Chapter 6: Prompt Engineering - Comprehensive Notes

1. Introduction to Prompt Engineering

Prompt engineering is the art and science of crafting inputs to guide Large Language Models (LLMs) toward desired outputs. This chapter explores techniques for effectively interacting with generative models like GPT through careful prompt design.

****Key Concepts:****
- LLMs generate text in response to prompts
- Prompt engineering enhances output quality
- Techniques range from simple instructions to complex reasoning frameworks

## 2. Choosing and Loading Text Generation Models

### Model Selection Considerations

****Proprietary vs. Open Source Models:****
- Proprietary models (GPT-4): Higher performance but less flexibility
- Open source models (Phi-3, Llama 2): More flexible, free to use

Foundation Models Overview:
```

| Model | Parameter Sizes |
|---|---|
| Llama | 7B/13B/33B/70B |
| StableLM | 3B/7B |
| Falcon | 7B/40B/180B |
| Llama 2 | 7B/13B/70B |
| Mistral | 7B |
| Phi-3 | 3.8B (mini) |

```

Figure 6-1: Foundation models available in different sizes

Recommendation: Start with smaller models like Phi-3-mini (3.8B parameters) which can run on devices with up to 8GB VRAM.

### Loading Models with Transformers

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)

tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")

# Create pipeline
pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
    max_new_tokens=500,
    do_sample=False,
)
```

## 3. Understanding Prompt Templates

When using chat models, prompts are converted into specific templates:

```
# User message
messages = [{"role": "user", "content": "Create a funny joke about chickens."}]

# Apply template
prompt = pipe.tokenizer.apply_chat_template(messages, tokenize=False)
print(prompt)
# Output: <s><|user|>Create a funny joke about chickens.<|end|><|assistant|>
```

Phi-3 Template Structure:

```
┌──────────────────────────────────────────────┐
│ <s>                                  │
│   <|user|> Create a funny joke about chickens.       │
│   <|end|>                             │
│   <|assistant|>                        │
│                                     │
└──────────────────────────────────────────────┘
```

Figure 6-2: Phi-3 chat template structure

- `<s>`: Beginning of sequence token
- `<|user|>`, `<|assistant|>`: Role tokens
- `<|end|>`: End of turn token

## 4. Controlling Model Output

LLMs generate text probabilistically. We can control this randomness through parameters:

### Temperature

Controls the randomness of output:
- Low temperature (0.2): Deterministic, focused output
- High temperature (0.8-1.0): Creative, diverse output

Effect of Temperature:

Low Temperature   High Temperature
car   car
truck   truck
...   ...
elephant   elephant
...   ...
probabilities   probabilities

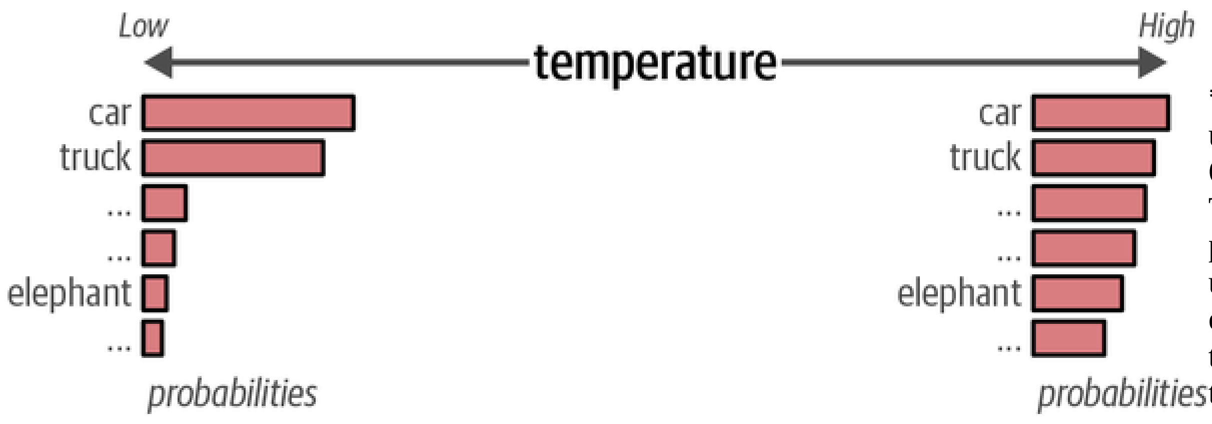


Figure 6-4: Temperature effect on token selection

```python
# Using temperature
output = pipe(messages, do_sample=True, temperature=1)
```

### Top-p (Nucleus Sampling)

Controls which subset of tokens the model considers:
- top_p = 0.1: Only top 10% probability tokens
- top_p = 1.0: All tokens considered

Effect of Top-p:
```
Low top_p    High top_p
```

```
   car                car
   truck               truck
   ...               ...
   elephant              elephant
   ...               ...
probabilities         probabilities
```

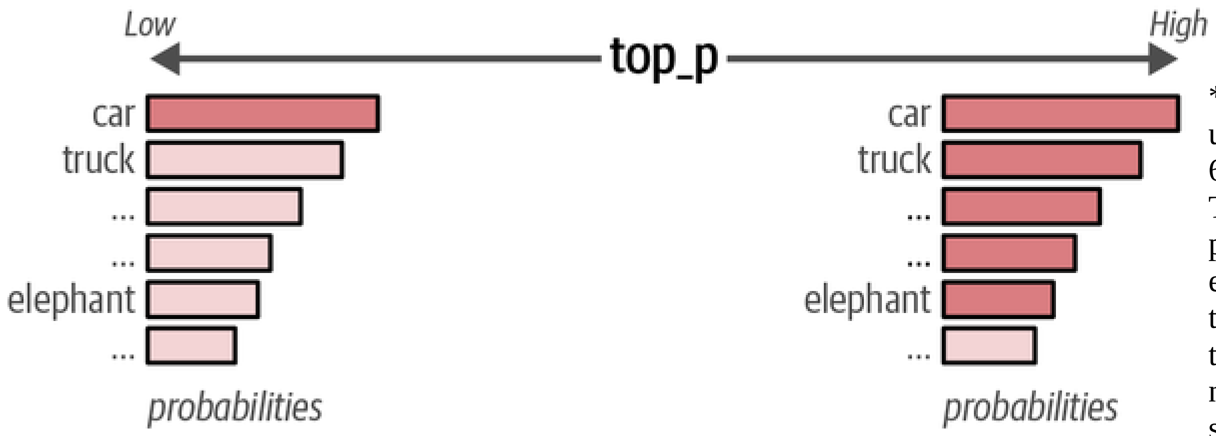


Figure 6-5: Top-p effect on token selection

```python
# Using top_p
output = pipe(messages, do_sample=True, top_p=1)
```

### Parameter Selection Guide

Use Case	Temperature	top_p	Description
Brainstorming	High	High	Diverse, creative ideas
Email generation	Low	Low	Predictable, formal tone
Creative writing	High	Low	Creative but coherent
Translation	Low	High	Accurate with varied vocabulary

Table 6-1: Parameter selection for different use cases

## 5. Fundamentals of Prompt Engineering

### Basic Prompt Components

Minimal Prompt:
```
Input: The sky is
Output: blue.
```

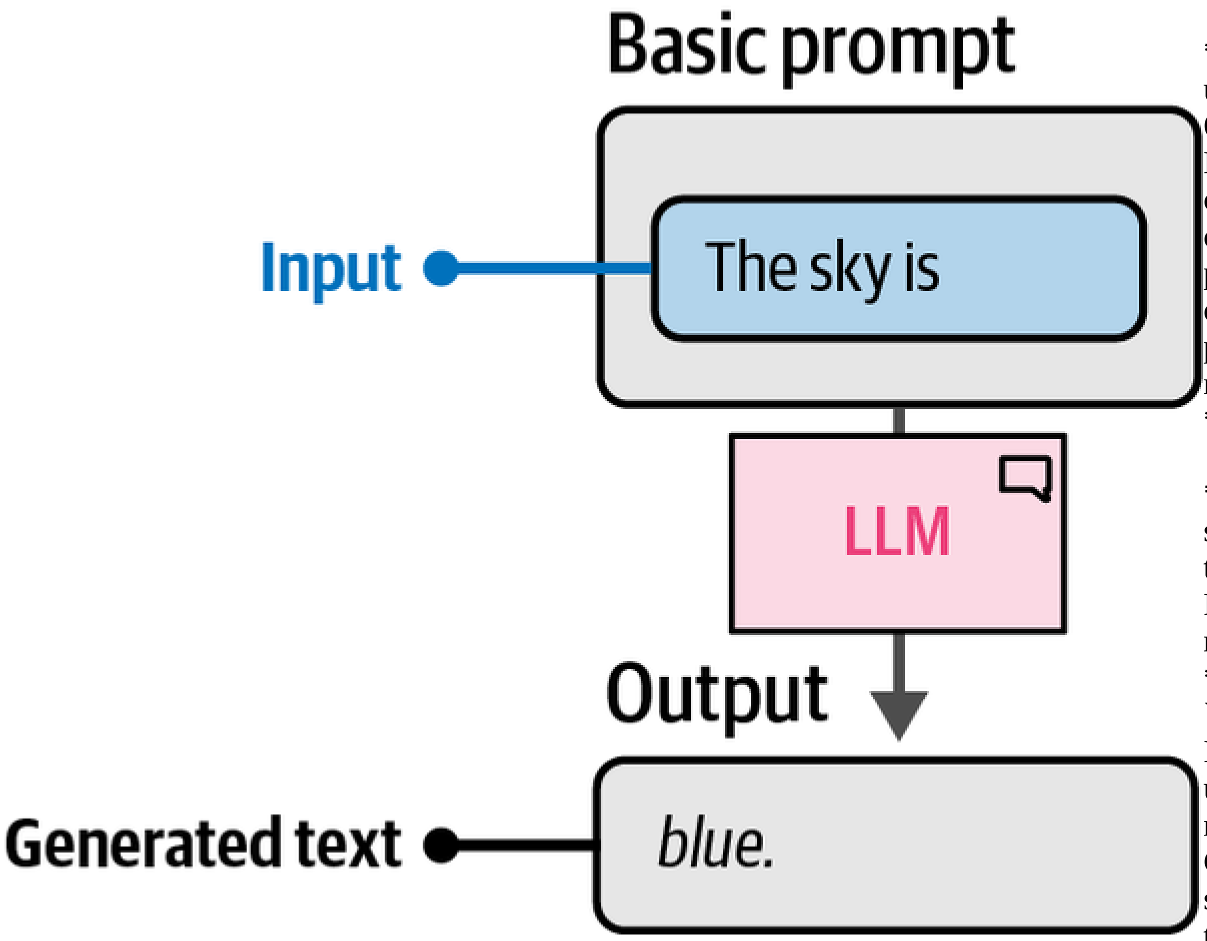


Figure 6-6: Basic completion prompt

Instruction Prompt:
```
Instruction: Classify the text into negative or positive.
Data: "This is a great movie!"
Output: The text is positive.
```

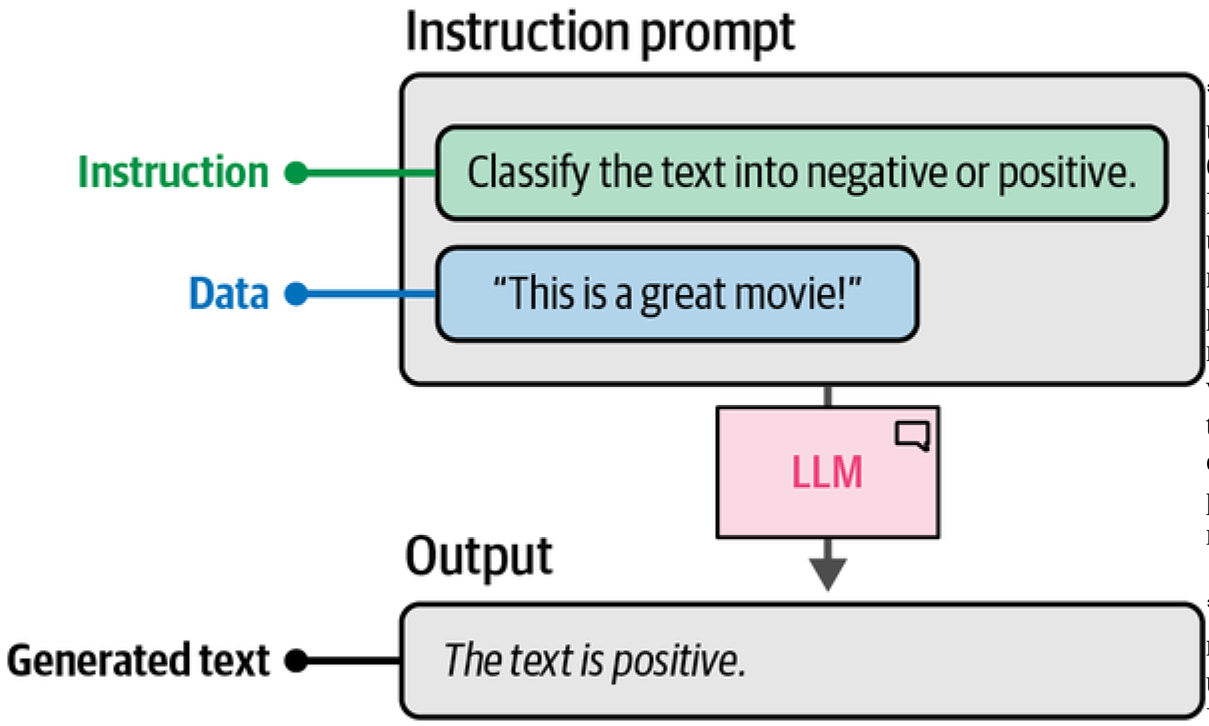


Figure 6-7: Instruction prompt with two components

Structured Prompt:
```
Instruction: Classify the text into negative or positive.
Data: Text: "This is a great movie!"
Output indicator: Sentiment:
Output: Positive
```

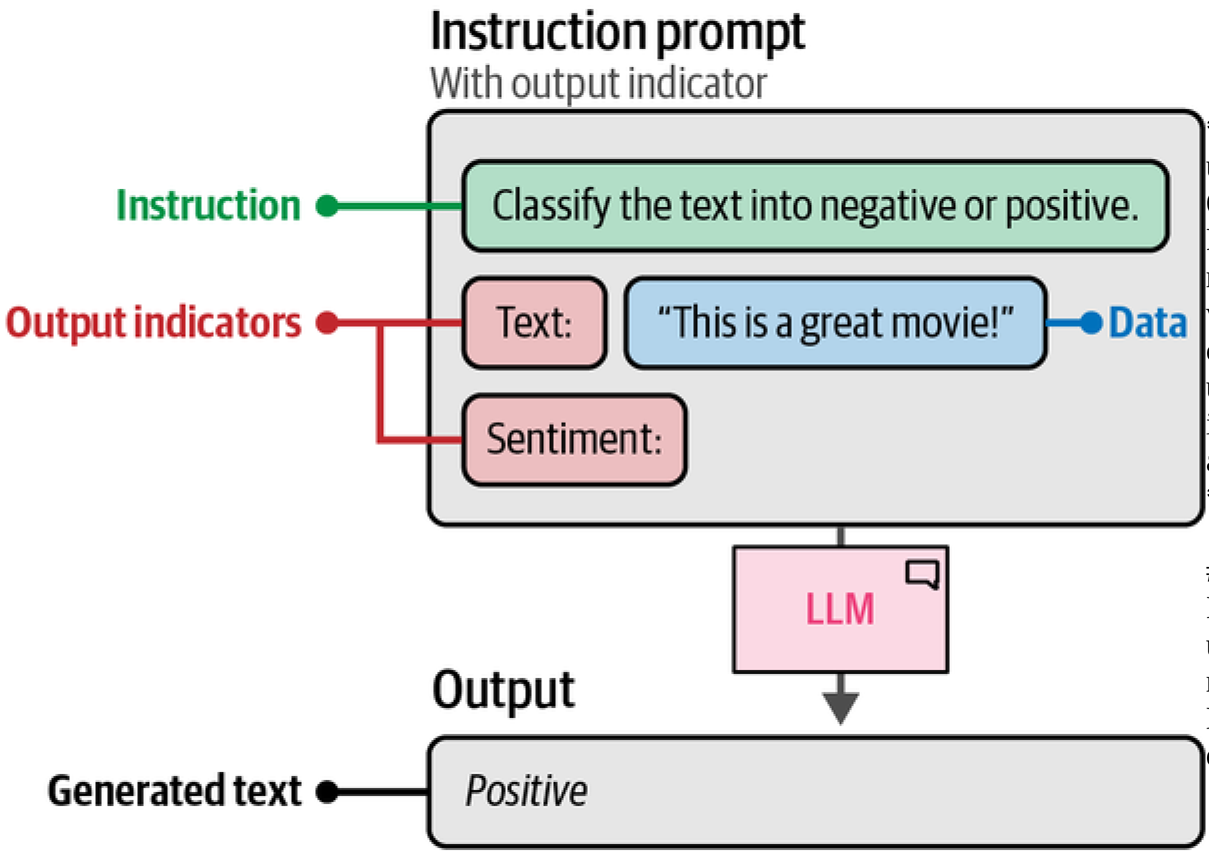


Figure 6-8: Prompt with output indicator

Instruction-Based

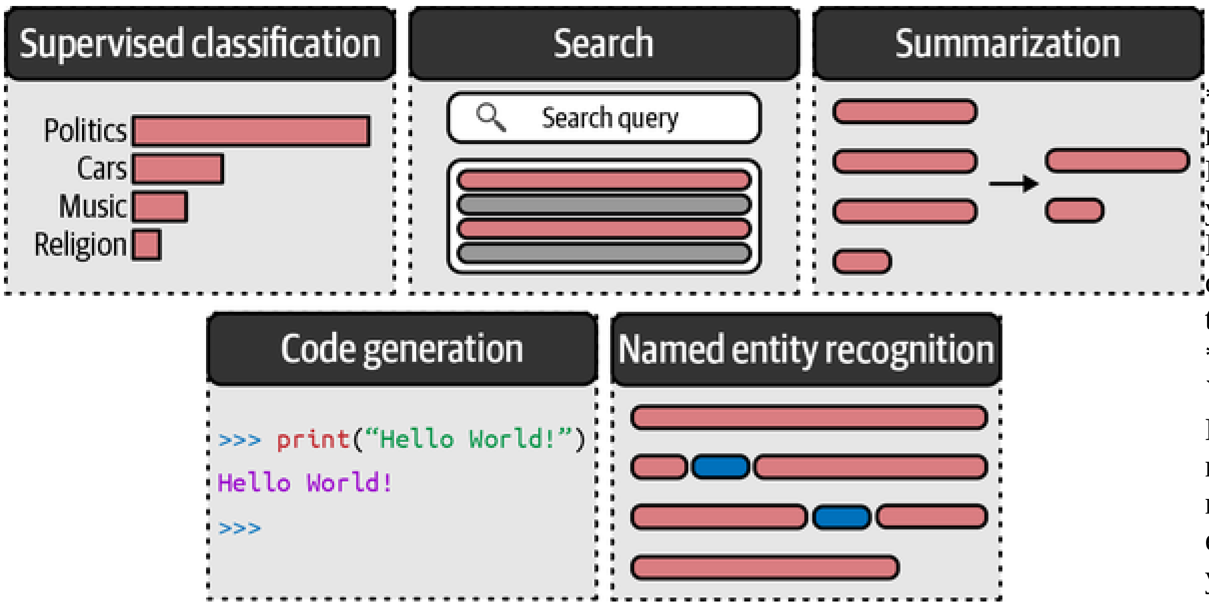
Prompting

Common use cases for instruction-based prompting:
- Summarization
- Classification
- Code generation
- Named entity recognition

Example Prompts by Use Case:

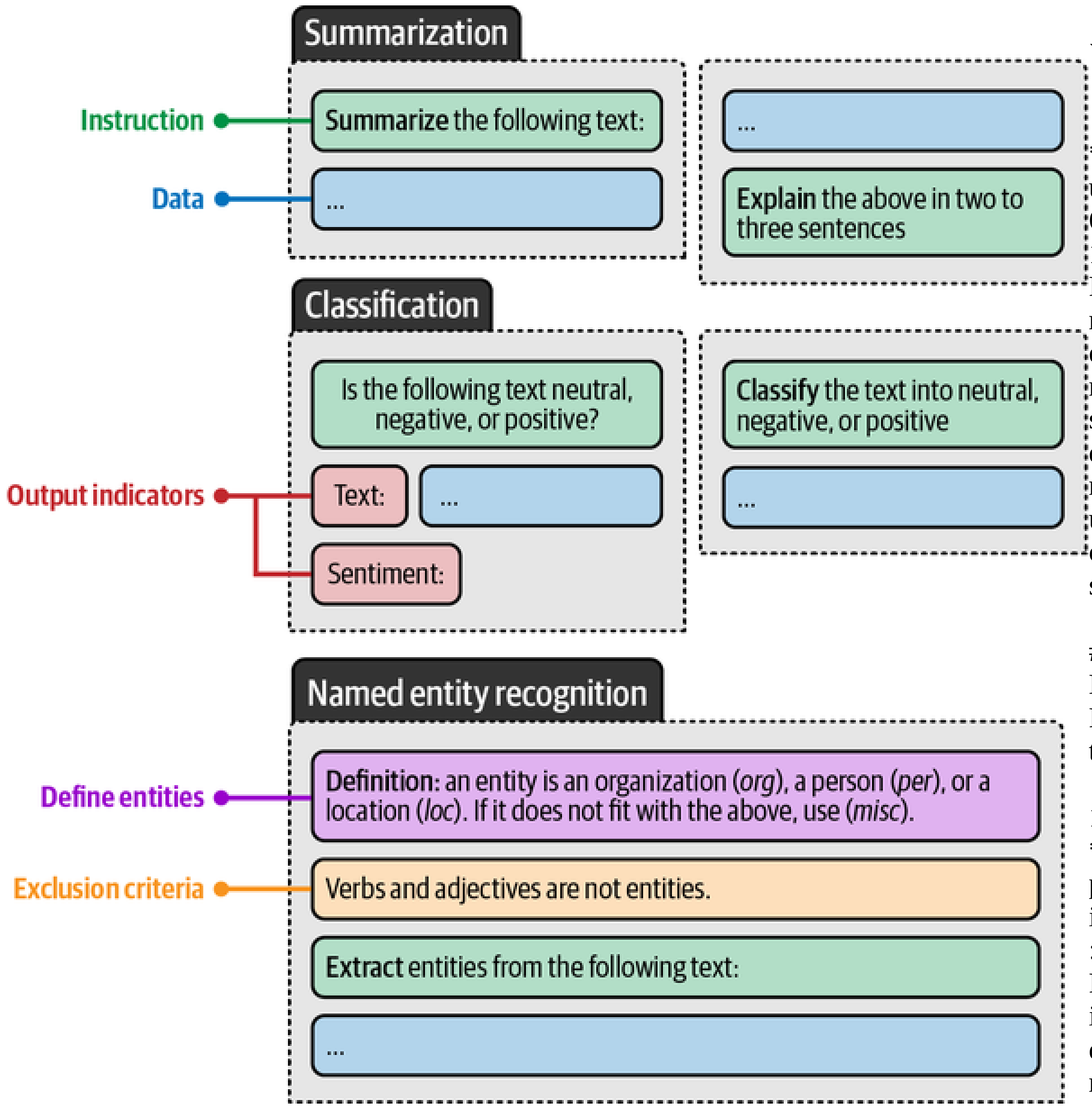
Summarization:
```
Summarize the following text: [text]
Explain the above in two to three sentences
```

Classification:
```
Is the following text neutral, negative, or positive?
Text: [text]
Sentiment:
```



```
>>> print("Hello World!")
Hello World!
>>>
```

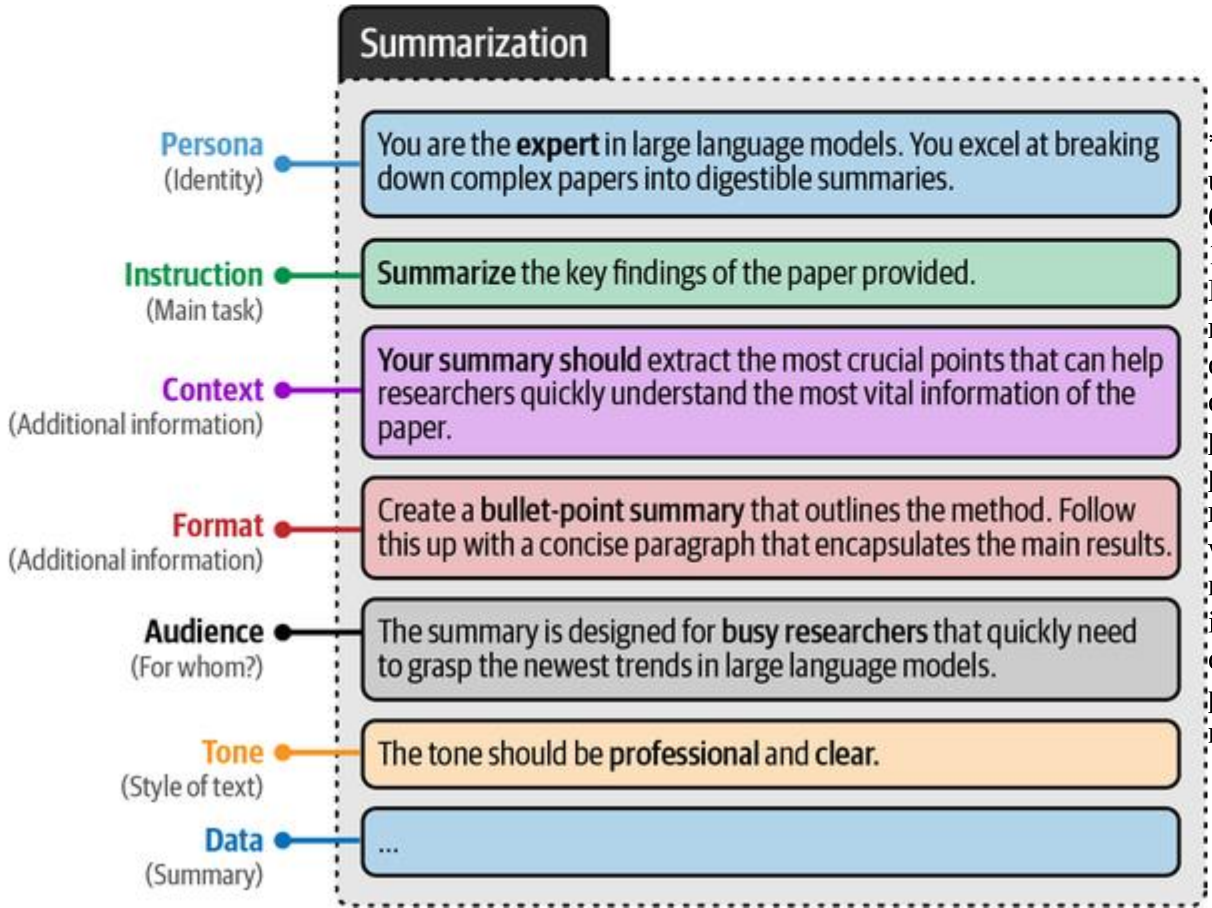
Named Entity Recognition:
```
Definition: an entity is an organization (org), a person (per), or a
location (loc). If it does not fit with the above, use (misc).
Extract entities from the following text: [text]
```


Summarization
Instruction
Summarize the following text:
Data
...
...
Explain the above in two to three sentences
Classification
Is the following text neutral, negative, or positive?
Output indicators
Text:
...
Sentiment:
Classify the text into neutral, negative, or positive
...
Named entity recognition
Define entities
Definition: an entity is an organization (org), a person (per), or a location (loc). If it does not fit with the above, use (misc).
Exclusion criteria
Verbs and adjectives are not entities.
Extract entities from the following text:
...

```

*Figure 6-10: Prompt examples for different use cases*

### Best Practices

1. **Specificity:** Precisely describe what you want
   - Instead of: "Write a description for a product"
   - Use: "Write a description for a product in less than two sentences and use a formal tone"

2. **Hallucination Mitigation:** Ask the model to respond with "I don't know" when uncertain

3. **Order Effects:** Place important instructions at the beginning (primacy effect) or end (recency effect)

## 6. Advanced Prompt Engineering

### Prompt Components

Complex prompts can include multiple components:
```

- **Persona:** "You are an expert in astrophysics"
- **Instruction:** The specific task to complete
- **Context:** Additional information about the problem
- **Format:** Required output structure
- **Audience:** Target audience for the output
- **Tone:** Desired style of the output
- **Data:** The actual input to process

Example Complex Prompt:
```
Persona: You are an expert in large language models. You excel at
breaking down complex papers into digestible summaries.

Instruction: Summarize the key findings of the paper provided.

Context: Your summary should extract the most crucial points that
can help researchers quickly understand the most vital information
of the paper.

Format: Create a bullet-point summary that outlines the method.
Follow this with a concise paragraph that encapsulates the main results.

Audience: The summary is designed for busy researchers that quickly
need to grasp the newest trends in large language models.

Tone: The tone should be professional and clear.

Data: [Text to summarize]
```



You are the **expert** in large language models. You excel at breaking down complex papers into digestible summaries.

Summarize the key findings of the paper provided.

Your summary should extract the most crucial points that can help researchers quickly understand the most vital information of the paper.

Create a **bullet-point summary** that outlines the method. Follow this up with a concise paragraph that encapsulates the main results.

The summary is designed for **busy researchers** that quickly need to grasp the newest trends in large language models.

The tone should be **professional** and **clear.**

...

Figure 6-11: Example of a complex prompt with multiple components

### Iterative Prompt Development

```
Iteration 1: Instruction + Data
Iteration 2: Persona + Instruction + Data
Iteration 3: Context + Tone + Instruction + Data
...
Iteration n: All relevant components
```

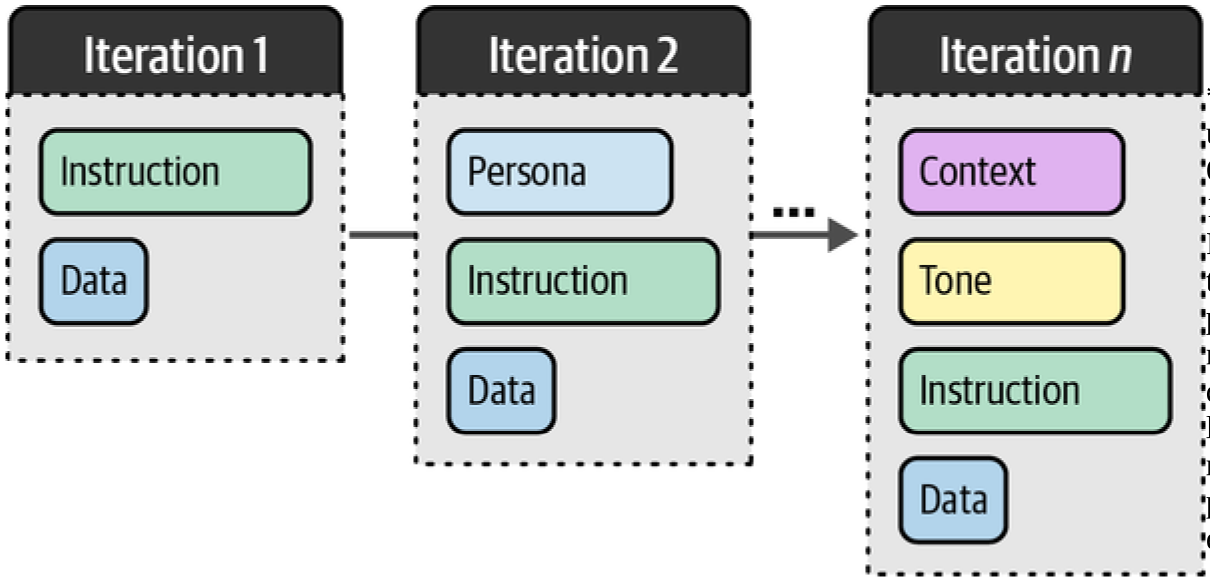


Figure 6-12: Iterative prompt development process

Implementation Example:
```python

# Prompt components

persona = "You are an expert in Large Language models..."
instruction = "Summarize the key findings of the paper provided.\n"
context = "Your summary should extract the most crucial points...\n"
data_format = "Create a bullet-point summary that outlines the method...\n"
audience = "The summary is designed for busy researchers...\n"
tone = "The tone should be professional and clear.\n"
text = "MY TEXT TO SUMMARIZE"
data = f"Text to summarize: {text}"

# Build prompt iteratively
query = persona + instruction + context + data_format + audience + tone + data
```

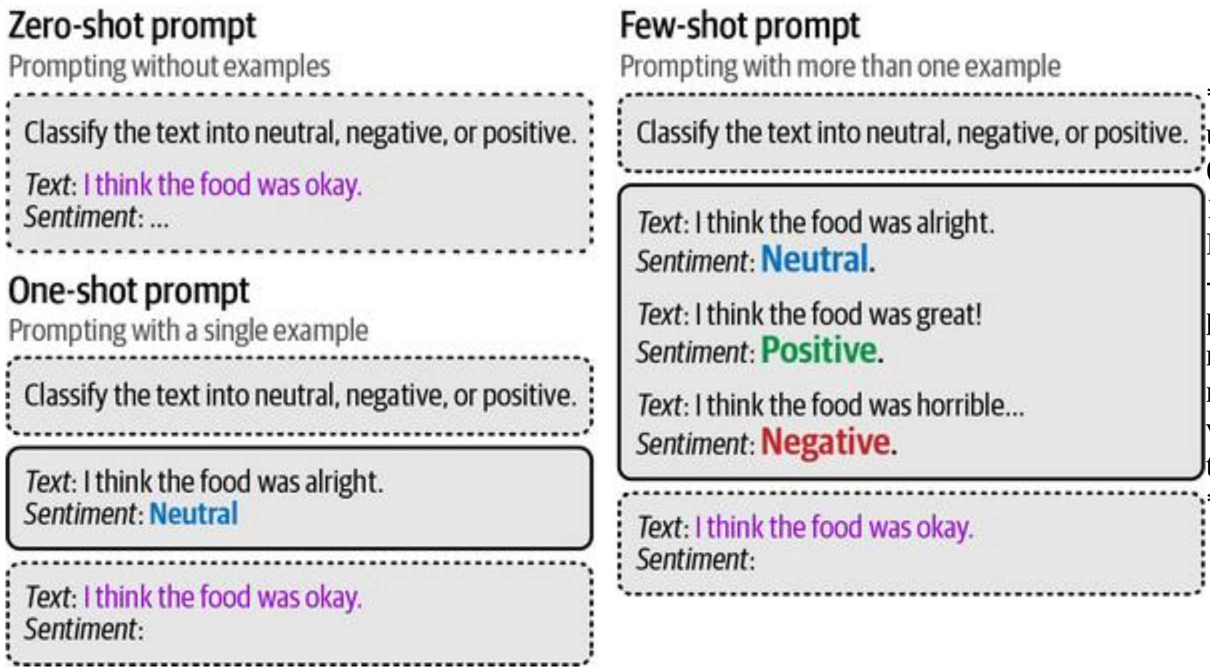
## 7. In-Context Learning

### Few-Shot Prompting

Zero-Shot: No examples provided
```
Classify the text into neutral, negative, or positive.
Text: I think the food was okay.
Sentiment:
```

One-Shot: Single example provided
```
Classify the text into neutral, negative, or positive.

Text: I think the food was alright.
Sentiment: Neutral

Text: I think the food was okay.
Sentiment:
```

Few-Shot: Multiple examples provided
```
Classify the text into neutral, negative, or positive.

Text: I think the food was alright.
Sentiment: Neutral.

Text: I think the food was great!
Sentiment: Positive.

Text: I think the food was horrible...
Sentiment: Negative.

Text: I think the food was okay.
Sentiment:
```



Zero-shot prompt
Prompting without examples

Classify the text into neutral, negative, or positive.
Text: I think the food was okay.
Sentiment: ...

One-shot prompt
Prompting with a single example

Classify the text into neutral, negative, or positive.

Text: I think the food was alright.
Sentiment: Neutral

Text: I think the food was okay.
Sentiment:

Few-shot prompt
Prompting with more than one example

Classify the text into neutral, negative, or positive.

Text: I think the food was alright.
Sentiment: Neutral.
Text: I think the food was great!
Sentiment: Positive.
Text: I think the food was horrible...
Sentiment: Negative.

Text: I think the food was okay.
Sentiment:

Figure 6-13: Few-shot prompting variations

### Example: Using Made-up Words

```python
one_shot_prompt = [
    {
        "role": "user",
        "content": "A 'Gigamuru' is a type of Japanese musical instrument. An example of a sentence that uses the word Gigamuru is:"
    },
    {
        "role": "assistant",
        "content": "I have a Gigamuru that my uncle gave me as a gift. I love to play it at home."
    },
    {
        "role": "user",
        "content": "To 'screeg' something is to swing a sword at it. An example of a sentence that uses the word screeg is:"
    }
]

outputs = pipe(one_shot_prompt)
print(outputs[0]["generated_text"])
# Output: During the intense duel, the knight skillfully screeged
# his opponent's shield, forcing him to defend himself.
```

## 8. Chain Prompting

Complex tasks can be broken down into sequential prompts:

```

1. Generate product name → 2. Generate slogan → 3. Generate sales pitch
```

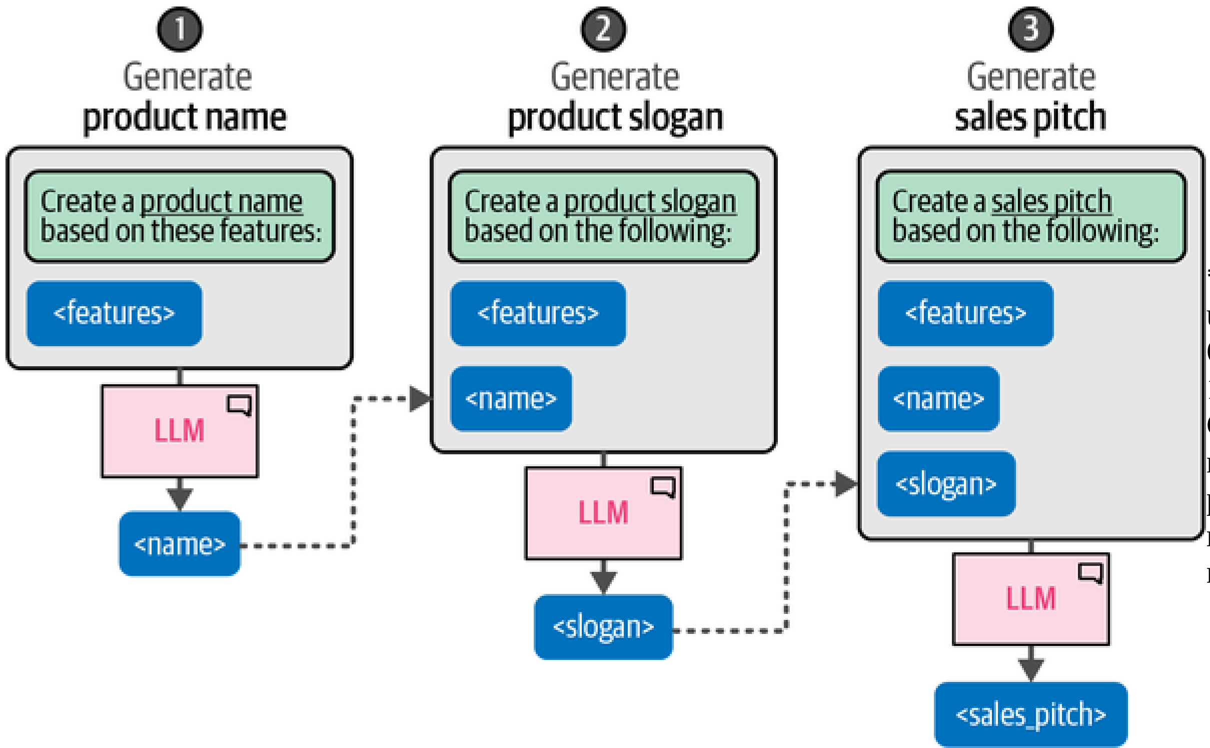


*Figure 6-14: Chain prompting

workflow*

Implementation Example:

```python
# Step 1: Generate product name and slogan
product_prompt = [
    {"role": "user", "content": "Create a name and slogan for a chatbot that leverages LLMs."}
]
outputs = pipe(product_prompt)
product_description = outputs[0]['generated_text']
# Output: Name: 'MindWeld Messenger'
#         Slogan: 'Unleashing Intelligent Conversations, One Response at a Time'

# Step 2: Generate sales pitch
sales_prompt = [
    {"role": "user", "content": f"Generate a very short sales pitch for the following product:
'{product_description}'"}
]
outputs = pipe(sales_prompt)
sales_pitch = outputs[0]['generated_text']
```

Applications of Chain Prompting:
- Response validation
- Parallel prompt processing
- Story writing
- Complex task decomposition

## 9. Reasoning with Generative Models

### System 1 vs. System 2 Thinking

- **System 1:** Automatic, intuitive, fast (default LLM behavior)
- **System 2:** Conscious, slow, logical (what we want to emulate)

### Chain-of-Thought (CoT) Prompting

Standard Prompting:
```
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls.
Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.
```

Chain-of-Thought Prompting:
```
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls.
Each can has 3 tennis balls. How many tennis balls does he have now?
```

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls.
5 + 6 = 11. The answer is 11.

Figure 6-15: Standard vs. chain-of-thought prompting

Implementation:
```python
cot_prompt = [
    {"role": "user", "content": "Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?"},
    {"role": "assistant", "content": "Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11."},
    {"role": "user", "content": "The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?"}
]

outputs = pipe(cot_prompt)
print(outputs[0]["generated_text"])
# Output: The cafeteria started with 23 apples. They used 20 apples,
# so they had 23 - 20 = 3 apples left. Then they bought 6 more apples,
# so they now have 3 + 6 = 9 apples. The answer is 9.
```

### Zero-Shot Chain-of-Thought

```python
zeroshot_cot_prompt = [
    {"role": "user", "content": "The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have? Let's think step-by-step."}
]

outputs = pipe(zeroshot_cot_prompt)
print(outputs[0]['generated_text'])
# Output: Step 1: Start with the initial number of apples, which is 23.
#         Step 2: Subtract the number of apples used to make lunch, which is 20.
#                 So, 23 - 20 = 3 apples remaining.
#         Step 3: Add the number of apples bought, which is 6. So, 3 + 6 = 9 apples.
#         The cafeteria now has 9 apples.
```

Figure 6-16: Zero-shot chain-of-thought prompting

Alternative CoT Triggers:
- "Take a deep breath and think step-by-step"
- "Let's work through this problem step-by-step"
- "Reason through this carefully before answering"

### Self-Consistency

Generate multiple responses and take the majority vote:

```
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls.
Each can has 3 tennis balls. How many tennis balls does he have now?

Let's think step-by-step.

→ Response 1: If he buys 2 cans of 3 tennis balls, then he has 2×3=6 tennis balls.
            A: The answer is 6.

→ Response 2: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls.
            5 + 6 = 11. A: The answer is 11.

→ Response 3: 2 cans of each 3 tennis balls totals 6 tennis balls. Add 5 on top.
            5 + 6 = 11. A: The answer is 11.

Majority vote: The answer is 11.
```

Figure 6-17: Self-consistency through majority voting

Advantage: Improved accuracy
Disadvantage: n times slower where n is number of samples

### Tree-of-Thought (ToT)

Explores multiple reasoning paths with intermediate evaluation:

```
        Problem
           │
   ┌───────────┴───────────┐
 Path A     Path B
   │          │
Evaluate   Evaluate
   │          │
  Keep      Prune
   │
Continue
```

Figure 6-18: Tree-of-thought exploration process

Zero-Shot Tree-of-Thought Implementation:
```python
zeroshot_tot_prompt = [
    {"role": "user", "content": "Imagine three different experts are answering this question. All experts will write down 1 step of their thinking, then share it with the group. Then all experts will go on to the next step, etc. If any expert realizes they're wrong at any point then they leave. The question is 'The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?' Make sure to discuss the results."}
]

outputs = pipe(zeroshot_tot_prompt)
print(outputs[0]['generated_text'])
# Output: Expert 1: Step 1 - Start with the initial number of apples: 23 apples.
#         Expert 2: Step 1 - Subtract the apples used for lunch: 23 - 20 = 3 apples remaining.
#         Expert 3: Step 1 - Add the newly bought apples: 3 + 6 = 9 apples.
#         Expert 1: Step 2 - Confirm the final count: The cafeteria has 9 apples.
#         Expert 2: Step 2 - Review the calculations: 23 - 20 = 3, then 3 + 6 = 9.
#                   The calculations are correct.
#         Expert 3: Step 2 - Agree with the result: The cafeteria indeed has 9 apples.
#         All experts agree on the final count: The cafeteria has 9 apples.
```

## 10. Output Verification

### Why Verify Output?

1. **Structured Output:** Ensure output follows required format (JSON, XML, etc.)
2. **Valid Output:** Prevent model from generating unauthorized responses
3. **Ethics:** Filter profanity, PII, bias, cultural stereotypes
4. **Accuracy:** Check for factual correctness and coherence

### Methods for Output Control

1. **Examples:** Provide few-shot examples of desired output
2. **Grammar:** Constrain token selection during generation
3. **Fine-tuning:** Train model on expected output formats (covered in Chapter 12)

### Providing Examples for Structured Output

Without Examples:
```python
zeroshot_prompt = [
    {"role": "user", "content": "Create a character profile for an RPG game in JSON format."}
]

outputs = pipe(zeroshot_prompt)
# Output might be incomplete or malformed JSON
```

With Examples:
```python
```

```
one_shot_template = """Create a short character profile for an RPG game. Make sure to only use this
format:

{
   "description": "A SHORT DESCRIPTION",
   "name": "THE CHARACTER'S NAME",
   "armor": "ONE PIECE OF ARMOR",
   "weapon": "ONE OR MORE WEAPONS"
}
"""

one_shot_prompt = [{"role": "user", "content": one_shot_template}]

outputs = pipe(one_shot_prompt)
print(outputs[0]["generated_text"])
# Output: {
#     "description": "A cunning rogue with a mysterious past, skilled in stealth and deception.",
#     "name": "Lysandra Shadowstep",
#     "armor": "Leather Cloak of the Night",
#     "weapon": "Dagger of Whispers, Throwing Knives"
# }
```

### Grammar-Constrained Sampling

Constrained Token Selection:
```
Q: Classify this sentence into positive, neutral, or negative:
"What a great movie!"

A:

Constrained sampling: Only allow "positive", "neutral", or "negative"
```

Figure 6-21: Constrained token selection for classification

Implementation with llama-cpp-python:
```python
from llama_cpp.llama import Llama
import json

# Load model
llm = Llama.from_pretrained(
   repo_id="microsoft/Phi-3-mini-4k-instruct-gguf",
   filename="*fp16.gguf",
   n_gpu_layers=-1,
   n_ctx=2048,
   verbose=False
)

# Generate with JSON grammar constraint
output = llm.create_chat_completion(
    messages=[
        {"role": "user", "content": "Create a warrior for an RPG in JSON format."},
    ],
    response_format={"type": "json_object"},
    temperature=0,
)['choices'][0]['message']['content']

# Parse and format JSON
json_output = json.dumps(json.loads(output), indent=4)
print(json_output)
```

Output Validation Process:
```
Prompt → LLM → Output → Validation LLM → Validated Output
```

Figure 6-19: Iterative output validation process

Structured Generation:
```
Expected format:
{
    "id": "{id}",
    "height": "{height}",
    "name": "{name}",
    "age": "{age}",
}

LLM fills in blanks:
{
    "id": "0",
    "height": "1.81 cm",
    "name": "Vincent",
    "age": "34",
}
```

Figure 6-20: Structured generation with known format

## 11. Mathematical Foundations

### Probability Distributions in LLMs

LLMs generate tokens based on probability distributions:

Token Probability Calculation:
\[P(w_t | w_{1:t-1}) = \frac{\exp(z_t / \tau)}{\sum_{j=1}^V \exp(z_j / \tau)} \]

Where:
- \(w_t \) is the next token
- \(w_{1:t-1} \) are the previous tokens
- \(z_t \) is the logit for token t
- \(\tau \) is the temperature parameter
- V is the vocabulary size

Temperature Effect:
\[P_{\tau}(w_t | w_{1:t-1}) = \frac{\exp(z_t / \tau)}{\sum_{j=1}^V \exp(z_j / \tau)} \]

- \(\tau \to 0 \): Deterministic selection (argmax)
- \(\tau = 1 \): Original distribution
- \(\tau > 1 \): Flatter distribution (more random)

Top-p Sampling:
\[\text{Select the smallest set } S \text{ such that } \sum_{w \in S} P(w) \geq p \]
Then sample from distribution renormalized over S.

## 12. Summary and Key Takeaways

### Prompt Engineering Principles

1. **Specificity is crucial:** Precise instructions yield better results
2. **Structure matters:** Use appropriate components (persona, instruction, context, etc.)
3. **Examples help:** Few-shot learning improves output quality and format adherence
4. **Iterate continuously:** Prompt development is an experimental process

### Advanced Techniques

1. **Chain-of-Thought:** Enables complex reasoning through step-by-step thinking
2. **Self-Consistency:** Improves accuracy through majority voting
3. **Tree-of-Thought:** Explores multiple reasoning paths
4. **Constrained Generation:** Ensures output follows required formats

### Output Verification

1. **Validation is essential** for production systems
2. **Multiple approaches** exist: examples, grammar constraints, fine-tuning
3. **Structured output** enables integration with other systems

### Future Directions

Prompt engineering continues to evolve with new techniques emerging regularly. The next chapter will explore how LLMs can use external memory and tools, building upon these prompt engineering foundations.

## References

1. Liu, N. F., et al. "Lost in the middle: How language models use long contexts." arXiv:2307.03172 (2023).
2. Li, C., et al. "EmotionPrompt: Leveraging psychology for large language models enhancement via emotional stimulus." arXiv:2307.11760 (2023).
3. Brown, T., et al. "Language models are few-shot learners." NeurIPS 33 (2020).
4. Wei, J., et al. "Chain-of-thought prompting elicits reasoning in large language models." NeurIPS 35 (2022).
5. Kojima, T., et al. "Large language models are zero-shot reasoners." NeurIPS 35 (2022).
6. Wang, X., et al. "Self-consistency improves chain of thought reasoning in language models." arXiv:2203.11171 (2022).
7. Yao, S., et al. "Tree of thoughts: Deliberate problem solving with large language models." arXiv:2305.10601 (2023).

These notes provide a comprehensive overview of prompt engineering techniques, from basic principles to advanced reasoning methods. The concepts and examples covered will help you effectively leverage LLMs for various applications while maintaining control over the generated output.