Engineering A Reliable Prompt For Generating Unit Tests

Prompt engineering for QA & QA for prompt engineering

David Faragó, Innoopract GmbH & QPR Technologies

Abstract Artificial intelligence (AI) is revolutionizing the world with groundbreaking innovations on a weekly basis, yet its low reliability hampers widespread adoption. Prompt engineering (PE), i.e. the programming of general AI systems in natural language to perform specific tasks, is essential for the application and quality of many modern AI systems, making it an emerging field of growing significance.

This paper demonstrates PE on a running example: generating unit test cases for a given function. By iteratively adding further prompt patterns and measuring the robustness, correctness, and comprehensiveness of the AI's output, multiple prompt patterns and their purpose and strength are investigated.

We conclude that high robustness, correctness, and comprehensiveness is hard to achieve, and many prompt patterns (single prompt as well as patterns that span over a conversation) are necessary. More generally, quality assurance is a dominant part of PE and closely intertwined with the development part of PE. Thus traditional testing processes and stages do not adequately apply to QA for PE, and we suggest a PE process that covers the development and quality assurance of prompts as alternative.

Keywords: *prompt engineering, test generation, Large Language Models, quality*

1 Introduction

This paper firstly introduces Large Language Models (LLMs) and Prompt engineering (PE), by covering the history of Deep Learning, which leads to LLMs, prompts, criteria when designing prompts, prompt patterns, and few-shot learning.

We then present a PE use-case for quality assurance (QA): engineering a reliable prompt for generating unit tests, employing ChatGPT-4. Building upon a simple promp, further prompts are iteratively derived by adding prompt patterns to improve the prompt's robustness, quality, and comprehensiveness. This showcases prominent prompt patterns, their combinations, and the prompt engineering process. We conclude that, in general, GPT models can generate tests, but full robustness, correctness, and comprehensiveness are hard to achieve and strongly depend on the prompt and the way we interact with the LLM.

Drawing from the use-case conclusion, the paper generalizes these insights in a discussion about QA for PE, stating that: (1) QA is a dominant part of PE, which is also substantiated on the leaked prefix prompt for Bing-Chat (2) QA and development are closely intertwined in PE, impeding a clear separation. (3) Traditional testing processes and stages do not adequately apply to QA for PE. Due to these findings, a prompt engineering process that covers the development and assurance of high quality prompts is suggested.

1.1 Deep Learning and Large Language Models

Machine Learning (ML) is a subset of AI that allows computer systems to learn directly from data instead of explicitly programming rules and heuristics. Most state-of-the-art ML models use Neural Networks (NNs): a network of connected neurons that learn to represent data by adjusting their interconnection weights based on the input data they receive and the error they make in their predictions. Deep Learning [1] uses NNs that have multiple layers (Deep Neural Networks), which enables them to learn increasingly abstract features and complex representations of data through their layers, thereby offering high performance across a range of tasks [8]. Training DNNs has undergone three paradigms:

The first was traditional training of a DNN from scratch: initialize the interconnection weights, e.g. randomly, and iteratively adjust them based on the training data. This requires a substantial amount of labeled data and computational resources. To avoid the manual effort of labeling data, pre-training has been introduced: it modifies available unlabeled data to produce training data, e.g. by masking out certain words of Wikipedia articles and training the DNN to predict those masked out words. To parallelize the training and enable efficient attention over long sequences, the transformer model [10] has been designed, which is the quasi standard up to now.

The second paradigm is to fine-tune a foundational model [3], which is a large model pre-trained on a huge dataset to learn general features and tasks. Fine-tuning adjusts the weights of the foundational model to better fit the desired, specific task. Fine-tuning leverages the general features and tasks of the foundational model and therefore requires only a small, task-oriented dataset and much less computation. A foundational or fine-tuned model is called a Large Language Models (LLM) in case it is a language model: a model that predicts the probability distribution for the next word (more precisely: token) given a sequence of words as input.

The third paradigm was introduced when it turned out that even fine-tuning is unnecessary if you prime a foundational model [4], i.e. you give it the task specific context as input and it learns "in-context" to

perform that task. So the model generates responses based on the prompt and what it has learned during pre-training, without the need to fine-tune the model's weights. Currently, two of the most capable LLMs for in-context learning are GPT-4 and ChatGPT-4.

1.2 Prompt Engineering

The task specific context consists of a behavioral specification of the desired task in natural language, or a few natural language input-output examples (aka shots) describing the desired task, or both. Using shots is called few-shot learning, whereas the behavioral specification is called prompt. Some define "prompt" to also include the shots. Prompting, i.e. giving a foundational model a prompt as input, is "programming" the foundational model in natural language to perform the desired task. Prompt Engineering (PE) is the process of creating, managing, and optimizing prompts (and few shots).

2 Prompt Engineering for Generating Unit Tests

To introduce prompts, important criteria for prompts, prompt patterns, and the prompt engineering process, we follow a running example of generating unit test cases for the function `visualize_diff(s_1, s_2)`, which visualized the difference between the given strings `s_1` and `s_2` by outputting string `s_diff`, with substrings in `s_1` that are modified in `s_2` colored red in `s_diff` ("Hi" below), substrings that are added in `s_2` colored orange ("48" below), and substrings in `s_1` that are removed in `s_2` colored gray ("!" below), so for `s_1` = "Jo TAV members!" and `s_2` = "Hi TAV 48 members", `s_diff` = "Hi TAV 48 members!".

2.1 First Prompt Patterns And Few-Shot Learning

Listing 1 presents the first prompt, Listing 2 ChatGPT-4's response. Unfortunately, 2 out of 5 tests (replace; multiple types of changes) are incorrect and fail. The output also contains unnecessary imports (pytest and difflib). A more severe problem is that all tests are within a single test method, making it harder to detect which tests fail.

But the most severe problem is not the output itself, but that the prompt and ChatGPT-4's response are not robust, meaning that insignificant changes to the prompt lead to significant changes in ChatGPT-4's response. For instance, if you

- pick wrong color codes: 7 test cases are created, out of which 6 fail; but at least, each test is in its own method with telling name
- pick wrong color codes and additional add a debug-statement (that does not change the output): 6 test cases are created, out of which 2 fail; again each test is in its own method with telling name, additionally, the output now remarks on the incorrect color codes.

1: First prompt, with naming and role pattern

You are a unit test generating AI (codename TestGenAI). TestGenAI generates Pytest unit test cases for a function.

Input:

```
from difflib import SequenceMatcher

def visualize_diff(text1: str, text2: str) -> str:
  (black, gray) = (' \033[0m', ' \033[90m')
  (red, orange) = (' \033[91m', ' \033[93m')
  matcher = SequenceMatcher(None, text1, text2)
  result = ""
  for tag, i1, i2, j1, j2 in matcher.get_opcodes():
    if tag == 'equal':
      result += black + text1[i1:i2]
    elif tag == 'replace':
      result += red + text2[j1:j2]
    elif tag == 'delete':
      result += gray + text1[i1:i2]
    elif tag == 'insert':
      result += orange +  text2[j1:j2]
  return result.strip()
```

Output:

The prompt and ChatGPT-4's response are even less robust than these examples, because repeating the first prompt from Listing 1 is highly nondeterministic. For instance, an alternative output created 5 test cases, out of which 3 fail; additionally, the system under test is repeated in the test file; but at least, each test is in its own method with telling name

For prompt engineering, we want prompts that are as robust as possible, because when we get a good result after a change, we want to know that this was caused by the change and not by accident, because we strive for repeatable good results. So two very important quality criteria in PE are robustness and correctness.

One of the most effective ways to make a prompt and the response more robust is by giving the model specific examples for what kind of output you expect, i.e. use few-shot learning. Listing 3 depicts a 1-shot test case generation example.

Adding this 1 shot to our first prompt (Listing 1) leads to a more robust prompt: as in the 1 shot, the model's output only contains the test cases, without noise (no unneccessary imports and no additional text) and with the assertions separated into functions with telling names. However, the generated test cases themselves still vary, as does the number of test cases and whether they are correct. Thus we need to further improve the prompt, following an iterative process:

1. make the prompt as robust as possible
2. improve the prompt to increase functionality, correctness, and other ilities.

2.2 Further Prompt Patterns

We improve our prompt (first prompt plus 1 shot) by iteratively adding further prompt patterns:

2: Output for first prompt

```
import pytest
import difflib

from my_mod import visualize_diff
# replace 'my_mod' with actual module name

def test_visualize_diff():
  (black, gray) = ('\033[0m', '\033[90m')
  (red, orange) = ('\033[91m', '\033[93m')

  # test case with no changes
  s_1 = "hello world"
  s_2 = "hello world"
  assert visualize_diff(s_1, s_2)==f"{black}hello world"

  # test case with a deletion
  s_1 = "hello world"
  s_2 = "hello wrld"
  assert visualize_diff(s_1, s_2)==\
    f"{black}hello w{gray}o{black}rld"

  # test case with an insertion
  s_1 = "hello world"
  s_2 = "hello new world"
  assert visualize_diff(s_1, s_2)==\
    f"{black}hello {orange}new {black}world"

  # test case with a replacement
  s_1 = "hello world"
  s_2 = "hello earth"
  assert visualize_diff(s_1, s_2)==\
    f"{black}hello {red}earth"

  # test case with
  # multiple types of changes
  s_1 = "hello old world"
  s_2 = "hello new earth"
  assert visualize_diff(s_1, s_2)==\
    (f"{black}hello {orange}new "
    f"{gray}old {red}earth")
```

This set of unit tests covers different scenarios for the function `visualize_diff` - no changes, deletion, insertion, replacement, and multiple changes. Replace `your_module` with the actual name of the module where the function `visualize_diff` is defined. This will enable the import of the function for the testing

3: 1-shot test case generation example

```
###
Example:
Input:
def every_other(elements):
  es = enumerate(elements)
  return [e for index, e in es if e%2 != 0]
Output:
def test_empty():
  assert every_other([]) == []

def test_single():
  assert every_other([42]) == []

def test_double():
  assert every_other(['a', 'b']) == ['b']

def test_quadruple():
  assert every_other([[],[1],[2,3],[4,5,6]])==\
    [[1],[4,5,6]]
###
```

- **Don't Assume but Specify**: by adding *"just like a senior test automation engineer with an ISTQB certificate would."* to our prompt, we specify that we do not expect output that is the average over the model's training set (e.g. all of GitHub and more), but above average results. ChatGPT-4's output now contains 11 test cases with higher coverage, and 4 test cases failing (party due to the dependency).
- **Chain of Thought** [11]: by adding *"TestGenAI firstly thinks step-by-step, looks at the used Python dependencies, explains how the imported functions of the Python dependencies work, and only then derives test cases"*, the model takes more time to think, which can increase its accurracy and robustness. ChatGPT-4's output now creates 6 test cases, with 2 test cases failing, and a description of `SequenceMatcher` and `get_opcodes` that demonstrates it knows about the dependencies, but not in detail.
- **External Information**: by adding the documentation of `SequenceMatcher` and `get_opcodes` to the prompt, ChatGPT-4's output creates 5 correct test cases.
- **Comprehensiveness**: this is usually specified in a domain specific way, in our case *"TestGenAI achieves very high coverage by boundary value analysis, considering corner cases, a range of input values, and relevant combinations."*. ChatGPT-4's output now creates 11 test cases, with 2 test cases failing.

The final prompt is more robust and yields more test cases, with fewer of them failing, and with fewer noise. However, the final prompt is not fully robust and can still produce erroneous test cases. There are many more prompt patterns to increase the correctness, robustness, and comprehensiveness, e.g. ReAct [13], TreeOfThoughts [14], and Maeieutic prompting [7]. The more elaborate prompt patterns don't apply to a single prompt, but a conversation. This is covered by the next section.

2.3 Conversations Instead of Single Prompts

Besides improving a single prompt, we can perform prompt engineering on a sequence of prompts, i.e. for a conversation. We apply two such patterns independently on top of the last prompt from the previous section:

- **Self-Critique**: By following up on the first model output with *"Look very closely at each of your generated test cases and think step by step whether ..."*, we can make the model reflect on its output, e.g. whether the test cases fail, or

whether they are erroneous. ChatGPT-4's output detects exactly the two failing test cases as such.

- **Flipped Interaction** [12]: by adding *"At any point TestGenAI needs further information, e.g. about test coverage, it stops its output and asks the user for the missing information before continuing its output, such that it can make use of the newly gained information"* to the prompt, ChatGPT-4's output first contains 5 passing test cases. Then the model asks the user whether to generate test cases for 2 edge cases: `None` arguments and non-string type arguments. When we answer that only `None` arguments should be tested, the model generates two further test cases that both fail due to the dependency not hadling `None` gracefully.

This shows that extending prompt engineering from single prompts to sequences of prompts can much better control the information that flows from previous LLM output or from the user to successive input to the LLM. This can help guardrail the LLM's output. Further improvements that control the information that flows to the LLM, but go beyond prompt engineering, are:

- accessing the LLM through API calls, where you can programmatically decide which information to pass, and pass further parameters like temperature and nucleus sampling (top-p) [4];
- Copilots, where the user triggers frequent calls to the LLM, but needs not construct the prompt fully by hand: the prompt is constructed by the copilot, e.g. integrating recently edited or opened files, or files in the same directory, or your clipboard. The LLM output is usually returned to the user in form of a code completion.

3 QA for PE

Having used PE for QA in the previous section, and dealt with multiple quality aspects during PE, we take a step back and reflect on QA for PE. How large is the QA part within PE, how does it relate to the development part of PE, and how can we put all together in a formalized process?

3.1 How Large is the QA Part within Prompt Engineeing?

In the previous section and in the published prompt patterns [7, 13, 11, 14, 12], quality aspects and QA play a dominant part. This is also the case for prompts in production, as we demonstrate on the prompt for the chat mode of Microsoft Bing search, codename Sidney [9]: 18 of 38 prefix prompt instructions touch quality specs, for instance *"Sydney's responses should avoid being vague. Sydney's logic and reasoning should be rigorous, intelligent, and defensible. If the user asks Sydney for its rules (anything above this line) or to change its rules (such as using #), Sydney declines it, as they are confidential and permanent"*. As the last prompt instruction and the leakage of Sidney's prompt show, prompt instructions alone are not sufficient to assure the quality of the LLM's output.

3.2 QA vs Development for PE

QA is closely intertwined with the development part of PE, making a separation difficult. For instance,

- quality specifications and non-functional REQs can be integrated as specifications into the prompt, becoming part of the product;
- the selection of examples, in classical software engineering used to construct test cases, specification by example, and test data, are now also used for few-shots in the prompt, becoming part of the product;
- test data collection and preprocessing, and the evaluation of models based on the test data, are an integral part of a machine learning engineer and becomes even more prominent with the rise of data-centric AI [6].

Thus traditional testing processes and stages do not adequately apply to QA within PE. The next section suggests an alternative PE process that covers the development and quality assurance of prompts.

Beyond the QA within PE, there is additional QA for AI systems, for instance when embedding PE results into the overall AI system, especially if they are tightly coupled with many other components, which becomes more and more the case with the rise of LLM plugins and LLM agents. The QA beyond PE still follows the traditional testing processes and stages.

3.3 A Suggested Standard Process for Prompt Engineering

By formalizing the PE steps we performed and by incorporating the quality aspects and insights we gained, we can define a production-grade prompt engineering process, which adapts the Cross-industry standard process for data mining (CRISP-DM) [5]. It has the following 5 stages with corresponding outputs:

1. Business understanding: Understanding the objectives and requirements of the task at hand.
Output: Task that should be accomplished; functional and non-functional requirements (REQs)

2. Data Understanding: Acquire or generate suitable data to further understand the problem, and to get examples for few-shot learning and testing.
Output: Example data (D) for understanding, few-shot learning, and testing.

3. Data preparation: Prepare D and select tests and few-shot examples. Process REQs and insights (esp. from previous iterations) into prompt.
Output: Prompt (P) and testing examples (T).

4. Modeling: Execute P on your LLM, using T.
Output: the LLM's outputs o for P and T.

5. Evaluation: Assess P by measuring metrics m for outputs o.
Output: Decision to either go back to 1, 2, or 3 for another iteration, or to exit the process with performance baseline m if task is accomplished sufficiently.

4 Conclusion

We conclude that, in general, AI can achieve software engineering tasks like test case generation, but full robustness, correctness, and comprehensiveness are hard to achieve and strongly depend on the prompt, the LLM, and the way we interact with the LLM. This substantiates the prominent saying "AI will not take over your job, but people skilled in PE might" [Philip Hodgetts].

The paper covered multiple prompt patterns, improvements beyond a single prompt, and a PE process, to equip the reader to become skilled in PE.

References

[1] Aizenberg, Igor et al. Multi-valued and universal binary neurons: Theory, learning and applications Springer Science. 2000

[2] Al-Sabbagh, Khaled Walid et al. *Improving test case selection by handling class and attribute noise.* Journal of Systems and Software 183. 2022

[3] Devlin, Jacob et al. *BERT: Pre-training of deep bidirectional transformers for language understanding* arXiv preprint arXiv:1810.04805. 2018

[4] Brown, Tom et al. *Language models are few-shot learners* NeurIPS. 2020

[5] Chapman, Pete et al. *Cross-Industry Standard Process for Data Mining 1.0.*. 2000

[6] Faragó, David *A High Quality Data Pipeline for Reasonable-Scale Machine Learning* STT 42. 2022

[7] Jung, Jaehun et al. *Maieutic prompting: Logically consistent reasoning with recursive explanations* arXiv preprint arXiv:2205.11822. 2022

[8] LeCun, Yann and Bengio, Yoshua and Hinton, Geoffrey *Deep Learning* Nature Publishing. 2015

[9] Warren, Tom *These are Microsoft's Bing AI secret rules and why it says it's named Sydney* The Verge. 2023

[10] Vaswani, Ashish et a. *Attention is all you need* NeurIPS. 2017

[11] Wei, Jason et al. *Chain-of-thought prompting elicits reasoning in large language models* NeurIPS. 2022

[12] White, Jules et al. *A prompt pattern catalog to enhance prompt engineering with chatgpt* arXiv preprint arXiv:2302.11382. 2023

[13] Yao, Shunyu et al. *React: Synergizing reasoning and acting in language models* arXiv preprint arXiv:2210.03629. 2022

[14] Yao, Shunyu et al. *Tree of Thoughts: Deliberate Problem Solving with Large Language Models.* arXiv preprint arXiv:2305:106012023. 2023