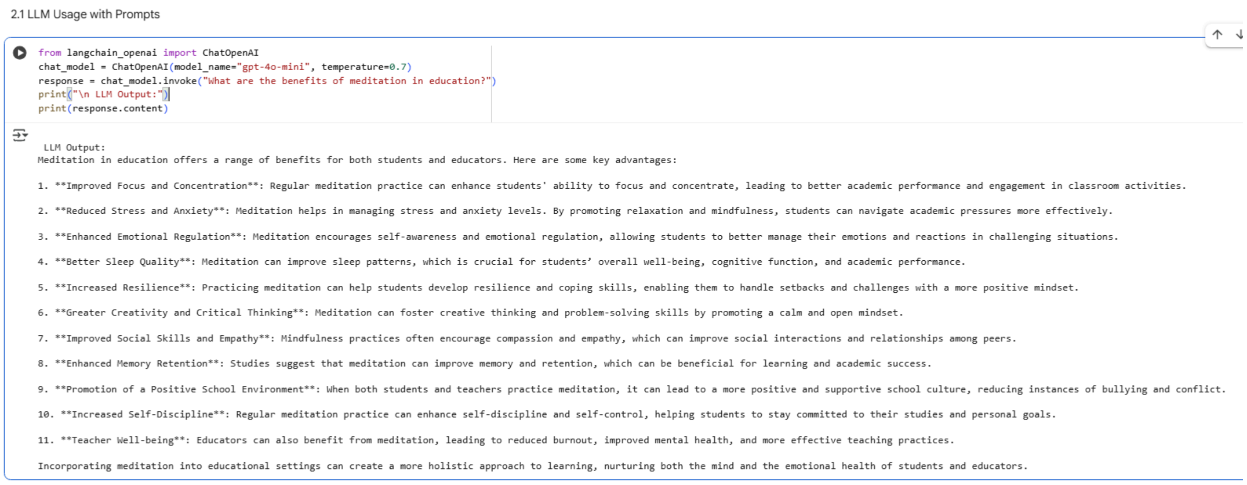
Assignment 2 - DATA266

San Jose State University

Padmanabh Rathi - 018180006

2. Explore the core components of LangChain (LLMs, Prompt Templates, Chains, etc). Experiment with each and describe how they interact. (5 points)

2.1 LLM Usage with Prompts


2.1 LLM Usage with Prompts

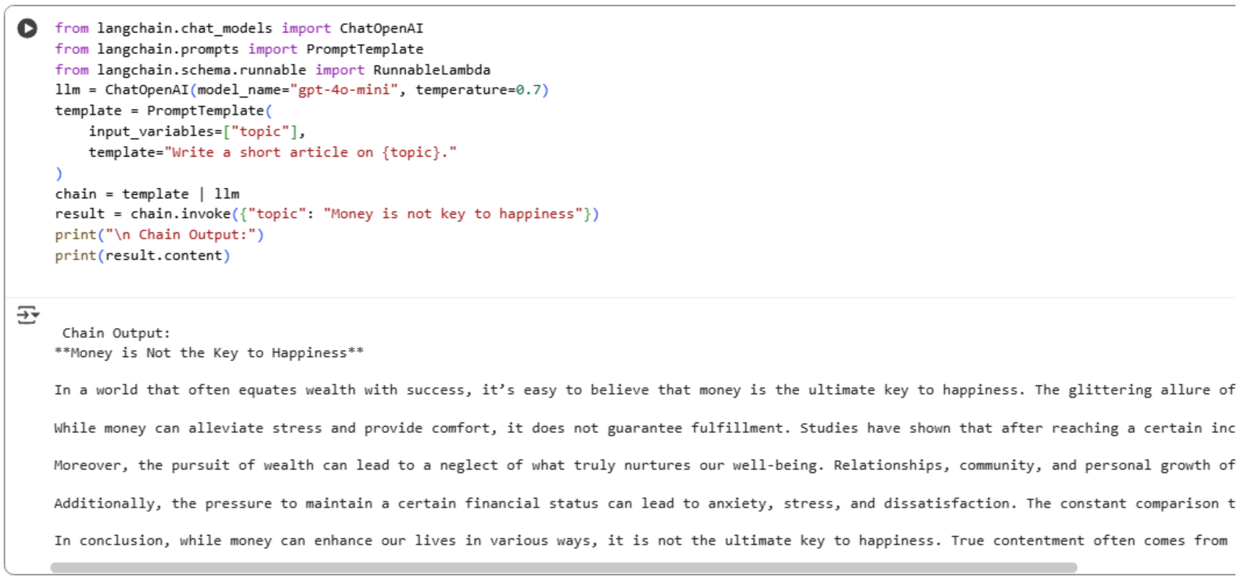
```
from langchain_openai import ChatOpenAI
chat_model = ChatOpenAI(model_name="gpt-4o-mini", temperature=0.7)
response = chat_model.invoke("What are the benefits of meditation in education?")
print("\n LLM Output:")
print(response.content)
```

```
 LLM Output:
Meditation in education offers a range of benefits for both students and educators. Here are some key advantages:

1. **Improved Focus and Concentration**: Regular meditation practice can enhance students' ability to focus and concentrate, leading to better academic performance and engagement in classroom activities.

2. **Reduced Stress and Anxiety**: Meditation helps in managing stress and anxiety levels. By promoting relaxation and mindfulness, students can navigate academic pressures more effectively.

3. **Enhanced Emotional Regulation**: Meditation encourages self-awareness and emotional regulation, allowing students to better manage their emotions and reactions in challenging situations.

4. **Better Sleep Quality**: Meditation can improve sleep patterns, which is crucial for students’ overall well-being, cognitive function, and academic performance.

5. **Increased Resilience**: Practicing meditation can help students develop resilience and coping skills, enabling them to handle setbacks and challenges with a more positive mindset.

6. **Greater Creativity and Critical Thinking**: Meditation can foster creative thinking and problem-solving skills by promoting a calm and open mindset.

7. **Improved Social Skills and Empathy**: Mindfulness practices often encourage compassion and empathy, which can improve social interactions and relationships among peers.

8. **Enhanced Memory Retention**: Studies suggest that meditation can improve memory and retention, which can be beneficial for learning and academic success.

9. **Promotion of a Positive School Environment**: When both students and teachers practice meditation, it can lead to a more positive and supportive school culture, reducing instances of bullying and conflict.

10. **Increased Self-Discipline**: Regular meditation practice can enhance self-discipline and self-control, helping students to stay committed to their studies and personal goals.

11. **Teacher Well-being**: Educators can also benefit from meditation, leading to reduced burnout, improved mental health, and more effective teaching practices.

Incorporating meditation into educational settings can create a more holistic approach to learning, nurturing both the mind and the emotional health of students and educators.
```

In the above snippet, I use a chat LLM directly with a plain string prompt. ChatOpenAI(model_name="gpt-4o-mini", temperature=0.7) creates the model client and sets decoding randomness. Calling invoke("What are the benefits of meditation in education?") sends that text as a single user turn to the API. The model returns an AIMessage object; response.content holds the generated text, which I print. There’s no prompt template, memory, retriever, or chain here—just a straight path: raw prompt → LLM → text output. This shows the basic interaction where the LLM is the only component and the prompt is just the input string.

2.2 Combining LLMs and Prompts

2.2 Combining LLMs and Prompts

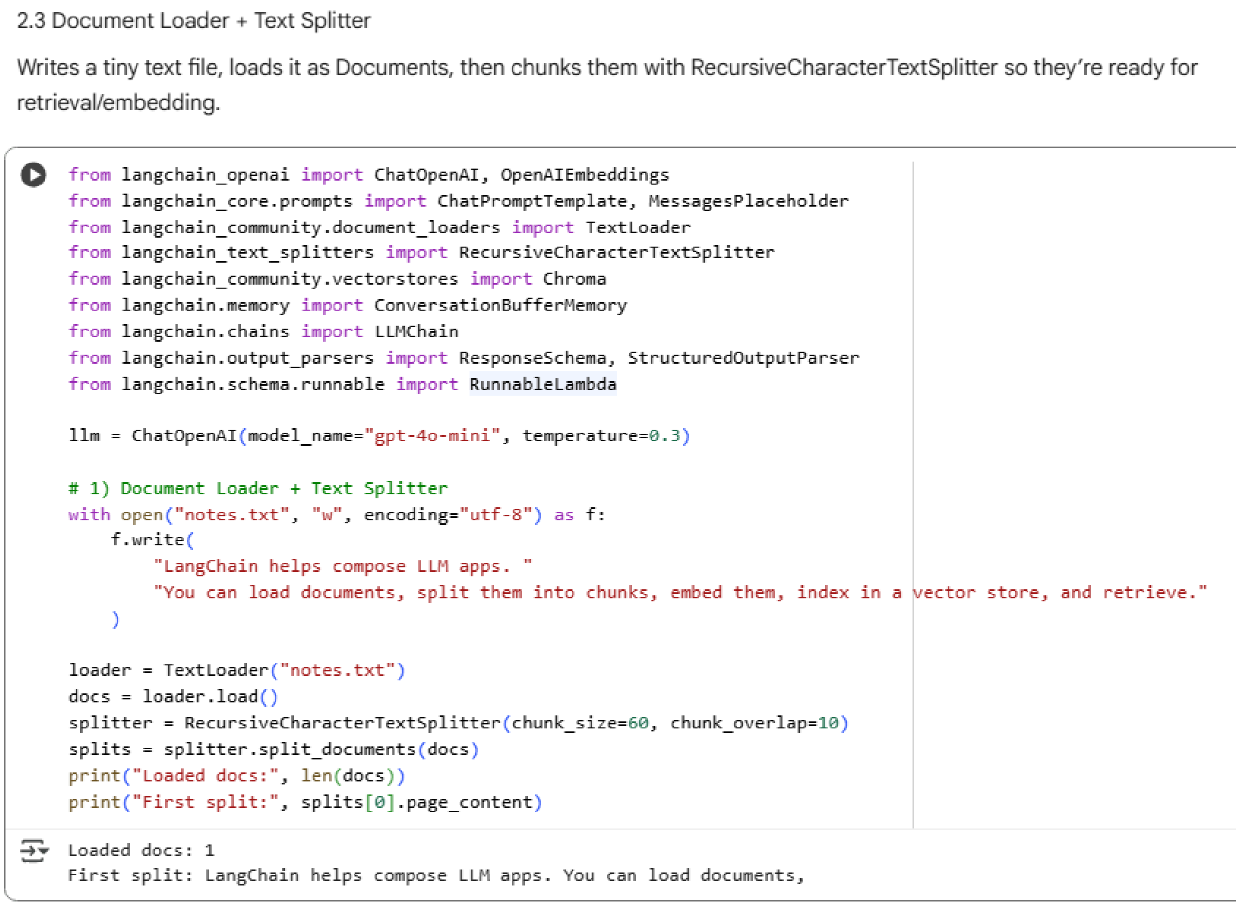


```python
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.schema.runnable import RunnableLambda
llm = ChatOpenAI(model_name="gpt-4o-mini", temperature=0.7)
template = PromptTemplate(
    input_variables=["topic"],
    template="Write a short article on {topic}."
)
chain = template | llm
result = chain.invoke({"topic": "Money is not key to happiness"})
print("\n Chain Output:")
print(result.content)
```

```
 Chain Output:
**Money is Not the Key to Happiness**

In a world that often equates wealth with success, it’s easy to believe that money is the ultimate key to happiness. The glittering allure of

While money can alleviate stress and provide comfort, it does not guarantee fulfillment. Studies have shown that after reaching a certain inc

Moreover, the pursuit of wealth can lead to a neglect of what truly nurtures our well-being. Relationships, community, and personal growth of

Additionally, the pressure to maintain a certain financial status can lead to anxiety, stress, and dissatisfaction. The constant comparison t

In conclusion, while money can enhance our lives in various ways, it is not the ultimate key to happiness. True contentment often comes from
```

This example composes a prompt with an LLM using LCEL. PromptTemplate defines a template with one variable {topic}. At runtime, calling chain.invoke({"topic": "Money is not key to happiness"}) fills that variable to produce the final prompt string. The LCEL pipe template | llm passes this rendered string directly into ChatOpenAI(model_name="gpt-4o-mini", temperature=0.7), which implicitly wraps the text as a human message and generates a response. The return value is an AIMessage; printing result.content shows the article. There’s no memory, retriever, or parser involved here—just variable substitution → LLM generation.

2.3 Document Loader + Text Splitter


2.3 Document Loader + Text Splitter

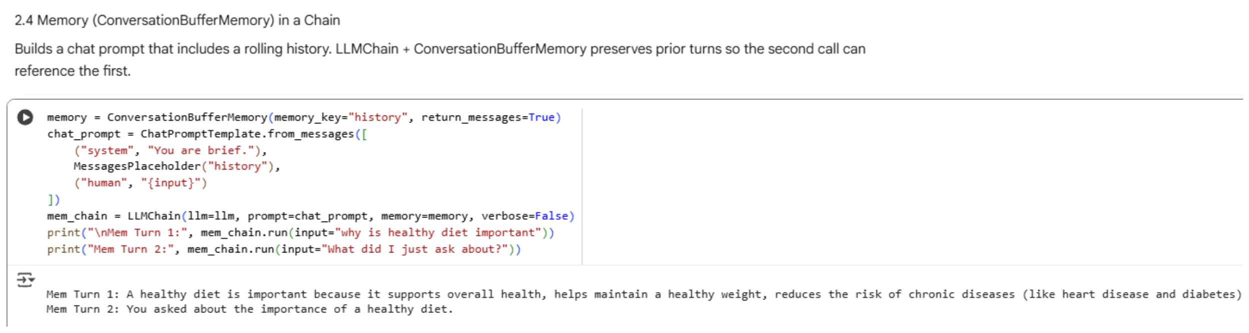
Writes a tiny text file, loads it as Documents, then chunks them with RecursiveCharacterTextSplitter so they're ready for retrieval/embedding.

```python
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain.memory import ConversationBufferMemory
from langchain.chains import LLMChain
from langchain.output_parsers import ResponseSchema, StructuredOutputParser
from langchain.schema.runnable import RunnableLambda

llm = ChatOpenAI(model_name="gpt-4o-mini", temperature=0.3)

# 1) Document Loader + Text Splitter
with open("notes.txt", "w", encoding="utf-8") as f:
    f.write(
        "LangChain helps compose LLM apps. "
        "You can load documents, split them into chunks, embed them, index in a vector store, and retrieve."
    )

loader = TextLoader("notes.txt")
docs = loader.load()
splitter = RecursiveCharacterTextSplitter(chunk_size=60, chunk_overlap=10)
splits = splitter.split_documents(docs)
print("Loaded docs:", len(docs))
print("First split:", splits[0].page_content)
```

```
Loaded docs: 1
First split: LangChain helps compose LLM apps. You can load documents,
```

I wrote a short note to disk, loaded it with TextLoader, and got a list containing one Document (text plus metadata). I then fed that list to RecursiveCharacterTextSplitter(chunk_size=60, chunk_overlap=10). The splitter reads each Document.page_content and produces overlapping chunks of up to 60 characters with a 10-character overlap to preserve context between chunks. In other words, the loader's job is to standardize raw text into Document objects; the splitter's job is to turn those documents into chunked units that are easier to index or pass to models. The printed result confirms the flow: one document in, and the first chunk shows the beginning of the note, clipped at the chosen size.

2.4 Memory (ConversationBufferMemory) in a Chain


2.4 Memory (ConversationBufferMemory) in a Chain

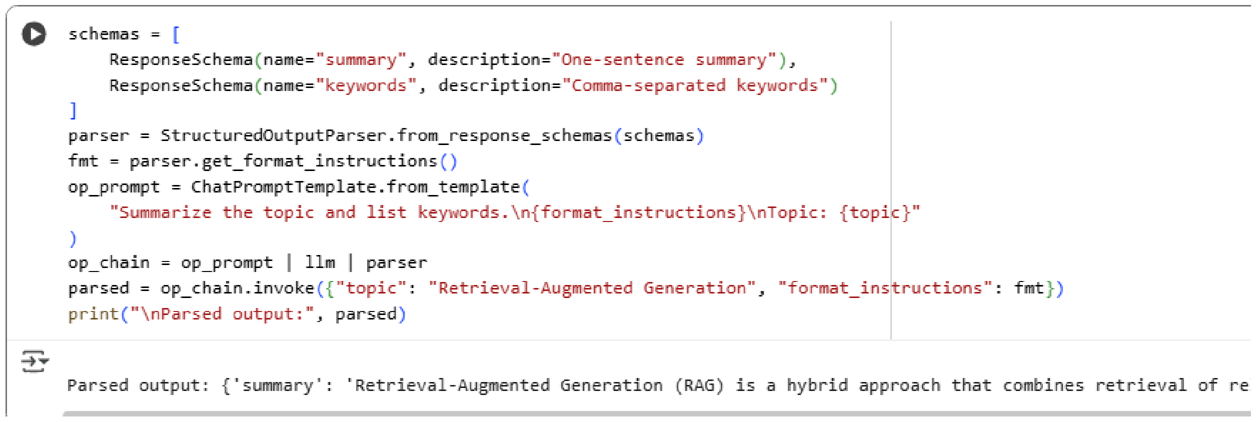
Builds a chat prompt that includes a rolling history. LLMChain + ConversationBufferMemory preserves prior turns so the second call can reference the first.

```
memory = ConversationBufferMemory(memory_key="history", return_messages=True)
chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "You are brief."),
    MessagesPlaceholder("history"),
    ("human", "{input}")
])
mem_chain = LLMChain(llm=llm, prompt=chat_prompt, memory=memory, verbose=False)
print("\nMem Turn 1:", mem_chain.run(input="why is healthy diet important"))
print("Mem Turn 2:", mem_chain.run(input="What did I just ask about?"))
```

```
Mem Turn 1: A healthy diet is important because it supports overall health, helps maintain a healthy weight, reduces the risk of chronic diseases (like heart disease and diabetes)
Mem Turn 2: You asked about the importance of a healthy diet.
```

Here, I connected a chat prompt to an LLM with a conversation memory so the second turn can reference the first. ConversationBufferMemory stores prior messages under the key "history". The prompt is built with three parts—a system message, a MessagesPlaceholder("history"), and the human slot {input}—so whatever the memory has gets injected into the prompt on each call. LLMChain ties the llm, chat_prompt, and memory together. On Turn 1, the memory is empty, so the model answers the question about a healthy diet and that exchange (user + AI) is saved to memory. On Turn 2, when I ask “What did I just ask about?”, the chain inserts the saved history via the placeholder, the model sees the prior question in-context, and responds correctly that I asked about the importance of a healthy diet. The memory_key="history" must match the placeholder name for this to work.

2.5 Output Parser (structured JSON)

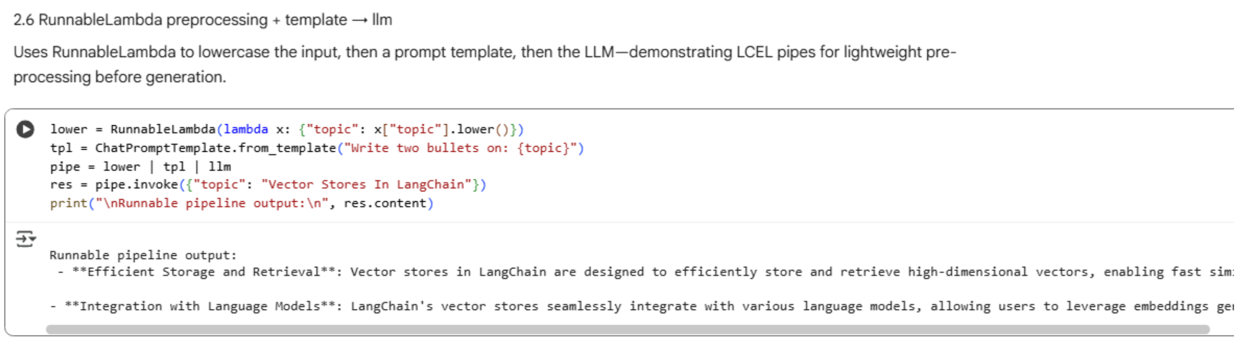


```
schemas = [
    ResponseSchema(name="summary", description="One-sentence summary"),
    ResponseSchema(name="keywords", description="Comma-separated keywords")
]
parser = StructuredOutputParser.from_response_schemas(schemas)
fmt = parser.get_format_instructions()
op_prompt = ChatPromptTemplate.from_template(
    "Summarize the topic and list keywords.\n{format_instructions}\nTopic: {topic}"
)
op_chain = op_prompt | llm | parser
parsed = op_chain.invoke({"topic": "Retrieval-Augmented Generation", "format_instructions": fmt})
print("\nParsed output:", parsed)
```

```
Parsed output: {'summary': 'Retrieval-Augmented Generation (RAG) is a hybrid approach that combines retrieval of re
```

I defined a two-field schema (summary, keywords) and built a StructuredOutputParser from it. The parser provides format_instructions, which I inject into the prompt so the model is told exactly how to structure its reply. The LCEL pipe op_prompt | llm | parser runs in three steps: the prompt renders with the topic and format rules → the LLM generates text following those rules → the parser reads that text and returns a Python dict matching the schema. The printed parsed value shows the final structured result with the required fields.

2.6 RunnableLambda preprocessing + template → llm


2.6 RunnableLambda preprocessing + template → llm

Uses RunnableLambda to lowercase the input, then a prompt template, then the LLM—demonstrating LCEL pipes for lightweight pre-processing before generation.

```
lower = RunnableLambda(lambda x: {"topic": x["topic"].lower()})
tpl = ChatPromptTemplate.from_template("Write two bullets on: {topic}")
pipe = lower | tpl | llm
res = pipe.invoke({"topic": "Vector Stores In LangChain"})
print("\nRunnable pipeline output:\n", res.content)
```

```
Runnable pipeline output:
 - **Efficient Storage and Retrieval**: Vector stores in LangChain are designed to efficiently store and retrieve high-dimensional vectors, enabling fast sim

- **Integration with Language Models**: LangChain's vector stores seamlessly integrate with various language models, allowing users to leverage embeddings ge
```

I added a lightweight preprocessing step before prompting the model. RunnableLambda lowercases the incoming topic and returns a dict shaped for the prompt ({"topic": ...}). That dict feeds ChatPromptTemplate, which formats the final message "Write two bullets on: {topic}". The LCEL pipe lower | tpl | llm runs these stages in order: preprocess → render prompt → call the chat model. The model returns an AIMessage; res.content prints the two bullets. There’s no memory or retrieval—just a sequential pipeline where the lambda prepares inputs for the template, and the template prepares text for the LLM.

3. Explore following optimization techniques that can be performed while training and document your findings with a basic example code snippets: (5 points)

3.1 Tensor Creation (CPU vs GPU)

Obvervation :

On a 5000×5000 matrix multiply, the CPU took 2.6765 s and the Tesla T4 GPU took 0.0819 s, which is about a 32.7× speedup (2.6765 / 0.0819). I used torch.cuda.synchronize() before timing to avoid async timing errors on GPU. This shows that for large dense tensor ops, running the compute on the GPU is far faster; if CUDA isn’t available the code falls back to CPU.

3.2 Weight Initialization

Observation :

my findings are that for the 3-layer MLP I used different inits per layer: fc1 = Xavier-Uniform with ReLU gain, fc2 = Kaiming-Normal (ReLU), fc3 = Normal(0, 0.02). The weight stats show near-zero means and the expected stds: fc1 std 0.3051 vs the Xavier-ReLU expectation ≈ 0.316 for (fan_in=8, fan_out=32); fc2 std 0.2457 vs He expectation √(2/32) ≈ 0.25; fc3 std 0.0188 close to the target 0.02. No training was run as this is just the initialization check.

3.3 Activation Checkpointing

Observation :

I wrapped the two hidden ReLU blocks (fc1→ReLU and fc2→ReLU) with torch.utils.checkpoint only during training. I also set x.requires_grad_(True) to satisfy the checkpoint requirement. With this, the model saves activation memory in the forward pass and recomputes those activations in backward, so peak memory goes down but each step takes a bit longer. In eval mode (or if I set use_checkpoint=False) it runs normally without checkpointing. I kept the same inits from 3.2 (Xavier for fc1, He for fc2, small normal for fc3). The loop doesn't print anything—it's just a regular training pass with checkpointing turned on.

3.4 Gradient Accumulation

Observation :

With gradient accumulation, I combined multiple micro-batches before updating the weights, so I got one optimizer step per accumulation window instead of per batch. The output shows exactly that: one full window produced effective_step=1 | loss=0.171629 (this loss is the scaled micro-batch loss from the last batch in that window), and then a second line effective_step=2 | (final partial window), which means the dataset size wasn't divisible by accumulation_steps and we applied one final update with a partial window. This confirms accumulation is working: fewer optimizer steps, stable per-step loss reporting, and no increase to the micro-batch size (so peak memory stays the same).

3.5 Mixed Precision Training

Obvervation :

For mixed precision, I ran the training loop with AMP (autocast + GradScaler) and gradient accumulation. The console showed two things: (1) FutureWarning messages saying the torch.cuda.amp API is deprecated—so I should switch to torch.amp.GradScaler('cuda') and torch.amp.autocast('cuda') going forward—and (2) the step prints: effective_step=1 | loss=0.170615 followed by effective_step=2 | (final partial window). That means one full accumulation window produced an update with a scaled loss of ~0.1706, and then the dataloader ended with a partial window that triggered a final update. No NaNs or errors showed up, so the scaler handled precision fine.