Building a Simple Agent Framework 2

Now, we are going to put the components together into a reusable agent class. This class will encapsulate the GAME components and provide a simple interface for running the agent loop. The agent will be responsible for constructing prompts, executing actions, and managing memory. We can create different agents simply by changing the goals, actions, and environment without modifying the core loop.

Let's take a look at our agent class:

```
class Agent:
    def __init__(self,
                 goals: List[Goal],
                 agent_language: AgentLanguage,
                 action_registry: ActionRegistry,
                 generate_response: Callable[[Prompt], str],
                 environment: Environment):
        """
        Initialize an agent with its core GAME components
        """
        self.goals = goals
        self.generate_response = generate_response
        self.agent_language = agent_language
        self.actions = action_registry
        self.environment = environment

    def construct_prompt(self, goals: List[Goal], memory: Memory,
actions: ActionRegistry) -> Prompt:
        """Build prompt with memory context"""
        return self.agent_language.construct_prompt(
            actions=actions.get_actions(),
            environment=self.environment,
            goals=goals,
            memory=memory
        )

    def get_action(self, response):
        invocation = self.agent_language.parse_response(response)
        action = self.actions.get_action(invocation["tool"])
        return action, invocation
```

```
    def should_terminate(self, response: str) -> bool:
        action_def, _ = self.get_action(response)
        return action_def.terminal

    def set_current_task(self, memory: Memory, task: str):
        memory.add_memory({"type": "user", "content": task})

    def update_memory(self, memory: Memory, response: str, result:
dict):
        """
        Update memory with the agent's decision and the
environment's response.
        """
        new_memories = [
            {"type": "assistant", "content": response},
            {"type": "user", "content": json.dumps(result)}
        ]
        for m in new_memories:
            memory.add_memory(m)

    def prompt_llm_for_action(self, full_prompt: Prompt) -> str:
        response = self.generate_response(full_prompt)
        return response

    def run(self, user_input: str, memory=None, max_iterations: int
= 50) -> Memory:
        """
        Execute the GAME loop for this agent with a maximum
iteration limit.
        """
        memory = memory or Memory()
        self.set_current_task(memory, user_input)

        for _ in range(max_iterations):
            # Construct a prompt that includes the Goals, Actions,
and the current Memory
            prompt = self.construct_prompt(self.goals, memory,
self.actions)

            print("Agent thinking...")
            # Generate a response from the agent
            response = self.prompt_llm_for_action(prompt)
            print(f"Agent Decision: {response}")

            # Determine which action the agent wants to execute
            action, invocation = self.get_action(response)
```

```
            # Execute the action in the environment
            result = self.environment.execute_action(action,
invocation["args"])
            print(f"Action Result: {result}")

            # Update the agent's memory with information about what
happened
            self.update_memory(memory, response, result)

            # Check if the agent has decided to terminate
            if self.should_terminate(response):
                break

        return memory
```

Now, let's walk through how the GAME components work together in this agent architecture, explaining each part of agent loop.

Step 1: Constructing the Prompt

When the agent loop begins, it first constructs a prompt using the `construct_prompt` method:

```
def construct_prompt(self, goals: List[Goal], memory: Memory,
actions: ActionRegistry) -> Prompt:
    """Build prompt with memory context"""
    return self.agent_language.construct_prompt(
        actions=actions.get_actions(),
        environment=self.environment,
        goals=goals,
        memory=memory
    )
```

This method leverages the `AgentLanguage` component to build a structured prompt containing:

- The agent's goals (what it's trying to accomplish)
- Available actions (tools the agent can use)
- Current memory context (conversation history and relevant information)

- Environment details (constraints and context for operation)

We are going to discuss the `AgentLanguage` in more detail later. For now, what you need to know is that it is responsible for formatting the prompt that is sent to the LLM and parsing the response from the LLM. Most of the time, we are going to use function calling, so the parsing will just be reading the returned tool calls. However, the `AgentLanguage` can be changed to allow us to also take the same agent and implement it without function calling.

Step 2: Generating a Response

Next, the agent sends this prompt to the language model:

```
def prompt_llm_for_action(self, full_prompt: Prompt) -> str:
    response = self.generate_response(full_prompt)
    return response
```

The `generate_response` function is a simple python function provided during initialization. This abstraction allows the framework to work with different language models without changing the core loop. We will use LiteLLM to call the LLM, but you could easily swap this out for any other LLM provider.

Step 3: Parsing the Response

Once the language model returns a response, the agent parses it to identify the intended action. The parsing will generally be just getting the tool calls from the response, however the agent language gets to decide how this is done. Once the response is parsed, the agent can look up the action in the `ActionRegistry`:

```
def get_action(self, response):
    invocation = self.agent_language.parse_response(response)
    action = self.actions.get_action(invocation["tool"])
    return action, invocation
```

The `action` is the interface definition of what the agent "can" do. The `invocation` is the specific parameters that the agent has chosen to use for this action. The `ActionRegistry` allows the agent to look up the action by name, and the `invocation` provides the arguments needed to execute it. We could also add validation at this step to ensure that the invocation parameters match the action's expected parameters.

Step 4: Executing the Action

The agent then executes the chosen action in the environment:

```
# Execute the action in the environment
result = self.environment.execute_action(action, invocation["args"])
```

The `Environment` handles the actual execution of the action, which might involve:

- Making API calls
- Reading/writing files
- Querying databases
- Processing data

Actions are defined in the `ActionRegistry` but executed within the context of the `Environment`, which provides access to resources and handles the mechanics of execution.

Step 5: Updating Memory

After execution, the agent updates its memory with both its decision and the result:

```
def update_memory(self, memory: Memory, response: str, result:
dict):
    """
    Update memory with the agent's decision and the environment's
response.
    """
    new_memories = [
        {"type": "assistant", "content": response},
        {"type": "user", "content": json.dumps(result)}
```

```
        ]
        for m in new_memories:
            memory.add_memory(m)
```

This creates a continuous record of the agent's reasoning and actions, which becomes part of the context of future loop iterations. The memory serves both as a record of past actions and as context for future prompt construction.

Step 6: Termination Check

Finally, the agent checks if it should terminate the loop:

```
def should_terminate(self, response: str) -> bool:
    action_def, _ = self.get_action(response)
    return action_def.terminal
```

This allows certain actions (like a "terminate" action) to signal that the agent has finished its work.

The Flow of Information Through the Loop

To better understand how these components interact, let's trace how information flows through a single iteration of the loop:

1. The `Memory` provides context about what the user has asked the agent to do and past decisions and results from the agent loop
2. The `Goals` define what the agent is trying to accomplish and rules on how to accomplish it
3. The `ActionRegistry` defines what the agent can do and helps lookup the action to execute by name
4. The `AgentLanguage` formats Memory, Actions, and Goals into a prompt for the LLM
5. The LLM generates a response choosing an action
6. The `AgentLanguage` parses the response into an action invocation, which will typically be extracted from tool calls

7. The `Environment` executes the action with the given arguments
8. The result is stored back in `Memory`
9. The loop repeats with the updated memory until the agent calls a terminal tool or reaches the maximum number of iterations

Creating Specialized Agents

The beauty of this framework is that we can create entirely different agents by changing the GAME components without modifying the core loop:

```
# A research agent
research_agent = Agent(
    goals=[Goal("Find and summarize information on topic X")],
    agent_language=ResearchLanguage(),
    action_registry=ActionRegistry([SearchAction(),
SummarizeAction(), ...]),
    generate_response=openai_call,
    environment=WebEnvironment()
)

# A coding agent
coding_agent = Agent(
    goals=[Goal("Write and debug Python code for task Y")],
    agent_language=CodingLanguage(),
    action_registry=ActionRegistry([WriteCodeAction(),
TestCodeAction(), ...]),
    generate_response=anthropic_call,
    environment=DevEnvironment()
)
```

Each agent operates using the same fundamental loop but exhibits completely different behaviors based on its GAME components.