Prompt Engineering Interactive Lab — Step-by-Step

---

0 — Goals

By the end of this lab you will be able to:

- Write precise prompts (role, few-shot, constraints).
- Force structured outputs (JSON) and validate them with Pydantic.
- Debug and iterate prompts.
- Use chain-of-thought and reasoning prompts safely.
- Use LangChain + Gemini agents to call local tools (file/directory creation) in a structured way.
- Build simple automated checks to validate prompt outputs.

---

1 — Prerequisites & Setup

1. Create a virtualenv and install packages:

```
python -m venv venv
source venv/bin/activate       # macOS/Linux
# venv\Scripts\activate        # Windows PowerShell
pip install --upgrade pip
pip install langchain-google-genai google-generativeai pydantic
python-dotenv
```

2. Put your API key in an environment variable (recommended):

```
export GOOGLE_API_KEY="your_gemini_api_key"    # macOS / Linux
setx GOOGLE_API_KEY "your_gemini_api_key"      # Windows (restart
shell)
```

Or use a `.env` file with python-dotenv in your scripts.

3. Minimal imports you'll use in exercises:

```
import os
from pydantic import BaseModel, Field, ValidationError
import json, re
# For LangChain:
from langchain_google_genai import ChatGoogleGenerativeAI
# For raw SDK (optional):
import google.generativeai as genai
```

---

2 — Lab 1 — Role Prompting (deterministic role + instructions)

Objective

Learn how explicit role & context steer outputs.

Task

Create a script `lab1_role.py`:

```
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash",
temperature=0.0)

prompt = """You are an expert HR interviewer.
Task: Given a candidate resume snippet, produce 3 targeted interview
questions (each one sentence).
Output: Plain text, 3 numbered lines only.
```

```
"""

text = "Candidate: 5 years backend experience in Python, worked on
payment microservices."

resp = llm.invoke({"input": prompt + "\n\n" + text})
print(resp.content)
```

Run & Expected

Run `python lab1_role.py`. You should get three concise numbered questions.
If you get extra explanation, tighten the prompt: add “Do not add explanations.”

---

3 — Lab 2 — Output Formatting + JSON + Pydantic validation

Objective

Force structured output and validate with Pydantic. Also learn to strip code fences.

Schema

```
from typing import Literal
class RouterSchema(BaseModel):
    selected_profile: Literal["hr","software engineer","product
manager"]
```

Task: file `lab2_json.py`

```
import os, json, re
from pydantic import BaseModel
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash",
temperature=0.0)

class RouterSchema(BaseModel):
```

```
    selected_profile: str  # we'll validate later with Literal or
manual check

prompt = """
You are a router. Allowed values: hr, software engineer, product
manager.
Task: Read the problem and output ONLY valid JSON like
{"selected_profile": "hr"}.
Do NOT include code fences or extra commentary.
Problem: I need to fix a bug that causes a null pointer error when
deserializing user objects.
"""

resp = llm.invoke({"input": prompt})
raw = resp.content.strip()

# remove code fences if present
if raw.startswith("```"):
    raw = re.sub(r"^```[a-zA-Z]*\n", "", raw)
    raw = raw.rstrip("`").strip()

try:
    parsed = json.loads(raw)
    # strict check
    if parsed.get("selected_profile") not in ["hr","software
engineer","product manager"]:
        raise ValueError("Invalid choice")
    print("SUCCESS:", parsed)
except Exception as e:
    print("PARSE FAIL:", raw)
    raise
```

Hints

- If Gemini returns code fences, the regex cleaning above handles it.
- If your output includes extra fields or text, instruct “Output ONLY valid JSON” and set `temperature=0`.

---

4 — Lab 3 — Few-Shot Prompting

Objective

Use examples to bias output format and style.

Example `lab3_fewshot.py`

```
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash",
temperature=0.0)

prompt = """
You are a classifier. Classify sentiment as positive/negative/neutral.

Example 1:
Text: "I love this product, works great!"
Label: positive

Example 2:
Text: "This is the worst purchase I made."
Label: negative

Now classify:
Text: "The app is okay, but crashes sometimes."
Label:
"""

resp = llm.invoke({"input": prompt})
print(resp.content)
```

Expected

Output should be `neutral` (or `negative` depending on wording). If wrong, add clarification or more examples.

---

5 — Lab 4 — Chain-of-Thought & Step-by-Step Reasoning

Objective

Get models to reveal reasoning (use carefully; for safety and hallucination concerns).

Task `lab4_cot.py`

```
from langchain_google_genai import ChatGoogleGenerativeAI
llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash",
temperature=0.2)

prompt = """
Solve the math problem step-by-step, then give the final answer.

Problem: If 12 people sit at a round table, how many unique seating
arrangements (up to rotation) exist?

Step-by-step:
"""

resp = llm.invoke({"input": prompt})
print(resp.content)
```

Note

If you require the LLM to *not* reveal chain-of-thought in public-facing apps, avoid requesting internal chains. For evaluation tasks, you can ask for "brief reasoning" or "high-level steps."

---

6 — Lab 5 — Tool Calling with LangChain Agent (create directories/files)

Objective

Let the agent decide to call a tool and run it. Use `STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION` agent type to avoid OpenAI function-call mismatch.

Files

- tools.py:

```
import os

def create_path(path: str, is_file: bool=False, content: str="") ->
str:
    path = path.strip()
    if is_file:
        os.makedirs(os.path.dirname(path) or ".", exist_ok=True)
        with open(path, "w", encoding="utf-8") as f:
            f.write(content)
        return f"File created: {path}"
    else:
        os.makedirs(path, exist_ok=True)
        return f"Directory created: {path}"
```

- agent_run.py:

```
import os
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.tools import tool
from langchain.agents import initialize_agent, AgentType
from tools import create_path

# set env var or rely on shell env
os.environ["GOOGLE_API_KEY"] = os.getenv("GOOGLE_API_KEY")

@tool
def create_path_tool(path: str, is_file: bool = False, content: str =
"") -> str:
    return create_path(path, is_file, content)

llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash",
temperature=0.0)
```

```
agent = initialize_agent(
    tools=[create_path_tool],
    llm=llm,
    agent=AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True,
)

prompt = """
Create a directory 'project' with 'src' and 'tests'. In 'src', make
'main.py' with content `print("Hello")`.
In 'tests', make 'test_main.py' with content `assert True`.
"""
agent.run(prompt)
```

Run

`python agent_run.py` — agent should call your tool and create files. Check local filesystem.

Safety

Restrict tool to a sandbox directory to prevent accidental writes outside project folder. Example: prefix paths with `./sandbox/`.

---

7 — Lab 6 — Debugging Prompts & Iteration

Process (repeatable)

1. Run prompt, inspect output.
2. If malformed:
    - Lower temperature (0–0.3) for deterministic outputs.
    - Add explicit format instructions (“JSON only”, “No explanation”, “No code fences”).
    - Provide examples (few-shot).

3. Add tests that validate outputs (see next section).

---

8 — Lab 7 — Build an Automated Checker (interactive)

Create `checker.py`:

```
import json, re
from pydantic import BaseModel, ValidationError
from typing import Literal

class ProfileRouter(BaseModel):
    selected_profile: Literal["hr","software engineer","product
manager"]

def clean_raw(raw: str) -> str:
    raw = raw.strip()
    if raw.startswith("```"):
        raw = re.sub(r"^```[a-zA-Z]*\n", "", raw)
        raw = raw.rstrip("`").strip()
    return raw

def validate_output(raw: str) -> bool:
    raw_clean = clean_raw(raw)
    try:
        parsed = json.loads(raw_clean)
    except json.JSONDecodeError:
        return False
    try:
        ProfileRouter(**parsed)
        return True
    except ValidationError:
        return False

# Example usage
raw_ok = '{"selected_profile":"software engineer"}'
raw_bad = '```json\n{"selected_profile": "doctor"}\n```'
```

```
print(validate_output(raw_ok), validate_output(raw_bad))  # True False
```

Use this in your prompts pipeline: if `validate_output` is False, re-run prompt with clarifications (auto-retry with an improved instruction).

---

9 — Exercises (interactive)

For each exercise, implement the prompt/code, run it, and use the validator where applicable.

1. Role Prompt (lab1): change role to "senior engineering manager" and produce 5 behavioral interview questions. Ensure numbered output only.
2. JSON Router (lab2): get the model to output `{"selected_profile": "<one-of-3>"}` for each of the following problems:
    - “The login page returns 500 for some users.” (expect software engineer)
    - “We need a hiring plan for new interns” (expect hr)
    - “We need to prioritize product roadmap 2026” (expect product manager)
3. Use `checker.validate_output` to assert correctness.
4. Few-shot classifier (lab3): supply 3 examples and classify new sentence.
5. Agent task (lab5): ask the agent to create nested directories and 2 files; ensure files exist and contain expected text.

Hints:

- For misformatted JSON, add: `If you cannot respond in valid JSON, return {"error":"unable_to_comply"}` so you can detect failures.
- Keep `temperature=0.0` for structured deterministic outputs.

---

10 — Solutions / Example Prompts (quick reference)

Role prompt:

```
You are a senior engineering manager. Produce 5 behavioral interview questions that assess leadership and technical judgment. Output exactly 5 numbered lines and nothing else.
```

JSON router prompt:

```
You are a router. Allowed outputs: hr, software engineer, product manager.
Task: Read the problem and return ONLY a JSON object: {"selected_profile":"<choice>"}.
No text, no code fences, no explanation.
Problem: {problem_here}
```

Few-shot example format:

```
Example:
Text: "I love the app's speed"
Label: positive
... (two more examples)
Now label:
Text: "{new_text}"
Label:
```

Agent tool description (LangChain uses this to build prompts automatically):

- name: create_path_tool
- description: "Creates a directory or file on local disk. Args: path (string), is_file (bool), content (string)."

---

11 — Rubric & What to Watch For

- Deterministic structured outputs: pass if JSON parses and passes Pydantic.

- Hallucination: watch for invented facts; ask for citations or say “I don’t know”.
- Overly verbose outputs: enforce “Output ONLY …” constraints.
- Code fences: handle with regex or instruct “no markdown or code fences”.

---

12 — Next Steps & Advanced Topics

- Prompt templates versioning (git prompts).
- Automatic prompt tuning (A/B prompts + scoring).
- Safety controls: filter unsafe instructions and sandbox tools.
- Human-in-the-loop: require confirmation before running destructive tools.