Mesa-LLM

Colin FRISCH - proposal refining

1. Paradigm/reasoning orchestration

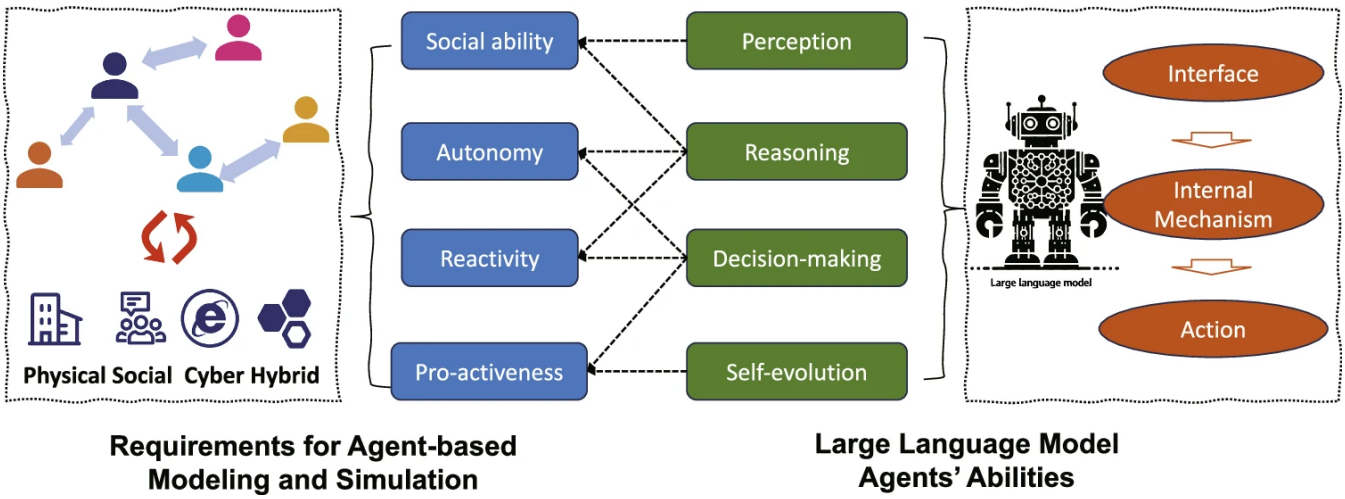

Fig. 1 - From: Large language models empowered agent-based modeling and simulation: a survey and perspectives[1]

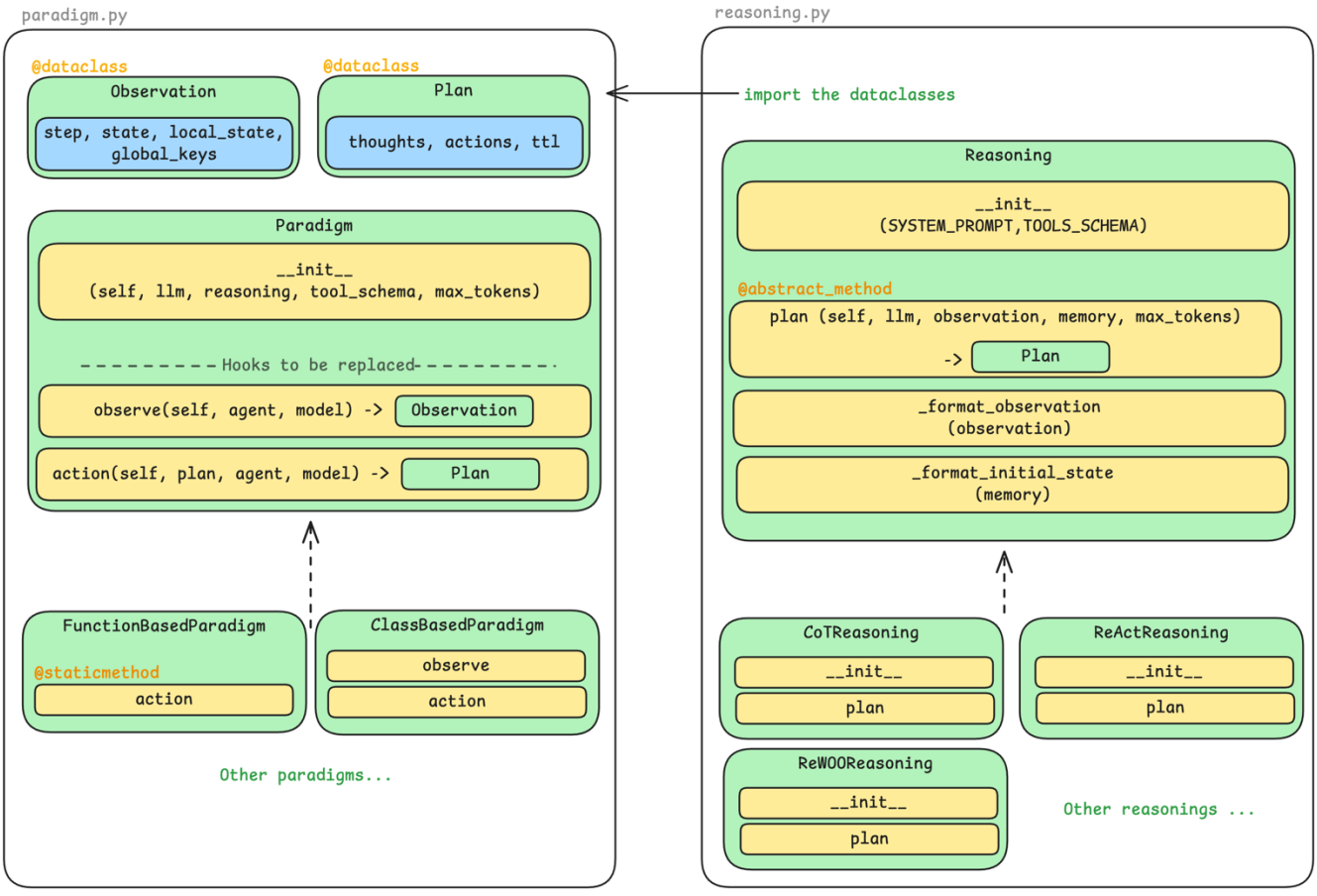

Figure 2 : Zoom on paradigm and reasoning -

paradigm.py

For modularity purposes, I still think the class separation is an intersting approach between Paradigm and Reasoning, as well as them being parents to the implementation of respectively paradigms (`FunctionBasedParadigm`, `ClassBasedParadigm`) and reasonings (`CoTReasoning`, `ReActReasoning`, etc.).

What I didn't developp much in the proposal and would like to insist on here is the fact that the paradigm is really the where the behavior of the LLMAgent is orchestrated and not only the way the LLM interracts with the environment. That is why, I think that it would also be usful to add an `observe()` method to the paradigm class as well as a new dataclass `Observation(step, self_state, local_state, global_keys)`.

```
paradigm.py
├── class Observation                                   ← dataclass
├── class Plan                                          ← dataclass
│
├── class Paradigm
│   ├── __init__()                                      ← init logic
│   ├── step()                                          ← main loop
│   ├── observe()                                       ← overridden
│   └── action()                                        ← overridden
│
├── class ClassBasedParadigm(Paradigm)
│   ├── observe()                                       ← custom class logic
│   └── action()                                        ← custom class logic
│
└── class FunctionBasedParadigm(Paradigm)
    @static_method
    └── action()                                        ← custom function logic
```

reasoning.py

After a bit more work and research, I think that the best way to implement reasoning is to view it only as planning. So, in order to have a unique but also scalable (for future improvements) format for the decisions of the LLM, I would like to introduce another dataclass : `Plan(thoughts, actions, ttl – time to live)`. The action attribute here is made to be a json serialized list of functions for tool usage.

```
reasoning.py
├── class Reasoning
│   └── plan()                                    ← base plan method
│   └── SYSTEM_PROMPT
│   └── SYSTEM_PROMPT
│
├── class CoTReasoning(Reasoning)
│   ├── __init__()                                ← prompt setup
│   └── plan()                                    ← chain-of-thought planning
│
├── class ReActReasoning(Reasoning)
│   └── plan()                                    ← ReAct loop
│   └── MAX_TURNS                                 ← max reasoning steps
│
└── class ReWOOReasoning(Reasoning)
    └── plan()                                    ← ReWOO planner
```

I suggested a complete improved implementation for these modules that I pushed in my abm-mesa-experimentations github repository in the *experimentations/mesa-llm subsection*.

2. Module LLM and batching

Modularity improvements :

During a personal project, I experimented a few LLM gateways. I found OpenRouter to be the most efficient and useful. I maintain that a good first step is to do as told in the proposal and test `LLMAgent` with only ChatGPT or/and Claude, but it could be a real advantage to pursue with OpenRouter as :

- With a single, OpenAI-compatible API, you can call over 300 models without juggling multiple keys or SDKs
- Its deployed on edge servers around the world → OpenRouter adds just ~30 ms of latency, so users get near-instant responses even at scale (+smart detour feature = always available)
- Built-in, real-time analytics (PostHog integration) = track latency, cost, and error rates without having to build separate monitoring tools

I think that it would be a good addition side by side with the huggingface pipeline for local model running.

Batching detailed :

The objective would be to make batching transparent to agent authors : agents still call `self.llm.generate(prompt)`. The batch layer lives under `ModuleLLM`. Also, it would be ideal to integrate the remaining inference parameter by the author in this module. This is a code architecture of how it could turn out, and a coded example is in the Appendix.

```
module_llm.py
├── class ModuleLLM
│   ├── async generate(prompt, **kwargs) -> str
│   ├── async generate_batch(prompts: List[str], params: Dict[str, Any]) -> List[str]
│   ├── async _init_pipeline() -> Any                ← client init
│   ├── async _call_hf(prompts, params)              ← HF batch call
│   └── async _call_openrouter(prompts, params)      ← OpenRouter batch call
│
│   └── _batch : BatchProcessor                      ← shared per config key
│   └── _pipeline : Optional[Client]                 ← lazy client instance
│   └── _config_key : Tuple                          ← provider, model_name, api_key…
│
└── class BatchProcessor
    ├── async enqueue(prompt, fut, params) -> None   ← adds prompt to queue
    ├── async _flush_loop()                          ← runs continuously
    └── async _flush_once()                          ← runs when flush triggered

    └── _queue : List[Tuple[prompt, future, params]] ← request queue
    └── _flush_event : asyncio.Event()               ← triggers flushing
    └── _task : asyncio.Task                         ← flush loop runner
```

`ModuleLLM` wraps the parameters into a `(prompt, Future, params)` tuple and enqueues it in the shared `BatchProcessor`.

`BatchProcessor` (`_flush_loop`) waits for either: the queue hitting max_batch (an explicit trigger), or the flush_interval timeout. When triggered, it calls its internal `_flush_once()`, which grabs all queued tuples at once, extracts a list of prompts and the shared paramsalls back to `ModuleLLM.generate_batch(prompts, params)`.

3. References

[1] Gao C., Lan X., Li N. *et al.* Large language models empowered agent-based modeling and simulation: a survey and perspectives. *Humanit Soc Sci Commun* 11, 1259 (2024). https://doi.org/10.1057/s41599-024-03611-3