LLM Agentic Systems

Calvin Chi

2025-07-18

Table of contents

Preface **3**

I Concepts **4**

1 Introduction **5**
1.1 Prompt 6
1.2 Tools 8
1.3 Memory 10

References **13**

2 Context Engineering **14**

References **16**

3 Reflection **17**
3.1 Self-Correction with External Feedback 18
3.2 ReAct 19
3.3 LLM as Optimizers 26

Preface

In March of 2025, I had the new exciting opportunity to join a new team developing foundational models and AI agents. Since the field of agents is new and rapidly evolving, the number of established texts or formal courses on AI agents is sparse. A common joke in this field is that by the time someone writes a book, half of the content becomes outdated when finished. Instead, most people that I know stay up to date by continually reading blogs, tweets, and papers. However, without a comprehensive text, it can be hard to see how concepts fit together within the bigger picture. This is true especially when definitions are still being debated, new perspectives are still being developed, and new terms are being invented. A recent example is the term "context engineering", which gained popularity after a discussion on X on June of 2025, where the term was introduced to unify the goals of prompt engineering, memory management, tool use, etc.

I started this online book as an attempt to organize and aggregate the lessons I learned along the way as I build agents. I use the structure of a book to organize ideas and concepts coherently, and choose the format to be online to easily incorporate updates in the field, which are frequent. The contents are drawn from my learnings from research papers, blogs, talks, and the practical experience of building agents. My goal is to write in sufficient levels of depth and detail to reveal how things work "underneath the hood", which may be difficult to see when most frameworks and agent-building tools abstract away those details. However, knowing these details is more empowering as it allows us to build more freely, concretely, and effectively, instead of attributing certain agentic abilties to "magic". Hence, this book will have a focus on describing low-level agent mechanisms with code illustrations when appropriate. Given how new and evolving the field of AI agents are, the definitions and perspectives of this book may not necessarily align with everyone's views or stand the test of time, nor are they meant to be comprehensive. Rather, the aim is to provide *one* valid mental model of how agents work to help people get started with building agents.

Part I

Concepts

1 Introduction

In artifical intelligence (AI), an agent is broadly defined as anything that can perceive and act in its own environment (Norvig and Intelligence 2002). With the rise of large language models (LLMs), LLMs are now used to power modern agentic systems by leveraging their much more powerful and generalized intelligence capabilities that emerged from scale (Brown et al. 2020; Wei et al. 2022). At its best, a LLM can dynamically decide the sequence of steps that need to be executed in order to accomplish a given task, essentially achieving autonomy.

In practice, agentic systems differ in the degree of reliance on the LLM as a decision maker, since the increased flexibility that LLMs provide comes with the cost of reliability. On one end of this spectrum is a LLM workflow, which has LLMs participate in a limited scope within a broader predefined workflow. The steps are pre-defined and the LLM is tasked with making some of the decisions. On the other end of the spectrum is a LLM agent, where the LLM directs its own workflow to accomplish a task - deciding *what* and *how* many steps to take. We can illustrate the difference between a workflow and an agent with a customer service chatbot example:

- **Workflow**: a potential workflow executes (1) intent classification by a LLM, (2) tool execution based on intent, and (3) LLM response generation, totaling three steps with each agent invocation. Based on the determined intent, only one tool is executed by following a pre-defined if-else control flow.
- **Agent**: given a set of tools, a LLM dynamically decides which tool to use in response to customer inquiry. In this process, multiple tools can be used any number of times, with the steps planned or decided by the LLM itself. Once the LLM determines it has collected sufficient information from tool-use to respond, it generates a final response to the customer.

While an agent can tackle tasks more adaptively, it also becomes less predictable and reliable. On the other hand, workflows are more deterministic and reliable, but are limited in their ability to tackle more open-ended tasks where there may not be one obvious approach. Choosing between a workflow and an agent requires considering the balance of flexibility and reliability needed for the application. In the rest of this book, the word *agent* will be used interchangeably with agentic systems, with the distinction between workflows and agents explicitly stated when necessary.

Building an agentic system from a LLM requires a prompt, tools, and memory. The prompt is piece of text that instructs the LLM on how to behave within the agent application. Tools

allow an agent to take actions and is typically accessed by an agentic system in the form of an API. Finally, memory allows an agent to act and behave in a contextualized manner, with user information or conversation history being common memory contexts. Each of these components are the building blocks that can be used to create and shape a LLM agent. Figure 3.1 illustrates an agentic system and its components:

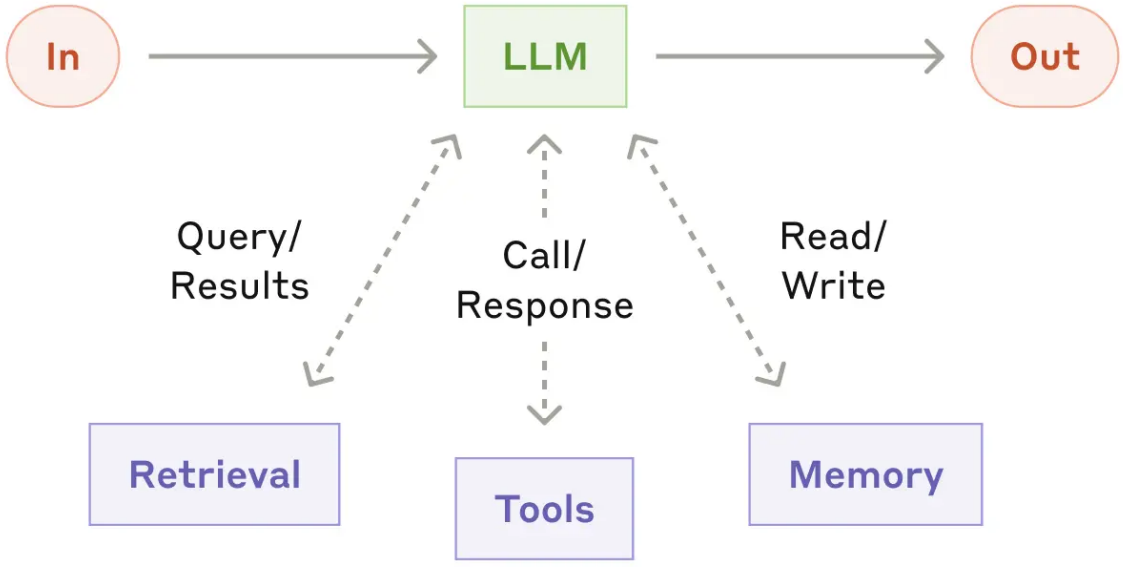

Figure 1.1: An agentic system and its components. Dotted and double-sided arrows indicate that the interaction is optional and bidirectional respectively. Additionally, the double-sideness implies that the interaction can be iterative, occuring multiple times until the LLM determines its task is done. Source: "Building Effective Agents" (https://www.anthropic.com/index/building-effective-agents).

1.1 Prompt

A prompt is a piece of text that instructs a LLM how to behave within an agent application. A prompt can be organized conceptually into a system prompt, contextual prompt, role prompt, and a user prompt. In the end, they are all concatenated together into a single text input when invoking the LLM (i.e. asking LLM to generate response).

- **System prompt**: contains high level instructions that should always be applied and thus is always part of the input text when invoking a LLM. Typically, the system prompt contains instructions asking the LLM to be a helpful and patient agent.

- **Role prompt**: in an agentic system, LLMs may be required to behave differently depending on the scenario. For example, in multi-agent collaboration where multiple specialized agents communicate together to solve as task, each specialized agent will need a role prompt. To implement this behavior, multiple role prompts are maintained and a specific role prompt is selected and concatenated with the remaining prompts depending on the scenario or role.
- **User prompt**: the question or instruction from the user of the agent application. The user prompt is typically appended to the end of the final prompt that is passed to the LLM.
- **Contextual prompt**: catch-all prompt for all contextual details needed for an agent to respond to a user request. For industry applications, this could be the account information of the user in the current conversation session. Having a contextual prompt is important for a good and safe user experience as it saves the user from having to state user information that might be later used by the agent.

Come LLM inference time, the process of putting together the final prompt typically involves concatenating the system prompt, one of the role prompts, the contextual prompt with contextual values filled in, and the user input. Below is an example for a bank agent chatbot, using AWS bedrock to access a LLM

```
import boto3

SYSTEM_PROMMPT = """
<instruction>
You are a helpful agent for XYZ bank. You are ALWAYS patient, helpful, and always try to
assist the user in the best way possible.
</instruction>
"""

ROLE_PROMPT_REPORTING = """
You are tasked with account reporting.
Use the following function to look up the account information:

{
    "function_name": "account_lookup",
    "description": "a tool to retrieve account information for a user.",
    "arguments": {
        "username": {"type": str, "description": "user name"},
        "security_code": {"type": str, "description": "security code"}
    }
}

NEVER reveal account Ids.
```

```
"""

CONTEXTUAL_PROMPT = """
Use below account information <account> about the customer:

<account>
Username: {username}
account_type: {account_type}
</account>
"""

user_input = "Can you get the ending balance of each month for 2024?"
bedrock_runtime = boto3.client("bedrock-runtime", region_name="us-west-1")
bedrock_runtime_response = bedrock_runtime.converse(
    modelId = "us.anthropic.claude-3-7-sonnet-20250219-v1:0",
    system = [
                {'text': SYSTEM_PROMMPT},
                {'text': ROLE_PROMPT_REPORTING},
                {'text': CONTEXTUAL_PROMPT.format(username = "caleb",
                                                  account_type = "savigns")}
    ],
    messages = [{"role": "user", "content": [{"text": user_input}]}]
)
```

According to Anthropic, using XML tags in your prompts can help Claude models parse specific components in your prompt more easily. For example, better identifying which part of the prompt is the system prompt by the `<instruction>` tag. As a heuristic, capitalize words for emphasis, such as the words "NEVER" or "ALWAYS".

1.2 Tools

The tools of an agent are the software services that a LLM can access via API calls, which gives the LLM a means to interact with the outside environment, and imbues an agent with specialized abilities. Common LLM tools include database access (for retrieval augmented generation (RAG)), web search, code interpreter, and calculator. For real-world agent applications, these tools can be specialized in-house services such as a recommendation system or placing an order.

Concretely, a LLM "accesses" tools by generating an API call string, typically in the standardized JSON format for ease of parsing. Then, the API call string is passed to the client side, which extracts key entities like the tool name and arguments, followed by making the API call to the

specified tool with the extracted arguments. While this is in principle possible with regular language models in the pre-LLM era, tool-use became more main stream as LLMs developed the instruction-following ability to generate API calls reliably if you simply provide the tool use instructions and tool documentation (e.g. tool name and required arguments) in the input prompt.

To illustrate the mechanism of tool-use, suppose we add to the LLM prompt the following documentation on a weather function so that the LLM knows how to generate the API call string when the user asks for the weather on a given day:

```
{
  "name": "get_temperature_by_day",
  "description": "Returns the forecasted temperature in Celsius for a specified day of the w
  "parameters": {
    "type": "object",
    "properties": {
      "day": {
        "type": "string",
        "description": "Name of the day of the week (e.g., 'Monday', 'Tuesday'). Case-insens
      }
    },
    "required": ["day"]
}
```

Additionally suppose in the prompt we instruct the model to generate the API call in JSON format for ease of parsing, for example:

```
{
    "name": "get_temperature_by_day",
    "arguments": {
        "day": "Tuesday"
    }
}
```

Then, the code to parse and execute the function could look like:

```
import json

def get_temperature_by_day(day):
    if day == "Tuesday":
        return 27
    return 30
```

```
tool_call_string = """
{
    "name": "get_temperature_by_day",
    "arguments": {
        "day": "Tuesday"
    }
}
"""

# Parsing LLM tool call string to extract tool name and argument
tool_call_json = json.loads(tool_call_string)
day = tool_call_json["arguments"]["day"]
func_name = tool_call_json["name"]

# Tool execution
temperature = globals()[func_name](day)
print("Temperature for {day} is {temp}C".format(day = day, temp = temperature))
```

```
Temperature for Tuesday is 27C
```

Figure 3.2 shows the life cycle of a function call, and shows that the role of a LLM in tool calling is to map the user question to the corresponding tool call JSON output.

1.3 Memory

Finally, memory allows an agent to accumulate conversation history as context, allowing responses to become highly contextualized and efficient. Perhaps inspired by the biological mind, people like to categorize agent memory into short-term or long-term memory, with implications for usage and implementation.

Short-term memory typically describes the conversation history of the current conversation session, which might revolve around solving a single task or topic. Like the RAM for computers, it acts as the working memory of the agent, which is typically stored in a buffer or list without further processing and passed to the LLM for each response.

Long-term memory refers to the collection of conversation history across sessions. Since each session may concern a different topic, long-term memory is typically only accessed by an agent when relevant to the current conversation, and is thus stored in external databases that can be retrieved (e.g. vector data store for semantic retrieval). When long-term memory is retrieved for the current conversation session, it may get summarized first before passing to the LLM in order

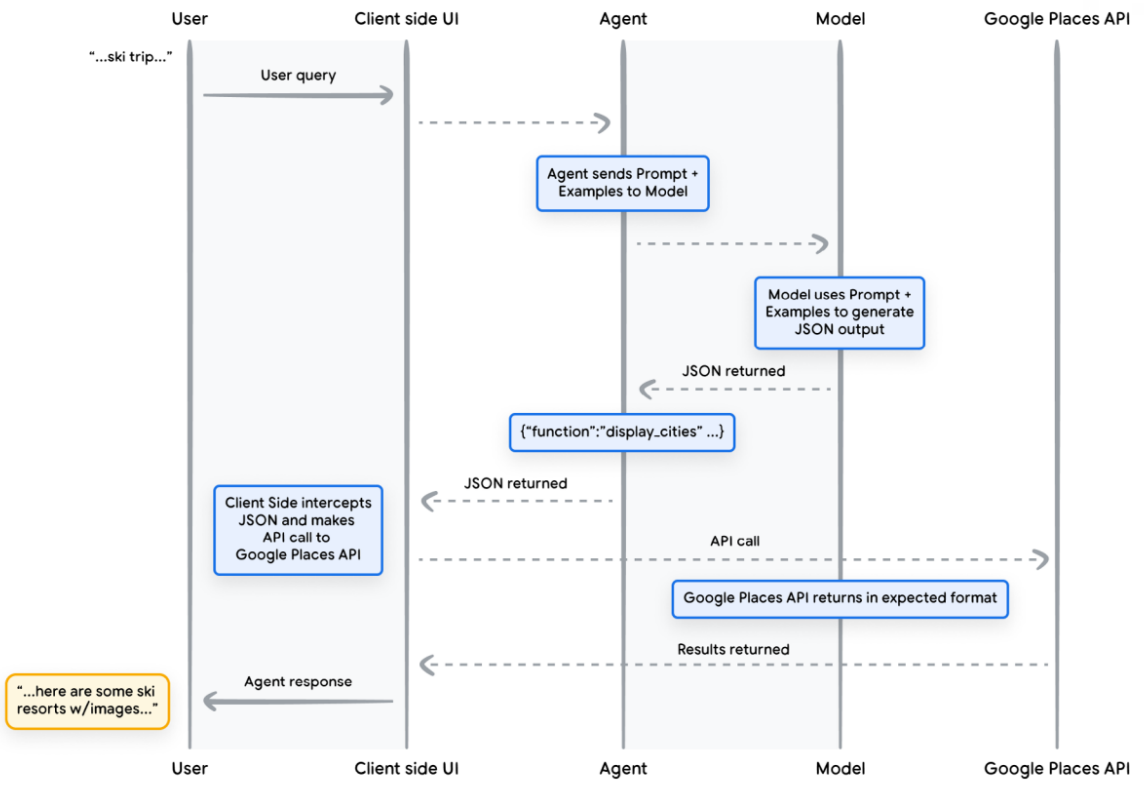

Figure 1.2: In this example life cycle of a function call, the agent processes the user request and decides to use the Google Places API. First, example API calls are added to the prompt for in-context learning, then the LLM generates the tool call JSON payload, which gets sent to the client side to process and make the API call. The API call results are then fed to the LLM for final response generation. Source: "Agents" (https://www.kaggle.com/whitepaper-agents).

to utilize its context window efficiently. An example of this long-term memory processing is in the multi-agent collaboration of ChatDev, where each long-term memory is the conversation history between two agents for solving a subtask (Qian et al. 2023). To start the next subtask, the solution to the previous subtask is extracted from long-term memory and loaded into the LLM context.

References

2 Context Engineering

LLMs are trending towards having longer context windows, capable of processing upwards of millions of tokens. This is regarded as beneficial overall, as larger contexts allow a model to process more information and solve more complex problems. However, research from Chroma showed that model performance degrades as context length increases, a phenomenom their research team coined "context rot" (Hong, Troynikov, and Huber 2025). Specifically, they observed this phenomenon by evaluating LLMs on the Needle in a Haystack (NIAH) task, where the LLM is instructed to answer a question where the answer (i.e. needle) is embedded in a larger, unrelated body of text (i.e. haystack). Traditionally, the needle for a question can be identified via lexical matching, or exact matching on words or phrases. For example:

> Question: Which book sparked my interest in AI?
>
> Needle: The book which sparked my interest in AI is "The Worlds I see".

A needle can also be identified through semantic matching or via a LLM's world knowledge that do not involve lexcial matching. For example:

> Question: Which professor had prior experience in theoretical research?
>
> Needle: Ten years ago, Ms. Carter spent a decade at the Institute for Advanced Study.

In the above example, in order to identify the needle, the LLM had to utilize its world knowledge that the Institute for Advanced Study is a center for theoretical research as well as semantic identification of the needle's relevance to the question. There are no overlapping words between the question and needle in this case.

Using the NIAH task, the Chroma team carried out a series of controlled experiments to show that performance generally degrades with longer contexts. The key findings are:

- When the task complexity is held constant by keeping the question-needle embedding cosine similarity the same, model performance degrades with context length. The performance decline tends to be more rapid for question-needle pairs that are more dissimilar.
- The decline in performance is also influenced by distractors (i.e. text chunks that are topically related to the needle but does not answer the question), the content of the haystack, and the structural coherency of the haystack (i.e. randomly shuffled sentences or not).

- No evidence that needle position in the haystack influences performance.

For a broader discussion on the types of long context problems, such poisoning, distraction, confusion, or clash, refer to this blog post by Drew Breunig.

Together, these findings suggest that managing the context of the LLM is important for effectiveness. Specifically, that the LLM's context window should be filled with relevant content for a given task, but no more, in order to be maximally effective. This context management effort is now coined the term "context engineering", commonly defined as "the art of providing all the context for the task to be plausibly solvable by the LLM". Context engineering regards everything that is inputted into a LLM as context, which includes memory, prompts, information from retrieved for RAG, etc.

A langchain blog post summarizes common context engineering patterns, which include writing context to offload information for later use, information selection for the context window, compression, and isolation. A few concrete tactics include performing RAG on tool descriptions to shrink the tool-selection space to a smaller and more relevant set of tools, periodically summarizing past conversations (implemented by Claude Code and ChatDev), and multi-agent architectures where subagents own their own isolated context so that the overall agentic system is effectively using an expanded context window.

Context compression is a key tactic used by ChatDev to implement multi-agent collaboration for software development (Qian et al. 2023), where the conversation history between two subagents for each software development phase is summarized into the solution. This solution then serves as the start context for the next phase of subagent-to-subagent dialog to tackle another set of tasks. This intuitively makes sense because only the solutions of subproblems are necessary for solving the bigger problem, and the process for solving the subproblems is usually irrelevant.

References

3 Reflection

Reflection is one of the most effective design patterns for agents, where a LLM enters into an iterative loop of problem solving and feedback until a solution is reached. This feedback can be external or intrinsic, where external feedback refers to receiving feedback from external evaluators such as humans, code execution output, or unit test results, and intrinsic feedback refers to self-critique from the same LLM on its previous response. The reflection process is depicted in Figure 3.1.

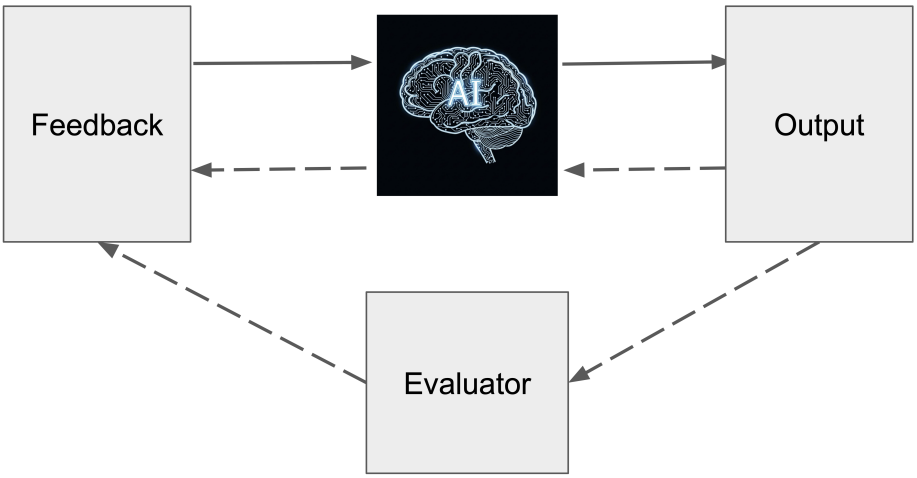

Figure 3.1: The reflection process of iteratively using feedback to improve LLM output. Dotted arrows indicate optionality in how the LLM output is processed into feedback, which can be provided by the same LLM (intrinsic feedback), by an external evaluator (external feedback), or both.

There has been some debate regarding whether self-feedback from the same LLM alone can drive performance improvements. Madaan *et al.* introduced Self-Refine, a reflection framework that relies completely on intrinsic feedback (Madaan et al. 2023). After the initial response construction, each iteration of Self-Refine consists of the steps of feedback and refinement. Each step uses the same LLM, but differs in the prompt that contains instructions (e.g. instructions for feedback or refinement) and few-shot examples. Empirically, this was shown to improve performance materially, with most of the improvement occurring in the initial iterations of reflection. However, the improvement is not necessarily monotonic, and Madaan *et al.* observed

that Self-Refine was most effective when the feedback is specific, actionable, or broken down into different evaluation dimensions. However, another group of researchers from Google DeepMind independently assessed instrinsic self-correction and found that its effectiveness was limited, instead causing performance degradation with each self-refine iteration (Huang et al. 2023). They concluded that this discrepancy was caused by flawed experimental design from the Self-Refine study where the complete set of requirements was included in the feedback prompt instead of the initial response instruction. As a result, it is unclear whether the improvement was due to "leakage" of requirements from the feedback or from the iterative process of self-improvement. When the complete set of requirements was included in the initial response instruction, Huang *et al.* observed that standard prompting outperformed Self-Refine.

These studies show that in order to reliably improve after reflection, external feedback is necessary. Intuitively, this makes sense because if the bottleneck of performance is due to the lack of certain parametric knowledge (i.e. knowledge trained into the weights of a model), then feedback from the same LLM is unlikely to provide the missing knowledge required to arrive at the solution. We next introduce self-correction from external feedback and ReAct as effective reflection examples that make use of external feedback or signal to drive performance beyond prompting a LLM in isolation.

3.1 Self-Correction with External Feedback

Multiple works of research has shown that external feedback reliably and materially improves performance on the problem an agent is trying to solve, with the LLM self-correcting its previous response using the provided feedback (Chen et al. 2023; Shinn et al. 2023; Gou et al. 2023). Gou *et al.* introduced a simple version of this by using a set of tools through which critiques on correctness are obtained. The critiques are used by the LLM to improve its previous response, and the loop terminates when the critiques determine the latest response to be correct (Gou et al. 2023). Shinn *et al.* introduced the Reflexion framework that added more bells and whistles by (1) integrating LLM-driven self-reflection with external feedback signal to serve as the final feedback and (2) incorporating short-term memory of conversation history (i.e. trajectory) and long-term memory of past verbal feedback. By using an evaluator LLM to score the latest LLM response and using another LLM to suggest actionable feedback based on the score, Reflexion improves the quality of the feedback. With memory of the LLM's past interactions with the envornment and their outcomes, the actor LLM essentially undergoes reinforcement learning based on in-context learning examples. The Reflexion framework is best illustrated with the diagram in Figure 3.2.

Finally, Chen *et al.* applied reflection to programming agents to simulate rubber duck debugging, where debugging is done by explaining code and following the execution results (Chen et al. 2023). Like Reflexion, the feedback step includes both the external signal (from code execution) and a LLM generated explanation of the code as the final feedback provided to the same

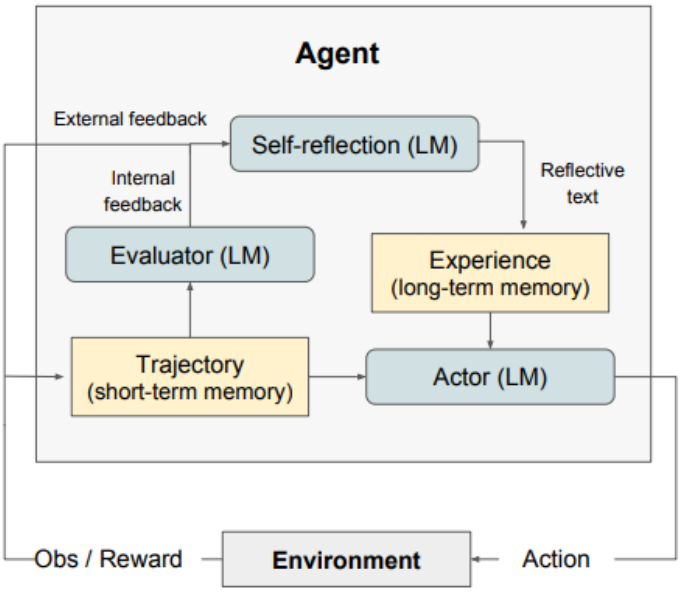

Figure 3.2: Reflexion framework. Source: "Reflexion: Language Agents with Verbal Reinforcement Learning" (https://arxiv.org/pdf/2303.11366).

LLM for the next iteration of code generation. In agreement with other research, the authors found that receiving feedback from code execution is important for improving performance consistently.

3.2 ReAct

ReAct, which stands for reasoning and acting, is a prompting technique to combine reasoning and acting with language models to solve diverse language reasoning and decision making tasks (Yao et al. 2023). By interspersing actions and subsequent observations with a LLM's reasoning traces, Yao *et al.* showed that this reduces hallucination and error propagation, demonstrating superior performance over baselines like standard prompting, chain-of-thought prompting (i.e. reasoning only), and acting only (i.e. ReAct without thoughts). It is believed that reasoning improves an agent's subsequent actions via mechanisms such as introducing notions of planning, injecting extra parametric knowledge, or by extracting important parts of observations.

Concretely, ReAct prompting induces a repeating pattern of thinking, acting, and receiving observation until a task is solved. Starting with a question, ReAct prompting invokes the LLM to output thinking traces on how to solve the problem, followed by another LLM invocation on what action to output given the thinking. The selected action is executed, then

the corresponding observation is added to the agent's short-term memory (i.e. conversation history), and the process repeats. Thus, the number of LLM calls is $O(N)$ in the number N of think, act, and observation iterations required to solve a problem, making ReAct prompting relatively expensive. However, ReAct improves the flexibility of an agent to address open-ended requests dynamically. Additionally, Yao *et al.* reported ReAct introduces strong generalization capabilities to new problems with just a few in-context examples.

We next use ReAct to implement an agent that uses reasoning and internet search to recommend where to live in the United States based on their preferences. Our ReAct prompt first describes to the LLM that we wish to solve the problem using a thinking, action, and observation pattern, instructing the LLM output format at each step, explaining the action space, and how to end. Then, this is followed by a one-shot example of a thought, action, and observation pattern used to address a sample question. The prompt ends by instructing the agent to *only* output tokens for action or thinking.

```
SYSTEM_PROMPT = """
You are an assistant trying to help me determine which city in the United States I should liv
"""

ROLE_PROMPT = """
Solve a question answering task with interleaving Thought, Action, Observation steps.
Respond ONLY with ONE JSON-complaint string of the format:

{{
    "type": "Thought" or "Action"
    "content": "content of Thought, Action, or Observation"
}}

The "content" of Action is either "Search[arguments]" or "Finish[answer]", where:

-`"Search[arguments]"` means to search the web with arguments
-`"Finish[answer]"` means to finish the steps with an answer. You should output `"Finish[answ
  to answer the question.

The "content" of Thought is a reasoning about what Action to take next based on previous Obse

Example pattern:

Question:
"Which city should I live in and what activities does it provide? I enjoy walkable cities."

Thought 1:
{{
```

```
    "type": "Thought"
    "content": "To find which city you should live in, I need to search for a list of most wa
                United States and their temperatures."
}}

Action 1:
{{
    "type": "Action"
    "content": "Search[Most walkable American cities]"
}}

Observation 1:
{{
    "type": "Observation"
    "content": "One of the most walkable cities in the U.S. is New York."
}}

Thought 2:
{{
    "type": "Thought"
    "content": "I need to search what hobbies or activities are available in New York."
}}

Action 2:
{{
    "type": "Action"
    "content": "Search[What hobbies or activities are available in New York?]"
}}

Observation 2:
{{
    "type": "Observation"
    "content": "From big highlights like Times Square to quaint walks locals love. New York C
                and adventure, offering an array of activities for everyone. Explore iconic l
                of Liberty, the Empire State Building, and Central Park, or marvel at art of
}}

Thought 3:
{{
    "type": "Thought"
    "content": "From the search results, New York City is the most walkable city in America w
                like exploring city landmarks or visiting museums."
```

```
}}

Action 3:
{{
    "type": "Action"
    "content": "Finish[I recommend you live in New York City, which is one of the most walkab
                        you can explore city landmarks Statue of Liberty, the Empire State Bu
                        at art of the Museum of Modern Art]"
}}

...

Return type and content for ONE step ONLY
"""
```

To implement the actual agent, we essentially wrap the LLM call within a while loop until a maximum number of steps is reached. The agent class is additionally equipped with:

- Short-term memory, implemented simply as a string in this protype.
- The internet search tool `DuckDuckGoSearchRun`, which simply accepts a natural language query and responds with a paragraph of text results.

Then, in each iteration we parse the LLM response to decide what action to take. The initial question and subsequent thoughts, actions, and observations are continually added to the conversation history to influence what the LLM will output next. For example, if the last item in the history is an observation, then the LLM will output a thinking trace based on following the ReAct prompt.

```
from langchain_community.tools import DuckDuckGoSearchRun
import json
import boto3
import warnings
warnings.filterwarnings('ignore')

class ReAct_agent:
    def __init__(self, client, system_prompt, role_prompt, max_steps=20, search_max_length=1
        self.client = client
        self.SYSTEM_PROMPT = system_prompt
        self.ROLE_PROMPT = role_prompt
        self.ddg_search = DuckDuckGoSearchRun()
        self.max_steps = max_steps
        self.step_num = 0
```

```
        self.search_max_length = search_max_length
        self.history = ""

    def ask(self, message, verbose = True):
        self.add_to_memory(message)
        model_response = self.invoke_llm(message)
        model_response_json = json.loads(model_response.replace("```json", "").replace("```"
        response_type = model_response_json["type"]
        while self.step_num < self.max_steps:
            self.add_to_memory(model_response)
            if response_type == "Action":
                prefix, content = self.parse_action(model_response_json["content"])
                if prefix == "Finish":
                    return content
                elif prefix == "Search":
                    observation = self.search(content)
                    if verbose:
                        print(json.dumps({"type": "Observation", "content": observation}, ind
                    self.add_to_memory(json.dumps({"type": "Observation", "content": observa
            model_response = self.invoke_llm(self.history)
            model_response_json = json.loads(model_response.replace("```json", "").replace("
            response_type = model_response_json["type"]
            if verbose:
                print(json.dumps(model_response_json, indent=4))
            self.step_num += 1

    def search(self, message):
        return self.ddg_search.invoke(message)[:self.search_max_length]

    def invoke_llm(self, message):
        bedrock_runtime_response = bedrock_runtime.converse(
            modelId = "us.anthropic.claude-3-7-sonnet-20250219-v1:0",
            system = [
                {'text': self.SYSTEM_PROMPT},
                {'text': self.ROLE_PROMPT}
            ],
            messages = [{"role": "user", "content": [{"text": message}]}]
        )
        return bedrock_runtime_response["output"]['message']['content'][0]['text']

    def add_to_memory(self, message):
        self.history += ", " + message
```

```
    def parse_action(self, message):
        start = message.find('[')
        end = message.find(']')
        prefix = message[:start]
        content = message[start+1:end]
        return prefix, content
```

Now we ask the agent where we should live if we prefer cities with access to nature and outdoor activities.

```
bedrock_runtime = boto3.client("bedrock-runtime", region_name="us-west-2")

agent = ReAct_agent(bedrock_runtime, SYSTEM_PROMPT, ROLE_PROMPT, max_steps = 15)

user_input = "Which city should I live in? I enjoy outdoor activities."
answer = agent.ask(user_input)

print(answer)
```


```
{
    "type": "Action",
    "content": "Search[best US cities for outdoor activities]"
}
{
    "type": "Observation",
    "content": "Nov 16, 2024 \u00b7 Adventure awaits in these top 10 US cities ! Discover des
round outdoor destination famous for its ski resorts, including the largest ski area in the U
}
{
    "type": "Thought",
    "content": "Based on the search results, I've found several cities that are known for out
}
{
    "type": "Action",
    "content": "Search[outdoor activities in Denver Colorado]"
}
{
    "type": "Observation",
    "content": "Mar 7, 2025 \u2014 There are tons of opportunities to go hiking, running and
fishing in and around the city. Denver is a paradise for cyclists with bike lanes and paved p
}
```

```
{
    "type": "Thought",
    "content": "Denver appears to be a great option for outdoor activities, with hiking, runn
fishing, cycling paths, and proximity to attractions like Rocky Mountain National Park, Pikes
}
{
    "type": "Action",
    "content": "Search[outdoor activities in Portland Oregon]"
}
{
    "type": "Observation",
    "content": "United States Oregon (OR) Portland Things to Do in Portland Outdoor Activitie
born cyclist Molly Sugar, Friends on Bikes diversifies the cycling scene by connecting and ho
binary people of color. Launched in 2017, this Portland -based group organizes monthly all-
levels social rides, as well as \u201cbike-packing\u201d trips, workshops and ... See full li
}
{
    "type": "Thought",
    "content": "I've gathered information about Denver and Portland, two cities known for out
fishing, cycling paths, and proximity to natural attractions like Rocky Mountain National Par
}
{
    "type": "Action",
    "content": "Search[outdoor activities in Seattle Washington]"
}
{
    "type": "Observation",
    "content": "From hiking & biking to kayaking & skiing add a dash of adventure to your vac
to spot for water activities in Seattle . Rent a kayak or paddleboard and enjoy gliding past
}
{
    "type": "Thought",
    "content": "I now have information about three top cities for outdoor enthusiasts: Denver
informed recommendation, I should look at one more city from the initial list - Salt Lake Cit
}
{
    "type": "Action",
    "content": "Search[outdoor activities in Salt Lake City Utah]"
}
{
    "type": "Observation",
    "content": "Founded in 2020, the Outdoor Market Alliance is a non-profit association dedi
driven initiatives. We are a collective of 25 independent sales and marketing \u2026 Apr 15,
```

```
largest city features fun outdoor recreation, historic sites, and stellar places to shop and
brand gear & clothing for camping, climbing, cycling, fitness, hiking, paddling, skiing, snow
brand gear, clothing\u2014and outdoor adventures! Plus rentals, classes, events, expert advic
op online and in-store. Explore 12 top spots with great access and hunter lodging. Outdoor Li
to publication of America\u2019s diehard hunters, shooters, and anglers since 1898. Our stori
}
{
    "type": "Thought",
    "content": "The search results for Salt Lake City didn't provide specific information abo
}
{
    "type": "Action",
    "content": "Finish[Based on your interest in outdoor activities, I recommend Denver, Colo
fishing in and around the city. You'll have access to world-class natural attractions like Ro
round, making it ideal for outdoor enthusiasts. Alternatives worth considering are Portland,
}
Based on your interest in outdoor activities, I recommend Denver, Colorado as an excellent ci
fishing in and around the city. You'll have access to world-class natural attractions like Ro
round, making it ideal for outdoor enthusiasts. Alternatives worth considering are Portland,
```


We see that with ReAct prompting, the agent first got a list of candidate cities, then searched each candidate city for specific outdoor activities before returning the final answer, which is Denver, Colorado.

3.3 LLM as Optimizers

Yang *et al.* from DeepMind showed that a LLM can be prompted to perform non-gradient optimization of numerical and discrete problems, using linear regression and the traveling salesman problem as example problems (Yang et al. 2023). The prompt includes a description of the problem and goal, followed by a set of candidate solutions and their corresponding scores, which the authors together refer to as the *meta-prompt*. By proposing new solutions and getting them scored, the LLM can utilize in-context learning to iterate on the problem. Crucially, this requires an external objective function evaluator to score solutions at each iteration, which are then added to the growing set of solution-score pairs. The solution-score pairs are sorted to presumably better help the LLM identify patterns that can be used to further refine the solution. See Figure 3.3. to see how a LLM can be used as an optimizer through prompting.

We can use the meta-prompt for linear regression from the paper to explain the prompt design:

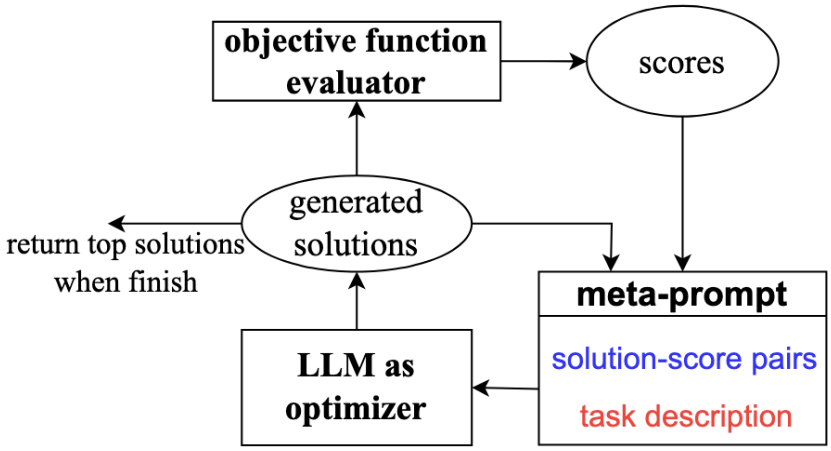

Figure 3.3: Optimization by prompting framework. Source: "Large Language Models as Optimizers" (https://arxiv.org/pdf/2309.03409).

> Now you will help me minimize a function with two input variables w, b. I have some (w, b) pairs and the function values at those points. The pairs are arranged in descending order based on their function values, where lower values are better..
>
> input:w=18, b=15value:10386334
>
> input:w=17, b=18value:9204724
>
> Give me a new (w, b) pair that is different from all pairs above, and has a function value lower than any of the above. Do not write code. The output must end with a pair [w, b], where w and b are numerical values..

In the above meta-prompt, the orange text are the instructions for linear regression and the blue text are the set of candidate solutions are the weight values w, b along with their sum of squared error (presumably).

As a more practical application, Yang *et al.* used the LLM to optimize the prompt for a given task. Specifically, the meta-prompt contains a statement that the goal is to generate a prompt (i.e. meta-instructions), examples of the output format, and past prompt-score pairs. After generation, a scorer LLM or objective function computes accuracy scores for these new prompts. The meta-prompt is then updated with the best new prompt-score pairs for further optimization.

Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. "Language Models Are Few-Shot Learners." *Advances in Neural Information Processing Systems* 33: 1877–1901.

Chen, Xinyun, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. "Teaching Large Language Models to Self-Debug." *arXiv Preprint arXiv:2304.05128*.

Gou, Zhibin, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. "Critic: Large Language Models Can Self-Correct with Tool-Interactive Critiquing." *arXiv Preprint arXiv:2305.11738*.

Hong, Kelly, Anton Troynikov, and Jeff Huber. 2025. "Context Rot: How Increasing Input Tokens Impacts LLM Performance." Chroma. https://research.trychroma.com/context-rot.

Huang, Jie, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. "Large Language Models Cannot Self-Correct Reasoning Yet." *arXiv Preprint arXiv:2310.01798*.

Madaan, Aman, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, et al. 2023. "Self-Refine: Iterative Refinement with Self-Feedback." *Advances in Neural Information Processing Systems* 36: 46534–94.

Norvig, P Russel, and S Artificial Intelligence. 2002. "A Modern Approach." *Prentice Hall Upper Saddle River, NJ, USA: Rani, M., Nayak, R., & Vyas, OP (2015). An Ontology-Based Adaptive Personalized e-Learning System, Assisted by Software Agents on Cloud Storage. Knowledge-Based Systems* 90: 33–48.

Qian, Chen, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, et al. 2023. "Chatdev: Communicative Agents for Software Development." *arXiv Preprint arXiv:2307.07924*.

Shinn, Noah, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. "Reflexion: Language Agents with Verbal Reinforcement Learning, 2023." *URL Https://Arxiv. Org/Abs/2303.11366* 1.

Wei, Jason, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, et al. 2022. "Emergent Abilities of Large Language Models." *arXiv Preprint arXiv:2206.07682*.

Yang, Chengrun, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2023. "Large Language Models as Optimizers." In *The Twelfth International Conference on Learning Representations*.

Yao, Shunyu, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. "React: Synergizing Reasoning and Acting in Language Models." In *International Conference on Learning Representations (ICLR)*.