# Best practices of Prompt

## Avoid negative prompting

Instead of instructing the model about the practices it should avoid, it prefers rephrasing them as the practices that are more appropriate.

## Avoid redundancy

In the process of eliminating redundancy from the model’s responses, it is possible to get carried away and include redundancy in the prompt itself. Thus, being precise with the language is always better.

## Avoid vagueness

Focus on one aspect of the model at a time. Be target specific and stick to it till consistency in the desired output is achieved.

## Direction

You’re not briefing the AI on what types of names you want. Do you want a single word or a concatenation? Can the words be made up or is it important they’re in real english? What sort of audience are you hoping to attract?

## *Format*

You’re getting back a list of newline separated names, of arbitrary length. When you run this prompt multiple times you’ll see sometimes it comes back with a numbered list, and often it has text at the beginning which makes it hard to parse programmatically.

## Evaluation

You have no feedback loop here to identify which names are good or bad, or to improve the quality of the name generator over time. If you can institute a rating system you can optimize the prompt to get better results, and identify the times when it fails.

## Division of Importance

How AI should decide the priority or each output

## The Five Principles of Prompting:

1. *Give Direction*: Describe what you’re imagining, to get an output matching your vision.
2. *Specify Format*: Define the response you want, and minimize time spent parsing errors.
3. *Provide Examples*: Insert examples in your prompts, to improve the reliability of the output.
4. *Evaluate Quality*: Identify errors and rate responses, testing what drives performance.
5. *Divide Labor*: Split tasks into multiple prompts, chained together for complex goals.

# Injection through Prompt

## Adversarial prompting

Malicious prompt engineering can be a subtle method of misconduct with the large language model. Malpractices like **prompt injection, prompt leaking, and jailbreaking** are used to intentionally generate undesired outputs. Adversarial prompting is generally intentional and occurs due to irresponsible prompt engineering. It can be prevented by including defending statements in the instruction element of the prompt. In addition, adversarial prompt detectors act as a defensive shield against any such malicious entries to the LLM.

## Faulty facts

This involves the hindrance with the data sources of the AI model. It is such a subtle challenge that without proper knowledge of the particular concept, it is difficult to identify the inaccuracy in the responses of the LLM. Factual inaccuracy can be prevented by providing suitable examples in the prompt or thoroughly examining the information sources

## Bias

Biased and opinionated content is forever a considerable challenge, especially in the case of conversational AI models. Improper prompt engineering may give rise to stereotypes, leading to biases. Thus, a strategically organized prompt engineering process plays a crucial role in identifying bias on behalf of the model and its remedy. To prevent the generation of bias, diversity in prompts is an essential action.

# Prompt Validation Approaches

<https://medium.com/@geoffdudgeon/test-driven-prompt-engineering-4c32c6c405b2>

# Prompt Tools Injection, Validation

<https://github.com/microsoft/promptbench>

PromptBench is a powerful tool designed to scrutinize and analyze the interaction of large language models with various prompts. It provides a convenient infrastructure to simulate black-box adversarial prompt attacks on the models and evaluate their performances. This repository hosts the necessary codebase, datasets, and instructions to facilitate these experiments.

Check our paper: [PromptBench: Towards Evaluating the Robustness of Large Language Models on Adversarial Prompts](https://arxiv.org/abs/2306.04528).

Prompt Engineering 101: Testing

<https://subedi.medium.com/prompt-engineering-101-testing-96221a07f709>

Prompt Engineering Best Practice Oreilly

https://www.oreilly.com/library/view/prompt-engineering-for/9781098153427/ch01.html

## Finding an alternative Prompt

Iterating on and testing prompts can lead to radical decreases in the tokens used, and therefore the cost and latency of your system. If you can find another prompt that performs equally as well (or better) but uses fewer tokens, you can afford to scale up your operation considerably. Often you’ll find in this process that many elements of a complex prompt are completely superflouous, or even counter productive.

# Philosophy behind testing LLM

1. Accurate information retrieval

2. Causal reasoning,

3. Logic

4. Incorporating intentional context.

### **Deterministic vs. non-deterministic tests**

**One way to categorize tests is by their predictability, dividing them into deterministic and non-deterministic tests. Deterministic tests are those where the inputs, process, and outputs are predictable and repeatable, producing the same results every time they run when given the same conditions. If a user applies input A, they’ll always receive output B. In most contexts, unit tests are considered deterministic tests, although some integration tests could be considered deterministic as well.**

* Similarity testing

from sentence\_transformers import SentenceTransformer

from sklearn.metrics.pairwise import cosine\_similarity

model = SentenceTransformer('all-MiniLM-L6-v2')  
expected\_embedding = model.encode("Rome is the capital of Italy.")actual\_embedding = model.encode("The capital of Italy is Rome.")  
similarity = cosine\_similarity([expected\_embedding], [actual\_embedding])[0][0]  
print(f"Similarity: {similarity}")

#### **Contextual Memory and Context Limitation**

LLM’s are stateless. Each incoming request is treated differently and processed independently. Application like chatbot requires to remember the context and previous interaction. How LLM powered app handles it ? What is the limitation ?

#### **Handling Errors and Failures**

Integration with external APIs introduces the possibility of network failures, timeouts, or other errors. How your application handles such scenarios can significantly impact user experience and overall application stability.

**Cache Strategy**

If the inputs repeat over multiple calls, its always advisable to maintain a cache and save the pre-processed LLM output there, thereby serving it directly from the cache preventing extra network hop

**User Privacy and Data Security**

 No sensitive proprietary data or personally identifiable information (PII) are exposed

**Language Quality**

Evaluate the quality of the response from a language perspective.

Evaluations along dimensions like tonality (if the response matches the desired brand tone), creativity, and interestingness of the response, etc. require additional information about the given task or persona