Prompt Engineering

Using Ollama & Open Source LLMs

Week 2

Lab 1 Retrospective!

Ollama is a free, open-source tool that simplifies running large language models (LLMs) on your local machine, allowing for off-line, private, and secure AI applications without needing cloud services. It handles model management, provides a command-line interface and API for interacting with models, and supports various open-source LLMs across macOS, Windows, and Linux. By enabling local execution, Ollama gives users full control over their data and empowers developers to build local chatbots, experiment with different models, and integrate AI into their own projects.

Get up and running with large language models, locally.

What is a Prompt?

A **prompt** is the input text you provide to a language model to generate a specific output.

- Can include instructions, questions, or context
- The model predicts what should come next
- Quality of prompt = Quality of output

Why Prompt Engineering?

You don't need to be a data scientist to write prompts, but crafting **effective prompts** requires skill.

Factors that affect prompts:

- Word choice, tone and style
- Structure and context

How LLMs Work

LLMs are **prediction engines**:

1. Take sequential text as input
2. Predict the next token
3. Add predicted token to sequence
4. Repeat

Your prompt sets up the LLM to predict the right sequence.

LLM Configuration: Output Length

Control response length with parameters:

```
response = requests.post('http://localhost:11434/api/generate',
    json={
        "model": "llama3.2",
        "prompt": "Explain quantum physics",
        "options": {
            "num_predict": 100  # Max tokens
        }
    })
```

Temperature Control

Temperature controls randomness (0.0 - 2.0):

- **Low (0.0-0.3):** Deterministic, focused
- **Medium (0.5-0.8):** Balanced creativity
- **High (0.9-2.0):** Very creative, random

```
"options": {
    "temperature": 0.1  # Deterministic
}
```

Temperature Examples

Temperature 0.0: Always picks most likely token

Temperature 1.0: Balances probability and randomness

Temperature 2.0: Nearly random selection

Higher temperature = More creative but less predictable

Top-K Sampling

Limits selection to top K most likely tokens:

```
"options": {
    "top_k": 40  # Consider top 40 tokens
}
```

- Lower top-K = More focused
- Higher top-K = More variety
- top-K of 1 = greedy decoding

Top-P (Nucleus Sampling)

Selects from tokens whose cumulative probability ≤ P:

```
"options": {
    "top_p": 0.9  # Top 90% probability mass
}
```

- 0.0 = greedy (most probable only)
- 1.0 = all tokens considered

Combining Parameters

Ollama applies settings in this order:

1. Top-K filters to K most likely
2. Top-P filters by cumulative probability
3. Temperature applied to remaining tokens

```
"options": {
    "temperature": 0.2,
    "top_k": 30,
    "top_p": 0.95
}
```

Recommended Starting Settings

For factual tasks:

```
{"temperature": 0.1, "top_k": 20, "top_p": 0.5}
```

For creative tasks:

```
{"temperature": 0.9, "top_k": 40, "top_p": 0.99}
```

For single correct answers:

```
{"temperature": 0.0}
```

Parameter	Controls...	Effect
Temperature	Randomness / creativity	Higher = more diverse/creative, lower = more deterministic/reliable
Top-p (nucleus sampling)	Probability mass of token choices	Smaller = only most likely tokens, Larger = more variety
Top-k	Fixed number of token choices	Lower = more focused, Higher = more exploratory

> **Temperature** and **top-p/k** are often used together. You usually *don’t* tune both top-p and top-k simultaneously — you pick one.

Parameters like **temperature**, **top-p**, and **top-k** control *how* a language model generates responses, and you can intentionally tune them to match the needs of **different phases of the software development lifecycle (SDLC)** — from brainstorming and design, to implementation, testing, and deployment.

How to Use Them Across the SDLC

Here’s how you can think about tuning these parameters in each major phase of software engineering:

1. Requirements Gathering & Ideation

Goal: Explore many possibilities and gather creative solutions.

- **Use:**
  - `temperature`: **0.8–1.0** (encourages creative brainstorming)
  - `top-p`: **0.9–1.0** (allow broad exploration of plausible tokens)
  - `top-k`: **50–100** (if using top-k, allow wide vocabulary)

Examples:

- Generate alternative user stories or acceptance criteria.
- Explore different design patterns or architectural options.
- Brainstorm features based on high-level goals.

2. System Design & Architecture

Goal: Explore options but still maintain technical soundness.

- **Use:**
    - `temperature`: **0.5–0.7** – balance between creativity and precision
    - `top-p`: **0.8–0.9** – keep responses technically plausible
    - `top-k`: **20–50** – moderate diversity

Examples:

- Suggest alternative API designs or data models.
- Draft possible microservice boundaries or deployment topologies.
- Explore pros/cons of design decisions.

3. Implementation (Coding)

Goal: Precise, deterministic code generation with minimal bugs.

- **Use:**
  - `temperature`: **0.0–0.3** – prioritize correctness and consistency
  - `top-p`: **0.5–0.8** – limit choices to most likely correct code
  - `top-k`: **5–20** – narrow vocabulary for reliable code generation

Examples:

- Autocomplete boilerplate code.
- Generate unit test scaffolding.
- Write database migration scripts.

> **Why lower temperature matters:** With `temperature=0`, the model becomes *deterministic* — it will always produce the same output for the same prompt, ideal for repeatable code generation.

4. Testing & QA

Goal: Explore edge cases and test scenarios creatively.

- **Use:**
    - `temperature` : **0.7–1.0** – encourage variety in test ideas
    - `top-p` : **0.9–1.0** – wide range of possible test inputs
    - `top-k` : **50+** – many test case suggestions

Examples:

- Generate boundary and negative test cases.
- Suggest potential failure modes.
- Propose fuzz test inputs or integration test scenarios.

> Use a higher temperature in test generation, then lower it again when generating actual test code for reliability.

5. Deployment & Maintenance

Goal: Reliable, repeatable automation scripts, docs, and changelogs.

- **Use:**
    - `temperature` : **0.0–0.4** – deterministic and reproducible
    - `top-p` : **0.5–0.8** – reduce chance of hallucinations
    - `top-k` : **5–20**

Examples:

- Generate CI/CD pipeline scripts.
- Automate release notes or deployment documentation.
- Create infrastructure-as-code templates.

Tuning LLM Parameters by SDLC Phase

SDLC Phase	Temperature	Top-p	Top-k	
Requirements & Ideation	0.8–1.0	0.9–1.0	50–100	Encourage creativity and exploration
Design & Architecture	0.5–0.7	0.8–0.9	20–50	Balance innovation with correctness
Implementation	0.0–0.3	0.5–0.8	5–20	Ensure reliable, deterministic code
Testing & QA	0.7–1.0	0.9–1.0	50+	Explore diverse test cases and edge conditions
Deployment & Maintenance	0.0–0.4	0.5–0.8	5–20	Consistent automation and documentation

Announcements

Course Website: https://github.com/abuach/cs450students

- Labs
- Slides (weekly)
- My Schedule

Announcements

Lab 1 Graded

Please upload your full `cs450` directory to Github.

HW1 graded

> **BTW: All large language models (LLMs) use temperature, top-p, and top-k**

LLMs like GPT-4, Claude, and Llama all have these parameters configurable, allowing users to control the creativity and randomness of the output by adjusting how the model selects the next word or token.

In General

Everything we learn about LLMs in this course will generalize to all large language models and their APIs, not just Ollama (and qwen2.5-coder).

Zero-Shot Prompting

Zero-shot = No examples provided

Simply describe the task:

```
prompt = """Classify movie reviews as POSITIVE, NEUTRAL or NEGATIVE.

Review: "Her" is a disturbing masterpiece.
Sentiment:"""
```

When Zero-Shot Fails

Zero-shot works for simple tasks, but struggles with:

- Complex reasoning
- Specific output formats
- Domain-specific knowledge
- Ambiguous instructions

Solution: Provide examples!

One-Shot Prompting

Provide **one example** to guide the model:

```
prompt = """Parse pizza orders to JSON:

EXAMPLE:
I want a small pizza with cheese and pepperoni.
{"size": "small", "ingredients": ["cheese", "pepperoni"]}

Now parse: I want a large pizza with mushrooms.
JSON:"""
```

Few-Shot Prompting

Provide **multiple examples** (typically 3-5):

```
prompt = """Parse pizza orders:

EXAMPLE 1: small cheese pizza
{"size": "small", "ingredients": ["cheese"]}

EXAMPLE 2: large pepperoni and mushroom
{"size": "large", "ingredients": ["pepperoni", "mushroom"]}

EXAMPLE 3: medium with olives and basil
{"size": "medium", "ingredients": ["olives", "basil"]}

Parse: extra large with everything
JSON:"""
```

Few-Shot Best Practices

- Use 3-6 examples minimum
- Make examples diverse
- Include edge cases
- Ensure high quality examples

Ollama API `role` Parameter Guide

In the **Ollama API** (and most modern LLM APIs), the `messages` you send are structured as a list of objects.
Each object has a `role` field that tells the model **who is “speaking.”**

The `role` Field – Meaning of Each Value

Role	Purpose	Typical Usage
system	Sets the **overall behavior, tone, or identity** of the model. It's like instructions to the assistant.	- Define the assistant's personality: `"You are a helpful tutor."` - Set constraints: `"Only answer in JSON."`
user	Represents input from the **end user** — usually the prompt or question you want answered.	- `"Explain quantum computing in simple terms."`
assistant	Represents the **model's previous responses.** Useful for maintaining conversation context.	- Stores previous replies so the model "remembers" what it said.

Example Conversation with Roles

```json
[
  {
    "role": "system",
    "content": "You are a concise technical assistant."
  },
  {
    "role": "user",
    "content": "Explain how blockchain works."
  },
  {
    "role": "assistant",
    "content": "Blockchain is a distributed ledger..."
  },
  {
    "role": "user",
    "content": "Can you give me an example?"
  }
]
```

- `system`: sets the model’s behavior.
- `user`: the actual prompts/questions.
- `assistant`: the model’s previous response (helps it stay consistent).

Lab 2!

Step-Back Prompting

Ask a **general question first**, then the specific one:

Step 1 - General:

```
What makes a good FPS game level?
```

Step 2 - Specific:

```
Using those principles, write a storyline
for an underwater research facility level.
```

Step-Back Benefits

- Activates relevant knowledge
- Reduces hallucinations
- Improves reasoning quality

Chain of Thought (CoT)

Prompt the model to show its **reasoning steps**:

```
prompt = """When I was 3, my partner was triple my age.
Now I am 20. How old is my partner?

Let's think step by step."""
```

CoT improves accuracy on complex reasoning tasks.

Why CoT Works

LLMs struggle with:

- Math problems
- Multi-step reasoning
- Logic puzzles

CoT forces the model to:

- Break down the problem
- Show intermediate steps
- Arrive at correct answer

Zero-Shot CoT

Simply add: **"Let's think step by step"**

```python
def zero_shot_cot(question, model="cs450"):
    prompt = f"{question}\n\nLet's think step by step."
    return call_ollama(prompt, model)
```

This simple phrase activates reasoning!

Few-Shot CoT

Provide examples **with reasoning**:

```
prompt = """Q: Brother was 2, I was double. Now I'm 40. Brother's age?
A: I was 2 × 2 = 4. Difference: 2 years. Now 40, so brother is 40 - 2 = 38.

Q: I was 3, partner was 3x my age. Now I'm 20. Partner's age?
A:"""
```

Shows the model HOW to reason.

When to Use CoT

Good for:

- Math problems
- Logic puzzles
- Multi-step reasoning
- Code generation with explanation
- Complex analysis

Not needed for:

- Simple questions
- Single-step tasks
- Creative writing

Self-Consistency

Generate **multiple reasoning paths** and pick most common answer:

```python
def self_consistency(prompt, model="cs450", n=5):
    answers = []
    for _ in range(n):
        response = call_ollama(prompt + "\n\nLet's think step by step.", model)
        # Extract final answer
        answer = extract_answer(response)
        answers.append(answer)

    # Return most common
    return max(set(answers), key=answers.count)
```

Self-Consistency Process

1. **Generate** multiple responses (high temperature)
2. **Extract** answer from each response
3. **Vote** - choose most frequent answer

Improves accuracy but increases cost!

Self-Consistency Example

Prompt sent 3 times may give:

```
Attempt 1: IMPORTANT ✓
Attempt 2: NOT IMPORTANT ✗
Attempt 3: IMPORTANT ✓

Final answer: IMPORTANT (majority vote)
```

Tree of Thoughts (ToT)

Extends CoT by exploring **multiple reasoning branches**:

```
         Question
         /  |  \
       /    |    \
   Path1  Path2  Path3
    / \     |      / \
   A   B    C     D   E
```

Evaluates multiple approaches simultaneously.

ToT vs CoT

Chain of Thought: Linear reasoning path

```
Q → Step1 → Step2 → Step3 → Answer
```

Tree of Thoughts: Branching exploration

```
Q → [Path1, Path2, Path3] → Evaluate → Best Answer
```

ToT Characteristics

- Explores multiple strategies
- Self-evaluates different paths
- Backtracks if needed
- More complex to implement
- Best for very complex problems

ReAct: Reason + Act

Combines **reasoning** with **actions** (tool use):

```
Thought: I need to find info about Metallica
Action: Search "Metallica band members"
Observation: James Hetfield, Lars Ulrich...
Thought: Now I need each member's children count
Action: Search "James Hetfield children"
...
```

ReAct Pattern

1. **Think** about what to do
2. **Act** - use a tool/API
3. **Observe** the result
4. Repeat until solved

This mimics human problem-solving!

ReAct Benefits

- Access to external information
- Can use calculators, databases, APIs
- More accurate than pure LLM reasoning
- Transparent decision-making process

This is the first step toward AI agents!

Automatic Prompt Engineering

Use an LLM to **generate prompts**:

```
meta_prompt = """Generate 10 different ways a customer might order:
"One Metallica t-shirt size S"

Keep the same meaning but vary the phrasing."""

variants = call_ollama(meta_prompt, "cs450")
```

Code Generation Best Practices

Be specific about:

- Language and version
- Input/output format
- Error handling requirements
- Coding style preferences

```
prompt = """Write a Python 3.10+ function using type hints
that validates email addresses using regex.
Include error handling and unit tests."""
```

Code Explanation

Ask LLM to explain code:

```
prompt = """Explain this Python code:

def fib(n):
    return n if n < 2 else fib(n-1) + fib(n-2)

Explain what it does and how it works."""
```

Helpful for understanding unfamiliar code!

Code Translation

Translate between languages:

```
prompt = """Translate this Bash script to Python:

#!/bin/bash
for file in *.txt; do
    mv "$file" "backup_$file"
done
"""

python_code = call_ollama(prompt, "cs450")
```

Code Debugging

LLMs can help find and fix bugs:

```
prompt = """This Python code has a bug:

def divide(a, b):
    return a / b

result = divide(10, 0)

Explain the bug and provide a fixed version."""
```

Code Review

Get suggestions for improvement:

```
prompt = """Review this code and suggest improvements:

def process_data(data):
    result = []
    for item in data:
        if item > 0:
            result.append(item * 2)
    return result

Focus on: performance, readability, Pythonic style."""
```

Best Practices

Best Practice: Action Verbs

Use clear action verbs:

- Analyze, Categorize, Classify
- Compare, Contrast, Create
- Define, Describe, Evaluate
- Extract, Generate, Identify
- List, Rank, Summarize
- Translate, Write

Best Practice: Be Specific

Vague prompts → Generic outputs

Vague:

```
Write about video games
```

Specific:

```
Write a 3-paragraph blog post about the top 5 retro game
consoles from the 1980s. Include release year and sales figures.
Use a conversational, nostalgic tone.
```

Best Practice: Instructions > Constraints

Don't say:

```
Write about games but don't mention violence or ratings
and don't list specific titles or...
```

Do say:

```
Write about game design principles focusing on mechanics,
storytelling, and player engagement.
```

Positive instructions work better!

Best Practice: Use Variables

Make prompts reusable:

```python
template = """You are a travel guide.
Tell me an interesting fact about {city}."""

cities = ["Amsterdam", "Tokyo", "Cairo"]

for city in cities:
    prompt = template.format(city=city)
    print(call_ollama(prompt))
```

Best Practice: Experiment with Formats

Try different input formats:

Question:

```
What makes the Dreamcast revolutionary?
```

Instruction:

```
Describe why the Sega Dreamcast was revolutionary.
```

Statement completion:

```
The Sega Dreamcast was revolutionary because...
```

Best Practice: Mix Up Examples

For classification, vary the order:

Bad:

```
Positive: "Great!"
Positive: "Amazing!"
Negative: "Terrible"
```

Good:

```
Positive: "Great!"
Negative: "Terrible"
Positive: "Amazing!"
Negative: "Awful"
```

> Lab 3!