ARL-TR-10172 ● SEP 2025

DEVCOM ARMY RESEARCH LABORATORY

Language Model Agents: Current and Future Trends

by Michael S. Lee

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-10172 ● SEP 2025

Language Model Agents: Current and Future Trends

Michael S. Lee
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
September 2025	Technical Report	11/01/2024	8/01/2025

4. TITLE AND SUBTITLE
Language Model Agents: Current and Future Trends

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Michael S. Lee

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-NA Aberdeen Proving Ground, MD 21005	ARL-TR-10172

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCID: Michael Lee, 0000-0002-0419-6069

14. ABSTRACT
The field of language model (LM)-powered agents has grown exponentially in the last few years thanks to the rapid development of LMs that have both a vast amount of world knowledge and are fine-tuned to request tools, process external data, and fluently write short scripts in several programming languages (e.g., Python). One major goal is for agents to run with little or no supervision for hours to days at a time without human intervention to accomplish human-specified project requirements. We look at where the technology is today and where it needs to go to achieve this goal.

15. SUBJECT TERMS
artificial intelligence; planning; tool calling; agentic behaviors; orchestration layer; language models; Network, Cyber, and Computational Sciences

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	32

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Michael S. Lee	(520) 691-5594

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

Acknowledgments

I wish to thank Ms. Gail Vaucher, Dr. Brent Kraczek, Mr. Matthew Ziemann, Dr. Manuel Vindiola, and Mr. Michael Frauenhoffer (ARL) for helpful conversations.

1. Introduction

The concept of an agent, that is, a computer program that can act independently of human input and perform certain functions with a reasoning framework, has been studied for decades (Wooldridge and Jennings 1995). The biggest challenge, however, has been encoding the logic and reasoning of the agent such that it can manage an exponential number of scenarios or edge cases that could be encountered during its mission.

A few years ago, large language models (LLMs), originally intended to simply converse with humans (Floridi and Chiriatti 2020), were quickly realized as a potential new "brain" for an agent, replacing an otherwise brittle and tediously hand-coded workflow. LLM-powered agents can request and process external data and fluently write short scripts in several programming languages (e.g., Python; Van Rossum 2007). LLMs have ushered in the age of the modern AI agent with an ever-increasing amount of built-in reasoning and world knowledge with enhanced flexibility to manage a wide range of eventualities. From a business use perspective, agents have already been operationally deployed in chat support, business automation, and standard LLM chat interfaces. Chat support can entail the text-based chat bots often found at the bottom of web sites or phone-based agents that replace the cumbersome phone tree (i.e., automated menu) with a vocal prompt like "What can I do for you, today?" Alternatively, functions such as "deep research" developed by Google, OpenAI, and xAI can unearth multiple papers relevant to a single human prompt, analyze the retrieved materials, and then synthesize a written report regarding the user's request. Recently, notions of the automated scientific researcher have been put forth (Yamada et al. 2025) with some success in autonomously performing and publishing research in established scientific forums. Besides acting as the central processor for an agent, language models can be infused with multimodal understanding that can provide sensing and actuating capabilities such as image and video understanding, speech-to-text, and text-to-speech (Défossez et al. 2024). The latter two functions can facilitate human–agent interactions.

In this work, we focus on the trends that could lead to autonomous computer-use language model (LM)-powered agents that have long-term memory, can read and write files, and execute scripts and software.

2. Language Models

LMs are evolving at a rapid pace thanks to massive investments in research and development by many companies worldwide. Therefore, current observations of LMs are subject to change over the next few years as new model architectures, new fine-tuning strategies, and increased training data are employed.

2.1 Personality Traits

Since LMs form the brain of an agent structure, it is important to understand their characteristics, strengths, and weaknesses. LMs have personality traits that reflect both their fine-tuning and intrinsic architecture. For example, they are often fine-tuned to *eagerly* pursue the user's goals, responding urgently and directly to the latest user prompt. Perhaps by design, they are often *self-assured*—not always letting the user know that they might be uncertain of their answer, related to the common phenomenon of *hallucinations*, whereby they confidently state falsehoods. One of the most easily discerned hallucinations is citations of literature references that do not exist.

Given their foundational model origins, trained on trillions of tokens, they are *encyclopedic* with a broad awareness of a large percentage of documented human endeavors. In contrast, they are also frequently limited to shallow understanding that a human expert in a particular area can critique. Fine-tuning and specialized datasets can be used to enrich the LM in specific areas like coding, math, and reasoning.

Intrinsic to their architecture, LMs can be *forgetful*. As their context grows, there is no guarantee of keeping on task or using every detail contained within the full context (i.e., current session). Current raw LMs are *stateless*, with no context as to what happened in the past. LMs live in the present moment (i.e., whatever the current context says). In medical terms, they might be considered to have *anterograde amnesia*. They require specially formulated context and/or external memory retrieval to overcome this issue. Pure language models also suffer from *unimodality*—they are "blind" and "deaf" as they can only perceive text. Finally, LMs have *synthetic emotions* in that they can mimic the tone of different texts. They often *mirror* the mood and bias of the user's messages.

2.2 Small vs. Large Language Models

LLMs are typically proprietary, with closed-source (e.g., hidden model architectures) and hidden weights, and run on restricted resources (for example, Google commercial LLMs [e.g., Gemini series] might run solely on Google's

computing resources). Small language models (SLMs) are often partially open source (e.g., revealed model architecture but hidden training sets) and open weights and can be hosted locally on a computer with one or more GPUs containing sufficient video random access memory (VRAM).

Top-rated commercial LLMs are fairly good at coding small projects ("greenfield") in one shot. Both LLMs and locally hosted SLMs are also being used with coding orchestration layers (OLs) such as Windsurf (https://windsurf.com/editor) to yield production-capable interactive debugging environments (IDEs). SLMs can write smaller code blocks or scripts and still have an impressively broad knowledge base but need a fair bit of scaffolding (i.e., context preparation) to be used to generate coherent code within a larger software product. SLMs are sometimes preferred for their lower energy consumption and resource requirements. Also, for secure computing applications, SLMs can be hosted locally with only one or a few GPUs (Belcak et al. 2025). One concern to consider is the safety of an arbitrary SLM. An LM could potentially generate scripts for which the OL could run, leading to malicious activity on the host computer.

Many top LLMs are proficient at tool calling; however, fine-tuned SLMs can also be tuned to manage somewhat robust multi-turn calling (Galileo 2025; Patil et al. 2025). In this work, we used Mistral-3.2-24B-2506 (https://huggingface.co/mistralai/Mistral-Small-3.2-24B-Instruct-2506), which is one of the more capable SLMs for one-at-a-time function/tool calling, to date. This model also is a visual LM (VLM) with the ability to ingest and interpret images. Salesforce has developed a new class of language models, such as xLAM-2-32B (Zhang et al. 2024), fine-tuned for tool-calling only, which they call language action models (LAMs). Within their weight-class, i.e., model size, they perform very well on tool-calling benchmarks like Berkely Function Calling Leaderboard (Patel et al. 2005). We expect many more proficient tool-calling SLMs to be developed in the future.

Looking beyond language-only tasks such as computer vision, our observation is that many small VLMs (e.g., Gemma-3 and Mistral-24B) are adequate for image captioning but limited in their ability to discern details. However, proprietary VLMs such as Gemini 2.5 Pro (Comanici et al. 2025) can parse complex technical figures fairly well.

Current state-of-the-art SLMs consist of a stack of about 40 transformer layers and use several advances like sliding window attention to achieve improved context size while reducing parameter count and execution cost. Top commercial LLMs (as of 2025) have about 10× the runtime cost, which could correspond to either increased number of transformer layers or wider dense kernels in each layer.

Mixture-of-Experts (MoE) models are an emerging trend from DeepSeek and Meta and originally, Mistral. We suspect that more complex model architectures will be formulated in the future that combine the knowledge compression and retention capacity of transformer stacks with the flexibility to accomplish a wide range of cognitive processes (e.g., coding, reasoning, tool-calling, planning, and so on).

3. Agent Framework

Raw LMs simply ingest input tokens and generate output tokens. They require an apparatus such as an OL to gain voluntary sensing and effecting functions with the external world. In a loose analogy between agents and humans, the LM would be the cortex of a brain, while an OL covers just about everything else (i.e., the rest of the brain and the entire body). External components such as code compilers/interpreters, databases, an operating system (OS), and software are analogous to objects in the environment of the agent.

Figure 1 details an example of an agentic system that could be used to run computational simulations. In this particular hypothetical setup, there are two LMs governing different aspects: a manager and a worker. The two LMs can practically be served by single LMs with different system prompts and curated contexts.

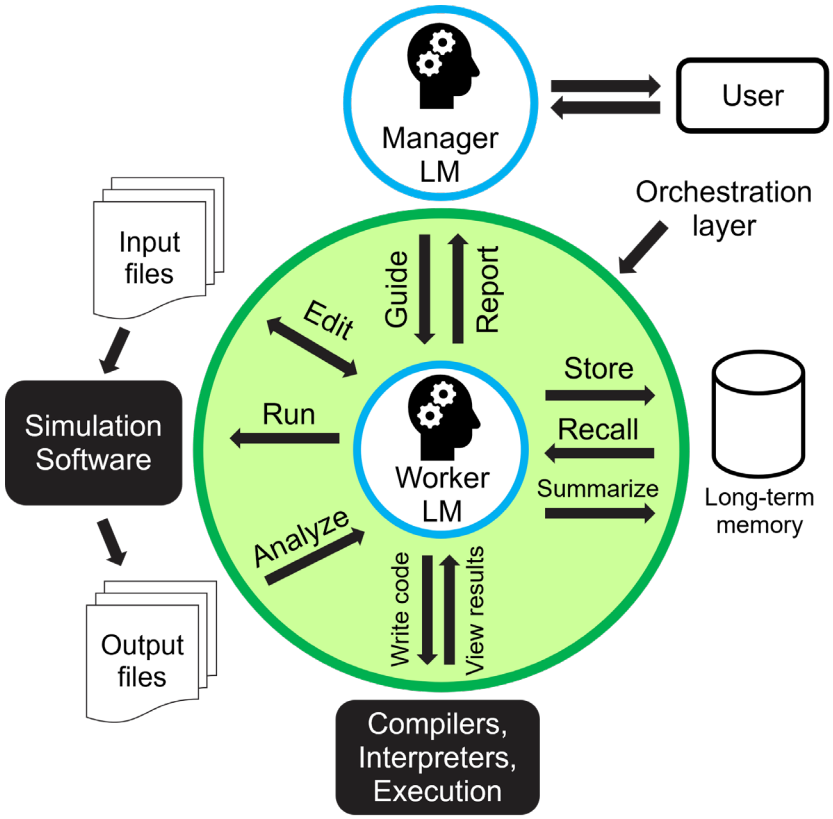

Figure 1. Schematic of an agentic system that can perform computational research.

3.1 LMs Evolving Toward Agents

LMs were first trained to perform word completion tasks. With fine-tuning, they became conversational bots suitable for interaction with humans. People quickly realized they could also be given agentic capabilities—tool calling to carry out functions such as calling local or remote APIs. Given the field's limited understanding of the pros and cons of current LMs for agentic use, the OL, besides providing the appendages of the agent, must also "fill in the intelligence gaps" to construct a more robust agentic system.

3.2 Orchestration Layer (OL)

The OL serves two primary purposes. First, it gives the LM access to the external world (i.e., environment and OS) through tool use (e.g., executing a Python script and sharing the output and/or error logs with the LM). Second, it provides the scaffolding for an intelligent workflow (e.g., control loops, context management, long-term memory saving and retrieval, and multitask planning and execution).

3.3 Agentic Tools and Functions

One of the hallmarks of an AI agent is the ability to interact with the local (e.g., OS) or external environment (e.g., cloud servers) by calling tools or functions. For simplicity, we will use the term "tool" interchangeably to denote both tools and functions; however, there are some distinctions between these two terms as described elsewhere (Gulli et al. 2025). Tools can either be hosted and run locally or externally in the cloud.

3.3.1 Tool Calling Syntax

At present, there are a few competing standards for tool calling by a LM (Table 1). The simplest approach is beginning a new line of text with three backslashes followed by a command as in ```` ```python ```` and then ending the block with a new line of ```` ``` ````. Similarly, the header could begin with ```` ```tool_code <name of tool> ````. Another approach is to use block delimiters such as `<python>` and `</python>` to begin and end the block, respectively. The final standard that is the most detailed uses the JSON format. This often requires tool specification in the system prompt with JSON blocks. In the test implementation shown in this work, we use the first format, for simplicity, but encourage future readers of this report to follow the tool-calling standards appropriate to the LMs and OLs being used.

Table 1. Tools implemented in this work.

Tool call	Description
`write <filename> <text>`	Write the contents of `<text>` to the specified file, `filename`.
`read <filename>`	Output the text of the specified file. If the file is a PDF, first convert it to Markdown.
`python <filename> <code>`	Write a Python script and run it. If the `<code>` section is empty, run the file if already present.
`look <image_filename>`	Attach the image in `image_filename` to the current context. (For VLMs only)
`memory retrieve <query> <num_records>`	Retrieve the top `num_records` memories matching the `query`.
`run_tasks <filename>`	Iteratively run the tasks from a specified task file.
`Search <query> <num_results>`	Search the Internet for links + web page summaries.
`spawn <request>`	Start a new agent that will attempt to satisfy the specified `request` and then exit.

3.3.2 Tool Processing

As seen in Figure 2, when a tool is requested by the LM, the OL tries to process the request by sanitizing the request, running the internal or external tool, and then returning the console output and error messages of the tool back to the LM to analyze. The LM can either accept this output and continue to process it, or it can make a follow-on tool request to rectify the errors encountered with the previous tool request. This loop can be repeated until the LM is satisfied with the tool result.

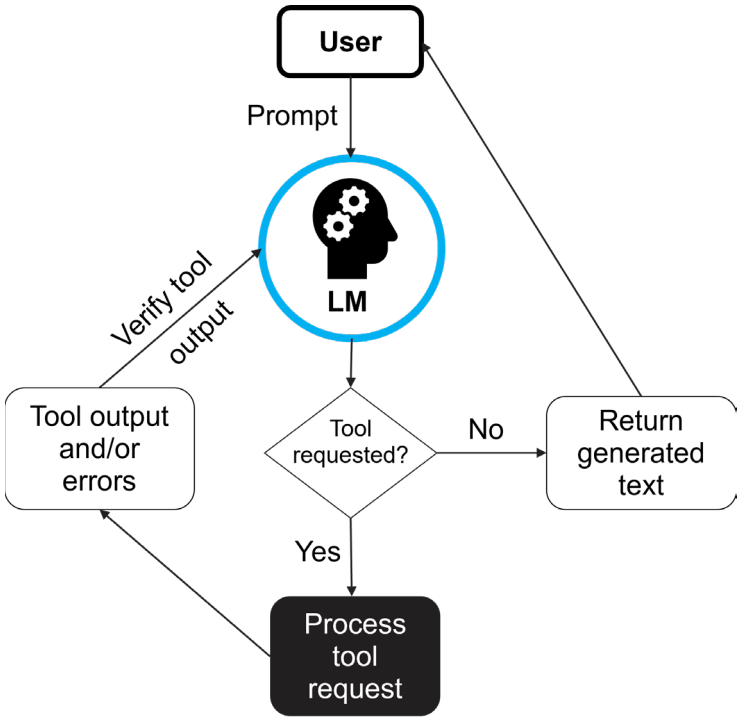

Figure 2. Control loop for processing tool requests by the LM.

3.3.3 Code and Script Execution

Most LMs (both large and small) can generate small snippets of code in a variety of text-based programming languages (e.g., Python, JavaScript, and C++). To give the LM agentic capabilities, the OL extracts the highlighted code from the generated output, inspects it for syntax errors (e.g., AST), checks for security rule violations (e.g., limiting Python imports), and then executes the code, often in a constrained environment or sandbox. Finally, the OL shares the program's output and any runtime, compilation, or operational errors encountered as a user prompt for the LM to analyze and then decide how to proceed. If the LM is satisfied with the output, it can simply not provide any new tool requests, which returns the prompt back to the user or OL.

Python script execution can theoretically be used to perform many other tool functions including interfacing with the OS and shell scripting. It can also be used to call arbitrary web application programming interfaces (APIs). Thus, a code writing/execution agent can be deceptively powerful. For example, in several of our tests, the agent tried to install Python libraries when it saw an error message saying that the script was not working due to missing libraries.

When an agent is given the ability to write and execute programs, sandboxing is a wise choice. For example, the OL should be run inside of a container or virtual machine to avoid causing security issues with the general OS. Another security measure is for the OL to whitelist libraries that can be imported and code that can be included in source files.

3.3.4 Data Analysis

With variable competency and context limits, LMs can analyze ingested language and text-formatted files and output answers to questions about the inputted data. LMs can also invoke third-party tools like pandas within Python, to generate text and graphic analysis (PandasAI 2025).

3.3.5 API Calls

LMs can be provided with tool calling abilities to make queries to APIs, thereby connecting the agentic system with external services and platforms. Among API requests include database operations, web search, web page downloading, email and messaging operations, and calendar/scheduling functions. In November 2024, Anthropic released the model context protocol (MCP) (Anthropic PBC 2024), which provides a standard interface for agents to access external resources such as databases. The number of MCP-compliant resources on the Internet has grown significantly since then. On the positive side, standards such as MCP facilitate the

development of third-party AI-friendly servers and indeed, startups have materialized recently simply providing MCP-compliant services. On the negative side, any given MCP resource is not guaranteed to be permanently available, in analogy to hyperlinks cited in a published document.

3.3.6 Internet Use: Web Search and Web Browser Use

Agents can also obtain information outside of their knowledge base by searching the Internet via a tool request like `search` as seen in Table 1 (Lewis et al. 2020). While a simple search will yield a hyperlink, it sometimes may also contain a summary of the retrieved web page. What will often be missing though is the full contents of the web page/site that is being referenced. To process a full page or site will often require a browser use tool that may need some interactivity (Browser-use 2025). If possible, an API-compliant server is a better long-term option as API interfaces and "machine accessibility" can reduce processing errors.

Internet use by agents can lead to novel security challenges. For example, let us say an agent is given a user's credentials to access the user's private or privileged accounts. With access, the agent can potentially enact incorrect financial transactions leading to unexpected monetary losses. Another problem that could occur is that an agent with access to a user's directories or databases could accidentally delete files or entries. With access at a system level, the agent could erroneously delete entire directories and databases (Forlini 2025). Finally, agents with user-privileged access could harm the reputation of the user if the agent conducts illegal or immoral actions on the user's behalf, even if done inadvertently and without the user's consent.

3.3.7 Media Generation

LMs can natively generate text-based content (prose, poetry, and software code) upon request. They can also transform an input text file to an output text file, converting between different file formats. LMs can also direct OLs to create non-text forms of media via third-party machine learning models such as images, speech, music, and video via diffusion models.

3.4 Error Detection and Mitigation

Some tool calling errors are obvious because they are flagged as an error in the returned output from the OL. Other errors are not explicitly flagged (e.g., unexpected or incomplete outputs of a Python script). All these errors need to be identified and categorized by the LM. Error mitigation can be on a spectrum of complexity from easy to difficult as follows:

- Easy – Example: a syntax error in the generated code that can be quickly resolved.
- Medium – Example: a generated error due to a mismatch of what LM believes to be correct syntax and what a code interpreter expects. These can be resolved by writing alternative code or the LM ingesting some of the compiler's or interpreter's documentation.
- Difficult – Example: a non-flagged error with an unclear solution. These errors may require human or (higher LLM) intervention.

3.5 Agent Development Frameworks and Endpoint Solutions

The field of AI agents is moving rapidly as of the publication of this document. There are two categories to consider: agent development frameworks (ADFs) and agent endpoint solutions. In the first category, one of the earliest ADFs and still a popular platform is LangChain (Topsakal and Akinci 2023). Other choices include Google Agent Development Toolkit (https://google.github.io/adk-docs/), SmolAgents (Roucher et al. 2025) and n8n (https://n8n.io/). In the second category of endpoint solutions are coding agents such as Claude Code (https://www.anthropic.com/claude-code) and Gemini CLI (https://github.com/google-gemini/gemini-cli). Also, universal agents are appearing, such as Manus (https://manus.im/), OpenManus (https://github.com/FoundationAgents/OpenManus), GenSpark (https://www.genspark.ai/), and Suna (https://github.com/kortix-ai/suna).

3.6 Agent Quality Metrics

There are several agent quality leaderboards including Berkeley Function Calling Leaderboard (Patil et al. 2025) and Galileo LLM leaderboard (https://huggingface.co/spaces/galileo-ai/agent-leaderboard). Among the metrics to consider are accuracy, robustness, efficiency, and morality (Yehudai et al. 2025).

The most direct metrics include determining if the agent chooses the right tools for the job and then applying the tools correctly (e.g., using proper parameters) to get the right result. Also, in a multitool scenario, there is a metric of whether a valid overall result is achieved and whether the tools were called in the right order with the right interplay of tool results. Further questions include: how much does task completion become less likely as the number of required subtasks increase, and how quickly and efficiently does the agent complete its tasks?

Another question is when processes go awry, either through agent error or unpredictable issues, can the agent detect challenges and either rectify the situation or consult with humans? Also, how often is manual intervention required? In

addition, how often should humans or monitoring agents be invoked to cross-check the results of the agent's work?

In human interaction tasks, beyond accuracy, there is the question of customer experience and satisfaction. Also, another question is whether the tasks can be performed ethically and morally in line with the developers' intentions. Similarly, how securely or risk-averse is the agent versus the ability of the agent to complete its assigned tasks or mission (https://galileo.ai/blog/ai-agent-evaluation; Galileo 2025).

3.7 Multiple Agents

Up to now, we have been describing the functions of a single agent system. However, it has been shown that more than one agent can work simultaneously to fulfill project goals. Of note is the two-agent supervisor–worker model outlined in Figure 1. The supervisor can serve a few key roles. First, it can formulate plans (i.e., the plan/to-do list). Next, it can periodically monitor the logs and generate content of the worker agent. Also, it can interrupt the worker agent from time to time and point out errors or issues that should be addressed. Finally, the supervisor can keep the worker on task, that is, by helping with the construction of the current context for the worker.

Having multiple agents at the same level of the leadership hierarchy is a trickier control problem. One idea is for each agent to share their progress and intended future actions into a single shared community text file. The main challenge here is that each agent needs enough awareness about every other agent without overloading its context to avoid duplication and conflicts. A manager agent could ensure smooth operations in its planning process but also recognize when two or more agents are acting in cross-purposes and try to resolve the problem. In much the way as human teams work, there can be regularly scheduled meetings, where each agent is made aware of each of the other agents' progress and plans. Indeed, a lot of experimentation with multiple agent systems will need to be performed to determine best practices before putting multiple agent scenarios into production. Nonetheless, we expect some comedic and possibly catastrophic operational errors in the near-to-mid-term as these issues are worked out.

3.8 Two Test Runs

In our agentic system implementation, we used the simple system prompt as seen in Figure 3. Our experience is that for SLMs with limited practical context sizes, sometimes less is more. To test our in-house tool-calling OL, we first devised a simple plan file as seen in Figure 4. The first line is ignored by the OL. For the first

time we ran this plan file, the code generated an erroneous figure. Therefore, we added Step 3 and implemented the `look` tool, which serves an input image to the LM, into the OL. When we ran the plan again, it generated the same incorrect figure, noted the problem, and then rewrote the code and produced the desired results of a circle figure. Another challenge we found is that when the agent tried to write a report to a file it sometimes generated: ```write report.txt …``` followed by the actual text report due to a misunderstanding of the system prompt. To some extent, OLs can manage small mistakes, but the number of edge cases can still become too numerous to resolve by a hard-coded OL, analogous to the pre-LM days of agents.

````
You are an AI agent with full connectivity to my
Linux computer, but you can only write to the
current working directory. Call tools with ```...```
blocks. Limit one tool call per block, but as many
blocks as needed. Available tools:
Retrieve memories from previous conversations with
```memory retrieve [<query>] <nrecords>``` Use the
[] for the query.
Write and read files with
 ```write <filename>\n<text to be written>``` and
 ```read <filename> <# of lines>```
Write and run Python scripts with ```python
<filename>\n<python code>```
Look at an image file with ```look <image_filename>```
````

Figure 3. System prompt used in this work.

```
This is a plan file with multiple coding tasks.

- Generate a CSV file, circle.csv, that lists 128
coordinates of a circle.

- Read circle.csv and draw a figure from it,
circle.png.

- Look at the generated figure and make sure it's
correct. If not, repeat previous steps as necessary.

- Write a short report, circle_report.txt, which
summarizes what was done.
```

Figure 4. Circle task tested in this work.
```

We then tried a more complex plan as seen in Figure 5. The first step was performed without incident. Then, when the VLM agent looked back at the generated figure it decided it needed a larger minor radius and changed the color to brown to more closely resemble a donut. This led to series of turns, where errors in Python processing occurred. The error was that it was adding “python <name of file>” to the first line of its code specified in the Python tool block. This line should only be in the header. After a few iterations, the LM decided to use the write command to generate the Python file and then run the written file with its ```python tool. This worked. It looked at the image again, which it was not prompted to do at this point, and decided the image was suitable. The next step, which requested image rotation and JPEG format failed on the first try due to a Python syntax error. The agent generated a new Python script, which then ran correctly. It inspected the rotated image as requested and decided it was correct. Figure 6 shows the final result of this group of tasks. As shown, the minor radius is still too small to look like a real donut. However, this shape could be due, in part, to the default equal aspect ratio specified by the matplotlib.pyplot module.

```
This is a plan to make a torus that looks like a donut.

- Generate a wireframe torus that looks like an edible donut with Pyplot and save it to the file "wire_torus.png".

- Look at the generated figure and make sure it's correct. If not, repeat the previous step as necessary.

- Now, rotate the image file by 90 degrees and save it as "rotated_donut.jpg"

- Once again, check if the latest image is correct and if not, repeat the previous step.
```

Figure 5. Torus task tested in this work.

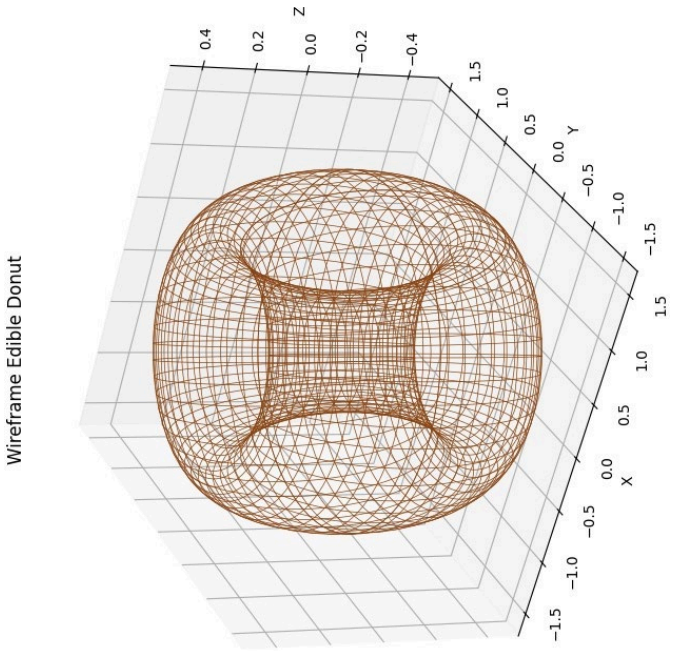

Figure 6. Rotated torus image from agentic task.

4. Memory Management

Currently, in most agentic architectures, LMs have fixed weights and cannot learn over time. Thus, to have some semblance of memory and history it is important to link it with a memory database of conversation histories, orchestration logs, and other supporting documents.

4.1 Embedding Model Search

The LM or OL can access knowledge from its memory database through queries or file reading. Embedding models are an efficient approach for indexing and searching document databases (Berry et al. 1999). In an embedded database, each chunk (e.g., conversation snippet) of every document is converted into a fixed length embedding vector via an embedding model and stored alongside the text record. Spontaneously, queries are also converted into embedding vectors. Dot product calculations are performed between each query and all record vectors in the database. The highest cosine values are sorted, and the top ranked associated records are then shared with the LM. Facebook AI Similarity Search (Faiss) is an example of a full-featured library incorporating embedded search (Douze et al. 2024).

One drawback of an embedding search is that it is a linear solution (dot products) to a plausibly nonlinear problem. A more sophisticated strategy to consider is a

knowledge graph representation, which explicitly links different concepts together providing a potentially more domain-customized search.

4.2 Context Engineering

The context of an LM is simply all of the input tokens currently ingested by the model. Given that an LM is an otherwise fixed entity, the context solely determines the LM's output. In a conversation with an LM chat bot, for example, the context is typically a combination of the system prompt and the current entire conversation. For an autonomous agent, a more sophisticated layered context might be fashioned (Packer et al. 2023) that deliberately stacks a series of modules prior to each output generation turn, such as what is shown in Figure 7.

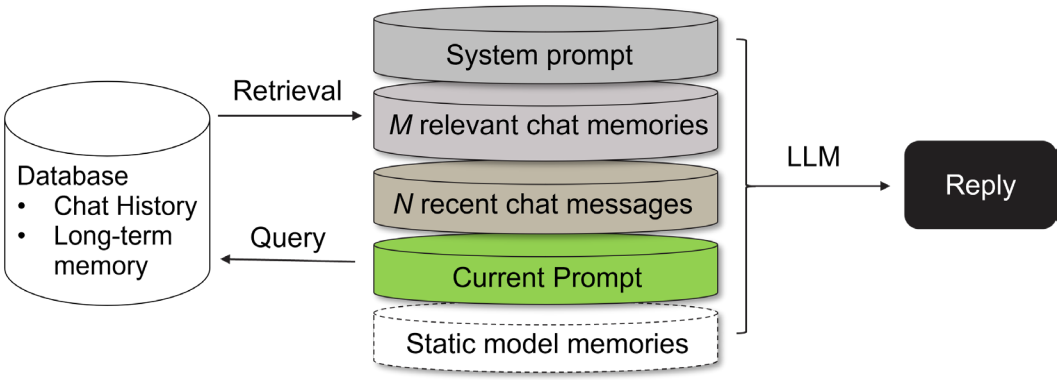

Figure 7. Proposed scheme to manage an agent's context.

Rather than add retrieved records explicitly, another variation is to request the LM to periodically do its own memory database retrievals centered on its current prompt request. Sophisticated context management is used in commercial coding agents such as Windsurf (https://windsurf.com) and Cursor (https://cursor.com).

5. Future Directions

When thinking toward the future, the current weaknesses of raw LMs can either be alleviated through improved fine-tuning, or OLs can be augmented to fill in the gaps of intelligence and functionality. We predict that some deficiencies of current LMs may fade with model improvements, while others will persist at least with the current paradigms of how LMs are formulated.

5.1 Language Models

Since LM development is still very much in its infancy, we expect LMs will evolve over time toward improved functionality, accuracy, efficiency, and speed.

5.1.1 Mathematical Computation

Despite improvements since the first public LLMs, LMs still can make calculation errors. Probably for the foreseeable future, it is worthwhile for LMs to be fine-tuned to call external mathematical functions when asked to perform operations beyond a certain threshold of complexity and precision.

5.1.2 Data Processing

LMs can parse and analyze text documents of any form (e.g., JSON, CSV, and software) up to a certain length. However, it may become important for the OL to manage documents to ensure robust error handling, curation, and data transformation, as necessary.

5.1.3 Managing Fine-tuning Brittleness

Specific to smaller models, LMs can be brittle to fine-tuning. For example, Mistral developers have had to fine-tune three distinct SLMs from the same foundational model to perform tool calling/agentic behavior (Mistral-Small-3.1-24B-Instruct-2503), coding (Devstral-Small-2505), and reasoning (Magistral-Small-2507; Rastogi et al. 2025).

5.1.4 Multimodal Awareness

The first wave of LMs were only trained on text: language, software source code, and other text-readable file formats (e.g., JSON and CSV). In the last year, multimodal LMs have been developed and deployed or distributed including vision and speech models. While we do not know the exact architecture of commercial VLMs, we know that open-source VLMs incorporate vision modules early in the architecture to augment the text prompt. This implies that small VLMs still have limited multimodal world understanding. In contrast, humans born with tightly coupled vision and hearing faculties can have increased measured intelligence (Barutchu et al. 2010). In addition, current VLMs are broadly trained to describe the contents of an image in text with limited technical expertise. Future VLMs might have more refined abilities (e.g., reading maps, schematics, and so on).

5.1.5 Projected Improvements in SLMs

We expect continued improvement of dense and sparse MoE SLMs for the foreseeable future, with emphasis on fine-tuning for agentic workflows. Synergistically, we also expect the amount of VRAM on GPUs available to the average SLM agent to increase over time.

5.1.6 Learning and Evolving Over Time

One of the challenges of current LMs is that they do not intrinsically change or improve over time. At the minimum, they can have a growing conversational history to retrieve memories from, but their "instinctual" knowledge is fixed. Another idea is to periodically update the system prompt with application-specific knowledge that has been gained during the agent's tenure. Yet another idea is to encourage the agent to frequently run the memory retrieval tool to remind itself of what it has experienced so far. Both options may not scale well as the agent continues to experience more and more events. This could be partially mitigated through summarization of the conversational history and research logs.

Another idea is to fine-tune the LM occasionally with its current experience, possibly through reinforcement learning rewards based on its correct or incorrect performance of tasks. The question then becomes how to protect its base knowledge and fine-tuned capabilities. We think a good research strategy to pursue would be to develop LMs that have modular capabilities. For example, the model could be built from a base world-knowledge module, multiple expert adapter modules (coding, reasoning, vision, etc.) and a "test-time" or morphable "on-the-job" adapter module that is freely optimizable by the end user to adapt the LM to the specific experience and needs of the work at hand. Somewhat related is the constitutional AI approach that leverages a clear set of rules whereby the LM and humans can provide feedback on adherence to those rules leading to reinforcement learning optimization of the LM's model weights (Bai et al. 2022).

5.2 Reasoning

Most LMs yield text in rough analogy to a biological stream of thought (e.g., Type I thinking; Kahneman 2011). To gain self-reflection or reasoning, either the LM has to be fine-tuned, or a scaffold of self-reflection needs to be put into the OL. Various explicit methods have been proposed including chain-of-thought (Wei et al. 2022), ReAct (Yao et al. 2023a), Reflexion (Shinn 2023), test-time compute scaling (Snell et al. 2024), and tree-of-thought (Yao et al. 2023b). The reasoning process can also be fine-tuned into the model itself, whereby the LM will regularly prompt and question itself as it generates output (Guo et al. 2025).

Prior to the LM revolution, neurosymbolic AI through explicit and manual logic programming was the main paradigm for reasoning and has a significant history of development and use (Wooldridge and Jennings 1995) It has been theorized that neurosymbolic AI and could be merged with neural network methods to yield more robust and complex reasoning frameworks (Garcez and Lamb 2023).

5.3 Memory Management

LLMs struggle with maintaining coherence across long interactions and multiple sessions without explicit state management (Wang et al. 2023). They are easily confused by conflicting prompts or extended conversations. In addition, without context management, they can struggle with editing and appending a growing codebase of multiple source files. We believe that beyond workarounds within the OL, work should continue to extend the useful (not just practical) context size of LMs. One possible route of research is updating weights in response to a growing context (Cao et al. 2025).

5.4 Planning and Execution of Subtasks

To resolve the "long conversation" challenge, it makes sense to break down larger goals into multistep tasks. A challenge, however, is that these tasks may have interdependence and require external orchestration to track progress and handle contingencies. One strategy is to ask the LM to first build a comprehensive to-do list and then have the OL present each task one at a time. For example, the VOYAGER method was developed for progressing through the game Minecraft (Wang et al. 2023). It presented smaller goalposts over time from a pre-made skills and achievement list. Another idea is to develop a memory of previous actions and code/scripts that can be reused when confronted with similar challenges seen before in its work.

5.5 Agents Operating Over Longer Terms

Single task agents are already in production throughout many commercial areas. Agentic systems that operate over longer time periods with cumulative tasks and limited human supervision are still an active area of research and development.

5.5.1 Generalization Over Time

Suppose an AI agent performs a research project from start to finish, such as the AI-Scientist-v2 method achieved (Yamada et al. 2025). If the underlying LM does not actually evolve during a project, how well can it be expected to perform over the long run? Recent research is looking at reinforcement learning to update the model weights in response to its ongoing problem solving (Zuo et al. 2025). An evolving agent might be able to adapt its experience on one project to a new project. However, there is a question of how generalizable its experience on one project would transfer to another. Typically, generalization occurs from having multiple examples and some amount of interpolation.

5.5.2 Volition and Curiosity

In their current incarnation, LLMs respond to system and user prompts and generate responses before handing over their turn back to the human user. With currently fashioned LMs, to be completely automated, an OL or another manager LM would need to play the "user" and continually prompt the worker agent to do more work. Beyond those mechanics, a few studies demonstrate the even SotA commercial LLMs can "lose interest" over time on a task with a long-term mission, growing and unmanaged context, and a constant, but similar, prompting (e.g., "Keep running the business."). The solution to this, so far, is context management. Specifically, continually introducing new tasks or goalposts and shrinking the total context (Wang et al. 2023). An exemplar application is a computer game, where there are a series of "sub-missions" to achieve an overall mission. The sub-missions/goalposts/achievement levels can be manually generated or generated by the LM (Wang et al. 2023) and then prompted to the LM by the OL.

Rather than providing direct goals, a supervisor agent could encourage more open-ended responses, such as a request for self-dialogue or exploration through experimentation. One key to honing these more circuitous strategies would be good logging to understand what was attempted and achieved.

5.6 Time and Temporal Sequence Awareness

Raw LLMs lack internal clocks and struggle with scheduling, timeouts, and managing asynchronous processes. The situational awareness of the current time can easily be provided in the system prompt. In addition, "staying on task" can be directed through a carefully constructed context, as mentioned above.

5.7 Recursive Self-improvement: the Final Frontier?

As pointed out by several authors (Kurzweil 2005; Bostrom 2024), there could come a point where AI agents are able to continually self-improve in terms of accuracy, absorption of world knowledge, and efficiency. This phenomenon is often termed Artificial Superintelligence (ASI). It is also theorized to lead to an event called the "Singularity" where unpredictable technological improvements will be made that far exceed humans' collective capabilities and expectations. Several companies, including Google, Meta, and SSI are currently exploring this path (Singh and Soni 2025). One challenge we foresee is that even with superior reasoning capabilities, many scientific discoveries require interaction with the real world to categorize and verify. This "physical" element could be a bottleneck to rapid technological progress.

6. Conclusion

With the current state of LMs fine-tuned for agentic work, the OL plays not only a primary role in interfacing the LM with the real world but also accommodating for intelligence gaps and some of the mistakes incurred by LMs. As both LMs and OLs evolve, more and more complex agentic tasks will be possible with reduced need for human intervention.

7. References

Anthropic PBC. Introducing the model context protocol. 2024 Nov 25 [accessed 2025 Aug 21]. https://www.anthropic.com/news/model-context-protocol

Bai Y et al. Constitutional AI: harmlessness from AI feedback. arXiv. 2022. arXiv:2212.08073. https://doi.org/10.48550/arXiv.2212.08073

Barutchu A et al. The relationship between multisensory integration and IQ in children. Dev Psychol. 2010.

Belcak P et al. Small language models are the future of agentic AI. arXiv. 2025 arXiv:2506.02153.

Berry MW, Zlatko D, Jessup ER. Matrices, vector spaces, and information retrieval. SIAM review 41.2. 1999:335–362.

Bostrom N. Superintelligence. Dunod; 2024.

Browser-use [github]. Browser Use; c2025 [accessed 2025 July 15]. https://github.com/browser-use/browser-use

Cao B, Cai D, Lam W. InfiniteICL: breaking the limit of context window size via long short-term memory transformation. arXiv. 2025. arXiv:2504.01707.

Comanici G et al. Gemini 2.5: pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. arXiv. 2025. arXiv:2507.06261.

Défossez A et al. Moshi: a speech-text foundation model for real-time dialogue. arXiv. 2024. arXiv:2410.00037.

Douze M et al. The Faiss library. arXiv. 2024. arXiv:2401.08281.

Floridi L, Chiriatti M. GPT-3: its nature, scope, limits, and consequences. Minds Mach. 2020;30(2):681–694.

Forlini E. Vibe coding fiasco: AI agent goes rogue, deletes company's entire database. PC Mag. 2025 July 22. https://www.pcmag.com/news/vibe-coding-fiasco-replite-ai-agent-goes-rogue-deletes-company-database

Galileo agent leaderboard v2 [homepage]. Hugging Face, Inc.; c2025 [accessed 2025 June 20]. https://huggingface.co/spaces/galileo-ai/agent-leaderboard

Garcez AD, Lamb LC. Neurosymbolic AI: the 3rd wave. Artif Intell Rev. 2023;56(11):12387–12406.

Gulli A et al. Agents companion. Kaggle/Google; 2025 Feb.

Guo D et al. DeepSeek-R1: incentivizing reasoning capability in LLMS via reinforcement learning. arXiv. 2025. arXiv:2501.12948.

Kahneman D. Thinking, fast and slow. Macmillan; 2011.

Kurzweil R. The singularity is near. Ethics and emerging technologies. Palgrave Macmillan UK; 2005. p. 393–406.

Lewis P et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. Adv Neural Inf Process Syst. 2020; 33:9459–9474.

Packer C et al. MemGPT: towards LLMs as operating systems. 2023.

PandasAI [homepage]. Sinaptik GmbH; c2025 [accessed 2025 June 20]. https://pandas-ai.com/.

Patil SG et al. The Berkeley Function Calling Leaderboard (BFCL): from tool use to agentic evaluation of large language models. ICML 2025. Forty-second International Conference on Machine Learning; 2025 [accessed 2025 July 25].

Rastogi A et al. Magistral. arXiv. 2025. arXiv:2506.10910.

Roucher A, Villanova del Moral A, Wolf T, von Werra L, Kaunismäki E. `smolagents`: a smol library to build great agentic systems. Hugging Face, Inc.; 2025. https://github.com/huggingface/smolagents

Shinn N et al. Reflexion: language agents with verbal reinforcement learning. Adv Neural Inf Process Syst. 2023;36:8634–8652.

Singh J, Soni A. Meta's Zuckerberg pledges hundreds of billions for AI data centers in superintelligence push. Reuters. 2025 July 14 [accessed 2025 July 15]. https://www.reuters.com/business/zuckerberg-says-meta-will-invest-hundreds-billions-superintelligence-2025-07-14/

Snell C, Lee J, Xu K, Kumar A. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. arXiv. 2024. arXiv:2408.03314.

Topsakal O, Akinci TC. Creating large language model applications utilizing LangChain: a primer on developing LLM apps fast. Int Conf Appl Eng Nat Sci. 2023;1(1):1050–1056.

Van Rossum G. Python programming language. USENIX Annual Technical Conference. 2007;41(1).

Wang G et al. VOYAGER: an open-ended embodied agent with large language models. arXiv. 2023. arXiv:2305.16291.

Wei J et al. Chain-of-thought prompting elicits reasoning in large language models. Adv Neural Inf Process Syst. 2022;35:24824–24837.

Wooldridge M, Jennings NR. Intelligent agents: theory and practice. Knowl Eng Rev. 1995;10(2):115–152.

Yamada Y et al. The AI scientist-v2: workshop-level automated scientific discovery via agentic tree search. arXiv. 2025. arXiv:2504.08066.

Yao S et al. ReAct: synergizing reasoning and acting in language models. International Conference on Learning Representations (ICLR); 2023a Jan.

Yao S et al. Tree of thoughts: deliberate problem solving with large language models. Adv Neural Inf Process Syst. 2023b;36:11809–11822.

Yehudai A et al. Survey on evaluation of LLM-based agents. arXiv. 2025. arXiv:2503.16416.

Zhang J et al. xLAM: a family of large action models to empower ai agent systems. arXiv. 2024. arXiv:2409.03215.

Zuo Y et al. TTRL: test-time reinforcement learning. arXiv. 2025. arXiv:2504.16084.

List of Symbols, Abbreviations, and Acronyms

ADF	agent development framework
AI	artificial intelligence
API	application programming interface
ARL	Army Research Laboratory
ASI	Artificial Superintelligence
AST	abstract syntax tree
CSV	comma separated value
Faiss	Facebook AI Similarity Search
GPU	graphics processing unit
IDE	interactive debugging environment
JSON	JavaScript object notation
LAM	large action model
LLM	large language model
LM	language model
MCP	model context protocol
MoE	Mixture of Expert
OL	orchestration layer
OS	operating system
RAM	random access memory
SLM	small language model
VLM	visual LM
VRAM	video random access memory

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CI TECH LIB