A Beginner-Friendly Tutorial on LLM-based Agents

Shuyue Hu, Siyue Ren, Yang Chen, Chunjiang Mu, Jinyi Liu, Zhiyao Cui, Yiqun Zhang, Hao Li, Dongzhan Zhou, Jia Xu, Qiaosheng Zhang, Chu Han, Yan Zheng, Jianye Hao, Zhen Wang

November 17, 2025

Abstract

This tutorial is designed for researchers and engineers who are interested in large language model (LLM)-based agents but have little or no prior experience with LLMs, machine learning, artificial intelligence, or programming. It offers a gentle yet comprehensive introduction, providing newcomers with an intuitive, high-level understanding of what LLM agents can do, while also serving as a practical guide for those who wish to explore how these agents actually work. Key takeaways from the tutorial include: (i) foundational concepts of LLMs and LLM-based agents; (ii) a curated set of nine concrete examples demonstrating essential agent capabilities and the mechanisms behind them across diverse domains, including scientific discovery, social simulation, software engineering, finance, biology, and medicine; (iii) practical tips for developing LLM agents; and (iv) a brief roadmap of advanced techniques in this rapidly evolving field, such as injecting domain knowledge, improving reasoning abilities, and managing long context.

1 Introduction

Today, we stand on the edge of a technological revolution driven by large language models (LLMs). With LLMs acting as the "brain", a new wave of artificial intelligence (AI) agents—known as LLM-based agents or simply LLM agents—is rapidly emerging [1, 2, 3, 4]. OpenAI's DeepResearch agent [5], for example, can gather, analyze, and synthesize hundreds of online sources to produce a comprehensive report in minutes. Claude Code [6] can navigate complex codebases, help with bug fixes, and manage Git workflows. Pactum's AI negotiator [7] closes deals on behalf of retail giants with vendors. Moreover, research also explores the application of LLM agents in fields like healthcare [8], education [9], customer support [10], scientific discovery [11], and finance [12].

Perhaps surprisingly, although the notion of AI agents has been central to AI research for decades, there remains no universally accepted definition of what precisely constitutes an agent [13, 14, 15, 16].[1] Still, several key capabilities are commonly agreed upon as characteristic features of LLM agents. These capabilities include: (i) memory, retaining, retrieving, and applying information from prior interactions, (ii) role-play, producing agent behaviors that faithfully reflect specified roles or personas, (iii) planning, which decomposes complex tasks into coherent, manageable steps, (iv) reflection, allowing agents to evaluate past actions to inform future decisions and behaviors, (v) tool use, selecting and invoking external tools or APIs to extend the agent's reach, and (vi) cooperation, working with other agents or

Shuyue Hu, Yang Chen, Yiqun Zhang, Dongzhan Zhou, Jia Xu, and Qiaosheng Zhang are with Shanghai Artificial Intelligence Laboratory. Siyue Ren, Chunjiang Mu, Zhiyao Cui, Hao Li, and Zhen Wang are with Northwestern Polytechnical University. Jinyi Liu, Yan Zheng, and Jianye Hao are with Tianjin University. Chu Han is with Guangdong Provincial People's Hospital. Siyue Ren, Jinyi Liu, Chunjiang Mu, and Hao Li contributed to this work during their internship at Shanghai Artificial Intelligence Laboratory.

[1]In seminal studies, Smith [14] define an agent as "a persistent software entity dedicated to a specific purpose", while Franklin and Graesser [15] define an agent "as a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to affect what it sense in the future." Wooldrige and Jennings [16] propose a weak (but relatively uncontentious) notion of agent to be "a hardware or a software-based computer system that enjoys autonomy, social ability, reactivity, and pro-activeness." Given the wide range of applications for agents, such disagreements are unavoidable. In practice, one need not worry excessively about definitions. As Andrew Ng notes (https://x.com/AndrewYNg/status/1801295202788983136), this diversity may actually foster greater inclusion, allowing more people to contribute to agent development.

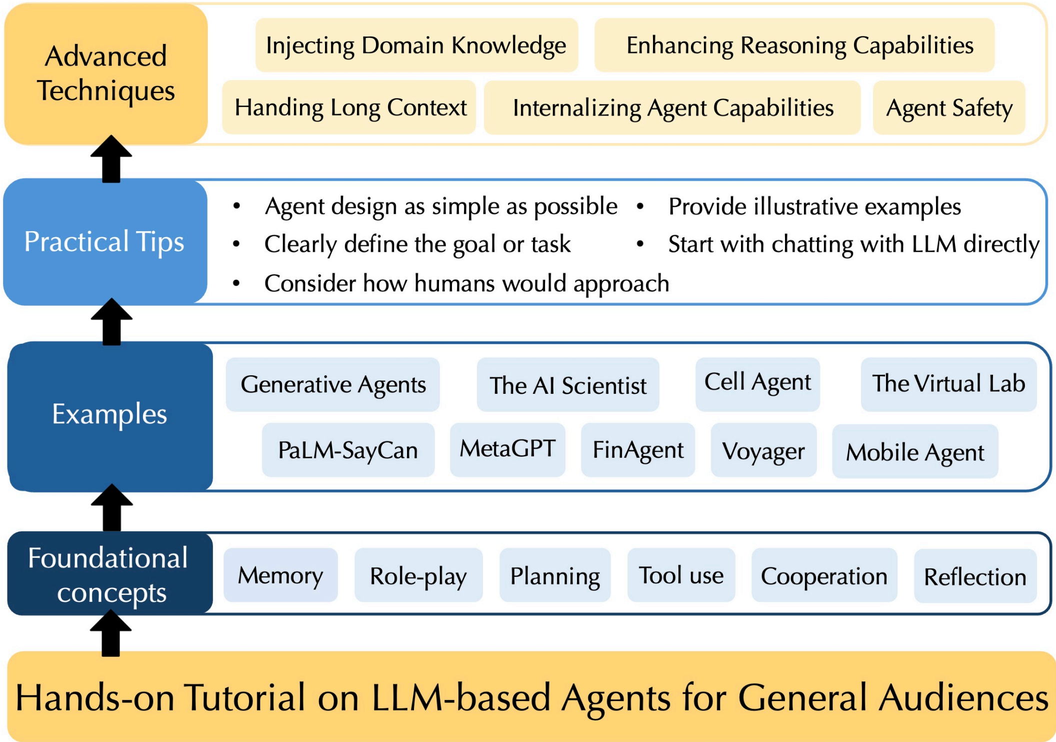


Figure 1: The overall structure of this tutorial.

humans to tackle problems too complex or time-consuming for a single agent. While these capabilities may not always manifest in every agent and typically require adaptation to specific applications, most agents display at least some of these abilities, distinguishing them from native LLMs, which focus primarily on question answering.

This tutorial provides an overview of recent developments in LLM agents for a non-AI researcher interested in the field. As visualized in Figure 1, we begin with foundational concepts of LLMs and key abilities of LLM agents (Section 2). We then present 9 concrete examples that show how LLM agents integrate these key abilities to perform complex tasks across various domains (Section 3), with a summary provided in Table 1. Note that these examples are intentionally selected to be representative and broad in scope, and have been carefully adapted and re-organized to clearly highlight the underlying agent capabilities. Each example is designed to be self-contained, allowing an audience to jump directly to a particular one, without needing to follow them in sequence. The primary approach involved in these examples is crafting prompts—providing specific instructions and contextual information in natural language for LLMs. As such, no coding skills or prior knowledge of LLMs, machine learning, or AI is required to understand these case studies. We believe that for a non-AI researcher, crafting prompts is sufficient to meet most development needs, given the current power of LLMs. To further assist, we offer practical tips for developing LLM agents (Section 4). A roadmap of advanced techniques, such as injecting domain knowledge to steer a general-purpose LLM into a powerful expert-level agent, enhancing the reasoning capabilities for complex tasks, and managing long context for multi-turn or extended workflows is briefly introduced in Section 5. We conclude this tutorial with a summary of key takeaways (Section 6).

2 Preliminaries: What you must know about LLM Agents

Large language models (LLMs), such the GPT, DeepSeek, Gemini, and Claude families, are deep learning models with billions of parameters, pre-trained on vast amounts of data. This data, typically sourced from human-generated content, includes web pages, codebases, textbooks, etc. Through training on these corpora, LLMs develop the ability to understand and generate language.

The fundamental operation that drives all LLM usage and applications is **next-token prediction**. A token is a basic unit of text that the model processes, which can represent words, characters, or subword units. Let x denote the textual input to the model, commonly referred to as the **prompt**. Given the prompt x, an LLM generates output by predicting one token at a time, ultimately producing a sequence of tokens $y = (y_1, y_2, \ldots, y_T)$. Formally, this can be represented as $p(y|x) = \prod_{t=1}^{T} p(y_t|x, y_1, y_2, \ldots, y_{t-1})$. When predicting the next token, both the prompt x and the generated tokens $(y_1, \ldots, y_{t-1})$ lie within a finite **context window**, which determines the maximum length of text the model can use at once.

LLMs are highly versatile. Due to the nature of next-token prediction, they can perform a wide range of tasks, as long as the task and the output can be expressed in text (input x and output y). Today, people can instruct LLMs using textual prompts to generate code, solve math problems, answer questions in healthcare, write essays, translate languages, create poetry, and much more. Given the current capabilities of LLMs, a prompt, which clearly provides instructions or questions and states the context, typically results in output of acceptable quality.

Since there is no unanimous agreement on the definition of agents, the differences between a LLM and a LLM agent are often blurred. In contrast to a single prompt–response interaction with an LLM, LLM agents typically manage multi-stage tasks that unfold through a structured **workflow**, rather than a one-shot model invocation. Moreover, LLM agents are often characterized by several additional capabilities: memory, role-play, planning, tool use, cooperation, and reflection. Note that these abilities can manifest in various forms and be applied in different ways, depending on the context. Below, we briefly outline each capability to provide a general sense of what they mean. Concrete examples of these abilities will be the focus of the next section.

Memory Memory is an agent's capacity to store, recall, and leverage information from prior interactions. Because LLMs have finite context windows and often struggle with very long inputs, simply retaining every detail might be neither practical nor effective. Memory allows an agent to selectively retain and retrieve information [17], merge similar information to reduce redundancy [18], reflect on information to distill high-level thoughts that help the agent generalize. As a result, the agent can build a useful knowledge base, draw on past experience to inform future actions, and remain efficient even as interactions grow over time. Memory can reside in the prompt-level memory or in external storage such as databases.

Role-play Role-play denotes an agent's capacity to take high-level descriptions (such as demographic information, professional role, or personality traits) as input [19], and generate behaviors that consistently reflect those attributes. Studies show that prompting an agent to adopt the persona of "a conservative, white, male, strong Republican" versus "a liberal, white, female, strong Democrat" elicits sharply different political attitudes [20]. Similarly, directing agents to "maximize self-interest" rather than "maximize collective benefit" produces pronounced variations in their willingness to cooperate [21].

Planning Planning is an agent's capacity to decompose a complex task or goal into a coherent sequence of actionable sub-tasks or steps [22]. LLMs still struggle with long-horizon tasks that demand multi-step reasoning and real-time adaptation [23]. Take "book a flight" as an example: the agent must open an airline or aggregator site, enter origin, destination, and dates, compare fares, choose seats, and complete payment. A single error can unravel the entire process, and interface changes on the website require on-the-fly adaptation. Effective planning allows an agent to decompose the task into manageable steps, iteratively refine each step with fine-grained details, and reflect on progress, adapting as conditions change.

Tool use Tool use refers to an agent's ability to select, compose, and invoke external functions, environments, or APIs [24]. LLMs are limited by their static training data; they may be imprecise with factual information, unreliable in performing calculations, and unaware of events occurring after their training cutoff date. A common manifestation of tool use is code generation, where the agent produces executable code that can be run in an external environment (e.g., generating Python scripts to analyze data, simulate processes, or visualize results) [25]. Moreover, an agent can also generate structured function calls to trigger predefined external tools such as calculators, weather APIs, or web search engines [24, 26]. Both the generated code and function calls are executed outside the LLM, and the resulting outputs are fed back into the agent's reasoning loop. This external execution enables the agent to provide more accurate, verifiable, and up-to-date responses than the LLM alone could achieve.

Examples	Domains	Memory	Role-play	Planning	Tool Use	Cooperation	Reflection
GenerativeAgents [31]	Social Simulation	✓	✓	✓			✓
The AI Scientist [11]	Scientific Discovery			✓	✓		✓
Cell Agent [34]	Biology	✓	✓	✓	✓	✓	✓
The Virtual Lab [35]	Medicine			✓	✓	✓	✓
MetaGPT [36]	Software Engineering		✓	✓	✓	✓	
FinAgent [12]	Finance	✓	✓				✓
Voyager	Game	✓		✓	✓		✓
Mobile Agent [32]	Mobile Devices	✓		✓	✓		✓
PaLM-SayCan [33]	Embodied AI			✓	✓		✓

Table 1: An overview of representative examples of LLM agents, their application domains, and demonstrated capabilities.

Cooperation Cooperation is the ability to cooperate with other agents or humans so that tasks too complex or time-inefficient for single agents can be accomplished effectively. This capability requires more than parallel effort. It includes inferring others' goals and intentions [27], communicating efficiently, dividing labor, aligning strategies [28], fostering trust, and resolving conflicts [29]. Sometimes, it may also require detecting deception or betrayal [30], and being prepared to respond appropriately to discourage such behavior and maintain group integrity.

Reflection Reflection mirrors the human ability to reconsider past behaviors and judgments to improve future decisions. It empowers agents to critique its performance, diagnose errors or gaps, and revise subsequent actions accordingly. Studies show that reflecting on past experience helps LLM agents distill prior interactions into more abstract and insightful thoughts [31], summarize overarching lessons from errors [12], and learn from mistakes to refine future behavior [11], thereby improving task performance, robustness, and long-term reliability in complex environments [32, 33].

3 Examples: What LLM agents can do and how they work

In this section, we present a carefully selected set of examples to illustrate the key agent capabilities and their underlying mechanisms (through prompt design) across various applications. These examples are summarized in Table 1, with each presented in a self-contained manner. Audiences can jump directly to a specific example without needing to follow them in sequence.

3.1 Generative Agents: Simulacra of Believable Human Behaviors

Generative Agents provides an LLM-based agent architecture designed to simulate human behaviors within a sandbox environment [31]. Because human behavior is inherently vast, dynamic, and context-dependent, traditional agents often struggle to produce behavior that appears genuinely human believable. *Generative Agents* address this by memorizing past experiences, reflecting on them, and generating plans and actions that remain coherent in the moment and over time. They can wake up at fixed times, prepare meals, and go to work; they can also initiate conversations with others, make plans for the future, and coordinate a party.

Role-play This capability is crucial for generative agents, as their high-level agent descriptions shape the types of agents they become. The agent descriptions are composed of three aspects: (i) *basic information*, which specifies attributes such as the agent's name, age, and innate traits; (ii) *current state*, which indicates what the agent is presently engaged in, for example, hosting a Valentine's Day party; (iii) *lifestyle*, which outlines the agent's daily routines, such as when it wakes up and goes to sleep. These aspects are not fixed and can be adapted based on different tasks. To illustrate, we present the description of generative agent Isabella Rodriguez, showing what such a high-level description looks like in practice. These agent descriptions guide their behavior in role-playing, such as by enabling them to generate conversations that reflect their characteristics and make decisions consistent with their preferences.

Example of an agent description

Basic information:
"*name*": "Isabella Rodriguez",
"*age*": 34,
"*innate*": "friendly, outgoing, hospitable",
"*learned*": "Isabella Rodriguez is a café owner of Hobbs Cafe who loves to make people feel welcome. She is always looking for ways to make the café a place where people can come to relax and enjoy themselves."
Current state: "Today is the day! Isabella Rodriguez is filled with excitement as the Valentine's Day party at Hobbs Cafe is finally here. She has been tirelessly planning and preparing for this event, ... Isabella is confident that her efforts will pay off and that the party will be a great success."
Lifestyle: "Isabella Rodriguez goes to bed around 11pm, awakes up around 6am."

Memory Memory in generative agents is stored in a JSON-format database that records and retrieves past information, ensuring consistency with prior experiences. Agent memory consists of three types: (i) *event*, which records observations of the environment or other agents (e.g., "Isabella Rodriguez is packing her bag"); (ii) *chat*, which captures conversations with other agents; and (iii) *thought*, which stores an agent's reflections (e.g., "Maria Lopez is planning something"). Retaining every detail of past history is impractical for LLMs. Generative Agents therefore retrieve memories based on three factors: *recency*, *relevance*, and *importance*. Recency, modeled with an exponential decay function, prioritizes recently accessed memories so that the most recent events remain salient. Relevance, computed via cosine similarity, assigns higher scores to memories related to the current situation. Importance distinguishes mundane from significant experiences by prompting LLMs to rate how meaningful a memory is. We provide an example that demonstrates how to instruct LLMs to assess importance.

Example of rating memory importance

INPUT:
Task: On a scale of 1 to 10, where 1 is purely mundane (e.g., brushing teeth, making the bed) and 10 is extremely poignant (e.g., a breakup, college acceptance), rate the likely poignancy of the following piece of memory.
Memory: buying groceries at The Willows Market and Pharmacy
Rating: <Completed by LLMs>

OUTPUT:
3

Reflection Over time, an agent's memory may accumulate excessive and redundant records; to address this, the agent can reflect on extended chats and summarizes them into higher-level thoughts. Below is a prompt template for condensing the conversation between Isabella and Francisco about preparing for the Valentine's Day party into one sentence.

Example of reflecting on chats to yield high-level thoughts

INPUT:
Chat:
(Isabella Rodriguez): Francisco! I'm so glad you're here early. The decorations are all set, and the guests are already asking about tonight's party. How are you feeling?
(Francisco Lopez): Honestly, Isabella, I'm thrilled! This is my first Valentine's event at Hobbs Cafe, and the atmosphere is perfect. I can't wait ......
Currently: "Today is the day! Isabella Rodriguez is filled with excitement as the Valentine's Day party at Hobbs Cafe is finally here. ...... Isabella is confident that her efforts will pay off and that the party will be a great success."
Task: Summarize the most relevant statements above that can inform (Isabella Rodriguez) in their conversation with Francisco Lopez.

OUTPUT:
Isabella Rodriguez and Francisco Lopez are discussing their excitement for the Valentine's Day party at Hobbs Cafe and working together to make his performance engaging and memorable.

Planning Plans specify a future sequence of actions that promote consistent behavior over time. In generative agents, a daily plan is generated based on the agent's character profile, covering its basic attributes, current state, and lifestyle. Each daily plan comprises ordered entries, each with a location, start time, and duration. After the plan list is generated, the agent executes the entries sequentially and, when circumstances change, adjusts it dynamically based on the current state. Below, we illustrate how Isabella Rodriguez's plan on the Valentine's Day can be generated and represented.

Example of Isabella Rodriguez's planning for Valentine's Day

INPUT:
Character:
Basic Information:
Name: Isabella Rodriguez
Age: 34 ......
Current State: Today is the day! Isabella Rodriguez is filled with excitement as the Valentine's Day party at Hobbs Cafe is finally here...
Lifestyle: Isabella Rodriguez goes to bed around 11pm, wakes up around 6am.

Current Date: Tuesday, February 14
Task: Today is (current date). Here is (agent's name)'s plan today in broad strokes (with times, e.g., lunch at 12:00 p.m., watch TV 7–8 p.m.):
1. 06:00 AM: Wake up and complete morning routine. 2. <completed by LLMs>

OUTPUT:
06:00 AM: Wake up and complete morning routine
07:00 AM: Prepare breakfast and review final details for the Valentine's Day party
08:00 AM: Open Hobbs Cafe and greet early customers
......
11:00 PM: Clean up with staff and reflect on the success of the event before going to bed

3.2 The AI Scientist: Towards Fully Automated Scientific Research

The *AI Scientist* [11] aims to independently carry out a scientific research process from idea generation to paper writing without any human involvement. It produces a research paper in three phases: (1) *idea generation*, (2) *experimental iteration*, and (3) *paper write-up*. First, the AI Scientist generate a diverse set of ideas that are different from existing literature. Then the AI Scientist executes the proposed experiments and visualizes its results during experimental iteration. In paper write-up, it produces a concise and informative write-up of its progress in the style of a standard machine learning conference in LaTeX. Afterward, an LLM-generated review is introduced to assess the quality of the generated paper for the AI Scientist to revise accordingly.

Planning Planning helps the AI Scientist break down a complex goal into smaller, sequential steps. For example, during experimental iteration, the AI Scientist plans the experimental steps for each selected idea, writes code and conducts experiments according to these plans. In the paper write-up phase, the AI Scientist organizes its scientific findings into research papers. Overall, the planning ability helps the AI Scientist organize its multi-step research process, allowing it to generate, improve, carry out ideas, and write research papers in a clear and purposeful way. We provide an example of how the AI Scientist generates an experimental plan below.

Example of planning an experiment list for the generated idea

INPUT:
Idea: Improving image classification through data augmentation.
Instruction: Your goal is to implement the given idea. Plan the list of experiments you would like to run. You are given a total of up to 5 runs to complete the necessary experiments.

OUTPUT:
Here's the list of experiments I propose:
1. Prepare Data: Load and inspect the CIFAR-10 dataset.
2. Build Model: Set up a simple convolutional neural network.
3. Train Baseline: Train the model without data augmentation and record accuracy.
4. Apply Augmentation: Add image transformations such as flipping, rotating, and adjusting the brightness.
5. Train Again: Retrain the model with augmentation and compare results.
6. Evaluate: Measure test accuracy and visualize performance improvements.

Tool use In all three phases, the AI Scientist extensively leverages tools. During idea generation, the AI Scientist evaluates the novelty of ideas by retrieving literature. It prompts an LLM to formulate search queries (such as related paper titles or keyword combinations), and then passes these queries to a built-in Python module that calls the Semantic Scholar API[2] to get related literature. In experimental iteration, to carry out experiments, the AI Scientist employs Aider[3], a coding assistant. The Aider takes the experimental plan generated by the LLM as input and then automatically generates code to execute the plan. In paper write-up, after the LLM generates the manuscript in LaTeX format, the AI scientist calls a LaTeX compiler to process the tex file and produces the corresponding PDF. The following is an example of generating a literature retrieval query.

Example of generating query for literature retrieval

INPUT:
Idea: Improving image classification through data augmentation.
Instruction: You have an idea and you want to check if it is novel. You will be given access to the Semantic Scholar API, which you may use to survey the literature and find relevant papers to help you make your decision. Please provide a search query to search the literature.

OUTPUT:
Query: Enhancing the image classification performance of convolutional neural networks using data augmentation.

The generated query is used as a parameter of the following built-in Python function, which retrieves related work by invoking Semantic Scholar API.

[2] https://www.semanticscholar.org/product/api#api-key
[3] https://github.com/Aider-AI/aider

Python code of literature retrieval

```python
def search_for_papers(...):
    results = requests.get(
        "https://api.semanticscholar.org/paper/search",
        headers={"API_KEY": API_KEY},
        params={
            "query": "Enhancing the image classification performance of convolutional
                     neural networks using data augmentation",
            "fields": "title,authors,year,abstract,citationCount",
        },
    )
    papers = results["data"]
    return papers
```

Below is an example showing the return value of the function "search_for_papers()", which is the retrieved papers.

Examples of the retrieved papers

```
{
    "title": "ImageNet Classification with Deep Convolutional Neural Networks",
    "authors": "Krizhevsky, Sutskever, Hinton",
    "year": 2012,
    "abstract": "We trained a large, deep convolutional neural network to classify
                 the 1.3 million high-resolution images in the LSVRC-2010...",
    "citationCount": 149105,
},
...
```

Reflection Human scientific research relies heavily on reflection, and the AI Scientist is no exception. When generating ideas, the AI Scientist use self-reflection to refine each proposed idea. During experiment iteration, it analyzes the experimental results and adjusts methods accordingly. In paper write-up, the AI Scientist perform two rounds of self-reflection section-by-section, aiming to improve the accuracy, conciseness, and coherence of each section. The most important reflection is based on the Automated Paper Reviewing mechanism. The PDF version of the generated paper is converted into text and reviewed by an LLM-based reviewer agent. The review feedback is used by the AI Scientist to improve the paper writing. An example for generating review comments is as follows.

Example of generating review comments by the reviewer agent

INPUT:
Instruction: You are an AI researcher who is reviewing a paper that was submitted to a prestigious ML venue. Be critical and cautious in your decision. If a paper is bad or you are unsure, give it bad scores and reject it. Below is a reviewing guidelines. When writing your review, please keep in mind that after decisions have been made, reviews and meta-reviews of accepted papers and opted-in rejected papers will be made public.
Paper:
Data Augmentation for Improved Image Classification
Abstract
This paper explores the use of data augmentation techniques to enhance the performance of convolutional neural networks in image classification tasks...
Introduction
Deep learning models, particularly convolutional neural networks (CNNs), have achieved remarkable success in image classification. However, their performance heavily depends on the availability of large, diverse datasets...
...
Guidelines:
Review Form
Below is a description of the questions you will be asked on the review form for each paper and some guidelines on what to consider when answering these questions...
1. Summary: Briefly summarize the paper and its contributions...
2. Questions: Please list up and carefully describe any questions and suggestions for the authors....
3. Limitations: Have the authors adequately addressed the limitations and potential negative societal impact of their work?...

OUTPUT:
Summary:
This paper explores the application of data augmentation techniques in image classification tasks...
Strengths: 1. Detailed ablation studies analyzing the contribution of different augmentation strategies. 2. ...
Weaknesses: 1. Lacks comparison with state-of-the-art automatic data augmentation methods. 2. ...
Questions: 1. Why was this specific combination of augmentation strategies chosen? 2. ...
Overall: 4
Confidence: 4
Decision: Reject

3.3 CellAgent: Automating Biological Data Analysis through Natural Language

CellAgent [34] is an autonomous, LLM-driven multi-agent framework that, given raw single-cell RNA sequencing (scRNA-seq) and spatial transcriptomics (ST) data plus natural language instructions, automates end-to-end analysis to deliver high-quality visualizations and processed data outputs. Traditional analysis in this field requires extensive programming and domain expertise, creating a significant barrier for many biologists. CellAgent addresses this by employing a hierarchical framework of specialized agents that plan, execute, and evaluate tasks, simulating a "deep-thinking" process to ensure high-quality, automated scientific discovery, with transparent reasoning, iterative refinement, and clear outputs across all steps.

Role-play CellAgent's architecture is built on a clear division of labor among three specialized agents: a *Planner*, an *Executor*, and an *Evaluator*. Each agent has a distinct role modeled after the workflow of a human expert. The Planner acts as a project lead, interpreting the user's high-level goal and breaking it down into a logical sequence of analytical steps. The Executor functions as a bioinformatician, selecting the appropriate tools, writing code, and running the analysis for each step. The Evaluator serves as a quality control specialist, assessing the output of the Executor to ensure biological accuracy and coherence, enabling iterative refinement without human intervention.

Example of role-playing as a planner

Name: Planner
Profile: A project manager with deep expertise in bioinformatics workflows.
Goal: To interpret a user's natural language request and decompose it into a logical, step-by-step analytical plan suitable for single-cell data analysis, leveraging expert-curated prompts to enhance domain-specific understanding.
Expert Knowledge: The planner should ground decisions in expert-provided principles and best practices for planning, including: QC → normalization → highly variable genes (HVGs) → dimensionality reduction (PCA, UMAP) → batch correction (if needed) → neighborhood graph → clustering → annotation → downstream analyses, ...
Output Format: Return 2 parts: *"Thought":* Brief rationale summarizing assumptions, data modality, and the high-level strategy. *"Plan":* An ordered list of actionable steps.

Example of role-playing as an executor

Name: Executor
Profile: A bioinformatician specializing in scRNA-seq data analysis.
Goal: Given a specific sub-task (e.g., "Perform cell type annotation"), select the best tool from the sc-Omni toolkit, generate correct Python/R code, and execute it in the code sandbox.
Constraint: Must use tools and parameters documented in sc-Omni. The available tools for this step are `CellTypist` | `scType` | `CellMarker2.0`. Ensure code is executable and handles data objects from previous steps correctly.
Output Format: Return 2 parts: *"Thought":* Brief rationale for tool selection, key parameters, and input assumptions. *"Code":* Provide an executable code block (Python or R).

Example of the Evaluator

Name: Evaluator
Profile: A quality control expert in bioinformatics, proficient in assessing the statistical and biological validity of single-cell analysis results.
Goal: To critically evaluate the results generated by the Executor. This includes analyzing plots (`UMAP` | `Heatmaps`) and quantitative metrics (`ARI` | `ASW`) to determine if the outcome is biologically meaningful and technically sound. If the result is suboptimal, provide specific, actionable feedback for improvement (e.g., "adjust parameter X," "try tool Y").
Constraint: Must base evaluations on established biological principles and predefined metrics, effectively replacing traditional manual assessments.
Output Format: Return 2 parts: *"Evaluation Thought":* Concise reasoning that interprets visualizations and metrics against biological/technical criteria; note any suspected issues (e.g., overclustering, batch effects). *"Feedback":* Actionable next steps.

Planning Planning in CellAgent is a hierarchical process managed by the Planner agent. Upon receiving a user's request in natural language, the Planner leverages its expert-curated knowledge (as given in their role-play prompt shown above) to decompose the complex task into a series of well-defined, sequential sub-tasks. This ensures the overall analysis follows a logically sound and scientifically valid workflow. For instance, a simple request for "clustering" is broken down into essential preprocessing steps before the final clustering is performed, mirroring best practices in the field.

Example of planning for clustering (Planner role)

INPUT:
Core Instruction: This is a 3k PBMCs dataset, please help me complete clustering.

OUTPUT:
Thought: To perform clustering, we must first ensure the data is properly prepared. I will create a plan that includes quality control, normalization, and dimensionality reduction before the final clustering step.
Plan:
"Quality Control": Filter out low-quality cells and genes.
"Normalization": Standardize expression values across cells.
"Identification of Highly Variable Genes": Focus the analysis on the most informative genes.
"Dimensionality Reduction": Project data into lower dimensions (e.g., PCA, UMAP).
"Clustering": Identify groups of similar cells.

Tool use The Executor agent is equipped with a comprehensive, expert-curated toolkit named sc-Omni. This toolkit consolidates a wide range of essential software for single-cell analysis (e.g., Scanpy, CellTypist, Harmony) and provides detailed documentation on their usage. The Executor intelligently select the most appropriate tool for a given sub-task, understand its parameters, and generate the corresponding code. This structured approach overcomes the limitations of general-purpose LLMs, which often lack the specialized knowledge to use these tools correctly.

Example of tool use (Executor role)

INPUT:
Current Sub-task: Step 5: Cell Type Annotation
Available Tools: sc-Omni toolkit, including `Cellmarker` | `CellTypist` | `scType`.

OUTPUT:
Thought: The current task is cell type annotation. I will consult the sc-Omni documentation to find suitable tools. Both "Cellmarker" and "Celltypist" are effective for this purpose. I will select Cellmarker for this step as it provides a robust database of markers for annotation.
Code:

```
# Generate Python code using the Cellmarker tool.
from sc_omni.tools import cellmarker
adata = cellmarker.annotate(adata, tissue='blood', cell_type='pbmc')
```

Reflection CellAgent incorporates a self-reflective optimization mechanism driven by the Evaluator agent. After the Executor runs a piece of code, the Evaluator assesses the quality and correctness of the output, such as plots or statistical results. This process replaces traditional manual assessment. If the results are suboptimal (e.g., poor cell clustering, insufficient batch correction), the Evaluator provides critical feedback and suggests corrective actions, such as adjusting hyperparameters or trying an alternative tool. This iterative loop of execution and reflection ensures the final results are robust and accurate.

Example of reflection (Evaluator role)

INPUT:
Last Operation: Executor ran clustering with leiden_resolution=0.5.
Current Result: UMAP plot showing poorly separated cell clusters. `<img>`
Task: Evaluate the clustering result and propose an optimization strategy.

OUTPUT:
Evaluation Thought: The clusters in the UMAP visualization are overlapping, suggesting the current resolution parameter is too low to capture the underlying biological heterogeneity. The quality is insufficient.
Feedback: The clustering is suboptimal. Re-run the execution step with a range of higher resolution values (e.g., 0.8, 1.0, 1.2) to achieve better separation.

Cooperation Cooperation in CellAgent follows a structured, hierarchical workflow rather than a conversational model. The agents coordinate by passing structured information sequentially. The Planner creates a high-level plan and passes the first sub-task to the Executor. The Executor completes the sub-task and hands its result (e.g., a processed data object and visualizations) to the Evaluator. The Evaluator provides feedback to the Executor, which may trigger a refinement of the current sub-task. Once a sub-task is successfully completed, the Executor proceeds to the next one in the plan, ensuring an orderly and systematic progression through the entire analysis.

Memory To enable efficient cooperation among agents and maintain context throughout a complex, multi-step analysis, CellAgent employs a global and local integrative memory control mechanism. This system systematically stores and manages historical information to optimize retrieval efficiency and enhance task execution. *Global Memory* maintains the high-level context, storing the complete analytical plan generated by the Planner and the final result for each step. This ensures all agent actions remain aligned with the overall scientific objective. *Local Memory*, in contrast, captures the operational details within each sub-task, including the specific code, parameters, and feedback from the Evaluator. This Memory mechanism is essential for ensuring consistency between sequential steps and enabling the iterative, self-reflective optimization loop.

3.4 The Virtual Lab: AI–human collaboration in medical research

The Virtual Lab [35] facilitates interdisciplinary research through collaboration between AI agents and human researchers. Initially, the human researcher defines two primary agents—a Principal Investigator (PI) and a Scientific Critic. The PI agent automatically assembles a specialized team of scientific agents tailored to the specific research topic. Research in the Virtual Lab occurs through two types of meetings: team meetings and individual meetings. In both cases, the human researcher provides an initial agenda to guide the discussion, and then the agents discuss how to address the agenda. After running multiple team or individual meetings in parallel, the PI agent conducts a final individual aggregation meeting. With assistance from the human researcher, the PI synthesizes the previously generated tool scripts and summarizes the outcomes of earlier discussions, constructing a comprehensive workflow tailored to the initial research topic. Finally, the human researcher leverages this finalized workflow, employing the generated tools to complete the research project.

Role-play The Virtual Lab introduces two key roles: a Principal Investigator (PI) agent acting as the primary coordinator, and a Scientific Critic agent that provides critical oversight and identifies potential risks. Together, they ensure high-quality decision-making and maintain workflow transparency. Below we use the PI agent as an example to illustrate the key elements of an agent role:

Example of role-playing as a principal investigator

Title: Principal Investigator
Expertise: Running a science research lab.
Goal: Perform research in your area of expertise that maximizes the scientific impact of the work.
Role: Lead a team of experts to solve an important scientific problem, make key decisions about the project direction based on team member input, and manage the project timeline and resources.

Reflection Once the human researcher defines a high-level goal, the PI agent determines which specialist scientist agents are needed (such as Sequence Designers, Structure Predictors, Energy Evaluators, and Evidence Retrievers) in an individual team generation meeting. Individual meetings allow an agent to focus on a specific task, the Scientific Critic provides targeted feedback after each response, guiding the agent toward improved outcomes. During the team generation meeting, the PI outlines each agent's title, expertise, goal, and role, the Scientific Critic then reviews these assignments and provides feedback to refine the roles before execution begins. The following example illustrates how a PI agent might generate a suitable team for a given research task and how the reflection process operates in practice.

Example of team generation (Principal Investigator)

INPUT:
Project: You are working on a research project to use machine learning to develop antibodies or nanobodies for the newest variant of the SARS-CoV-2 spike protein ...
Task: Generate a team of 3 scientists to help you with this project.

OUTPUT:
Agent 1:
Title: Computational Biologist
Expertise: Protein structure prediction and modeling.
Goal: Develop and refine predictive models for protein-antibody interactions.
Role: Provide insights into molecular dynamics and structural constraints impacting antibody design.
Agent 2: ...
...

Example of reflection (Scientific Critic)

INPUT:
Chat History: ...
Task: Please critique (agent)'s most recent answer.

OUTPUT:
As the Scientific Critic, I would like to address ...

Cooperation Team meetings involve collaborative discussions among the PI, scientist agents, and the Scientific Critic. The human researcher sets an agenda, and the PI initiates the conversation by outlining key questions. Each agent then provides their insights in turn. After several discussion rounds, the PI synthesizes the input, summarizes important decisions, and identifies next steps. The human researcher can quickly understand outcomes by reviewing only the PI's concise summary. An example of a team cooperation scenario is provided below.

Example of team meeing start (Principal Investigator)

INPUT:
Agenda (Provided by human researcher): You are working on a research project to use machine learning to develop antibodies or nanobodies for the newest variant of the SARS-CoV-2 spike protein. Please create ...
Agenda questions (Provided by human researcher):
1. Will you design standard antibodies or nanobodies?
2. ...
Team members: Computational Biologist, Scientific Critic, ...
Task: Provide your initial thoughts on the agenda as well as any questions you have to guide the discussion among the team members.

OUTPUT:
Thank you, everyone, for joining this important meeting. Our task is to ...

Example of team member discussion (Computational Biologist)

INPUT:
Chat History: ...
Task: Provide your thoughts on the discussion (round 2 of 5).

OUTPUT:
As the Computational Biologist, I will focus on ...

Example of summarizing the discussion (Principal Investigator)

INPUT:
Chat History: ...
Task: Summarize the meeting in detail for future discussions, provide a specific recommendation regarding the agenda, and answer the agenda questions (if any) based on the discussion while strictly adhering to the agenda rules (if any).

OUTPUT:
Summarization of each team member:
Computational Biologist: ...
...
Recommendation: Based on the input from the team, I recommend focusing on ...
Answers of agenda questions:
1. Will you design standard antibodies or nanobodies?
Answer: Nanobodies.
Justification: Nanobodies offer superior stability, tissue penetration, and ...
2. ...
Next Steps:
1. *Data Acquisition and Verification*: Collaborate with partners and use automated alerts to acquire and verify the latest cross-reactivity data for selected nanobodies.
2. *Model Development*: ...

Tool use After finalizing project details in the team meeting, the agents move forward with the help of human researchers to complete the steps of tool generation and tool implementation. In the tool generation phase, the agents hold another team meeting to collaboratively choose the computational tools they will use. Here is an example showing how the team generates appropriate tools during the meeting.

Example of tool generation discussion (Computational Biologist)

INPUT:
Project: You are working on a research project to ...
Summaries of the previous meetings: Your team previous decided to modify existing nanobodies to improve their binding to the newest variant of the SARS-CoV-2 spike protein ...
Chat History of current meeting: ...
Task: Please list several tools (5-10) that would be relevant.

OUTPUT:
... Apply AlphaFold or Rosetta to model the 3D structures of the top-ranked nanobody variants ...

Once the toolset is agreed upon, the implementation phase begins. First, the PI holds an individual meeting to assign specific computational tools to each specialist agent. Next, each specialist agent conducts their own individual meetings, guided and assisted by the human researcher, to generate structured scripts for the assigned tools. Under the guidance of the human researcher, the agents iteratively refine these scripts through feedback. The human researcher then manually stores the finalized scripts for subsequent implementation. The following example illustrates how a specialist agent generates a structured Rosetta script under the guidance of a human researcher:

Example of tool implementation (Computational Biologist)

INPUT:
Project: You are working on a research project to ...
Decision: Your team previous decided to modify existing nanobodies to improve their binding to the newest variant of the SARS-CoV-2 spike protein.
Task*(Provided by human researcher)***:** Now you must use Rosetta to calculate the binding energy of nanobody-antigen complexes. You must do this in **3 parts**. First, write a complete RosettaScripts XML file ...

OUTPUT:
Part 1: RosettaScripts XML for Binding Energy Calculation

```
<xml>
    ...
</xml>
```

Part 2: Rosetta Command Example

```
```bash
    rosetta_scripts.default.linuxgccrelease ...
```
```

Part 3: Python Script to Process Score Files

```
import os
import pandas as pd
...
```

3.5 MetaGPT: Multi-agent framework that write code efficiently

MetaGPT [36] is an LLM-based multi-agent framework designed to mirror established software engineering practices. Modern software engineering typically decompose complex workflows into distinct processes to manage complexity, enhance productivity, and ensure software quality. MetaGPT replicates this process by: (i) explicitly assigning specialized roles to agents; (ii) structuring detailed planning and task decomposition; (iii) leveraging tools for automated code generation; and (iv) facilitating iterative debugging through inter-agent cooperation.

Role-play Role-play serves as a foundational capability within MetaGPT, shaping how individual agents function and collaborate. In MetaGPT, roles include *Product Manager*, *Architect*, *Project Manager*, *Engineer*, and *QA Engineer*, where the *Product Manager* defines requirements, the *Architect* designs system structure, the *Project Manager* oversees workflows, the *Engineer* implements code, and the *QA Engineer* ensures quality. Specifically, role-play defines the agent's identity, responsibilities, and interaction constraints through four components: (i) *Name*, (ii) *Profile*, detailing the agent's professional identity, such as a Product Manager or Software Architect, along with their area of expertise, (iii) *Goal*, clarifying the agent's primary tasks within the team, for instance, generating product requirements or performing code reviews; and (iv) *Constraint*, specifying explicit standards, such as adhering to coding conventions and ensuring modularity, readability, and maintainability. To illustrate this clearly, we take the *Product Manager* as an example.

Example of role-playing as a product manager

Name: Alice
Profile: Product Manager
Goal: Create a Product Requirement Document or market research/competitive product research
Constraint: Utilize the same language as the user requirements for seamless communication

Planning In MetaGPT, *planning* involves systematically decomposing complex software engineering tasks into structured, actionable procedures clearly assigned to specific agent roles. For example, the *Product Manager* analyzes user requirements and produces a detailed Product Requirements Document

(PRD) that explicitly defines product goals, user stories, and a prioritized requirement pool. This structured PRD serves as a foundational functional breakdown, guiding subsequent agents through detailed system design, task distribution, and code implementation.

Example of planning for product requirement document

INPUT:
Agent description: You are Alice, a Product Manager responsible for analyzing user requirements and creating clear, structured Product Requirement Documents (PRDs)
High-level user requirements: Make the 2048 sliding tile number puzzle game
Task: Generate a structured Product Requirement Document (PRD) clearly including well-defined product goals that describe the intended outcomes and value proposition, detailed user stories explicitly covering core functionalities from the user's perspective, a comprehensive competitive analysis systematically identifying existing solutions and analyzing their strengths and weaknesses, and a prioritized requirement pool organizing all identified tasks by priority levels

---

OUTPUT:
Product Goals: Simple, responsive, user-friendly 2048 puzzle game for desktop and mobile.
User Stories: Quickly start games, slide tiles to merge numbers, clearly view scores, easily reset.
Competitive Analysis: Original 2048 is simple but limited; mobile versions have good UI but intrusive ads; Threes! is polished yet paid.
Prioritized Requirement Pool: Core gameplay mechanics (high), score tracking and reset functionality (medium), responsive design and customization (low).

Cooperation Cooperation within MetaGPT revolves around the shared message pool mechanism, which enables structured and efficient collaboration among specialized agents. Agents publish their structured outputs, such as product requirement documents and system designs, directly into a global message pool, allowing access by other agents. Agents selectively subscribe to messages relevant to their roles, reducing information overload and ensuring efficient collaboration. This publish-subscribe approach ensures efficient division of labor. Below, we illustrate how the *Product Manager* assigns tasks to the *Architect*.

Example of the product manager assigning tasks to the architect

INPUT:
Agent description: You are Alice, a Product Manager responsible for analyzing user requirements and creating clear, structured Product Requirement Documents (PRDs)
Task: You are the Product Manager, responsible for assigning tasks to the Architect. You must ensure all necessary information is included in your message because you are the sole source of information for your team. When publishing messages: Clearly specify the task. Provide relevant paths, links, environment details, programming languages, frameworks, requirements, and constraints. Address the message explicitly to the receiving agent. Wrap the parameters in JSON format: *content* for task details, and *send_to* for the receiving agent's name.

OUTPUT:
Message:

```
{
    "content": "Bob (Architect), the PRD is ready at docs/PRD.md.
                Please draft the system design accordingly.",
    "send_to": "Bob"
}
```

Tool use Tool use is essential in software development, enabling efficient interaction with external resources and real-world development environments. In MetaGPT, specialized agents employ dedicated tools aligned with their roles to streamline workflows and reduce errors. Tools are strictly aligned with agent roles: Product Managers employ market analysis tools, Architects utilize design visualization tools, and Engineers leverage coding tools (Editor, Terminal, Code Review) to streamline task execution and reduce errors. Below, we provide a concrete example illustrating how the *Engineer* agent utilizes these tools in practice.

Example of the engineer reviewing the system design using the `editor.open_file` tool

INPUT:
Agent description: You are Alex, an autonomous Engineer who writes elegant, production-ready code for game, app, and web projects. You implement features following system designs, prioritizing code quality, maintainability, and deployment readiness.

<System Design provided by the Architect>

Current Task: Implement `game.py` according to the system design, which includes the Game class with grid management, scoring logic, and tile movement methods.

While coding, you have access to specialized tools such as:
Tools for Editor:

- `open_file(path)`: Opens a file and returns its content.
- `write(path, content)`: Creates or overwrites a file with complete content.
- ...

Tools for Terminal: `run_command(cmd, parameters)` executes shell commands and returns output. The `cmd` parameter accepts any valid shell command (e.g., `ls`, `cat`, `cd`).
Tools for CodeReview: `review(patch_path, output_file)` analyzes code changes and generates review comments.
...
Analyze the current task, then select the most appropriate tool to proceed. Your response must include: (1) **Reason** - explain why you need this tool, (2) **Tool** - specify the exact tool name (e.g., Editor.open_file), and (3) **Tool Parameters** - provide all required parameters in JSON format.

OUTPUT:
Reason: Before implementing game.py, I need to carefully review the complete system design. Opening the system design document allows me to reference these specifications while coding.
Tool: Editor.open_file
Tool Parameters:

```
{
  "path": "docs/system_design.json"
}
```

After the LLM determines the required tool and specifies its parameters, the agent runs a shell command to execute the tool and retrieves the requested file's contents. The obtained file data is then incorporated into the agent's prompt, thereby improving the reliability of subsequent coding tasks.

3.6 FinAgent: A Multimodal Foundation Agent for Financial Trading

FinAgent [12] is a multimodal foundation agent designed for financial trading that addresses adaptability and generalization through three key components: (i) a *market intelligence module*, extracting and summarizing key insights from diverse market information, including daily updates on stock news, prices, and monthly and quarterly financial reports; (ii) a *reflection module*, performing dual-level reflection on market dynamics and past decisions; and (iii) a *tool-augmented decision-making module*, enhancing trading actions through expert-informed guidance.

Role-play FinAgent is guided by a high-level role description that ensures consistent expertise in market analysis and financial decision-making. This role-play specification contains three aspects: the definition of the agent's role, the context of the financial market environment, and the agent's analytical capabilities for trading.

Example of role-playing as an expert financial trader

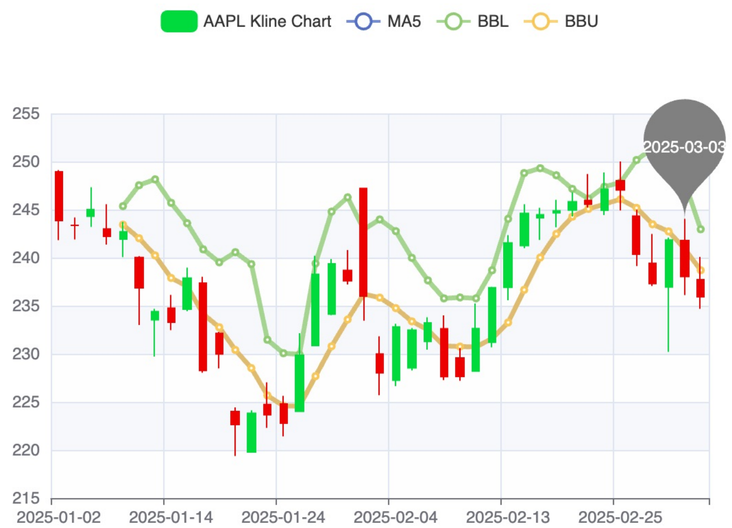
Role: You are an expert trader who have sufficient financial experience and provides expert guidance.
Context: Imagine working in a real market environment where you have access to various types of information (e.g., daily real-time market price, news, financial reports, professional investment guidance and market sentiment) relevant to financial markets. You will be able to view visual data that contains comprehensive information, including Kline charts accompanied by technical indicators, historical trading curves and cumulative return curves. And there will be some auxiliary strategies providing you with explanations for trading decisions.
Capabilities: You are capable of deeply analyzing, understanding, and summarizing information, and use these information to make informed and wise trading decisions (i.e., BUY, HOLD and SELL).

Memory Memory plays a crucial role in multimodal LLM-based financial trading agents. In FinAgent, the memory module adopts a vector-based storage architecture consisting of three main components: (i) *summary of market intelligence*, which collects, summarizes, analyzes, and extracts key insights from diverse multimodal financial data sources; (ii) *summary of low-level reflection*, which identifies potential patterns in price movements and derives insights from market intelligence summaries; and (iii) *summary of high-level reflection*, which stores and reflects on past trading decisions. The stored summaries are subsequently used as inputs for other modules. For example, summaries of both past and latest market intelligence are provided as input data to the reflection prompts.

Reflection FinAgent incorporates a dual-level reflection mechanism: (i) *Low-level reflection*, which analyzes price change data to reason about and identify potential patterns in market movements; and (ii) *High-level reflection*, which focuses on past trading decisions, summarizing overarching lessons from both successes and failures to recommend improvements and suggest corrective trading actions.

Example of analyzing market dynamics with low-level reflection

INPUT:
Summaries of latest market intelligence: Positive sentiment prevails with expectations around Apple's new AR/VR...The overall market sentiment appears POSITIVE in the medium term...
Summaries of past market intelligence: Apple shows resilience and innovation strength, which are likely to maintain investor confidence and could drive the company's market value upward...
Kline chart:



Task: As the above Kline chart shows, Short-Term (provided by *Market intelligence*): Over the past 1 day, the price movement ratio has shown an increase of 1.60%, Medium-Term (provided by *Market intelligence*): ... You should analyze the above information on the reasoning that lead to past to now price movements...

Desired format:
reasoning: This field will be used for trading decisions...provide the detailed reasoning to determine how the above information that lead to the price movements...
query: (tailored to enable focused retrieval of specific types of information) This field will be used to retrieve past reasoning for price movements...

OUTPUT:
reasoning: The short-term price increase of 1.60% for Apple Inc. (AAPL) can be attributed to recent positive market sentiment surrounding Apple's upcoming AR/VR product line. Over the past week, AAPL has seen a 3.38% rise can be linked to positive expectations of new product releases...
query: Short-term growth in AAPL's price is driven by reactions to product excitement and concerns, medium-term gains come from the positive...

Example of analyzing past decisions with high-level reflection

INPUT:
Summaries of latest market intelligence: Positive sentiment prevails with expectations around Apple's new AR/VR...The overall market sentiment appears POSITIVE in the medium term...
Summaries of past market intelligence: Apple shows resilience and innovation strength, which are likely to maintain investor confidence and could drive the company's market value upward...
Reasoning of low-level reflection: The short-term price increase of 1.60% for AAPL can be attributed to recent positive market sentiment surrounding Apple's upcoming AR/VR product line...
Historical action and reasoning: Trading decision and reasoning made by your assistant for the past 14 days are as follows: 2023-06-01. BUY... The analysis indicates a bullish future trend, aligning with the need to consider a BUY in such a scenario...
Trading chart:

Task in desired format:
reasoning: You should reflect on the decisions made at each time were right or wrong and give reasoning...
improvement: If there are bad decisions, are you likely to revise them and maximize the return? If so, how would you revise them?...
summary: Provide a summary of the lessons learnt from the success / mistakes that can be adapted to future trading decisions...
query: (tailored to enable focused retrieval of specific types of information) This field will be used to retrieve past reflection of the decisions, so you should...

OUTPUT:
reasoning: The BUY decision made on 2023-06-01 appears to have been correct. This decision aligned well with the market sentiment and the Kline chart analysis...
improvement: However, constant vigilance and responsiveness to both upward and downward market signals are crucial. For future scenario balancing, it could involve dynamic trade adjustments such as setting stop-losses to mitigate risks...
summary: The success of the BUY decision on 2023-06-01 underscores the importance of comprehensive market analysis, incorporating factors such as market sentiment...
query: The price movement analysis and technical indicators provided a compelling case for a bullish trend, thereby justifying the action to buy...

3.7 Voyager: Exploring, Learning, and Adapting Autonomously in Minecraft

Voyager [37] is designed to enable lifelong learning in the video game Minecraft. Minecraft is a 3D open-world sandbox featuring diverse biomes, dynamic day–night cycles, various creatures, and a rich variety of blocks and items that can be gathered, crafted, and used for building or exploration. The Voyager's objective is to explore the game world, acquires diverse skills, and makes novel discoveries without manually designed reward functions or human intervention. It operates by generating its own curriculum of tasks, completing them through code-based actions, and refining its behavior via iterative feedback and self-verification. Through this process, the agent builds a growing library of reusable and compositional skills that support progressively more sophisticated behaviors.

Planning Voyager uses the automatic curriculum module for planning within the game, including automatically setting tasks and decomposing high-level tasks into sub-tasks. The automatic curriculum module autonomously generates new tasks based on its current abilities and the state of the environment. Then, it decomposes high-level tasks into actionable sub-tasks based on LLM's internal knowledge. The planning ability of the automatic curriculum module enables Voyager to gradually progress from completing simple tasks to tackling more difficult ones. In addition, when generating the skill code, Voyager also produces the corresponding plan. We show an example of an automatically generated task below.

Example of an automatically generated task by the automatic curriculum module

INPUT:
Instruction: You are a helpful assistant that tells me the next immediate task to do in Minecraft. My ultimate goal is to discover as many diverse things as possible, accomplish as many diverse tasks as possible and become the best Minecraft player in the world. You should act as a mentor and guide me to the next task based on my current learning progress.
Game State:
Biome: ...
Time: ...
Nearby blocks: ...
Other blocks that are recently seen: ...
Nearby entities (nearest to farthest): ...
Health: ...
Hunger: ...
Position: ...
Equipment: ...
Inventory (xx/36): ...
Completed and Failed Tasks:
Completed tasks so far: ...
Failed tasks that are too hard: ...

OUTPUT:
Reasoning: The inventory is empty now, chop down a tree to get some wood.
Task: Obtain a wood log.

Tool use Voyager controls the game character to complete tasks proposed by the automatic curriculum module by leveraging the LLM generate action code for the character. The action code is composed of Mineflayer[4] API functions, which enable the game character to perform specific actions in Minecraft, such as moving, mining, crafting, and fighting. The use of Mineflayer first allows the LLM to directly access structured game state and high-level action interfaces, eliminating the need to interpret complex visual inputs and frequent low-level decision-making, thereby greatly simplifying environment interaction and improving efficiency. More importantly, each generated Mineflayer function is stored as an independent skill, which can be retrieved for new tasks and combined to form more complex behaviors.

[4] https://mineflayer.com

Example of the generated code for obtaining a wood log

INPUT:
Instruction: You are a helpful assistant that writes Mineflayer javascript code to complete any Minecraft task specified by me. Here are some useful programs written with Mineflayer APIs. You should Write an async function taking the bot as the only argument. Reuse the above useful programs as much as possible. Your function will be reused for building more complex functions. Therefore, you should make it generic and reusable.
Code Example: {programs}
Game State:
Biome: ...
Time: ...
Nearby blocks: ...
Other blocks that are recently seen: ...
Nearby entities (nearest to farthest): ...
Health: ...
Hunger: ...
Position: ...
Equipment: ...
Inventory (xx/36): ...
Task: Obtain a wood log
Context: ...
Critique: ...

OUTPUT:
Explain:
1. First check if a wooden pickaxe is in the inventory.
2. If not, craft one. If the wooden pickaxe is available, equip the wooden pickaxe in the hand.
3. ...
4. ...

Code:

```
async function mineCobblestone(bot) {
  let woodenPickaxe = bot.inventory.findInventoryItem(mcData.itemsByName["
      wooden_pickaxe"].id);
  if (!woodenPickaxe) {
    bot.chat("Crafting a wooden pickaxe.");
    await craftWoodenPickaxe(bot);
    woodenPickaxe = bot.inventory.findInventoryItem(mcData.itemsByName["wooden_pickaxe
        "].id);
  }
  ...
}
```

Memory Voyager exhibits memory through its skill library, which is extensible and stores all verified executable programs. The skill library is implemented based on a vector database, enabling Voyager to retrieve similar skills when facing new tasks and to compose complex skills through the combination of existing skills. In detail, When a new task is proposed by the automatic curriculum, Voyager queries the skill library using the embedding of the task description and environment feedback to retrieve relevant skills. These retrieved skills are provided to an LLM, which can reuse or compose them to generate new executable programs. Verified programs are then added back to the skill library as a new skill, enabling Voyager to continually expand its repertoire of modular and compositional behaviors for solving increasingly complex and novel tasks.

Reflection The iterative prompting mchanism endows Voyager with reflection ablitity. After executing each generated program, Voyager observes environmental feedback, execution errors and self-verification for checking task success. By incorporating signals such as execution errors and success checks, Voyager continuously revises its action codes until the task succeeds. This trial-and-error cycle mirrors reflective reasoning, allowing the agent to learn from its mistakes and autonomously improve its action. Below is an example of Voyager's self-verification.

Example of self-verification for checking task success

INPUT:
Instruction: You are an assistant that assesses my progress of playing Minecraft and provides useful guidance. You are required to evaluate if I have met the task requirements. Exceeding the task requirements is also considered a success while failing to meet them requires you to provide critique to help me improve.
Game State:
Biome: ...
Time: ...
Nearby blocks: ...
Other blocks that are recently seen: ...
Nearby entities (nearest to farthest): ...
Health: ...
Hunger: ...
Position: ...
Equipment: If I have better armor in my inventory, you should ask me to equip it.
Inventory (3/36): {'crafting_table': 1, 'spruce_planks': 6, 'stick': 4}
Task: Craft a wooden pickaxe

OUTPUT:
Reasoning: You have enough materials to craft a wooden pickaxe, but you didn't craft it.
Success: False.
Critique: Craft a wooden pickaxe with a crafting table using 3 spruce planks and 2 sticks.

3.8 Mobile Agent: Operate the mobile phone as smoothly as a human

Mobile agent [32] is designed to assist users by simplifying interactions with mobile devices, making complex operations easier and more intuitive. Instead of manually handling intricate tasks or repeatedly switching between apps, users simply provide high-level instructions in natural language. The agent then interprets these instructions, identifies the necessary steps, and automatically performs the required actions—such as tapping buttons, entering text, or navigating across apps. This approach significantly streamlines task completion, reducing user effort and minimizing errors.

Planning The agent initiates the task by interpreting the user's instruction in the context of the device's current state, typically represented by the latest screenshot. Using this combined understanding, it formulates an operational plan and identifies the optimal next action. This planning process effectively translates the user's high-level objective into precise and actionable steps, ensuring that task execution remains organized and directed toward achieving the intended goal. An illustrative example of this planning step is provided below, guided by the provided prompt, the agent engages in structured reasoning to formulate an action articulated in natural language.

Example of planning next action

INPUT:
Instruction: Set a clock for 7:00 a.m.

Screenshot: 

History: {...}
Task: Think and select the next action. (Swipe, Tap, Type ...)

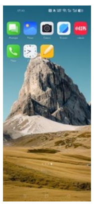
OUTPUT:
Operation Thought: Currently on the home screen we need to first **tap the clock icon** ...
Action: Tap(500,800)

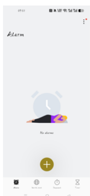
Tool use After generating the next action, the agent directly inputs this action into an action mapping tool, which then converts the natural-language action into executable instructions for the device. An abstract action such as `Tap(500,800)` is transformed by the code translator into low-level commands, for example `adb shell input tap 500 800`. These commands are then executed through the Android Debug Bridge (ADB), which directly interacts with the mobile operating system. In this way, the agent connects high-level planning with concrete device operations. Actions such as launching an application, selecting an interface element, swiping across the screen, or entering text can thus be automatically performed. By bridging abstract decisions and real interactions, the agent progressively fulfills the user's goal without requiring detailed manual intervention.

Reflection After each operation is executed, the agent evaluates its effectiveness by comparing the screenshot before and after the action. If the outcome aligns with the intended goal, the agent returns to the planning phase for the next iteration, continuing this cycle until the entire instruction is successfully completed. If the outcome diverges from the intended goal, such as when an incorrect page is opened or a gesture fails to produce any effect, the agent detects the error and initiates corrective measures. This reflective mechanism prevents errors from propagating, ensuring that the task remains aligned with the original objective and that overall performance remains reliable. The example below illustrates how the agent evaluates the success of an operation by comparing screenshots.

Example of reflection for checking operation success

INPUT:
Last Screenshot: *Current Screenshot:*





Latest Operation: ...tap the clock icon ...
Task: Compare and determine whether the latest operation was successful.

OUTPUT:
By comparing these two images, it is found ... so this operation is **successful**.

Memory Throughout the process, the agent maintains a memory of task-relevant information that may be required in later steps. This includes recording intermediate results and preserving contextual details across applications. By recalling such information when necessary, the agent provides continuity for multi-step and cross-application tasks, thereby avoiding the loss of critical content and ensuring coherent task completion.

3.9 PaLM–SayCan: A Clear, Grounded Agent Pattern

Embodied agents must turn open-ended language into safe, effective behavior amid partial observability, long horizons, hardware limits, and real-time uncertainty. Typical failures include vague plans that ignore geometry, hallucinated capabilities the robot does not have, and unsafe actions a robot cannot execute. PaLM–SayCan addresses these issues by *separating* high-level reasoning from low-level feasibility. At each step it asks: *What is useful to do next?* (**Say**) and *What can the robot actually do now?* (**Can**). Only actions that are both useful and feasible are executed. At decision time, the agent takes the user's instruction, its current state, and a library of named skills. The LLM ranks the skills by how helpful each one would be right now (*Say*). A pre-trained affordance model then estimates how likely each skill is to succeed in the current scene (*Can*). The system combines usefulness and feasibility, executes the top skill, logs the outcome, updates state, and repeats until the goal is reached or no safe, feasible action remains.

Planning The LLM acts as a *semantic planner over a fixed skill set*. It does not invent new primitives; it scores which provided skill is most useful now, optionally revealing short reasoning for interpretability.

Example of planning (Say)

INPUT:
Role: Semantic planner for a mobile manipulator. Select the most useful next skill from a list.

- *Instruction:* e.g., "I spilled my drink."
- *Available Skills:* e.g., `["find a sponge", "pick up sponge", "find an apple"]`
- *Task Progress:* e.g., "No cleanup tools found yet."

Constraints: Score *every* listed skill with a probability (scores sum to 1). Use concise, commonsense reasoning. Do not propose skills outside the list.

OUTPUT:
Utility scores: — find sponge: 0.85; pick up sponge: 0.10; find apple: 0.05.
Rationale: Locate a cleaning tool before manipulation; ignore irrelevant goals.

Tool Use SayCan treats low-level abilities as tools that must be *grounded* in two ways. First, **text-to-skill grounding**: each skill exposes a human-readable name and description (and optional arguments) so the LLM can align language to capabilities. Second, **execution grounding**: each skill has an executor with preconditions and postconditions; execution returns a structured outcome (success/failure and observations) that updates the progress log and the robot's state. Adding capabilities is modular: register a new skill by its name/description, provide (or train) its executor and an affordance estimator, and it immediately becomes available to the planner.

Example of skill schema

Name: `open_refrigerator`
Description: "Open the refrigerator door using the handle."
Preconditions: handle visible and within reach; gripper free
Postconditions: door angle $> 20^\circ$
Executor: grasp $\rightarrow$ pull $\rightarrow$ release
Affordance: $A_{\text{open\_refrigerator}}(x)$ predicts success from detections, depth, reachability.

Reflection Reflection is the feasibility check that keeps the plan honest. For each available skill, a pre-trained *affordance function* (learned from robot experience) estimates the likelihood of success in the current scene. Unlike the "Say" step, the "Can" step is not generated by an LLM. Instead, it is computed by a pre-trained affordance model. In the original SayCan implementation, this model is a language-conditioned value function. This value function is trained on a large dataset of real-world robot rollouts from task execution. Its specific function is to take the robot's current observation state and a natural language description of a skill (e.g., "pick up the apple") as input. It then outputs a scalar score that predicts the expected probability of successfully executing that skill. In the final decision-making phase, the system integrates the outputs from both modules: "Say" (LLM) provides the linguistic usefulness probability for each skill, and "Can" (Affordance Model) provides the physical feasibility probability for each skill. The system multiplies these two scores to obtain a final score for each skill, ultimately selecting the skill with the highest score for execution.

Example of reflection (Can) and decision

INPUT:
Instruction: "I'd like a cold apple."
Scene: The robot is far from a closed refrigerator; the handle is visible but out of reach.
Skills: [1] navigate to refrigerator, [2] open refrigerator, [3] pick up apple.
Planning (Say) utilities: navigate: 0.50; open: 0.40; pick up: 0.10.
Affordance (Can) estimates: navigate: 0.95 (clear path); open: 0.20 (too far); pick up: 0.05 (not visible).

OUTPUT: Combined scores favor *navigate to refrigerator*. After moving, the affordance for *open refrigerator* rises, so the next step becomes opening the door, followed by picking up the apple.

4 Practical Tips: How to build your first LLM agent

In this section, we present tips that we believe to be helpful for general audiences to design agents.

Keep your agent design as simple as possible. First attempts are rarely seamless, so the most important principle is to aim for the simplest viable design and minimal complexity. Building an LLM agent is already much easier than it was just a few years ago—back when the first version of ChatGPT appeared—because today's models are much more capable. Their strengthened reasoning skills mean that tasks once solved only through long, fine-grained prompt chains can now often be handled with a single, concise prompt.

Clearly define the goal or task for your agent. The most important aspect of agent design is to clearly specify what you want your agent to accomplish. Take time to articulate the agent's goal or task, and make sure this is clearly stated in your prompt. For example, you might begin your prompt with: "Your goal is to ..." This directness helps the agent stay focused and increases the likelihood of successful outcomes.

Guide your agent by considering how humans would approach the same task. If simply setting the goal does not lead to desired agent behaviors, a helpful strategy is to craft prompts that outline high-level milestones to guide agent behaviors. Think about how you would brief a person on the same task. For example, if you are building an essay-writing agent, a human researcher would propose an idea, review relevant literature, suggest a solution, and validate that solution. Structure your prompt around these stages, e.g., "First, propose an original idea on [topic]; then scan the literature ...", and let the agent handle the finer details.

Provide illustrative examples. Examples can accelerate understanding for both humans and LLM agents. If you need structured output, show the desired format, e.g., "Produce the result in this JSON schema: ...". When a specific methodology matters (e.g., a clinical guideline for diagnosis and treatment), include step-by-step examples that embody each required step. Well-chosen examples help guide the agent's behavior and reduce ambiguity. Note that this practice is often referred as in-context learning in the literature.

Start with chatting with LLMs directly, and restrict the use of agent framework. There are many agent frameworks, such as LangGraph, Dify, Coze, Bedrock, etc, and some (e.g., Coze) even offer a drag-and-drop graphical user interface. However, they often create extra layers of abstraction that can hide the raw prompts and responses you need to understand and debug, while also tempting you to over-engineer. In our experience, newcomers learn more and iterate faster by chating with the model directly (e.g., via ChatGPT), then calling the LLM API, and only later adopting an agent framework. As our examples show, many agent capabilities can be achieved with just a few well-crafted lines of prompt.

5 Advanced Techniques: What is happening at the frontier

5.1 Injecting Domain Knowledge

Injecting domain knowledge is a critical step to steer a general-purpose LLM into a powerful, expert-level agent. The method chosen for this task depends on the scale of the knowledge, how frequently it changes, and the desired behavior of the agent. There are three primary techniques: *In-Context Learning*, *Retrieval-Augmented Generation (RAG)*, and *Fine-Tuning*.

In-Context Learning The most straightforward way to inject domain knowledge is to place the relevant information directly within the LLM's context window. This method leverages the model's in-context learning ability, allowing it to use the provided text as a direct source of truth for generating a response. As the user can dynamically insert any piece of information on the fly without requiring any additional infrastructure, in-context learning is very agile and is especially effective when the knowledge is limited in scope, such as some trading rules for the stock market or the clinical summary for a patient. Below we provide an example of a medical care scenario.

Example of Injecting Domain Knowledge through In-Context Learning

Task: Triage a patient's chief complaint to recommend an appropriate care setting.
Prompt: You are an experienced triage nurse. Use the following medical knowledge: chest pain with exertional onset, radiating to the jaw/arm, and associated with diaphoresis (sweating) is a classic presentation of *cardiac ischemia* and is an emergency requiring immediate ER evaluation. Non-radiating, pleuritic pain (sharp, worse on inspiration) is more suggestive of musculoskeletal or pulmonary issues like costochondritis or pneumonia, which may be handled by urgent care. Now triage this complaint: "I have a crushing feeling in my chest that started after I mowed the lawn, and I'm sweating but feel cold."
Output: This is a high-risk presentation. The exertional onset (mowing the lawn), crushing quality, and associated diaphoresis (sweating) are hallmark red flags for acute coronary syndrome (e.g., a heart attack). This patient requires immediate emergency room evaluation. Do not delay.

However, this method is not well-suited for extensive knowledge bases. Directly including large volumes of text in every prompt can be expensive (as the API costs are usually calculated on the number of input and output tokens), especially in multi-turn dialogues. Moreover, too long context can "crowd out" the space needed for the user's query, the conversation history, and the model's generated output, potentially leading to truncated or less coherent responses. We will discuss how to handle the long context window in Section 5.3.

Retrieval-Augmented Generation When dealing with a large corpus of documents, RAG [38] offers a more scalable and efficient solution. Instead of attempting to fit all possible information into the context window, RAG operates on a "just-in-time" basis by retrieving only the most relevant snippets of knowledge in response to the user's query. Taking the medical scenario as an example, this allows an agent to pull specific symptom descriptions for cardiac ischemia from a vast medical database to provide precise, expert-level advice. Standard RAG often treats knowledge chunks as independent, parallel pieces of information, but the user can also explore different ways to construct the knowledge base according to their needs. For example, by incorporating a knowledge graph, the system can represent the intricate relationships between different concepts and entities, and thus fetches not just directly relevant text but also interconnected information. Finally, RAG is not limited to a static set of documents. The retrieval mechanism can be extended to pull information from a wide variety of sources, including structured databases (e.g., via text-to-SQL queries) or live web searches, ensuring the agent has access to the most current information available.

Fine-Tuning A third approach is to embed domain knowledge directly into the LLM's parameter weights through fine-tuning. This process involves further training the pre-trained model on a custom dataset specific to the target domain. *Full-Parameter Fine-Tuning* would modify all of the model's weights, which offers the highest potential for performance but is extremely resource-intensive. In contrast, methods as *LoRA* (Low-Rank Adaptation) [39] or *adapters* [40] freeze most of the model's parameters and only train a small set of new ones, known as *Parameter-Efficient Fine-Tuning (PEFT)*. This dramatically reduces computational costs and makes fine-tuning more accessible. Fine-tuning does more than just teach the model facts; it changes its behavior and makes the knowledge an intrinsic part of its reasoning process. Meanwhile, since the knowledge is stored in the model's weights, the user does not need to include large amounts of text in the prompt during inference. This leads to faster response times and lower operational costs, which is especially beneficial for high-volume applications.

5.2 Enhancing Reasoning Capabilities

Reasoning underpins many core agent abilities, including planning, tool use, and cooperation. Recent advances have significantly strengthened the reasoning capabilities of LLMs. For example, OpenAI's o-series and DeepSeek-R1 are LLMs specifically enhanced for reasoning. Compared with standard LLMs (such as the GPT-4 series) from the same providers, these models are explicitly trained to perform multi-step thinking before producing an answer. As a result, they often demonstrate superior performance on complex tasks such as mathematics and programming, though they also tend to produce substantially longer outputs and consume more tokens.

Chain-of-Thought This technique [41] is the most simple technique for enhancing a model's reasoning ability. It encourages models to generate intermediate reasoning steps by adding a simple instruction to the prompt—"Let's think step by step". This prompt guides the model to engage in multi-step reasoning before arriving at its final answer, rather than responding directly. Despite its simplicity, this method yields substantial improvements in reasoning performance. Building on this, *long CoT* [42] extends the depth and complexity of reasoning traces by encouraging models to elaborate over multiple stages, simulate nested sub-decisions, and maintain coherence across extended contexts. *Self-consistency* [43] aggregates reasoning from multiple CoT samples to increase reliability, while *Tree-of-Thoughts* (ToT) [44] generalizes CoT by searching over multiple reasoning paths before settling on a final solution.

Test-Time Scaling *Test Time Scaling (TTS)* [45] is another simple yet powerful strategy for enhancing the reasoning capabilities of LLMs at inference time, without requiring retraining or architectural changes. The core idea is to generate multiple reasoning samples–such as diverse CoT traces–and select the most consistent or confident output. This leverages the model's ability to explore varied reasoning paths and self-correct through redundancy. Two basic and popular TTS methods are *Beam Search* [46] and *Monte Carlo Tree Search (MCTS)* [47, 48, 49]. Beam Search maintains a fixed number of top candidate sequences based on likelihood scores, promoting diversity while preserving high-probability paths. MCTS adaptively explores the reasoning space by simulating reasoning paths and using statistical heuristics to guide search toward promising paths. These techniques allow LLMs to scale their deliberative capacity, improving reliability and reducing hallucinations in tasks that demand multi-step reasoning, such as math word problems, commonsense inference, and strategic planning.

Reinforcement Learning When one wants to enhance the reasoning capability of LLMs at training time, the technique of *Reinforcement learning (RL)* provides an almost standard approach. sRL enables the model to learn from experience by exploring alternative reasoning paths and improving through reward signals. These rewards may be verifiable, when grounded in objective correctness (e.g., comparing against labeled solutions), or they may quantify alignment with human preferences, when derived from human feedback. For example, *REINFORCE* is an early RL method that realized this idea in LLMs. More recently, advanced techniques have built on this foundation to offer greater stability and efficiency. *Proximal Policy Optimization (PPO)* [50] emphasizes training stability, while *Direct Preference Optimization (DPO)* [51] offers a way to directly align with human preference without explicit reward signals. *Group Relative Policy Optimization (GRPO)* [52] enhances reasoning by contrasting the reward-gain across a relatively small group of reasoning paths, offering more computational efficiency.

5.3 Managing Long Context

As LLM-based agents often engage in multi-turn, prolonged, or complex workflows, a key challenge they may encounter is the extremely long context, which significantly raises computational costs, slows down inference [53], and often reduces accuracy due to models' bias towards recent inputs [54, 55]. Therefore, naively using maximal-length contexts remains impractical for real-world agents. To address this, there are three primary techniques: *Context Compression*, *Context Reuse*, and *Hierarchical Context Management*.

Context Compression The first strategy is *context compression*, which reduces the prompt to the minimum necessary set. In *Retrieval-Augmented Generation (RAG)*, a large knowledge source is segmented and indexed so that, at query time, only the top-k relevant chunks are inserted into the prompt instead of the whole corpus [56, 57]. In addition, agents can *summarize or merge prior turns* to keep the running transcript short [31]; and they can offload long-term information to external memory (e.g., databases or vector stores) and re-inject it only on demand, so the working prompt stays small by default [17].

Context Reuse The second strategy is *context reuse*. Intuitively, *prompt caching* lets the system treat an unchanged prefix as "already processed", so it does not need to be recomputed on every turn. Concretely, servers reuse precomputed key–value attention states to skip redundant prefix computation, reducing latency and cost [58, 59]. To make caching effective, maintain a stable prefix—system role, shared tools, policies—so cache hits remain consistent across multi-turn conversations. When tool availability must change, avoid editing the prompt; keep all tool definitions in the prefix and switch tools on or off during decoding (*mask, don't remove*) to preserve a byte-identical, cache-friendly prefix while constraining actions [60].

Hierarchical Context Management The third strategy is *hierarchical context management*: rather than continually expanding a single transcript, we structure contexts according to role and scope. An orchestrator manages global goals and constraints, delegating detailed execution to specialized sub-agents [61], each operating with concise, role-specific prompts. Between these agents, structured summaries, such as key inputs, assumptions, and outputs, are passed instead of lengthy transcripts, ensuring each stage uses a minimal, cache-friendly context. Additionally, capabilities are organized into modular and reusable skill bundles [62], loaded into context only as needed, further limiting prompt size and complexity. Such hierarchical designs are increasingly common in frameworks and community practices focused on sub-agent orchestration and modular skill management [63, 64, 65, 66].

5.4 Internalizing Agent Abilities

As AI agents continue to advance, a recent trend is to internalize agent abilities directly within the model itself. This allows the model to perform agent-like behaviors (such as planning, tool use, and multi-step task execution) or integrate more seamlessly with agent frameworks that extend these abilities. For example, Claude Code [6] incorporates structured reasoning and tool-usage capabilities optimized for autonomous coding workflows. It can generate code, run tasks, analyze results, and iterate based on feedback. Similarly, OpenAI's Operator [67], which combines the multimodal perception of GPT-4o with advanced reasoning through reinforcement learning, includes a built-in browser that the model can observe and control. It can inspect webpages, type, click, and scroll—interacting with the environment through standard mouse and keyboard actions. While internalizing agent abilities is fundamentally training the model, many techniques used for training LLMs can be applied. The key distinction, however, lies in the data: agent-centric training requires constructing datasets that capture agent behaviors—demonstrations of actions, sequences, and decision-making in interactive environments [68, 69, 70].

5.5 Agent Safety

Agent safety is a critical concern for deploying LLM-based agents in the real world. This requires them to remain aligned, controllable, and recoverable even under uncertainty. While this direction is relatively underexplored, four key strategies emerge as promising for addressing the safety concerns. First, establish rigorous permission management and controlled environmental affordances [71] to ensure safe operations, when agents invoke tools, execute physical operations, access critical business assets, or interact with sensitive data. Second, architect agents from security-hardened components, e.g. scaffolds fortified against hijacking and code injection, memory modules that enforced confidentiality [72], and trustworthy MCP (Model Context Protocol) channels secured through cryptographic verification. Third, build system resilience by serializing security states into snapshots, enabling restoration and rollback to the last verified configuration, or fallback to hardened policies upon detecting anomalies, goal drift, or unauthorized actions. Fourth, introduce supervisory agents when building multi-agent systems to prevent collusion or rouge behaviors arising from information asymmetry and cascade amplification [73].

6 Conclusions

This tutorial provides a gentle yet comprehensive introduction of LLM-based agents for general audiences. We outline the foundational concepts of LLMs and LLM agents, and illustrate 6 key capabilities commonly recognized as characteristic of LLM agents, including memory, role-play, planning, reflection, tool use, and cooperation. We further present 9 carefully selected, self-contained examples that demonstrate how these capabilities are designed and operate across diverse applications, including biology, soft engineering, finance, embodied AI, etc. To help newcomers design their own LLM agents, we also provide practical tips, such as: keeping the agent design as simple as possible, clearly defining the task or goal, guiding the agent by considering how humans would approach the same task, supplying illustrative examples, and starting with direct use of LLM APIs. Finally, for readers interested in exploring more advanced techniques, we offer a brief roadmap of current frontiers, including methods for injecting domain knowledge into agents, enhancing reasoning abilities, handling long contexts, internalizing agent capabilities, and addressing safety concerns. We hope this tutorial helps lower the entry barrier to understanding and developing LLM agents, empowering a broader audience to build their own agents effectively.

References

[1] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*. International Joint Conferences on Artificial Intelligence Organization, 2024, pp. 8048–8057.

[2] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.

[3] S. Ren, Z. Gan, Z. Yin, J. Shao, and S. Hu, "An ai researchers' perspective: At the crossroad of llms, agent-based modeling, and complex systems: Comment on" llms and generative agent-based models for complex systems research" by y. lu et al," *Physics of life reviews*, vol. 53, pp. 215–217, 2025.

[4] Y. JIANG, S. YANG, S. TANG, S. ZHENG, and J. CAO, "A comprehensive survey of llm-driven collective intelligence: Past, present, and future," 2025.

[5] OpenAI, https://openai.com/index/introducing-deep-research/.

[6] Anthropic, https://www.anthropic.com/claude-code.

[7] Pactum, https://pactum.com/.

[8] J. Qiu, K. Lam, G. Li, A. Acharya, T. Y. Wong, A. Darzi, W. Yuan, and E. J. Topol, "Llm-based agentic systems in medicine and healthcare," *Nature Machine Intelligence*, vol. 6, no. 12, pp. 1418–1420, 2024.

[9] S. Xu, X. Zhang, and L. Qin, "Eduagent: Generative student agents in learning," *arXiv preprint arXiv:2404.07963*, 2024.

[10] H. Su, W. Luo, Y. Mehdad, W. Han, E. Liu, W. Zhang, M. Zhao, and J. Zhang, "LLM-friendly knowledge representation for customer support," in *Proceedings of the 31st International Conference on Computational Linguistics: Industry Track*. Abu Dhabi, UAE: Association for Computational Linguistics, Jan. 2025, pp. 496–504.

[11] C. Lu, C. Lu, R. T. Lange, J. Foerster, J. Clune, and D. Ha, "The ai scientist: Towards fully automated open-ended scientific discovery," *arXiv preprint arXiv:2408.06292*, 2024.

[12] W. Zhang, L. Zhao, H. Xia, S. Sun, J. Sun, M. Qin, X. Li, Y. Zhao, Y. Zhao, X. Cai *et al.*, "A multimodal foundation agent for financial trading: Tool-augmented, diversified, and generalist," in *Proceedings of the 30th acm sigkdd conference on knowledge discovery and data mining*, 2024, pp. 4314–4325.

[13] Y. Shoham, "Agent-oriented programming," *Artificial intelligence*, vol. 60, no. 1, pp. 51–92, 1993.

[14] D. C. Smith, A. Cypher, and J. Spohrer, "Kidsim: Programming agents without a programming language," *Communications of the ACM*, vol. 37, no. 7, pp. 54–67, 1994.

[15] S. Franklin and A. Graesser, "Is it an agent, or just a program?: A taxonomy for autonomous agents," in *International workshop on agent theories, architectures, and languages*. Springer, 1996, pp. 21–35.

[16] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *The knowledge engineering review*, vol. 10, no. 2, pp. 115–152, 1995.

[17] W. Zhong, L. Guo, Q. Gao, H. Ye, and Y. Wang, "Memorybank: Enhancing large language models with long-term memory," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 17, 2024, pp. 19 724–19 731.

[18] S. Ren, Z. Cui, R. Song, Z. Wang, and S. Hu, "Emergence of social norms in generative agent societies: principles and architecture," *arXiv preprint arXiv:2403.08251*, 2024.

[19] M. Shanahan, K. McDonell, and L. Reynolds, "Role play with large language models," *Nature*, vol. 623, no. 7987, pp. 493–498, 2023.

[20] L. P. Argyle, E. C. Busby, N. Fulda, J. R. Gubler, C. Rytting, and D. Wingate, "Out of one, many: Using language models to simulate human samples," *Political Analysis*, vol. 31, no. 3, pp. 337–351, 2023.

[21] Z. Wang, R. Song, C. Shen, S. Yin, Z. Song, B. Battu, L. Shi, D. Jia, T. Rahwan, and S. Hu, "Overcoming the machine penalty with imperfectly fair ai agents," *arXiv preprint arXiv:2410.03724*, 2024.

[22] X. Huang, W. Liu, X. Chen, X. Wang, H. Wang, D. Lian, Y. Wang, R. Tang, and E. Chen, "Understanding the planning of llm agents: A survey," *arXiv preprint arXiv:2402.02716*, 2024.

[23] L. E. Erdogan, N. Lee, S. Kim, S. Moon, H. Furuta, G. Anumanchipalli, K. Keutzer, and A. Gholami, "Plan-and-act: Improving planning of agents for long-horizon tasks," *arXiv preprint arXiv:2503.09572*, 2025.

[24] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," *Advances in Neural Information Processing Systems*, vol. 36, pp. 68 539–68 551, 2023.

[25] M. A. Islam, M. E. Ali, and M. R. Parvez, "Mapcoder: Multi-agent code generation for competitive problem solving," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 4912–4944.

[26] R. Nakano, J. Hilton, S. Balaji, J. Wu, L. Ouyang, C. Kim, C. Hesse, S. Jain, V. Kosaraju, W. Saunders *et al.*, "Webgpt: Browser-assisted question-answering with human feedback," *arXiv preprint arXiv:2112.09332*, 2021.

[27] G. Li, H. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "Camel: Communicative agents for" mind" exploration of large language model society," *Advances in Neural Information Processing Systems*, vol. 36, pp. 51 991–52 008, 2023.

[28] W. Chen, Y. Su, J. Zuo, C. Yang, C. Yuan, C.-M. Chan, H. Yu, Y. Lu, Y.-H. Hung, C. Qian *et al.*, "Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors," in *ICLR*, 2024.

[29] S. Ren, W. Fu, X. Zou, C. Shen, Y. Cai, C. Chu, Z. Wang, and S. Hu, "Beyond the tragedy of the commons: Building a reputation system for generative multi-agent systems," *arXiv preprint arXiv:2505.05029*, 2025.

[30] Y. Lan, Z. Hu, L. Wang, Y. Wang, D. Ye, P. Zhao, E.-P. Lim, H. Xiong, and H. Wang, "Llm-based agent society investigation: Collaboration and confrontation in avalon gameplay," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 128–145.

[31] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, "Generative agents: Interactive simulacra of human behavior," in *Proceedings of the 36th annual acm symposium on user interface software and technology*, 2023, pp. 1–22.

[32] J. Wang, H. Xu, H. Jia, X. Zhang, M. Yan, W. Shen, J. Zhang, F. Huang, and J. Sang, "Mobile-agent-v2: mobile device operation assistant with effective navigation via multi-agent collaboration," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, 2024, pp. 2686–2710.

[33] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," *arXiv preprint arXiv:2204.01691*, 2022.

[34] Y. Xiao, J. Liu, Y. Zheng, X. Xie, J. Hao, M. Li, R. Wang, F. Ni, Y. Li, J. Luo *et al.*, "Cellagent: An llm-driven multi-agent framework for automated single-cell data analysis," *arXiv preprint arXiv:2407.09811*, 2024.

[35] K. Swanson, W. Wu, N. L. Bulaong, J. E. Pak, and J. Zou, "The virtual lab of ai agents designs new sars-cov-2 nanobodies," *Nature*, pp. 1–3, 2025.

[36] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin *et al.*, "Metagpt: Meta programming for a multi-agent collaborative framework." International Conference on Learning Representations, ICLR, 2024.

[37] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *arXiv preprint arXiv:2305.16291*, 2023.

[38] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, "A survey on rag meeting llms: Towards retrieval-augmented large language models," in *Proceedings of the 30th ACM SIGKDD conference on knowledge discovery and data mining*, 2024, pp. 6491–6501.

[39] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, "Lora: Low-rank adaptation of large language models." *ICLR*, vol. 1, no. 2, p. 3, 2022.

[40] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. De Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-efficient transfer learning for nlp," in *International conference on machine learning*. PMLR, 2019, pp. 2790–2799.

[41] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[42] Q. Chen, L. Qin, J. Liu, D. Peng, J. Guan, P. Wang, M. Hu, Y. Zhou, T. Gao, and W. Che, "Towards reasoning era: A survey of long chain-of-thought for reasoning large language models," *arXiv preprint arXiv:2503.09567*, 2025.

[43] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *The Eleventh International Conference on Learning Representations*, 2023.

[44] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *Advances in neural information processing systems*, vol. 36, pp. 11 809–11 822, 2023.

[45] Q. Zhang, F. Lyu, Z. Sun, L. Wang, W. Zhang, W. Hua, H. Wu, Z. Guo, Y. Wang, N. Muennighoff *et al.*, "A survey on test-time scaling in large language models: What, how, where, and how well?" *arXiv preprint arXiv:2503.24235*, 2025.

[46] M. Ott, M. Auli, D. Grangier, and M. Ranzato, "Analyzing uncertainty in neural machine translation," in *International Conference on Machine Learning*. PMLR, 2018, pp. 3956–3965.

[47] D. Zhang, S. Zhoubian, Z. Hu, Y. Yue, Y. Dong, and J. Tang, "Rest-mcts*: Llm self-training via process reward guided tree search," *Advances in Neural Information Processing Systems*, vol. 37, pp. 64 735–64 772, 2024.

[48] D. Zhang, X. Huang, D. Zhou, Y. Li, and W. Ouyang, "Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b," *arXiv preprint arXiv:2406.07394*, 2024.

[49] Y. Xie, A. Goyal, W. Zheng, M.-Y. Kan, T. P. Lillicrap, K. Kawaguchi, and M. Shieh, "Monte carlo tree search boosts reasoning via iterative preference learning," *arXiv preprint arXiv:2405.00451*, 2024.

[50] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[51] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in neural information processing systems*, vol. 36, pp. 53 728–53 741, 2023.

[52] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[53] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 16 344–16 359.

[54] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," 2023. [Online]. Available: https://arxiv.org/abs/2307.03172

[55] X. Zhang, Y. Chen, S. Hu, Z. Xu, J. Chen, M. Hao, X. Han, Z. Thai, S. Wang, Z. Liu, and M. Sun, "∞Bench: Extending long context evaluation beyond 100K tokens," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 15 262–15 277. [Online]. Available: https://aclanthology.org/2024.acl-long.814/

[56] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, D. Metropolitansky, R. O. Ness, and J. Larson, "From local to global: A graph rag approach to query-focused summarization," 2025. [Online]. Available: https://arxiv.org/abs/2404.16130

[57] C.-Y. Chang, Z. Jiang, V. Rakesh, M. Pan, C.-C. M. Yeh, G. Wang, M. Hu, Z. Xu, Y. Zheng, M. Das, and N. Zou, "MAIN-RAG: Multi-agent filtering retrieval-augmented generation," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 2607–2622. [Online]. Available: https://aclanthology.org/2025.acl-long.131/

[58] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 611–626. [Online]. Available: https://doi.org/10.1145/3600006.3613165

[59] Y. Liu, H. Li, Y. Cheng, S. Ray, Y. Huang, Q. Zhang, K. Du, J. Yao, S. Lu, G. Ananthanarayanan, M. Maire, H. Hoffmann, A. Holtzman, and J. Jiang, "Cachegen: Kv cache compression and streaming for fast large language model serving," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 38–56. [Online]. Available: https://doi.org/10.1145/3651890.3672274

[60] Y. Ji, "Context engineering for ai agents: Lessons from building manus," https://manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus, 2025, technical report.

[61] Anthropic, "Claude code: Subagents," https://docs.anthropic.com/en/docs/claude-code/sub-agents, 2025, product documentation.

[62] ——, “Agent skills: Overview,” https://docs.anthropic.com/en/docs/agents-and-tools/agent-skills/overview, 2025, product documentation.

[63] T. Wu, Y. Li, Y. Song *et al.*, “Autogen: Enabling next-gen llm applications via multi-agent collaboration,” arXiv preprint arXiv:2308.08155, 2023. [Online]. Available: https://arxiv.org/abs/2308.08155

[64] Microsoft, “Semantic kernel documentation: Skills and plugins,” https://learn.microsoft.com/en-us/semantic-kernel/, 2024, community framework docs on packaging skills/plugins.

[65] CrewAI, “Crewai documentation,” https://docs.crewai.com/, 2024, community framework docs on role- and tool-based crews.

[66] LangChain, “Langgraph documentation,” https://python.langchain.com/docs/langgraph/, 2024, community docs on graph-structured, multi-actor workflows.

[67] OpenAI, “Introducing operator,” https://openai.com/index/introducing-operator/, 2025.

[68] K. Team, Y. Bai, Y. Bao, G. Chen, J. Chen, N. Chen, R. Chen, Y. Chen, Y. Chen, Y. Chen *et al.*, “Kimi k2: Open agentic intelligence,” *arXiv preprint arXiv:2507.20534*, 2025.

[69] J. Qiu, X. Qi, T. Zhang, X. Juan, J. Guo, Y. Lu, Y. Wang, Z. Yao, Q. Ren, X. Jiang *et al.*, “Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution,” *arXiv preprint arXiv:2505.20286*, 2025.

[70] Y. Qin, Y. Ye, J. Fang, H. Wang, S. Liang, S. Tian, J. Zhang, J. Li, Y. Li, S. Huang *et al.*, “Ui-tars: Pioneering automated gui interaction with native agents,” *arXiv preprint arXiv:2501.12326*, 2025.

[71] Y. Son, M. Kim, S. Kim, S. Han, J. Kim, D. Jang, Y. Yu, and C. Y. Park, “Subtle risks, critical failures: A framework for diagnosing physical safety of llms for embodied decision making,” in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025, pp. 25 703–25 744.

[72] X. Liu, M. Chen, A. Patel, and K. Wang, “Structured memory and modular adaptation in large language agents,” in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

[73] Cooperative AI Foundation, “Multi-agent risks from advanced ai,” Cooperative AI Foundation, Tech. Rep., 2025, technical Report.