

# LLM Agentic Tutorial Examples

## 1. Basic LLM call: Establish LLM API Connection

```
In [1]: import requests
import json

def call_llm(messages, max_tokens=250, temperature=0.1):
    """Connect to LLM API and return the response."""
    url = "http://cs-513-ml003:3000/chat"
    headers = {"X-API-Key": "AccGPT-API", "Content-Type": "application/json"}
    payload = {
        "model": "llama3-70b-8192",
        "temperature": temperature,
        "messages": messages,
        "max_tokens": max_tokens
    }
    response = requests.post(url, headers=headers, json=payload)
    response.raise_for_status()
    return response.json()

system_prompt = "You are an accelerator physics chatbot. Respond short and c
user_query = "What is CERN?"

messages = [
    {"role": "system", "content": system_prompt},
    {"role": "user", "content": f"{user_query}"}
]

response = call_llm(messages, max_tokens=50)
output = response.get("choices", [{}])[0].get("message", {}).get("content",
print("LLM Response:\n", output)
```

LLM Response:

CERN (European Organization for Nuclear Research) is a research organization that operates the Large Hadron Collider (LHC), a powerful particle accelerator, to study the fundamental nature of matter and the universe.

## 2. Simple ReAct Agent

Just iteratively call LLMs

```
In [2]: def react_agent(task_description):
    # Step 1: Reasoning
    print("Agent: Thinking about the task...")
    reasoning_prompt = f"Explain step-by-step (in 2 steps) how to approach t
    reasoning_resp = call_llm([{"role": "user", "content": reasoning_prompt}
    reasoning_text = reasoning_resp.get("choices", [{}])[0].get("message", {
```

```

# Step 2: Execute the first step
print("Agent: Executing the first step...")
first_step_prompt = (
    f"Based on the reasoning below, what is the first step to solve the
    f"Reasoning: {reasoning_text}\n"
    "Now, perform that first step."
)
first_step_resp = call_llm([{"role": "user", "content": first_step_prompt}], max_tokens=100)
first_step_text = first_step_resp.get("choices", [{}])[0].get("message", "")

# Step 3: Execute the second step
print("Agent: Executing the second step...")
second_step_prompt = (
    f"Based on the reasoning and the result of the first step below, what is the next step?
    f"Reasoning: {reasoning_text}\n"
    f"First step result: {first_step_text}\n"
    "Now, perform that second step."
)
second_step_resp = call_llm([{"role": "user", "content": second_step_prompt}], max_tokens=100)
second_step_text = second_step_resp.get("choices", [{}])[0].get("message", "")

# Step 4: Final Answer
print("Agent: Formulating final answer...")
final_prompt = (
    f"Using the result from the second step, provide a concise final answer to the task.
    f"Second step result: {second_step_text}"
)
final_resp = call_llm([{"role": "user", "content": final_prompt}], max_tokens=100)
final_text = final_resp.get("choices", [{}])[0].get("message", {}).get("text", "")

return {
    "reasoning": reasoning_text,
    "first_step": first_step_text,
    "second_step": second_step_text,
    "final_answer": final_text
}

# Example
task_description = "How do cyclotrons work?"
agent_output = react_agent(task_description)

for key, value in agent_output.items():
    print("\n-----")
    print(f"{key.capitalize()}: \n{value}\n{'-'*60}")

```

Agent: Thinking about the task...  
Agent: Executing the first step...  
Agent: Executing the second step...  
Agent: Formulating final answer...

-----  
Reasoning:

Here's a 2-step approach to understanding how cyclotrons work:

**\*\*Step 1: Understand the Basic Principle\*\***

A cyclotron is a type of particle accelerator that uses a magnetic field to accelerate charged particles, such as protons or ions, to high speeds. The basic principle of a cyclotron is based on the concept of circular motion. Imagine a charged particle moving in a circular path under the influence of a magnetic field. As the particle moves in a circle, it experiences a constant force perpendicular to its direction of motion, which causes it to accelerate. The key insight is that the magnetic field can be designed to keep the particle on a circular path while increasing its speed.

**\*\*Step 2: Break Down the Cyclotron's Components and Operation\*\***

To understand how a cyclotron works, break down its components and operation into the following steps:

\* **\*\*Dee electrodes\*\***: The cyclotron consists of two D-shaped electrodes (called "dees") that are placed in a vacuum chamber. The dees are connected to a high-voltage alternating current (AC) power source.

\* **\*\*Magnetic field\*\***: A strong magnetic field is applied perpendicular to the plane of the dees. This field causes the charged particles to move in a

-----  
First\_step:

Let's perform the first step: **\*\*Understand the Basic Principle\*\***.

A cyclotron is a type of particle accelerator that uses a magnetic field to accelerate charged particles, such as protons or ions, to high speeds. The basic principle of a cyclotron is based on the concept of circular motion. Imagine a charged particle moving in a circular path under the influence of a magnetic field. As the particle moves in a circle, it experiences a constant force perpendicular to its direction of motion, which causes it to accelerate. The key insight is that the magnetic field can be designed to keep the particle on a circular path while increasing its speed.

In essence, the basic principle of a cyclotron relies on the combination of circular motion and a magnetic field to accelerate charged particles.

-----  
Second\_step:

Now that we have a solid understanding of the basic principle of a cyclotron, let's break down its components and operation to gain a deeper understanding of how it works.

**\*\*Step 2: Break Down the Cyclotron's Components and Operation\*\***

To understand how a cyclotron works, let's examine its components and operation:

\* **Dee electrodes**: The cyclotron consists of two D-shaped electrodes (called "dees") that are placed in a vacuum chamber. The dees are connected to a high-voltage alternating current (AC) power source. This AC power source creates an electric field that oscillates at a specific frequency, which is crucial for the acceleration process.

\* **Magnetic field**: A strong magnetic field is applied perpendicular to the plane of the dees. This field causes the charged particles to move in a circular path, as we understood from the basic principle.

\* **Particle acceleration**: When a charged particle, such as a proton or ion, is injected into the cyclotron, it is attracted to one of the dees due to the electric field. As the particle approaches the dee, it gains kinetic energy and starts moving in a circular path due to the magnetic field. When the particle reaches the gap between the dees, it is accelerated again.

-----  
-----  
Final\_answer:  
Here is a concise final answer to the task:

A cyclotron works by using a combination of electric and magnetic fields to accelerate charged particles, such as protons or ions, in a circular path. The device consists of two D-shaped electrodes (dees) connected to a high-voltage AC power source, which creates an oscillating electric field, and a strong magnetic field applied perpendicular to the dees. As a charged particle is injected, it is attracted to one of the dees, gains kinetic energy, and moves in a circular path due to the magnetic field. The particle accelerates as it passes through the gap between the dees, gaining more energy with each cycle, until it reaches the desired energy level.

## Agent Loop for Autonomous Math Tasks

Provide the LLM with possible functions ("tools") to call on its own.

```
In [3]: # --- Extract Answer from Response ---
def extract_answer(response):
    """Extracts the answer text from the LLM response."""
    if "choices" in response and len(response["choices"]) > 0:
        return response["choices"][0]["message"]["content"]
    return "No answer received."

# --- Math Functions for Agent ---
def function_add(args):
    return str(args[0] + args[1])

def function_subtract(args):
    return str(args[0] - args[1])
```

```

def function_multiply(args):
    return str(args[0] * args[1])

def function_divide(args):
    return str(args[0] / args[1]) if args[1] != 0 else "Error: Division by zero"

def function_exponentiate4(args):
    return str(args[0] ** 4)

# Mapping function names to their implementations.
available_functions = {
    "function_add": function_add,
    "function_subtract": function_subtract,
    "function_multiply": function_multiply,
    "function_divide": function_divide,
    "function_exponentiate4": function_exponentiate4,
}

# --- Autonomous Math Agent ---
def agent_loop(max_iterations=10):
    """
    Autonomous math agent loop:
    - Instructs the LLM about the task and available math functions.
    - Receives JSON instructions on which function to call next.
    - Executes the function, updates the conversation history, and repeats
    """
    system_message = (
        "You are an autonomous math solver tasked with computing the following expression:\n"
        "((10 + 5) * (2^4) - (30 / 2))\n"
        "You have access to these functions:\n"
        "1. function_add([a, b]) -> returns a + b\n"
        "2. function_subtract([a, b]) -> returns a - b\n"
        "3. function_multiply([a, b]) -> returns a * b\n"
        "4. function_divide([a, b]) -> returns a / b\n"
        "5. function_exponentiate4([x]) -> returns x raised to the power of 4\n"
        "\n"
        "IMPORTANT: Do not include any extra commentary, multiple JSON objects, or text before or after the JSON object.\n"
        "You must output exactly one valid JSON object. When calling a function, the JSON object must be:\n"
        '{"action": "call_function", "function": "<function_name>", "args": [a, b]}\n"
        "When you have reached your final answer, output exactly one JSON object with the following structure:\n"
        '{"action": "finish", "final_answer": "<result>"}\n'
        "Only output valid JSON with no additional text."
    )

    conversation = [{"role": "system", "content": system_message}]

    for iteration in range(max_iterations):
        response = call_llm(conversation, max_tokens=150)
        decision_text = extract_answer(response)
        print(f"LLM Decision (iteration {iteration+1}): {decision_text}")

        try:
            decision = json.loads(decision_text)
        except json.JSONDecodeError as e:
            print("Error parsing JSON decision:", e)
            break

```

```

    action = decision.get("action")

    if action == "call_function":
        func_name = decision.get("function")
        args = decision.get("args")
        if func_name in available_functions:
            print(f"Calling function: {func_name} with args: {args}")
            result = available_functions[func_name](args)
            print(f"Result from {func_name}: {result}")

            conversation.append({"role": "function", "name": func_name,
                                "content": result})
            conversation.append({"role": "user", "content": f"Function {func_name} returned {result}"})
        else:
            print(f"Error: Function {func_name} is not available.")
            break

    elif action == "finish":
        print("\nFinal Answer:", decision.get("final_answer"))
        return decision.get("final_answer")
    else:
        print("Unrecognized action. Ending agent loop.")
        break

print("Max iterations reached without a final answer.")
return None

# --- Example Call ---
print("\nStarting autonomous math agent...\n")
agent_loop()

```

Starting autonomous math agent...

```

LLM Decision (iteration 1): {"action": "call_function", "function": "function_add", "args": [10, 5]}
Calling function: function_add with args: [10, 5]
Result from function_add: 15
LLM Decision (iteration 2): {"action": "call_function", "function": "function_exponentiate4", "args": [2]}
Calling function: function_exponentiate4 with args: [2]
Result from function_exponentiate4: 16
LLM Decision (iteration 3): {"action": "call_function", "function": "function_multiply", "args": [15, 16]}
Calling function: function_multiply with args: [15, 16]
Result from function_multiply: 240
LLM Decision (iteration 4): {"action": "call_function", "function": "function_divide", "args": [30, 2]}
Calling function: function_divide with args: [30, 2]
Result from function_divide: 15.0
LLM Decision (iteration 5): {"action": "call_function", "function": "function_subtract", "args": [240, 15.0]}
Calling function: function_subtract with args: [240, 15.0]
Result from function_subtract: 225.0
LLM Decision (iteration 6): {"action": "finish", "final_answer": "225.0"}

```

Final Answer: 225.0

Out[3]: '225.0'

## Compare response to plain LLM

```
In [4]: query = "Calculate: ((10 + 5) * (2^4) - (30 / 2))"

messages = [
    {"role": "system", "content": system_prompt},
    {"role": "user", "content": f"{query}"}
]

response = call_llm(messages, max_tokens=250, temperature=0.01)
output = response.get("choices", [{}])[0].get("message", {}).get("content",)
print("LLM (without RAG) Response:\n", output)
```

LLM (without RAG) Response:

Let's calculate:

$$(10 + 5) = 15$$

$$(2^4) = 16$$

$$(15 * 16) = 240$$

$$(30 / 2) = 15$$

$$\text{So, } ((10 + 5) * (2^4) - (30 / 2)) = 240 - 15 = 225$$

There are also specialized libraries which handle agentic tasks simpler. E.g. Langchain, llama-index, ...