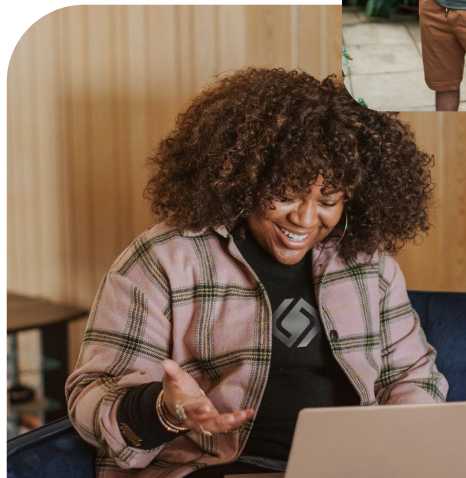**Bloom Institute of Technology**

AI For Developer Productivity

Welcome to class! Today, we will introduce the syllabus, discuss our approach to learning at BloomTech, and begin talking about transformers.

Syllabus

Objective:
Equip you with the required knowledge and experience to design, build, and deploy agents and multi-agent systems within your company.

Why:
So you can use agents in your workday to work faster and more efficiently.

How:
1. Teach you the skills required to master agent creation using **live instruction**.
2. Showcase and build implementations of these agents during **guided projects**.
3. Guide you in creating a **capstone multi-agent system** to bring back to your team and use.

Week	Sprint	Topics
1	RAG	LLMs, Open-source LLMs, Tracing LLMs with LangSmith
2		Retrieval-Augmented Generation, RAG Agent with Pinecone
3	Chaining	Chaining, LCEL, building functions/tools
4		Agents 101, Coder Agent using LangChain
5	Agents	Deploying agents using AWS-Bedrock
6		Building performative agents in production
7	Multi-Agent Systems	Building multi-agent systems with CrewAI and LangChain
8		Deploying and scaling multi-agent systems with LangGraph

Guided Projects

We Do, Then You Do:
- Watch us implement an agent.
- Add your own spin and build something useful for you.

Pass/Fail Criteria:
- Each project is graded as pass or fail.
- To pass, you must meet the minimum requirements set forth in the README for each project.
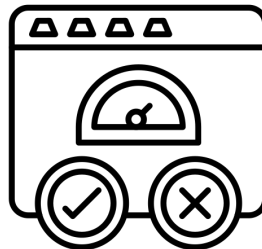
Stretch Goals:
- This is your opportunity to let your creativity shine and go beyond the minimum requirements.
- To get the most out of this course, make each project reusable within your company ecosystem.
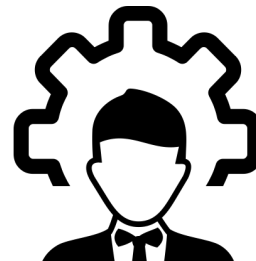
Technical Writer Agent
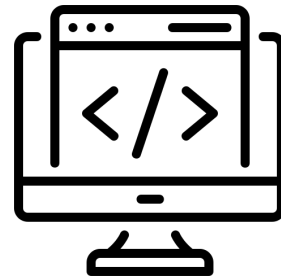
Developer Agent

QA Tester Agent

Managing Agent

Capstone

In the last two weeks of the course, you will **build a capstone project designed for direct implementation within your team at your company**. Our goal is to provide you with tangible code and guidance on creating practical solutions, not just theoretical knowledge about using AI agents.

Overview of Tasks

1. **Identify a Problem:** Work by yourself or collaborate with your team to find a problem suitable for automation or AI solutions.

2. **Design the Solution:** Plan the implementation, including the agent's tasks, training strategy, and performance metrics.

3. **Code a Prototype:** Develop a functional prototype, integrating the agent with necessary systems. The goal is MVP, not something with every single feature you want.

4. **Test in Your Ecosystem:** Deploy and test the prototype, gathering feedback and refining it based on real-world performance.

Course Schedule

Week/Days	Live Classes
Week 1: June 18-20	Live Instruction, Regular Tues/Thurs Schedule, Optional Office Hours: Wed, June 19.
Week 2: June 25-27	Guided Project, Regular Tues/Thurs Schedule, Optional Office Hours: June 26.
Summer Break	**No Class June 30- July 7 2024.**
Week 3: July 9-11	Live Instruction, Regular Tues/Thurs Schedule, Optional Office Hours: Wed, July 10.
Week 4: July 16-18	Guided Project, Regular Tues/Thurs Schedule, Optional Office Hours: July 17.
Week 5: July 23-25	Live Instruction, Regular Tues/Thurs Schedule, Optional Office Hours: Wed, July 24.

Week/Days	Live Classes
Week 6: Jul 30-Aug 1	Guided Project, Regular Tues/Thurs Schedule, Optional Office Hours: July 31.
Week 7: Aug 6-8	Live Instruction, Regular Tues/Thurs Schedule, Optional Office Hours: Wed, August 7.
Week 8: Aug 13-15	Guided Project, Regular Mon/Wed Schedule, Optional Office Hours: August 14.
Week 9: Aug 20-22	Capstone, Regular Tues/Thurs Schedule, Optional Office Hours: Wed, August 21.
Week 10: Aug 27-29	Capstone, Regular Mon/Wed Schedule, Optional Office Hours: Tues, August 28.

Demystifying LLMs

To make the most out of this course, understand that LLMs are powerful tools for next token generation, leveraging advanced statistical analysis beyond simple models.

Bloom Institute of Technology

Core Competencies

The student must demonstrate...

1. Understanding LLM Intuition (5 min)
2. Understanding of the various types of LLM messages (5 min)
3. Zero/one/few shot prompting (5 min)
4. Context filling prompts (10 min)
5. Building chat histories, putting these practices in use (10 min)

LLM Intuition

Text generation is a well-established task with effective prior techniques. LLMs are **fundamentally "just" next token predictors**, enhanced by complex statistical analysis rather than simple Markov chains. LLMs are inherently probabilistic and should be viewed as engineering tools, not magical solutions.

```python
from collections import defaultdict
import random

class MarkovChain:
  def __init__(self, text: str):
    self.next_word_ref = defaultdict(list)
    words = text.split(" ")
    for i, word in enumerate(words[:-1]):
      self.next_word_ref[word].append(words[i + 1])

  def generate_text(self, start_word: str ="", output_length: int = 100) -> str:
    if not start_word:
      start_word = random.choice(sorted(self.next_word_ref))
    output = [start_word]
    for i in range(output_length):
      if output[i] in self.next_word_ref:
        output.append(random.choice(self.next_word_ref[output[i]]))
      else:
        # Edge case for last word, no next word in corpus so pick a random one
        output.append(random.choice(sorted(self.next_word_ref)))
    return " ".join(output)
```

Test and access the full [Markov Chain code here](#).

System vs. User Prompts

User Messages: Contain the task at hand and are sent by the user.

System Messages: Sent by the developer and carry more weight, guiding the LLM on how to respond.

Impact: System messages influence the behavior more significantly than user messages.

Best Practices: Combine both message types for more accurate LLM responses.

```python
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_openai import ChatOpenAI
from langchain_community.llms.ollama import Ollama

llm = ChatOpenAI(model="gpt-3.5-turbo")
# llm = Ollama(model="llama2") # for Ollama users

#Step 1
text = "What would be a good company name for a company that
makes colorful socks?"
messages = [HumanMessage(content=text)]

# Step 3
# system_text="You are a sarcastic bot that gives helpful advice"
# messages = [SystemMessage(content=system_text),
HumanMessage(content=text)]

# Step 2
llm.invoke(messages)
```

Zero shot, one shot, few shot...

Zero/one/few shot prompts involve adding examples to prompts to achieve better results. This simple yet effective technique proves useful in various aspects down the road.

Zero shot:

```python
# Initialize Langchain with OpenAI's
GPT-3.5-turbo model
llm = OpenAI(api_key=openai_api_key,
model="gpt-3.5-turbo")

# Define a function to generate a
response using zero-shot learning
def generate_response_zero_shot(review):
  prompt = f"As a customer support
  representative, write a response to
  the following review:\n\n{review}\n\n
  Response:"
  response = llm(prompt)
  return response
```

Few shot:

```python
few_shot_prompt = """
Few-Shot Examples:
Review: "I had a great experience with your
product, but I encountered a small issue."

Response: "Thank you for your feedback! We're
glad to hear you had a great experience
overall. Please let us know more about the
issue you encountered so we can assist you
further."

Review: "Your product didn't meet my
expectations."
Response: "We're sorry to hear that our
product didn't meet your expectations. Could
you please provide more details about your
experience so we can address your concerns."
"""
```

Context Within Prompts

Give a role:

```
# Define the system prompt message
system_prompt_message =
HumanMessage(content="""
System Prompt: As a technical support
specialist, provide clear and concise
instructions or solutions tailored to the
user's problem.
""")


# Define the user's problem message
user_problem_message =
HumanMessage(content="""
User's Problem: My computer is running slow
and freezing frequently. What should I do?
""")


# Generate responses using the ChatOpenAI object
response_with_role_playing = openai(
[system_prompt_message, user_problem_message])
```

Decorate a prompt:

```
# Define prompts with and without SQL schema
prompt_with_schema = """
System Prompt: Write a SQL query to retrieve
all employees from the 'employees' table who
have a salary greater than $50,000.

SQL Schema:
Table: employees
Columns: id (INTEGER), name (TEXT), salary
(INTEGER)
"""


prompt_without_schema = """
System Prompt: Write a SQL query to retrieve
all employees who have a salary greater than
$50,000.
"""


# Generate responses using OpenAI
response_with_schema = openai(
prompt_with_schema)
```

Saving Chat History

LLMs leverage chat histories to maintain context, though older context might be discarded due to size constraints or reduced relevance. These limitations can be mitigated through prompt engineering.

Create system prompt and initialize LLM:

```python
from langchain_core.messages import AIMessage,
HumanMessage
from langchain_core.prompts import
ChatPromptTemplate, MessagesPlaceholder
from langchain_openai import ChatOpenAI

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are a helpful assistant.
            Answer all questions to the best
            of your ability.",
        ),

MessagesPlaceholder(variable_name="messages"),
    ])

  chat = ChatOpenAI()
```

Save messages and continue conversation:

```python
messages =  [
        HumanMessage(content="Translate this
        sentence from English to French: I love
        programming."),AIMessage(content="J'adore
        la programmation."), HumanMessage(
        content="What did you just say?")
    ]

result = chat.invoke(messages)
messages.append(result)
print(messages)

# The code below continues the conversation!
while True:
        prompt = input("Prompt: ")
        messages.append(HumanMessage(
        content=prompt))
        result = chat.invoke(messages)

        messages.append(result)
        print(messages)
```

Hands-on Homework.

Develop a conversation between two LLMs discussing the process of building a CRUD app, leveraging different perspectives and techniques to maintain chat history effectively.

Consider approaches such as summarization and proactive context management to prevent the conversation from going stale.