benefit from precise instructions
- Maximizing code performance, from planning to execution
    - Excellent at building applications in one shot
        - Use GPT 5's thorough planning and self-reflection capabilities
    - Surrounding context important
    - Details about product behavior

Prompt reasoning models not needed, we use GPT

Claude

Claude documents the following best practices for their prompt engineering order from broadly effective to more specialized techniques [2]:

- Prompt generator
- Be clear and direct
- Use examples (multishot)
- Let claude think
- Use XML tags
- Give claude a role
- Prefill claude's response
- Chain complex prompts
- Long context tips

They also mentioned some general principles

- Be explicit with your instructions
- Add context to improve performance
- Be vigilant with examples and details
- Long-horizon reasoning and state tracking (not applicable, multi-task)
    - Context awareness and multi-window workflows (only sonnet 4.5)
    - Multi-content window workflows
    - State management best practices
- Communication style

Guidance for specific situations

- Balance verbosity
- Tool usage patterns
- Control the format of responses
- Research and information gathering
- Subagent orchestration
- Model self-knowledge
- Leverage thinking & interleaved thinking capabilities
- Document creation
- Optimize parallel tool calling
- Reduce file creation in agentic coding
- Enhance visual and frontend code generation
- Avoid focusing on passing tests and hard-coding
- Minimizing hallucinations in agentic coding
- Migration considerations

Prompt generator is not needed

Use prompt templates

- API calls will consist of two types of content
  - Fixed content: context/instructions that remain constant across multiple interactions
  - Variable content: dynamic elements that change each request
- You should always use prompt templates when you expect any part to be repeated

Prompt improver is not needed

Be clear and direct

- Give contextual information
- Be specific about what you want claude to do
- Provide instructions as sequential steps

Use examples (multishot prompting)

- Must be
  - Relevant
  - Diverse
  - Clear (use tag wrap <example> or <examples>)

Let claude think (Chain of Thought)

- Balance performance and latency
- How (basic to advanced)

    - Basic: "think step by step"
    - Guided: outline specific steps for claude to follow
    - Structured: use XML tags

Use XML tags

- Game-changer when having multiple components <context>, <instruction>, <example>
- Should nest tags when needed for hierarchical content

Give claude a role

- Using the system parameter to give it a role

Prefill claude's response

- Done in the assistant message

Chain complex prompts is something we don't do

Long context tips

- Place it at the top
- Structure with xml tags
- Ground responses in quotes

Extended thinking tips

- Requires the extended thinking mode of a model
- Comes with structures

Use model snapshots

Combined prompt

As we can see from both LLMs best practices, we can apply many different techniques to strive for better performing models. Apart from using the few-shot prompting, which is a variable in our experiment, we use the following best practices for our prompt:

- Use of XML formatting/structure
- Use precise explanation and description
- Let the LLM act as a person

This leads to the following prompt for **zero-shot**:

<instruction>

You, a software developer, need to insert code documentation to make the Python code more readable and comprehensible. This code documentation should be done on all functions and classes to explain what they do. The documentation of a function should be written underneath the function header and the documentation of a class should be written underneath the class header. As an input, you will get a Python file that doesn't contain any comments and only consists of a class and functions. You will then insert the comments into that file to complement the code. The output should be the same python file with the same code but with the added documentation for functions and classes.

<\instruction>

<*file*>

And the following prompt for **multi-shot**:

<instruction>

You, a software developer, need to insert code documentation to make the Python code more readable and comprehensible. This code documentation should be done on all functions and classes to explain what they do. The documentation of a function should be written underneath the function header and the documentation of a class should be written underneath the class header. As an input, you will get a Python file that doesn't contain any comments and only consists of a class and functions. You will then insert the comments into that file to complement the code. The output should be the same python file with the same code but with the added documentation for functions and classes.

<\instruction>

<examples>

<example 1 input file> *file* <\example 1 input file>

<example 1 output file> *file*<\example 1 output file>

<example 2 input file>*file* <\example 2 input file>

<example 2 output file> *file*<\example 2 output file>

<example 3 input file> *file*<\example 3 input file>

<example 3 output file> *file*<\example 3 output file>

<\examples>

<*file*>

Prompt evolution

To evaluate the prompt, we use the test set of our project. From this test set, we strip the comments, apply the LLMs to generate comments and see if the output is as expected, but also compare with the stripped comments to see if they line-up.

First prompt

For the first prompt, we use the best practices prompt.

Input:

<instruction>

You, a software developer, need to insert code documentation to make the Python code more readable and comprehensible. This code documentation should be done on all functions and classes to explain what they do. The documentation of a function should be written underneath the function header and the documentation of a class should be written underneath the class header. As an input, you will get a Python file that doesn't contain any comments and only consists of a class and functions. You will then insert the comments into that file to complement the code. The output should be the same Python file with the same code but with the added documentation for functions and classes.

</instruction>

<file>

qiskit-transpiler-coupling.py

</file>

Output GPT-5:

Here's your **Python file with added documentation** for every class and function, following your instructions:

<first prompt result GPT5.py>

Would you like me to export this documented version as a **.py file** for download?

Output Claude Sonnet 4:

<first prompt result claude.py>

Second prompt

From the first prompt result, we see that GPT gives more information than we eventually want as an output. So we must specify more what we should expect in the output.

Input:

<instruction>

You, a software developer, need to insert code documentation to make the Python code more readable and comprehensible. This code documentation should be done on all functions and classes to explain what they do. The documentation of a function should be written underneath the function header and the documentation of a class should be written underneath the class header. As an input, you will get a Python file that doesn't contain any comments and only consists of a class and functions. You will then insert the comments into that file to complement the code. The output should be the same Python file with the same code but with the added documentation for functions and classes. Also, the output file should only be the Python file and nothing else.

</instruction>

<file>

qiskit-transpiler-coupling.py

</file>

Output GPT5:

<second prompt result GPT5.py>

Output Claude sonnet 4:

<second prompt result claude.py>

Third prompt:

The second prompt gave only the desired output as requested. However, in this version, nothing had been said about the readability of in-line documentation. By saying that the readability should be improved by adding comments between the lines within functions, we can get better documentation.

Input:

<instruction>

You, a software developer, need to insert code documentation to make the Python code more readable and comprehensible. This code documentation should be done on all functions and classes to explain what they do. The documentation of a function should be written underneath the function header and the documentation of a class should be written underneath the class header. To improve the readability of a function, comments between the lines in functions might also be needed. As an input, you will get a Python file that doesn't contain any comments and only consists of a class and functions. You will then insert the comments into that file to complement the code. The output should be the same Python file with the same code but with the added documentation for functions and classes. Also, the output file should only be the Python file and nothing else.

</instruction>

<file>

qiskit-transpiler-coupling.py

</file>

Output GPT5:

<third prompt result GPT5.py>

Output Claude sonnet 4:

<third prompt result claude.py>

Fourth prompt:

The third prompt gave more information, but also redundancy. By specifying omitting redundancy of inline level and function level, the readability would likely to be improved

Input:

<instruction>

You, a software developer, need to insert code documentation to make the Python code more readable and comprehensible. This code documentation should be done on all functions and classes to explain what they do. The documentation of a function should be written underneath the function header and the documentation of a class should be written underneath the class header. To improve the readability of a function, comments between the lines in functions might also be needed but don't make it redundant if not necessary. As an input, you will get a Python file that doesn't contain any comments and only consists of a class and functions. You will then insert the comments into that file to complement the code. The output should be the same Python file with the same code but with the added documentation for functions and classes. Also, the output file should only be the Python file and nothing else.

</instruction>

<file>

qiskit-transpiler-coupling.py

</file>

Output GPT5:

<fourth prompt result GPT5.py>

Output Claude sonnet 4:

<fourth prompt result claude.py>

Final prompt result

For zero-shot:

<instruction>

You, a software developer, need to insert code documentation to make the Python code more readable and comprehensible. This code documentation should be done on all functions and classes to explain what they do. The documentation of a function should be written underneath the function header and the documentation of a class should be written underneath the class header. To improve the readability of a function, comments between the lines in functions might also be needed but don't make it redundant if not necessary. As an input, you will get a Python file that doesn't contain any comments and only consists of a class and functions. You will then insert the comments into that file to complement the code. The output should be the same Python file with the same code but with the added documentation for functions and classes. Also, the output file should only be the Python file and nothing else.

</instruction>

<file>

qiskit-transpiler-coupling.py

</file>

For multi-shot:

<instruction>

You, a software developer, need to insert code documentation to make the Python code more readable and comprehensible. This code documentation should be done on all functions and classes to explain what they do. The documentation of a function should be written underneath the function header and the documentation of a class should be written underneath the class header. To improve the readability of a function, comments between the lines in functions might also be needed but don't make it redundant if not necessary. As an input, you will get a Python file that doesn't contain any comments and only consists of a class and functions. You will then insert the comments into that file to complement the code. The output should be the same Python file with the same code but with the added documentation for functions and classes. Also, the output file should only be the Python file and nothing else.

<\instruction>

<examples>

<example 1 input file> *file* <\example 1 input file>

<example 1 output file> *file*<\example 1 output file>

<example 2 input file>*file* <\example 2 input file>

<example 2 output file> *file*<\example 2 output file>

<example 3 input file> *file*<\example 3 input file>

<example 3 output file> *file*<\example 3 output file>

<\examples>

<file>

file

<\file>