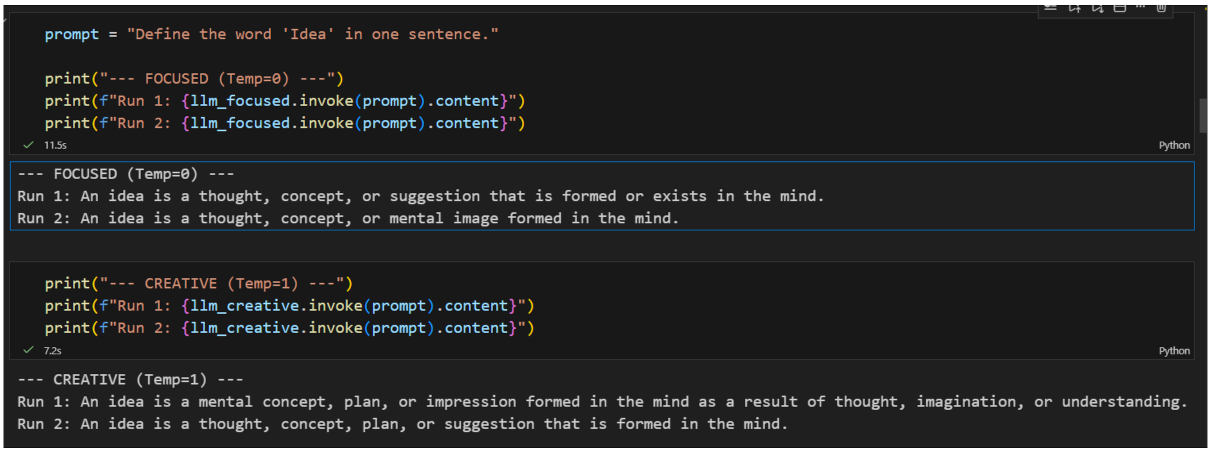
GenAI Unit 2 Lab

Name	Dhanya Prabhu
SRN	PES2UG23CS169
Section	C

Part 1a:



```
prompt = "Define the word 'Idea' in one sentence."

print("--- FOCUSED (Temp=0) ---")
print(f"Run 1: {llm_focused.invoke(prompt).content}")
print(f"Run 2: {llm_focused.invoke(prompt).content}")
```

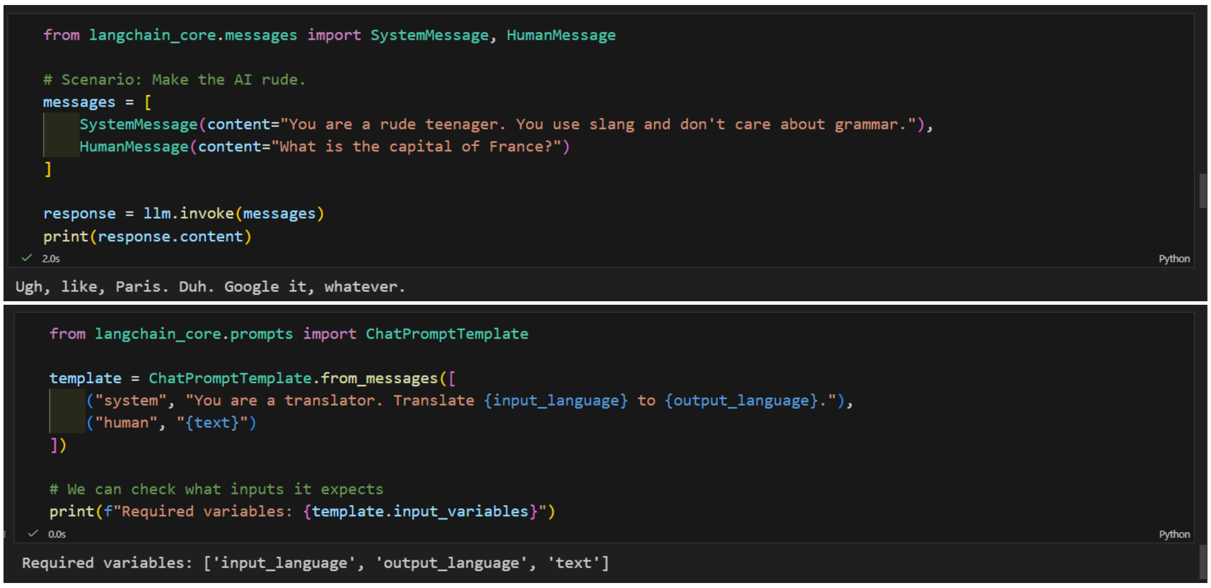
```
--- FOCUSED (Temp=0) ---
Run 1: An idea is a thought, concept, or suggestion that is formed or exists in the mind.
Run 2: An idea is a thought, concept, or mental image formed in the mind.
```

```
print("--- CREATIVE (Temp=1) ---")
print(f"Run 1: {llm_creative.invoke(prompt).content}")
print(f"Run 2: {llm_creative.invoke(prompt).content}")
```

```
--- CREATIVE (Temp=1) ---
Run 1: An idea is a mental concept, plan, or impression formed in the mind as a result of thought, imagination, or understanding.
Run 2: An idea is a thought, concept, plan, or suggestion that is formed in the mind.
```

Lower temperature gives deterministic output. Higher temperature gives more varied and creative responses.

Part 1b:

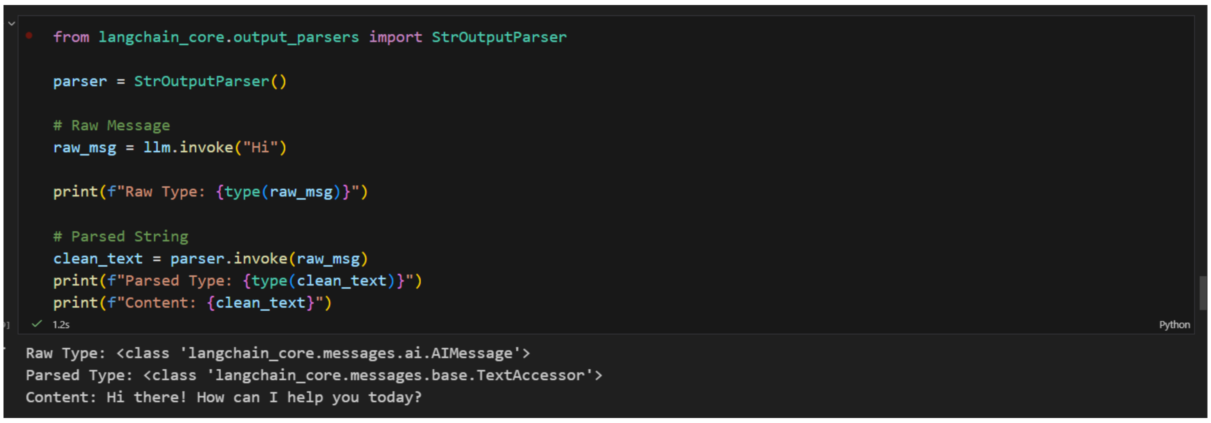


```
from langchain_core.messages import SystemMessage, HumanMessage

# Scenario: Make the AI rude.
messages = [
    SystemMessage(content="You are a rude teenager. You use slang and don't care about grammar."),
    HumanMessage(content="What is the capital of France?")
]

response = llm.invoke(messages)
print(response.content)
```

```
Ugh, like, Paris. Duh. Google it, whatever.
```

```
from langchain_core.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages([
    ("system", "You are a translator. Translate {input_language} to {output_language}."),
    ("human", "{text}")
])

# We can check what inputs it expects
print(f"Required variables: {template.input_variables}")
```

```
Required variables: ['input_language', 'output_language', 'text']
```

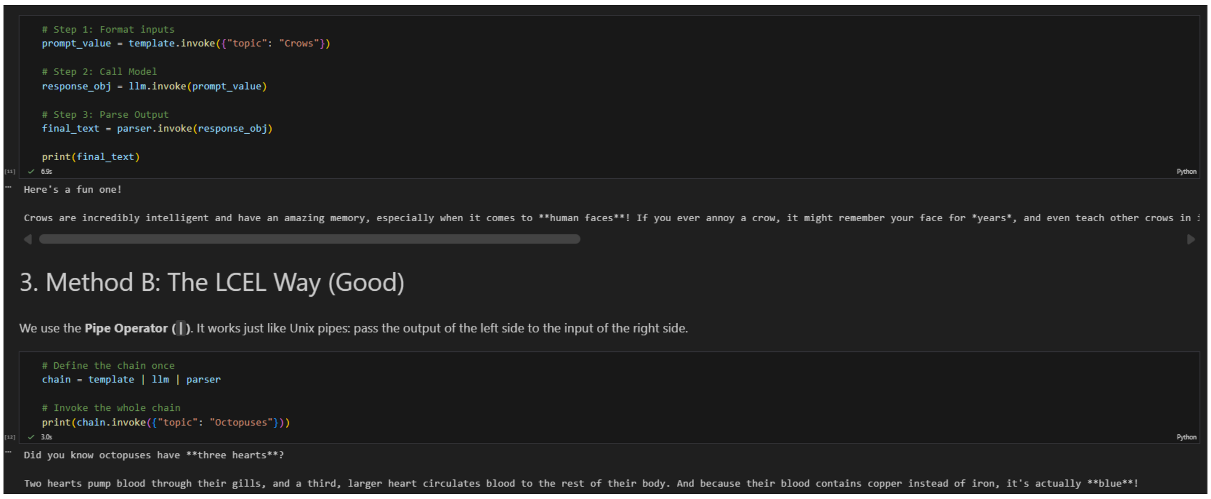


```python
from langchain_core.output_parsers import StrOutputParser

parser = StrOutputParser()

# Raw Message
raw_msg = llm.invoke("Hi")

print(f"Raw Type: {type(raw_msg)}")

# Parsed String
clean_text = parser.invoke(raw_msg)
print(f"Parsed Type: {type(clean_text)}")
print(f"Content: {clean_text}")
```

```
Raw Type: <class 'langchain_core.messages.ai.AIMessage'>
Parsed Type: <class 'langchain_core.messages.base.TextAccessor'>
Content: Hi there! How can I help you today?
```

Using System and Human messages changes the tone and behavior of the model. ChatPromptTemplate helps structure inputs clearly and makes prompts reusable. Structured prompting improves control over output style.

Part 1c:



```python
# Step 1: Format inputs
prompt_value = template.invoke({"topic": "Crows"})

# Step 2: Call Model
response_obj = llm.invoke(prompt_value)

# Step 3: Parse Output
final_text = parser.invoke(response_obj)

print(final_text)
```

```
Here's a fun one!

Crows are incredibly intelligent and have an amazing memory, especially when it comes to **human faces**! If you ever annoy a crow, it might remember your face for *years*, and even teach other crows in i
```

3. Method B: The LCEL Way (Good)

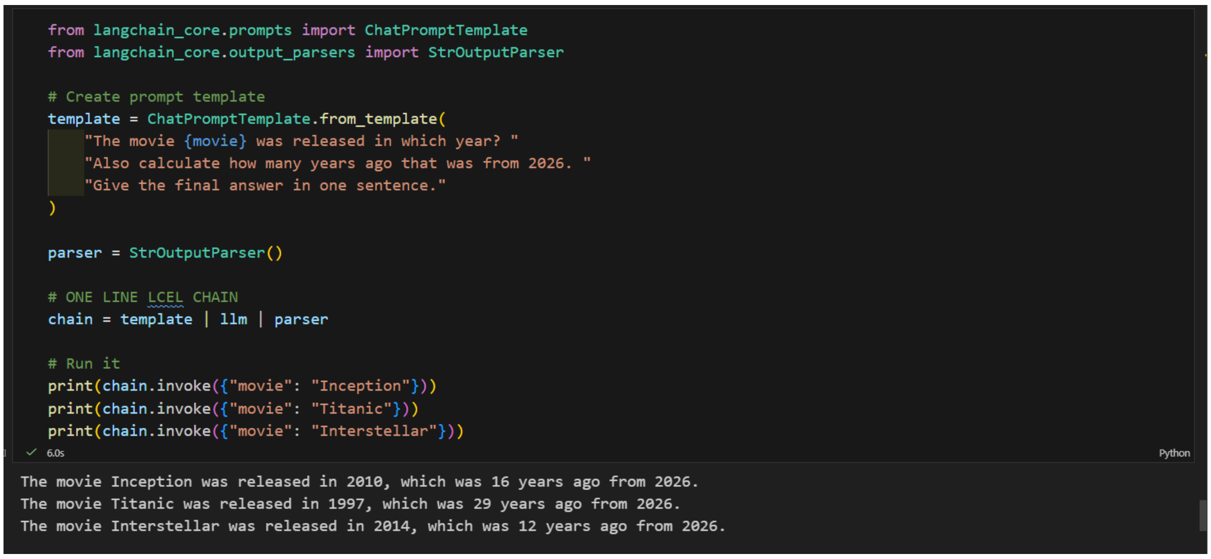
We use the **Pipe Operator** (|). It works just like Unix pipes: pass the output of the left side to the input of the right side.

```python
# Define the chain once
chain = template | llm | parser

# Invoke the whole chain
print(chain.invoke({"topic": "Octopuses"}))
```

```
Did you know octopuses have **three hearts**?

Two hearts pump blood through their gills, and a third, larger heart circulates blood to the rest of their body. And because their blood contains copper instead of iron, it's actually **blue**!
```

LCEL simplifies chaining by using the pipe (|) operator to connect prompt, model, and parser. It makes the workflow cleaner and more readable compared to manual chaining. The movie assignment successfully demonstrated dynamic input handling.

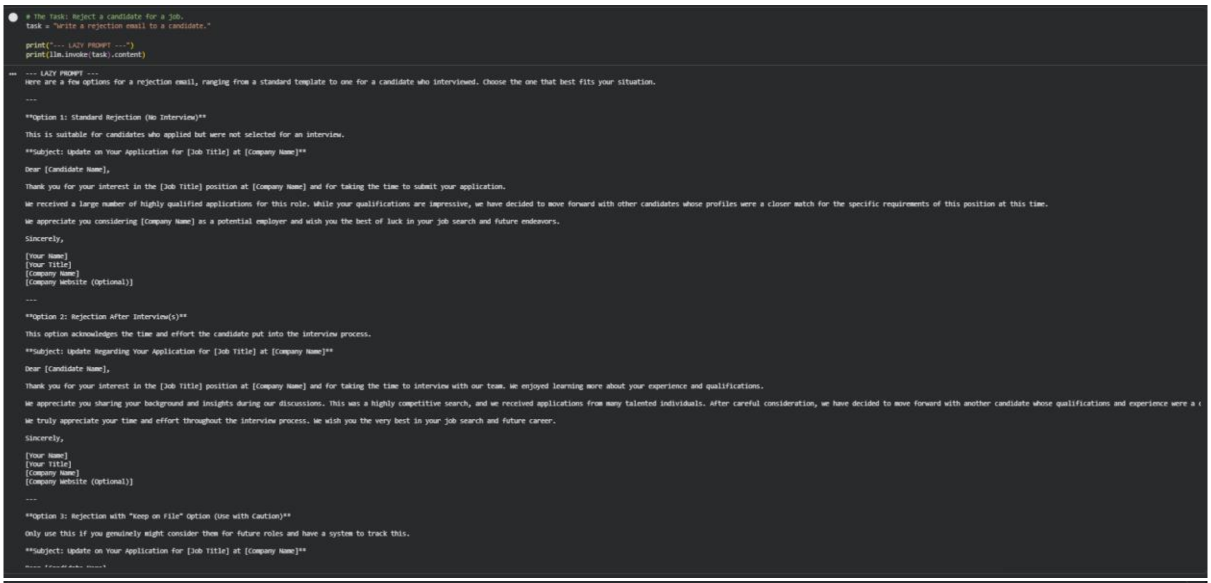
Assignment part:



```python
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Create prompt template
template = ChatPromptTemplate.from_template(
    "The movie {movie} was released in which year? "
    "Also calculate how many years ago that was from 2026. "
    "Give the final answer in one sentence."
)

parser = StrOutputParser()

# ONE LINE LCEL CHAIN
chain = template | llm | parser

# Run it
print(chain.invoke({"movie": "Inception"}))
print(chain.invoke({"movie": "Titanic"}))
print(chain.invoke({"movie": "Interstellar"}))
```

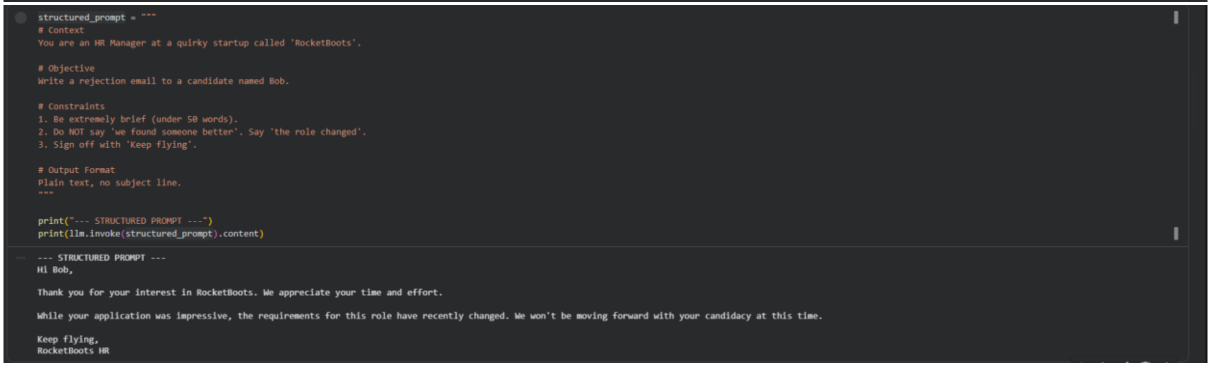
```
The movie Inception was released in 2010, which was 16 years ago from 2026.
The movie Titanic was released in 1997, which was 29 years ago from 2026.
The movie Interstellar was released in 2014, which was 12 years ago from 2026.
```

Part 2a:



```python
# The Task: Reject a candidate for a job.
task = "Write a rejection email to a candidate."

print("--- LAZY PROMPT ---")
print(llm.invoke(task).content)
```

```
--- LAZY PROMPT ---
Here are a few options for a rejection email, ranging from a standard template to one for a candidate who interviewed. Choose the one that best fits your situation.

---

**Option 1: Standard Rejection (No Interview)**

This is suitable for candidates who applied but were not selected for an interview.

**Subject: Update on Your Application for [Job Title] at [Company Name]**

Dear [Candidate Name],

Thank you for your interest in the [Job Title] position at [Company Name] and for taking the time to submit your application.

We received a large number of highly qualified applications for this role. While your qualifications are impressive, we have decided to move forward with other candidates whose profiles were a closer match for the specific requirements of this position at this time.

We appreciate you considering [Company Name] as a potential employer and wish you the best of luck in your job search and future endeavors.

Sincerely,

[Your Name]
[Your Title]
[Company Name]
[Company Website (Optional)]

---

**Option 2: Rejection After Interview(s)**

This option acknowledges the time and effort the candidate put into the interview process.

**Subject: Update Regarding Your Application for [Job Title] at [Company Name]**

Dear [Candidate Name],

Thank you for your interest in the [Job Title] position at [Company Name] and for taking the time to interview with our team. We enjoyed learning more about your experience and qualifications.

We appreciate you sharing your background and insights during our discussions. This was a highly competitive search, and we received applications from many talented individuals. After careful consideration, we have decided to move forward with another candidate whose qualifications and experience were a c

We truly appreciate your time and effort throughout the interview process. We wish you the very best in your job search and future career.

Sincerely,

[Your Name]
[Your Title]
[Company Name]
[Company Website (Optional)]

---

**Option 3: Rejection with "Keep on File" Option (Use with Caution)**

Only use this if you genuinely might consider them for future roles and have a system to track this.

**Subject: Update on Your Application for [Job Title] at [Company Name]**
```



```python
structured_prompt = """
# Context
You are an HR Manager at a quirky startup called 'RocketBoots'.

# Objective
Write a rejection email to a candidate named Bob.

# Constraints
1. Be extremely brief (under 50 words).
2. Do NOT say 'we found someone better'. Say 'the role changed'.
3. Sign off with 'Keep flying'.

# Output Format
Plain text, no subject line.
"""

print("--- STRUCTURED PROMPT ---")
print(llm.invoke(structured_prompt).content)
```

```
--- STRUCTURED PROMPT ---
Hi Bob,

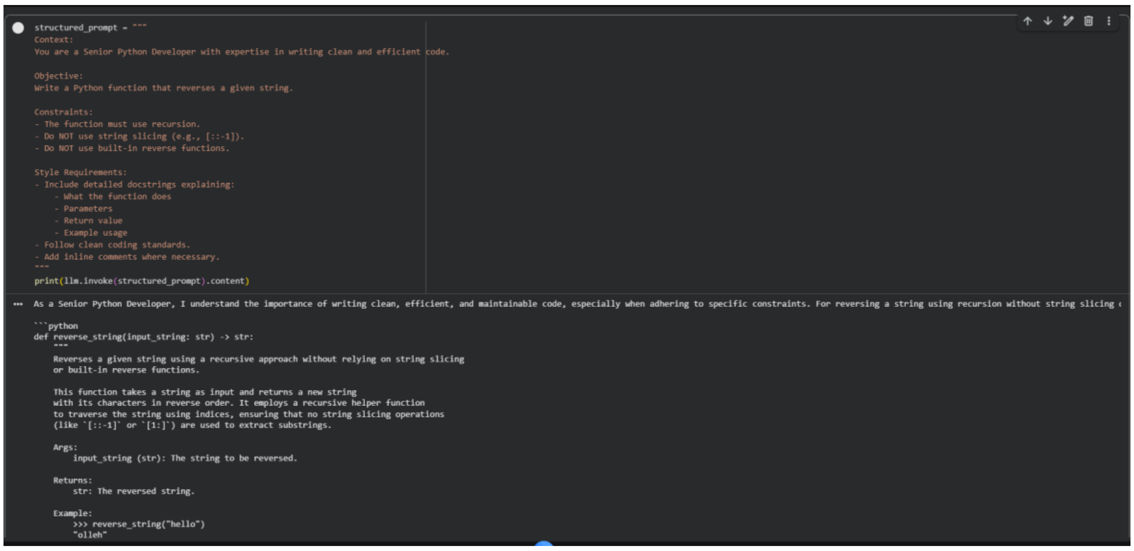
Thank you for your interest in RocketBoots. We appreciate your time and effort.

While your application was impressive, the requirements for this role have recently changed. We won't be moving forward with your candidacy at this time.

Keep flying,
RocketBoots HR
```

Providing clear context, objective, constraints, and style requirements resulted in high-quality and constraint-compliant code. The model correctly avoided string slicing and used recursion as required. Structured prompting improved output precision.

Assignment:



```
structured_prompt = """
Context:
You are a Senior Python Developer with expertise in writing clean and efficient code.

Objective:
Write a Python function that reverses a given string.

Constraints:
- The function must use recursion.
- Do NOT use string slicing (e.g., [::-1]).
- Do NOT use built-in reverse functions.

Style Requirements:
- Include detailed docstrings explaining:
    - What the function does
    - Parameters
    - Return value
    - Example usage
- Follow clean coding standards.
- Add inline comments where necessary.
"""
print(llm.invoke(structured_prompt).content)
```

~~~
As a Senior Python Developer, I understand the importance of writing clean, efficient, and maintainable code, especially when adhering to specific constraints. For reversing a string using recursion without string slicing

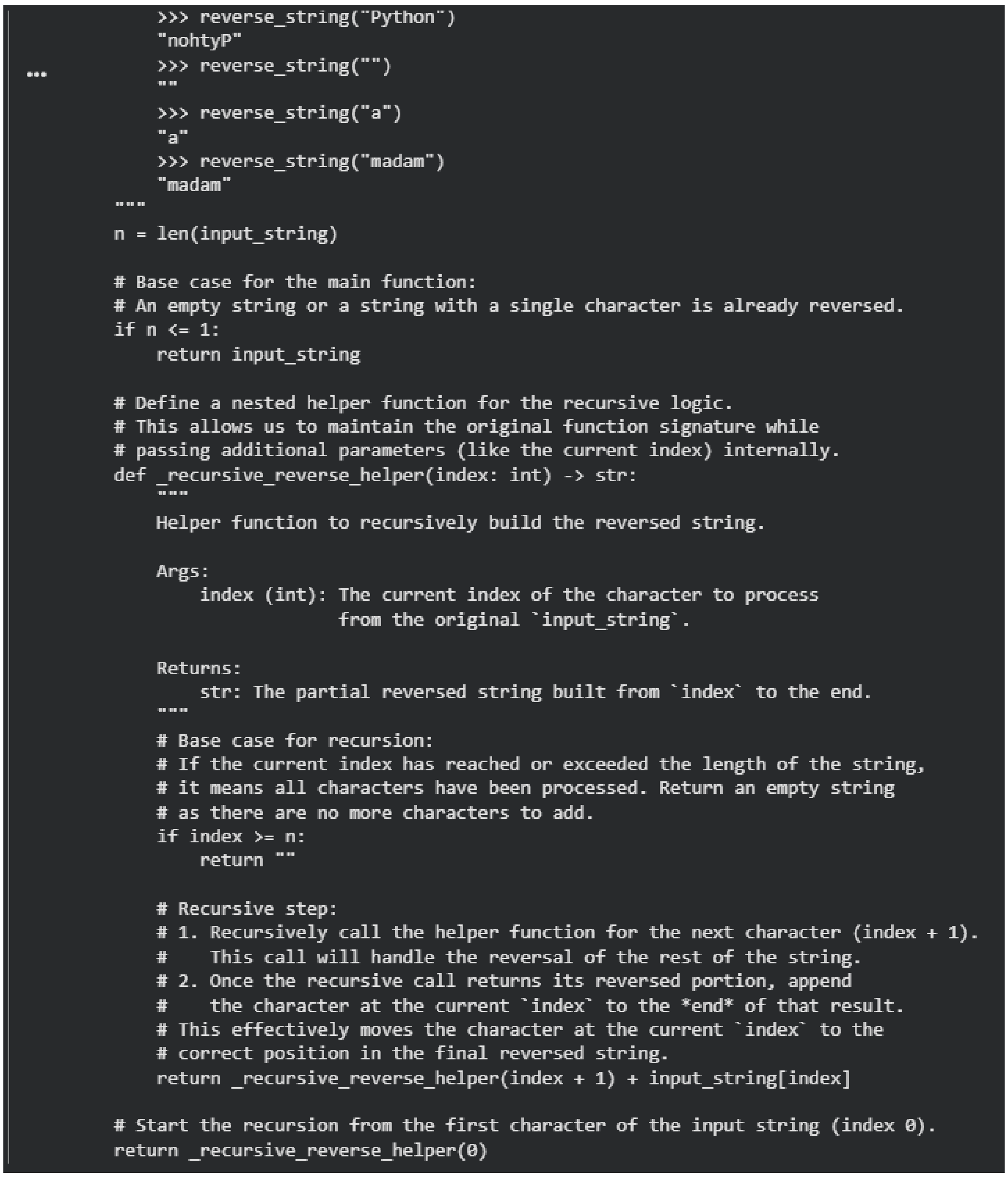
```python
def reverse_string(input_string: str) -> str:
    """
    Reverses a given string using a recursive approach without relying on string slicing
    or built-in reverse functions.

    This function takes a string as input and returns a new string
    with its characters in reverse order. It employs a recursive helper function
    to traverse the string using indices, ensuring that no string slicing operations
    (like `[::-1]` or `[1:]`) are used to extract substrings.

    Args:
        input_string (str): The string to be reversed.

    Returns:
        str: The reversed string.

    Example:
        >>> reverse_string("hello")
        "olleh"
~~~



```
        >>> reverse_string("Python")
        "nohtyP"
        >>> reverse_string("")
        ""
        >>> reverse_string("a")
        "a"
        >>> reverse_string("madam")
        "madam"
    """
    n = len(input_string)

    # Base case for the main function:
    # An empty string or a string with a single character is already reversed.
    if n <= 1:
        return input_string

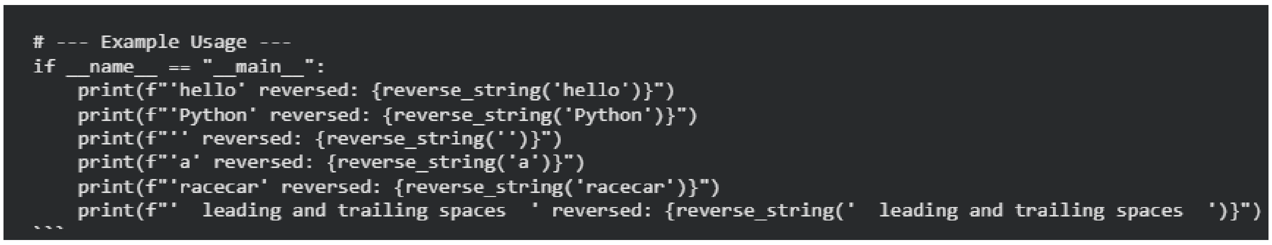
    # Define a nested helper function for the recursive logic.
    # This allows us to maintain the original function signature while
    # passing additional parameters (like the current index) internally.
    def _recursive_reverse_helper(index: int) -> str:
        """
        Helper function to recursively build the reversed string.

        Args:
            index (int): The current index of the character to process
                         from the original `input_string`.

        Returns:
            str: The partial reversed string built from `index` to the end.
        """
        # Base case for recursion:
        # If the current index has reached or exceeded the length of the string,
        # it means all characters have been processed. Return an empty string
        # as there are no more characters to add.
        if index >= n:
            return ""

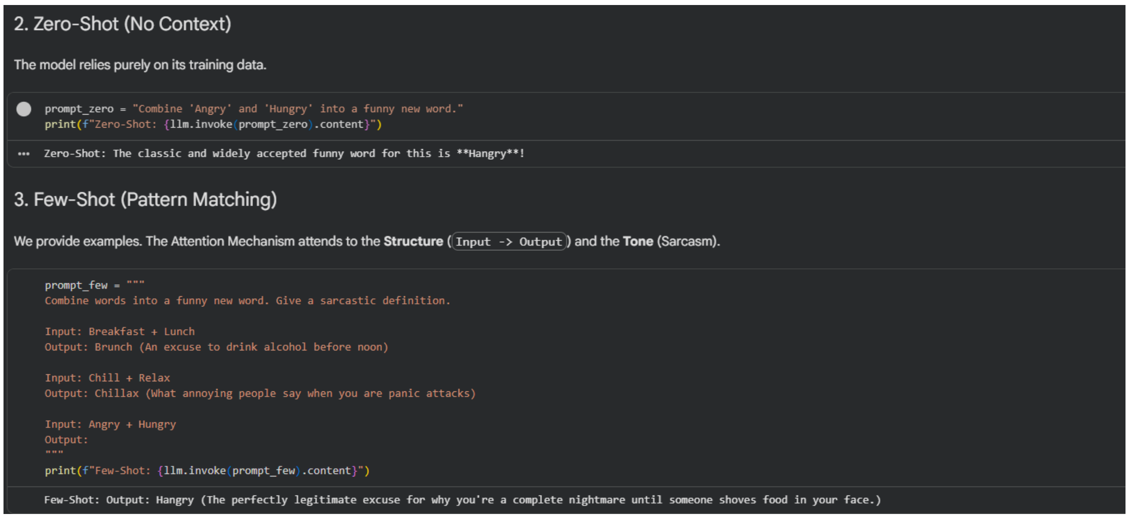
        # Recursive step:
        # 1. Recursively call the helper function for the next character (index + 1).
        #    This call will handle the reversal of the rest of the string.
        # 2. Once the recursive call returns its reversed portion, append
        #    the character at the current `index` to the *end* of that result.
        # This effectively moves the character at the current `index` to the
        # correct position in the final reversed string.
        return _recursive_reverse_helper(index + 1) + input_string[index]

    # Start the recursion from the first character of the input string (index 0).
    return _recursive_reverse_helper(0)
```



~~~~python
# --- Example Usage ---
if __name__ == "__main__":
    print(f"'hello' reversed: {reverse_string('hello')}")
    print(f"'Python' reversed: {reverse_string('Python')}")
    print(f"'' reversed: {reverse_string('')}")
    print(f"'a' reversed: {reverse_string('a')}")
    print(f"'racecar' reversed: {reverse_string('racecar')}")
    print(f"'  leading and trailing spaces  ' reversed: {reverse_string('  leading and trailing spaces  ')}")
```
~~~~

Part 2b:



2. Zero-Shot (No Context)

The model relies purely on its training data.

```python
prompt_zero = "Combine 'Angry' and 'Hungry' into a funny new word."
print(f"Zero-Shot: {llm.invoke(prompt_zero).content}")
```

```
Zero-Shot: The classic and widely accepted funny word for this is **Hangry**!
```

3. Few-Shot (Pattern Matching)

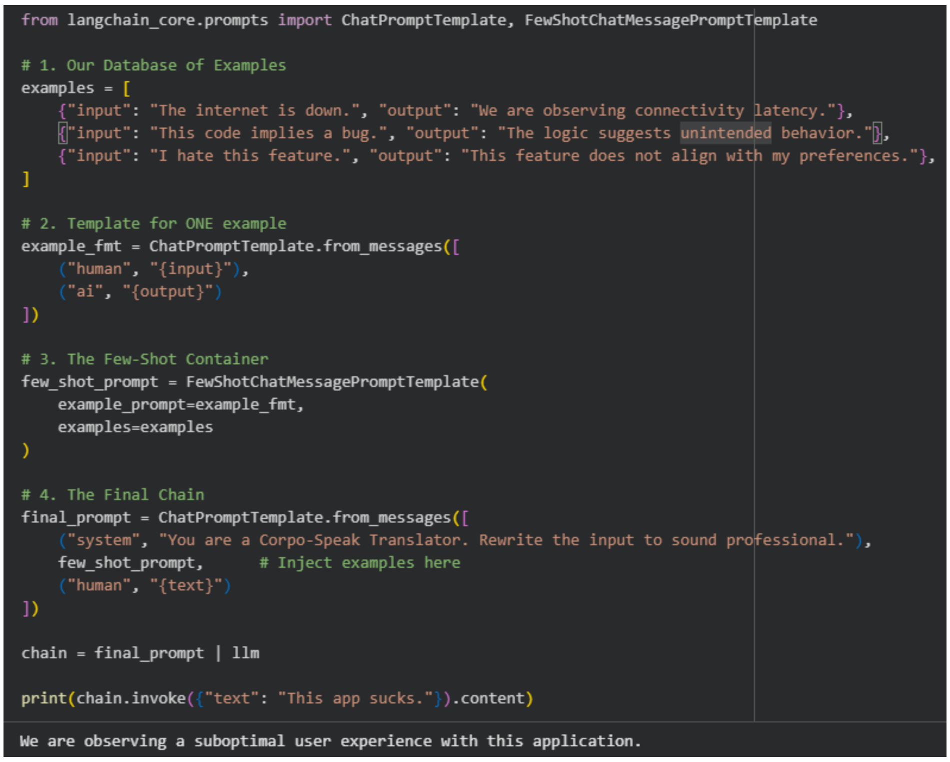
We provide examples. The Attention Mechanism attends to the **Structure** (`Input -> Output`) and the **Tone** (Sarcasm).

```python
prompt_few = """
Combine words into a funny new word. Give a sarcastic definition.

Input: Breakfast + Lunch
Output: Brunch (An excuse to drink alcohol before noon)

Input: Chill + Relax
Output: Chillax (What annoying people say when you are panic attacks)

Input: Angry + Hungry
Output:
"""
print(f"Few-Shot: {llm.invoke(prompt_few).content}")
```

```
Few-Shot: Output: Hangry (The perfectly legitimate excuse for why you're a complete nightmare until someone shoves food in your face.)
```

Few-shot examples guided the model to follow the required format and tone. The output was more consistent compared to zero-shot prompting. This demonstrates how examples influence model behavior.

Part 2c:

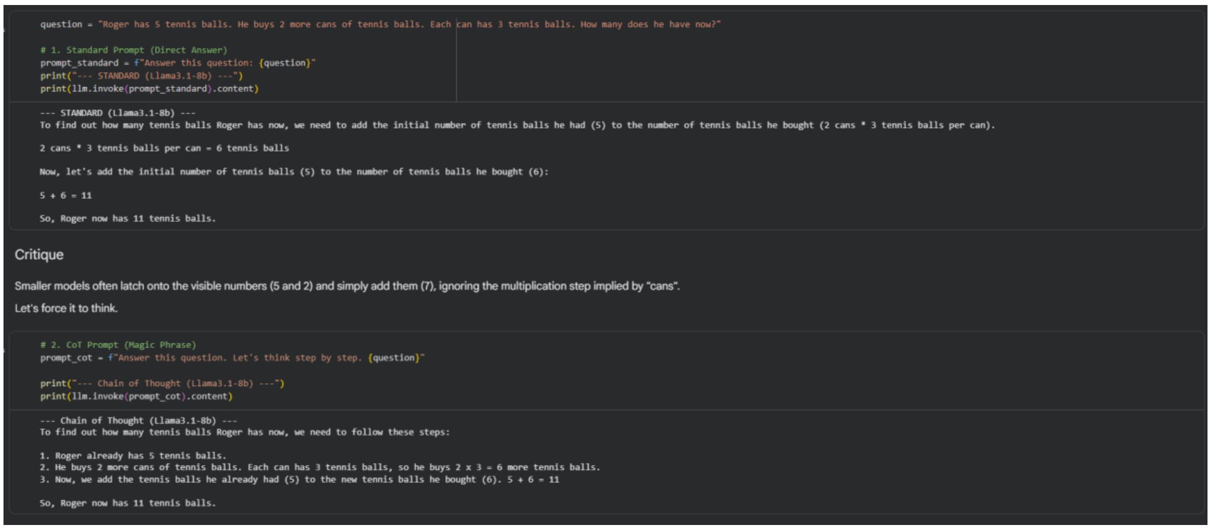


```python
from langchain_core.prompts import ChatPromptTemplate, FewShotChatMessagePromptTemplate

# 1. Our Database of Examples
examples = [
    {"input": "The internet is down.", "output": "We are observing connectivity latency."},
    {"input": "This code implies a bug.", "output": "The logic suggests unintended behavior."},
    {"input": "I hate this feature.", "output": "This feature does not align with my preferences."},
]

# 2. Template for ONE example
example_fmt = ChatPromptTemplate.from_messages([
    ("human", "{input}"),
    ("ai", "{output}")
])

# 3. The Few-Shot Container
few_shot_prompt = FewShotChatMessagePromptTemplate(
    example_prompt=example_fmt,
    examples=examples
)

# 4. The Final Chain
final_prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a Corpo-Speak Translator. Rewrite the input to sound professional."),
    few_shot_prompt,      # Inject examples here
    ("human", "{text}")
])

chain = final_prompt | llm

print(chain.invoke({"text": "This app sucks."}).content)
```

```
We are observing a suboptimal user experience with this application.
```

Using FewShotChatMessagePromptTemplate allowed structured insertion of examples into the final prompt. This improved response professionalism and consistency. It shows how pattern-based prompting enhances output reliability.

Part 3a:



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

# 1. Standard Prompt (Direct Answer)
prompt_standard = f"Answer this question: {question}"
print("--- STANDARD (Llama3.1-8b) ---")
print(llm.invoke(prompt_standard).content)
```

```
--- STANDARD (Llama3.1-8b) ---
To find out how many tennis balls Roger has now, we need to add the initial number of tennis balls he had (5) to the number of tennis balls he bought (2 cans * 3 tennis balls per can).

2 cans * 3 tennis balls per can = 6 tennis balls

Now, let's add the initial number of tennis balls (5) to the number of tennis balls he bought (6):

5 + 6 = 11

So, Roger now has 11 tennis balls.
```

Critique

Smaller models often latch onto the visible numbers (5 and 2) and simply add them (7), ignoring the multiplication step implied by "cans".

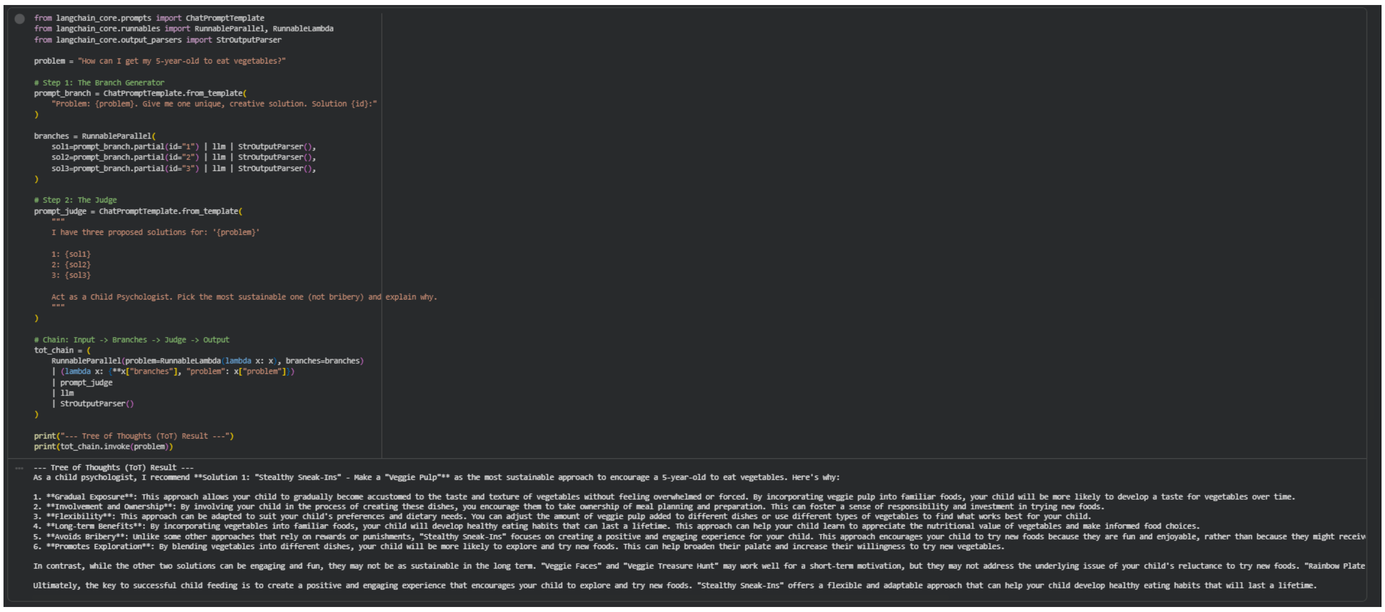
Let's force it to think.

```
# 2. CoT Prompt (Magic Phrase)
prompt_cot = f"Answer this question. Let's think step by step. {question}"

print("--- Chain of Thought (Llama3.1-8b) ---")
print(llm.invoke(prompt_cot).content)
```

```
--- Chain of Thought (Llama3.1-8b) ---
To find out how many tennis balls Roger has now, we need to follow these steps:

1. Roger already has 5 tennis balls.
2. He buys 2 more cans of tennis balls. Each can has 3 tennis balls, so he buys 2 x 3 = 6 more tennis balls.
3. Now, we add the tennis balls he already had (5) to the new tennis balls he bought (6). 5 + 6 = 11

So, Roger now has 11 tennis balls.
```

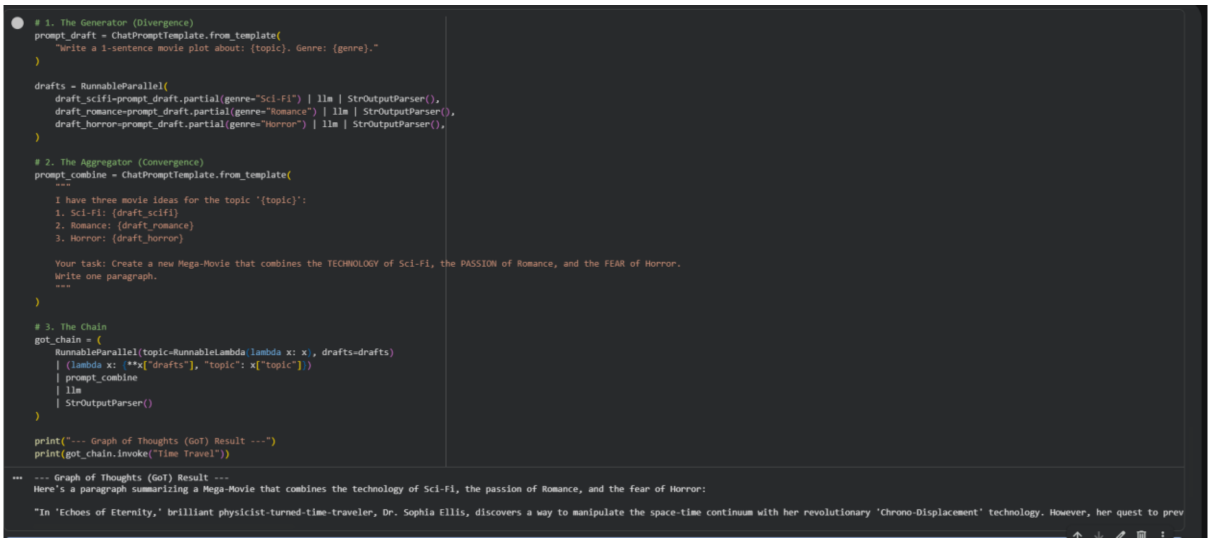
Zero-shot prompting relies only on model knowledge and may produce generic responses. Few-shot prompting provides structured examples, resulting in more controlled and context-aware outputs. Few-shot performed better in tone and format alignment.

Part 3b:



```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableParallel, RunnableLambda
from langchain_core.output_parsers import StrOutputParser

problem = "How can I get my 5-year-old to eat vegetables?"

# Step 1: The Branch Generator
prompt_branch = ChatPromptTemplate.from_template(
    "Problem: {problem}. Give me one unique, creative solution. Solution {id}:"
)

branches = RunnableParallel(
    sol1=prompt_branch.partial(id="1") | llm | StrOutputParser(),
    sol2=prompt_branch.partial(id="2") | llm | StrOutputParser(),
    sol3=prompt_branch.partial(id="3") | llm | StrOutputParser(),
)

# Step 2: The Judge
prompt_judge = ChatPromptTemplate.from_template(
    """
    I have three proposed solutions for: '{problem}'

    1: {sol1}
    2: {sol2}
    3: {sol3}

    Act as a Child Psychologist. Pick the most sustainable one (not bribery) and explain why.
    """
)

# Chain: Input -> Branches -> Judge -> Output
tot_chain = (
    RunnableParallel(problem=RunnableLambda(lambda x: x), branches=branches)
    | (lambda x: {**x["branches"], "problem": x["problem"]})
    | prompt_judge
    | llm
    | StrOutputParser()
)

print("--- Tree of Thoughts (ToT) Result ---")
print(tot_chain.invoke(problem))
```

```
--- Tree of Thoughts (ToT) Result ---
As a child psychologist, I recommend **Solution 1: "Stealthy Sneak-Ins" - Make a "Veggie Pulp"** as the most sustainable approach to encourage a 5-year-old to eat vegetables. Here's why:

1. **Gradual Exposure**: This approach allows your child to gradually become accustomed to the taste and texture of vegetables without feeling overwhelmed or forced. By incorporating veggie pulp into familiar foods, your child will be more likely to develop a taste for vegetables over time.
2. **Involvement and Ownership**: By involving your child in the process of creating these dishes, you encourage them to take ownership of meal planning and preparation. This can foster a sense of responsibility and investment in trying new foods.
3. **Flexibility**: This approach can be adapted to suit your child's preferences and dietary needs. You can adjust the amount of veggie pulp added to different dishes or use different types of vegetables to find what works best for your child.
4. **Long-term Benefits**: By incorporating vegetables into familiar foods, your child will develop healthy eating habits that can last a lifetime. This approach can help your child learn to appreciate the nutritional value of vegetables and make informed food choices.
5. **Avoids Bribery**: Unlike some other approaches that rely on rewards or punishments, "Stealthy Sneak-Ins" focuses on creating a positive and engaging experience for your child. This approach encourages your child to try new foods because they are fun and enjoyable, rather than because they might receiv
6. **Promotes Exploration**: By blending vegetables into different dishes, your child will be more likely to explore and try new foods. This can help broaden their palate and increase their willingness to try new vegetables.

In contrast, while the other two solutions can be engaging and fun, they may not be as sustainable in the long term. "Veggie Faces" and "Veggie Treasure Hunt" may work well for a short-term motivation, but they may not address the underlying issue of your child's reluctance to try new foods. "Rainbow Plate

Ultimately, the key to successful child feeding is to create a positive and engaging experience that encourages your child to explore and try new foods. "Stealthy Sneak-Ins" offers a flexible and adaptable approach that can help your child develop healthy eating habits that will last a lifetime.
```

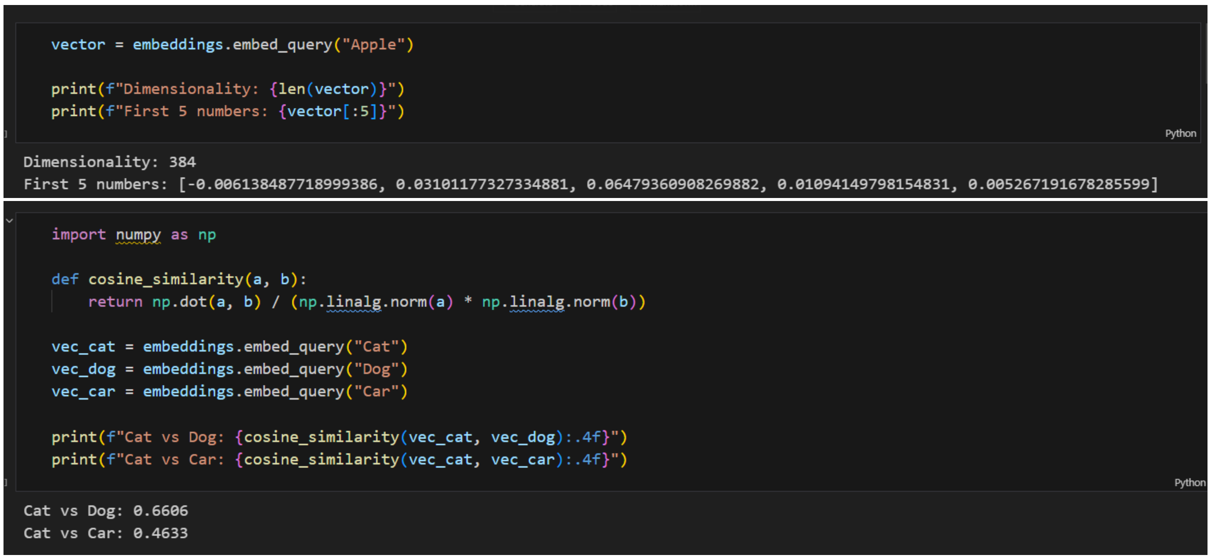


```python
# 1. The Generator (Divergence)
prompt_draft = ChatPromptTemplate.from_template(
    "Write a 1-sentence movie plot about: {topic}. Genre: {genre}."
)

drafts = RunnableParallel(
    draft_scifi=prompt_draft.partial(genre="Sci-Fi") | llm | StrOutputParser(),
    draft_romance=prompt_draft.partial(genre="Romance") | llm | StrOutputParser(),
    draft_horror=prompt_draft.partial(genre="Horror") | llm | StrOutputParser(),
)

# 2. The Aggregator (Convergence)
prompt_combine = ChatPromptTemplate.from_template(
    """
    I have three movie ideas for the topic '{topic}':
    1. Sci-Fi: {draft_scifi}
    2. Romance: {draft_romance}
    3. Horror: {draft_horror}

    Your task: Create a new Mega-Movie that combines the TECHNOLOGY of Sci-Fi, the PASSION of Romance, and the FEAR of Horror.
    Write one paragraph.
    """
)

# 3. The Chain
got_chain = (
    RunnableParallel(topic=RunnableLambda(lambda x: x), drafts=drafts)
    | (lambda x: {**x["drafts"], "topic": x["topic"]})
    | prompt_combine
    | llm
    | StrOutputParser()
)

print("--- Graph of Thoughts (GoT) Result ---")
print(got_chain.invoke("Time Travel"))
```

```
--- Graph of Thoughts (GoT) Result ---
Here's a paragraph summarizing a Mega-Movie that combines the technology of Sci-Fi, the passion of Romance, and the fear of Horror:

"In 'Echoes of Eternity,' brilliant physicist-turned-time-traveler, Dr. Sophia Ellis, discovers a way to manipulate the space-time continuum with her revolutionary 'Chrono-Displacement' technology. However, her quest to prev
```

Chain-of-Thought prompting encouraged step-by-step reasoning, improving logical accuracy. The structured reasoning helped the model avoid mistakes and explain intermediate steps clearly. This technique enhances reasoning tasks.

Part 4a:



```python
vector = embeddings.embed_query("Apple")

print(f"Dimensionality: {len(vector)}")
print(f"First 5 numbers: {vector[:5]}")
```

Python

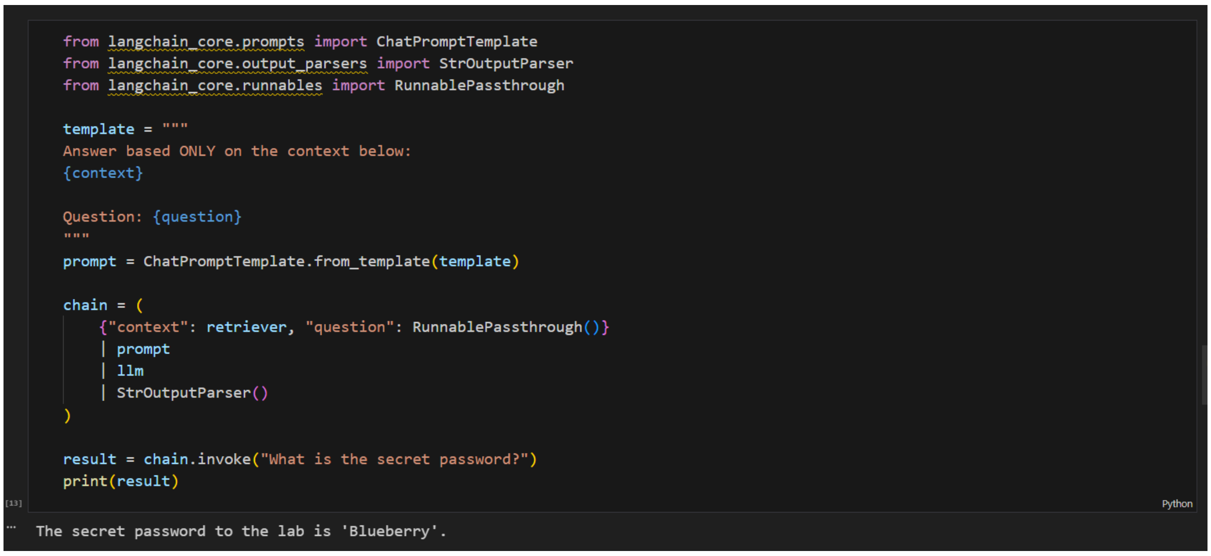
```
Dimensionality: 384
First 5 numbers: [-0.006138487718999386, 0.03101177327334881, 0.06479360908269882, 0.01094149798154831, 0.005267191678285599]
```

```python
import numpy as np

def cosine_similarity(a, b):
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

vec_cat = embeddings.embed_query("Cat")
vec_dog = embeddings.embed_query("Dog")
vec_car = embeddings.embed_query("Car")

print(f"Cat vs Dog: {cosine_similarity(vec_cat, vec_dog):.4f}")
print(f"Cat vs Car: {cosine_similarity(vec_cat, vec_car):.4f}")
```

Python

```
Cat vs Dog: 0.6606
Cat vs Car: 0.4633
```

Part 4b:

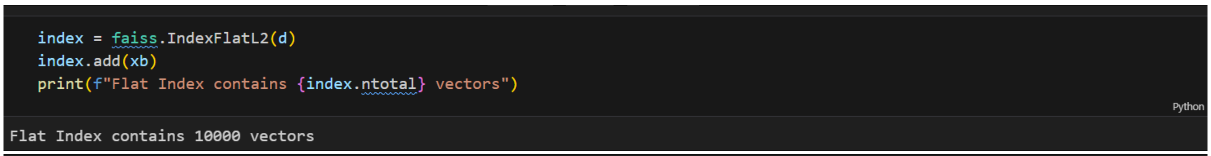


```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

template = """
Answer based ONLY on the context below:
{context}

Question: {question}
"""
prompt = ChatPromptTemplate.from_template(template)

chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)

result = chain.invoke("What is the secret password?")
print(result)
```

[13] Python

```
The secret password to the lab is 'Blueberry'.
```

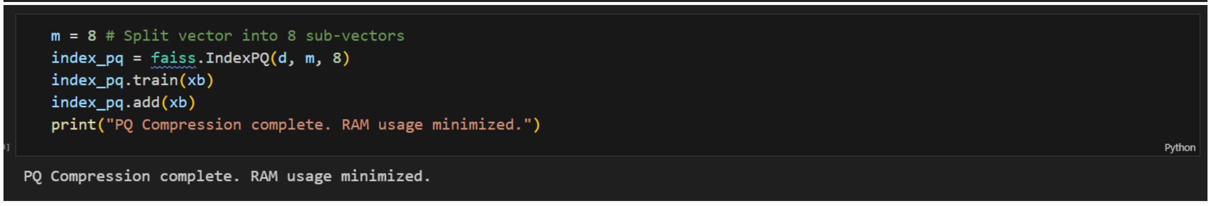
Part 4c:



```
index = faiss.IndexFlatL2(d)
index.add(xb)
print(f"Flat Index contains {index.ntotal} vectors")
```

Python

```
Flat Index contains 10000 vectors
```



```
m = 8 # Split vector into 8 sub-vectors
index_pq = faiss.IndexPQ(d, m, 8)
index_pq.train(xb)
index_pq.add(xb)
print("PQ Compression complete. RAM usage minimized.")
```

Python

```
PQ Compression complete. RAM usage minimized.
```