AgentChain: A framework for building and chaining LLM agents

Abdelhadi Azzouni
hadiazzouni@heycli.com

Abstract

In this paper, we first introduce the Agent Oriented Programming (AOP) as a new paradigm to build software using LLM agents. AOP is mainly inspired by (and an adaptation of) Object Oriented Programming (OOP) and the Actor Model to LLM agents [1]. Then we propose AgentChain as a simple implementation framework for AOP. AgentChain facilitates working with LLM-based agents through modularity and composability.

1 Introduction

Powerful LLMs, such as GPT-4, are very good at reasoning over text, which opens what seems to be endless possibilities when hooked to code. For example, you can ask GPT-4 to scan a web page for vulnerabilities and propose fixes. You can then hook the output to some code that executes these fixes.

LLMs should not be looked at as databases, where data is stored at training time and retrieved at inference time. Instead, LLMs should be regarded as reasoning engines that need external tools to reliably achieve objectives. For example, LLMs may hallucinate WW2 events or mess up Gmail API calls. However, given access to Wikipedia and the Gmail API documentation, LLMs have higher chance of successfully responding to questions or executing tasks.

With that in mind, we suggest that a new Software Paradigm is emerging and will be centered around LLM agents. An LLM agent is a software entity, or a programme, that uses an LLM for reasoning and a set of tools to execute tasks. In this paper, we attempt to define this new paradigm.

2 State of the art

Although some tools have been developed recently [2], these tools do not follow any specific framework, which makes their code relatively hard to understand and extend.

LangChain [3] offers ways to create agents, however, although LangChain is relatively new, it is already bloated. It intermingles prompt engineering and

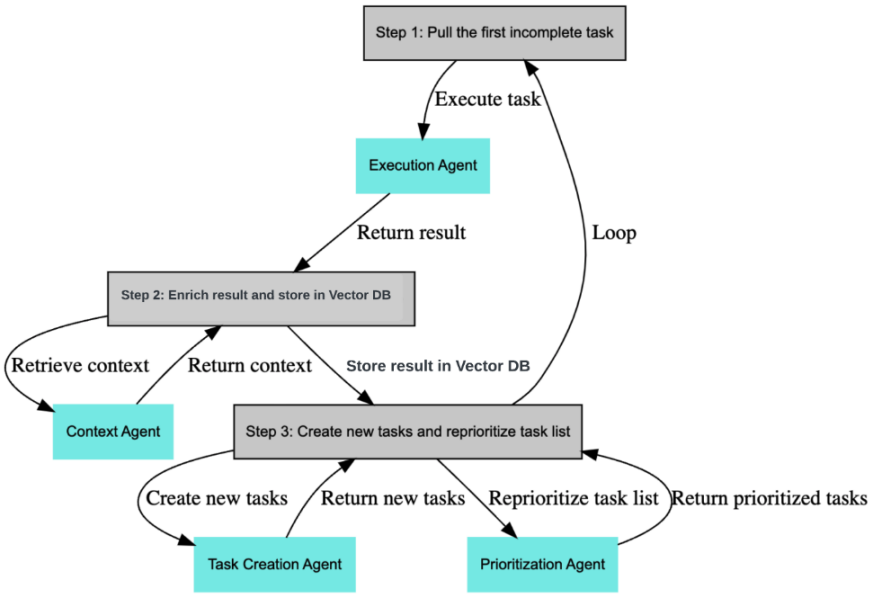

Figure 1: BabyAGI workflow

agent design elements without a coherent framework, which is leading to ever growing complexity of the library. In addition, LangChain does not offer a framework for composing multiple agents together.

For building individual agents, we can summarise the main existing approaches as follows:

- ReAcT [4]: not really an LLM agent framework but we mention it here as authors usually mix it with LLM agents. The Reason and Act (ReAct) framework is a few shots prompting techniques that teaches the LLM to reason before answer.
- Plan and execute (AutoGPT, BabyAGI [2]): both projects lack a rigorous definition of an agent and seem to mix agents with LLM prompts. They work by first planning the objective into tasks then executing tasks one by one. However, AutoGPT and BabyAGI do not follow a clear framework for planning and aggregating task results to achieve the final objective. They can run indefinitely, execute redundant tasks or completely diverge from the initial objective. They also do not handle recursion and aggregation (similar to map reduce, as we will explain later) or chaining multiple agents.

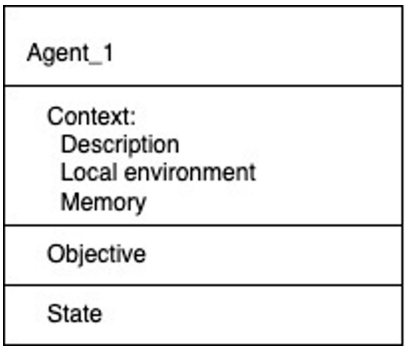

Figure 2: Agent class

3 Agent Oriented Programming

3.1 Definitions

We define the following entities:

- **Agent**: a software entity that is defined by its *context, objective and state*.
- **Context**: text that describe the agent, its environment and its memories
- **Agent description**: text that describes the capabilities and tools available to the agent
- **Local environment**: a collection of textual passages that describe parts of the global environment that are relevant to the agent. The environment (global and local) evolves over time.
- **Agent's memory**: a collection of textual passages that record tasks, results and other information that the agent encounters during its lifetime.
- **Objective**: text that describes either:
  - A task that the agent needs to perform (usually a high level, complex task). Eg, send prospecting emails to clients
  - or a result that the agent needs to achieve, Eg. emails sent to clients

  An agent can only have one objective at a given time.
- **State**: text (word) that indicates if the agent is: running, idle (stopped) or terminated.

  An agent has only one state at a given time.

Note: the only unique attribute of an agent is its description. Two agents can have the same objective, global context etc but two agents cannot have the same description. If two agents have the same description then they are considered the same agent.

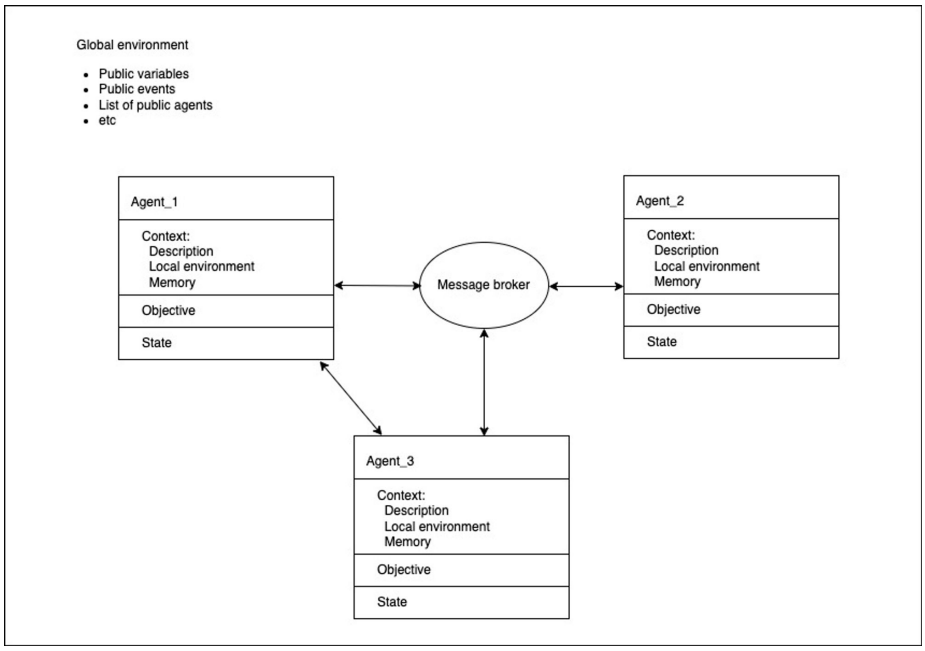

Figure 3: Agents exchange messages via a message broker

- **Global environment**: a collection of textual passages that contain information to be shared to all agents. For example:
  - Public variables
  - Public events
  - List of public agents
  - etc
- **Message**: textual passages exchanged between agents, synchronously or asynchronously via a message broker.
- **Task**: a textual description of one or more actions to be performed by an agent. A task can be:
  - **Atomic**: task that is meant to be executed with a low level tool without need for LLM intervention. An atomic task can be a command line, line of code, function, set of functions etc.., as long as an LLM intervention is not needed to guide the execution or check intermediary outputs.
  - **Composite**: (we also call them "Objectives") complex task that needs planning to decompose it into a sequence of lower level tasks.

- **Organisation**: set of 2 or more agents related by a common objective, a common owner or some other common attribute.
- **Tool**: a software programme used to execute tasks. There are 2 types of tools:
  - **Universal tools**: tools that can execute a wide range of tasks. For example, the command line terminal is a universal tool. Python interpreter is a universal tool, too. However, Gmail is not a universal tool as it can only send, receive and edit emails.
  - **Specialised tools**: tools that are designed for specific tasks. For example, Gmail is an email tool. Airbnb API is a specialised tool etc

Notes:

1. LLMs are specialised tools, as they can only output text from an input text. A function that calls an LLM is a specialised tool as well.
2. Tools can use other tools, just like programmes in general can use other programmes. For example, the python interpreter can use the Airbnb API.
3. LLM agents, as described above, are universal tools. An LLM agent can use other LLM agents as tools.

4 AgentChain

We borrow concepts from OOP, distributed systems and the Agile methodology literature to design AgentChain.

4.1 When to create an agent vs a function

A reasonable question to ask when working with LLMs is: should a given task be implemented as a function that uses an LLM or should it be implemented as an agent. We suggest the following rule of thumb: If the objective requires cognitive planning into tasks and iterative reasoning over intermediary results, then an agent is more suitable. In all other cases, simple functions are simpler and enough. For example:

Objective1: *"Parse a text file and extract all references to food."*. Available tools: file system. This objective could be achieved in one shot by the LLM, so no need to build an agent for it. In this case, the function using the LLM should be considered as a tool not an agent.

Objective2: *"Create an EC2 instance."* Available tools: AWS CLI. This objective is a high level task that needs to be planned into lower level tasks, including for example:

- translate the objective to an AWS CLI command: awsec2 run-instances –image-id <image-id> –count 1 –instance-type <instance-type> –key-name <key-name> –security-group-ids <security-group-id> –subnet-id <subnet-id>
- List the available images: "aws ec2 describe-images" then pick and return one of them.
- List the available instance types: "aws ec2 describe-instance-types" then pick and return one of them.
- List the available key pairs: "aws ec2 describe-key-pairs" then pick and return one of them.
- List the available security groups: "aws ec2 describe-security-groups" then pick and return one of them.
- List the available subnets: "aws ec2 describe-subnets" then pick and return one of them.
- Replace the placeholders with valid values
- Execute the command

An agent is clearly more suitable for this case.

4.2 When to use one agent vs multiple agents

This issue is similar to the one around objects in OOP. For example, you may create classes: Car, Wheel, Chair, Engine etc or you may just create the Car class and make Wheel, Chair, Engine etc attributes of the Car. This depends on the problem you are modelling.

Similarly, you may create a single agent for the "create an EC2 instance" like above, or you may create multiple agents: EC2 instance creator, Instance type manager, Key pairs manager etc if each of these need to handle specific cases or perform complex reasoning on intermediary results. It all depends on what level of reasoning over intermediary results you may want.

4.3 A framework for building a (universal) LLM agent

As discussed in the "State of the art" section, there is still no intuitive unified framework for building a single LLM agent. A few demos were built using different approaches [2].

Here we propose an intuitive framework for building a universal LLM agent, that is coherent with the AOP paradigm. As defined in the "Definitions" sections, there are two types of tasks:

- **Atomic**: that can be executed by tools without need for planning and reasoning

- **Composite**: that need LLM intervention for planning and reasoning over the input and intermediary results.

Agents are only relevant when solving complex tasks. To solve a complex task, the agent does the following:

Input: Given an objective O:

1. Plan: this step is similar to the *Map* step in Map-Reduce.
   - Divide the task into subtasks, with defined inputs and outputs
   - For each subtask, decide and attribute priority
   - Decide concurrency amongst tasks
   - For each subtask, pick the most useful tool (Note: it can be another agent, slave agent)

   Note: it is important to note that the Plan can be updated over time based on changes in the context that is accessible to the agent.
2. For each subtask, launch execution according to decided priority and concurrency
3. For each subtask execution, evaluate the returned result:
   (a) If the result is good: keep the result
   (b) Else, launch a corrective task
4. Aggregate and return: this step is similar to the *Reduce* step of Map-Reduce. It consists of Aggregating results of all subtask executions and return the final result. Eg, by filling placeholders and returning the final command output.

Output: Return a final result for O.

As can be noticed, this framework is an adaptation of the ReAct framework [4] to distributed agents and complex tasks. We suggest the name: PAIR for: Plan, Act, Iterate, Return. Note that this framework allows for *recursion*: a subtask can be the objective of another agent, which also can use other agents as tools and so on.

4.3.1 Agent hierarchy

There are two types of hierarchies:

a. Parenthood hierarchy

The ideas of parenthood and inheritance are simple yet powerful: When you want to create a new agent and there is already an agent that includes some of the code that you want, you can derive your new agent from the existing agent. In doing this, you can reuse the context and capabilities of the existing agent without having to write (and debug!) them yourself.

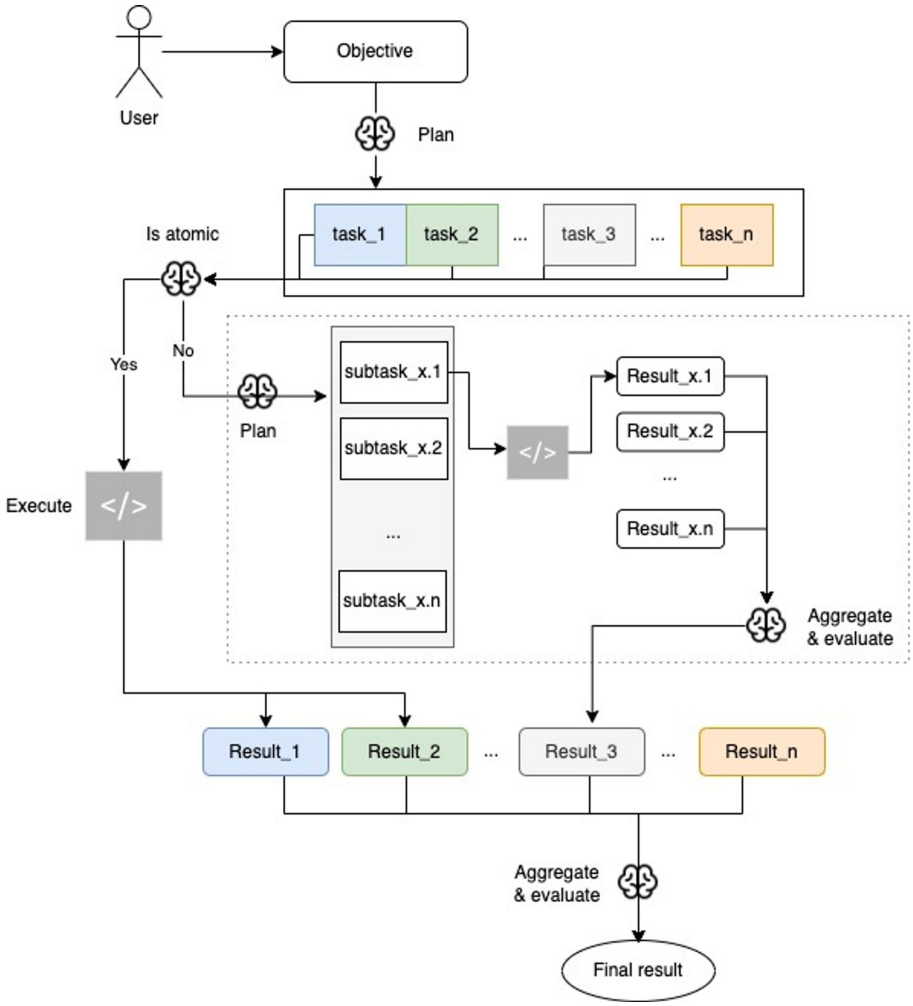

Figure 4: AOP allows for recursively planning and executing complex tasks

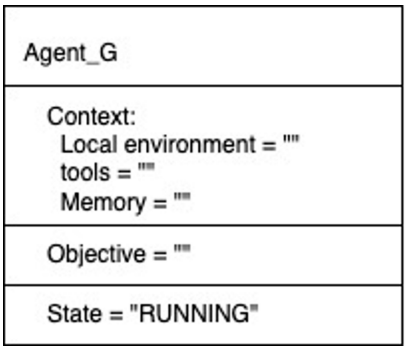

Figure 5: The genesis agent

The root of the parenthood hierarchy is the Genesis agent (equivalent to the Object class in java for example). It is defined as follows:

An agent inherits all static attributes from its parent: Description, tools and local environment.

Examples:

Suppose we have a travel agent that can plan trips using the following tools:

- Airbnb API
- Google maps
- Google flights

We can expand this agent by giving it an additional tool: Gmail, for example, to send an email to all travellers with a summary of the trip.

b. Authority hierarchy

Agents not only have children and parents, they also have authority hierarchy. For some tasks, an agent A may need to use another agent B as a tool. In this case, we call A a master agent and B a slave agent. Any agent that is directly created by a human user is a root agent in the authority hierarchy. The authority hierarchy can have one or more root agents.

Examples:

The travel agent above can be used by a personal assistant agent that has more authority. The personal assistant agent does travel and personal event planning for example. In this case, the personal assistant uses the travel agent as a tool. The terminology master-slave is particularly useful when talking about the technical implementation of both agents.

4.3.2 Agent chaining (The orchestrator agent)

Agent chaining (coordination or orchestration) is particularly important when agents need to be executed in a specific order to achieve a specific objective. Agents can run in sequence or in parallel depending on their objectives and

dependancies. The role of the orchestrator is to define overlaps between objectives and define task dependancies globally across agents. It achieves this by maintaining a centralised ledger of all tasks, and messages exchanged between agents.

Example

We create an autonomous agent organisation with the goal to automate admin tasks for a startup CEO. Suppose the org is composed of the following agents:

- accounting agent
- tax agent
- HR agent
- travel agent

These agents have access to tools, documents and databases necessary to achieve their objectives. The user (CEO) instructs the agents with the following tasks:

- accounting agent: set all client_x invoices to "paid" in the accounting system and send receipt emails to the client.
- tax agent: file this quarter's VAT report
- HR agent: how many employees are still in their trial period?
- travel agent: book a trip to go see client_x

Without an orchestrator agent, these objectives could result in redundant 6 tasks or bad outcome as a result of bad execution order (VAT tax filing before setting invoices for example).

Another example

Agent 1: Objective is to increase website traffic by 20Agent 2: Objective is to improve website user experience to reduce bounce rate by 15% in the next quarter.

Tasks for Objective 1:

1. Conduct keyword research to identify high-traffic keywords 2. Optimize website content with identified keywords 3. Share blog posts on social media platforms to attract more visitors 4. Run targeted ads to drive traffic to the website 5. Collaborate with other websites to generate backlinks to our website

Tasks for Objective 2:

1. Analyze user behavior on the website to identify pain points 2. Improve website speed and loading time 3. Optimize website design and layout for better user experience 4. Simplify website navigation to reduce confusion and frustration 5. Create engaging and informative content to increase user engagement

As you can see, there is overlap between some of these tasks. For example, optimizing website content with high-traffic keywords from Objective 1 could

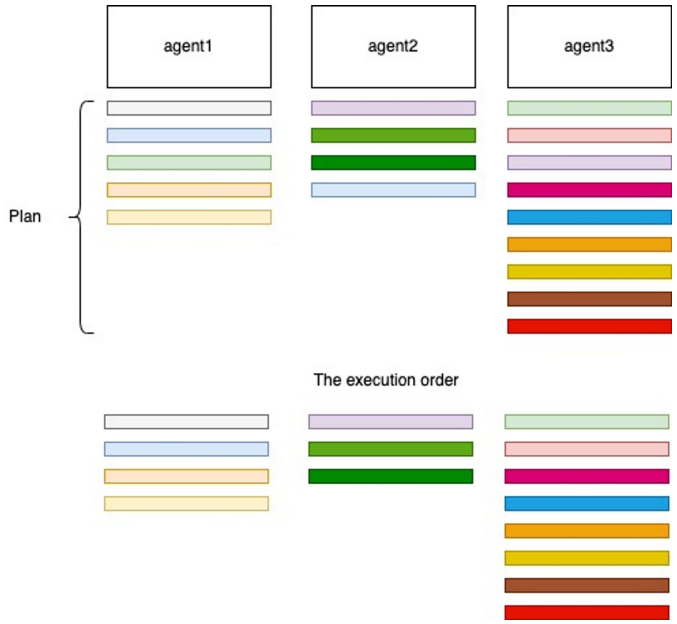

Figure 6: The orchestrator agent makes sure no redundant tasks are executed

also improve user experience by making the website more relevant and informative. Similarly, creating engaging and informative content for Objective 2 could also drive more traffic to the website, helping to achieve Objective 1.

In this case, an orchestrator agent is needed to orchestrate the two agents and make sure:

- tasks are not redundant
- reuse common results

The orchestrator agent has visibility into all tasks and execution results of agents under management. The orchestrator needs to have authority over managed agents to be able to distribute, correct or remove tasks from them.

Note: the orchestrator can be the developer, by hard coding orchestration rules, or the user during runtime. In complex systems, where managing agents require reasoning about their state, we recommend creating an orchestrator agent instead of hard coding orchestration rules.

4.3.3 Agent lifecycle management

Agents can be created, updated and killed by their manager (a designated agent, developer via code rules or user during runtime). Similar to the orchestrator, in complex systems, where managing agents require reasoning about their state,

we recommend implementing the lifecycle manager as an agent instead of hard coding lifecycle rules.

The lifecycle agent is an infrastructure agent. Its objective is to create, update and kill other agents according to its meta plan. It must have access to objectives, tasks, task status, and agent states of all agents under management. ***You can think of it as the HR department of the agent organisation.***

4.3.4 Agent monitoring

Monitoring agents includes:

- make sure all planned tasks are aligned with rules defined by the root agents. - make sure all results, including all intermediary results, are in line with rules defined by the root agents - collect and report infrastructure information, including: agent deployment environment, execution metrics, resource usage etc - etc

These tasks can be implemented as one or different agents depending on the desired rules and constraints to be put in place.

5 Conclusion

LLMs are changing the way we write software, moving from pre-defined static code written by humans to dynamically generated code based on a changing context. The software building block used to be functions, classes and objects. In the new paradigm, the building blocks are:

- a reasoning engine: LLM
- LLM agents
- and tools

in this paper we contributed some ideas to inch closer to this new paradigm. We will keep iterating on this paper and we will release the AgentChain code in the near future.

References

[1] Hewitt, Carl. "Actor model of computation: scalable robust information systems." arXiv preprint arXiv:1008.1459 (2010).

[2] Zhao, Guodong Troy. "A comprehensive and hands-on guide to autonomous agents with GPT."

[3] LangChain documentation. https://python.langchain.com/en/latest/index.html

[4] Yao, Shunyu, et al. "React: Synergizing reasoning and acting in language models." arXiv preprint arXiv:2210.03629 (2022).