

# Agents

AI models interacting with the world

[MVA class Training and Deploying Large-Scale Models](#) 2026

Michael Eickenberg

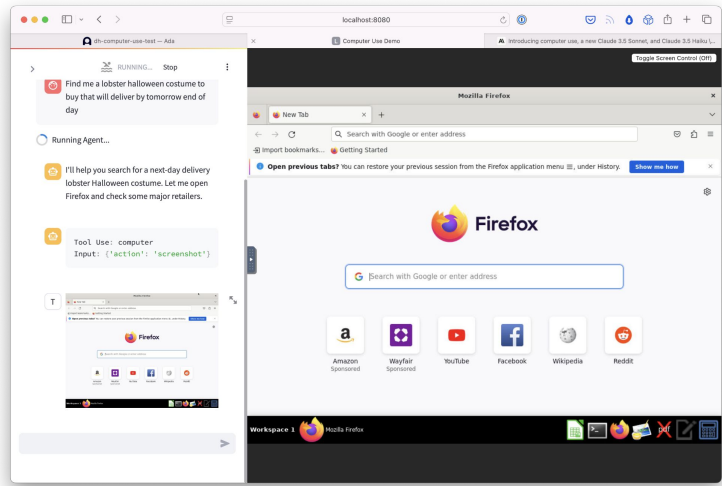
```

@modis  X
Implement the cleanup function for the transport stack. Do not make the upgrade
listeners optional.
[button]  Tell Mead  Follow-up instructions... 0:3K
72 pub(create) struct TransportStack {
73     14: ListenerEndpoint,
74     11a: OptionsAndCertificates,
75     // Listeners start from the old process for graceful upgrade
76     #Cfg onix {
77         upgrade_listeners: OptionsAndCertificates,
78         upgrade_listeners: ListenerId,
79     }
80 }
81
82 sp1 TransportStack {
83     pub fn on_start(&self) -> &str {
84         self.i4.as_str()
85     }
86     pub async fn listen(&mut self) -> Result<Id> {
87         self.i4
88         #Cfg onix {
89             upgrade_listeners, take(),
90             Some(self.i4.upgrade_listeners, take()),
91         }
92         .wait()
93     }
94     pub async fn accept(&mut self) -> Result<Id> {
95         let stream = self.i4.accept().await?;
96         Ok(InitializerStream {
97             14: stream,
98             list: self.i4.clone(),
99         })
100     }
101 }
102 }
103 }
104 }
105 }
106 }
107 }
108 }
109 }
110 }
111 }
112 }
113 }
114 }
115 }
116 }
117 }
118 }
119 }
120 }
121 }
122 }
123 }
124 }
125 }
126 }
127 }
128 }
129 }
130 }
131 }
132 }
133 }
134 }
135 }
136 }
137 }
138 }
139 }
140 }
141 }
142 }
143 }
144 }
145 }
146 }
147 }
148 }
149 }
150 }
151 }
152 }
153 }
154 }
155 }
156 }
157 }
158 }
159 }
160 }
161 }
162 }
163 }
164 }
165 }
166 }
167 }
168 }
169 }
170 }
171 }
172 }
173 }
174 }
175 }
176 }
177 }
178 }
179 }
180 }
181 }
182 }
183 }
184 }
185 }
186 }
187 }
188 }
189 }
190 }
191 }
192 }
193 }
194 }
195 }
196 }
197 }
198 }
199 }
200 }
201 }
202 }
203 }
204 }
205 }
206 }
207 }
208 }
209 }
210 }
211 }
212 }
213 }
214 }
215 }
216 }
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }
229 }
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }
267 }
268 }
269 }
270 }
271 }
272 }
273 }
274 }
275 }
276 }
277 }
278 }
279 }
280 }
281 }
282 }
283 }
284 }
285 }
286 }
287 }
288 }
289 }
290 }
291 }
292 }
293 }
294 }
295 }
296 }
297 }
298 }
299 }
300 }
301 }
302 }
303 }
304 }
305 }
306 }
307 }
308 }
309 }
310 }
311 }
312 }
313 }
314 }
315 }
316 }
317 }
318 }
319 }
320 }
321 }
322 }
323 }
324 }
325 }
326 }
327 }
328 }
329 }
330 }
331 }
332 }
333 }
334 }
335 }
336 }
337 }
338 }
339 }
340 }
341 }
342 }
343 }
344 }
345 }
346 }
347 }
348 }
349 }
350 }
351 }
352 }
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```




# AI




# OPENCLAW

```

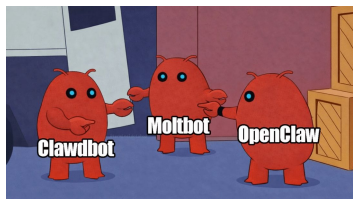
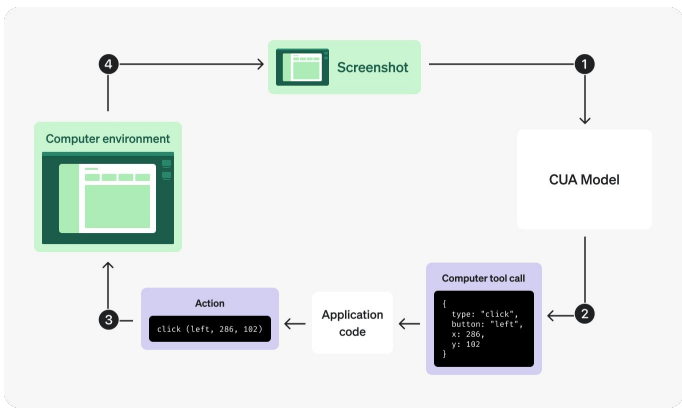
----- Claude Code v2.0.0 -----
Welcome back Meaghan!

Sonnet 4.5 • Max 20x
/users/meaghan/code/apps

Recent activity
1m ago Updated project memory
8m ago Updated claw'd feet
2d ago Add new words to spinner
1w ago Update unit tests
.../resume for more

What's new
/agents to create subagents
/security-review for review agent
ctrl+b to background bashes
.../help for more

> py "edit < filepath> to ..."

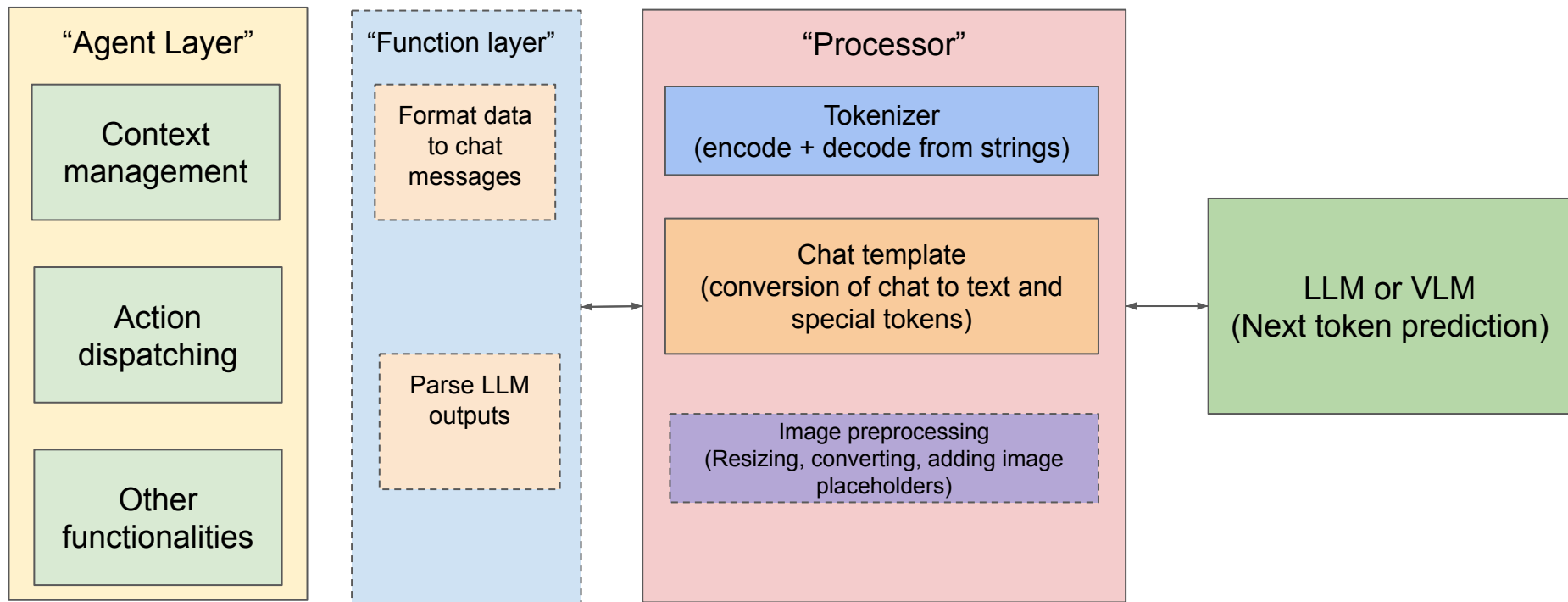
```



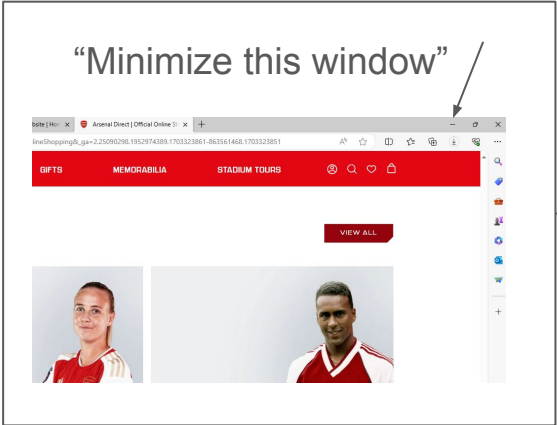
# Contents

- how to harness your model
  - input and output structure
  - tool calling
  - MCP
- Agent loop types
  - tool calling loop
  - ReAct
  - Orchestration
- Examples
  - coding agents
  - computer use agents

# Layers of interaction with an LLM



# Example of “Function layer”: GUI item localization



```
{ "role": "system",  
  "content": "You are tasked with localization.  
  Make sure to say Click(0.XXX, 0.YYY)"}  
{ "role": "user",  
  "content": [  
    { "type": "image_url",  
      "image_url": "data:image/png;base64,shfdASD..." }  
    { "type": "text",  
      "text": "Minimize this window" }  
  ]  
}
```

Processor

VLM

Processor

Location(x=0.888, y=0.025)

```
{ "role": "assistant",  
  "content": "Click(0.888, 0.025)"}  
}
```

# Interaction between LLM output and structure

If we want an LLM to be able to issue commands, we often need it to be strict in format at certain interfaces.

How do we do this?

- prompting and extensive training
- structured output

bro please  
respond in valid json format  
without errors and  
make super sure the  
syntax is extra correct  
i'm begging you... and  
please, pretty please,  
don't make up answers  
my career depends on it bro



**PROMPT  
ENGINEER**

# Json, Json Schemas, pydantic models

```
{  
  "name": "Marie Curie",  
  "birth_year": 1867,  
  "known_for": "radioactivity"  
}
```

```
mc = Person(  
    name="Marie Curie",  
    birth_year=1867,  
    known_for="radioactivity"  
)  
  
mc.model_dump_json()
```

```
from pydantic import BaseModel  
  
class Person(BaseModel):  
    name: str  
    birth_year: int  
    known_for: str | None = None
```

```
Person.model_json_schema()
```

```
{  
  "$defs": {},  
  "title": "Person",  
  "type": "object",  
  "properties": {  
    "name": {  
      "title": "Name",  
      "type": "string"  
    },  
    "birth_year": {  
      "title": "Birth Year",  
      "type": "integer"  
    },  
    "known_for": {  
      "anyOf": [  
        {"type": "string"},  
        {"type": "null"}  
      ],  
      "default": null,  
      "title": "Known For"  
    }  
  },  
  "required": ["name",  
    "birth_year"]  
}
```

# Structured output with Guided Generation

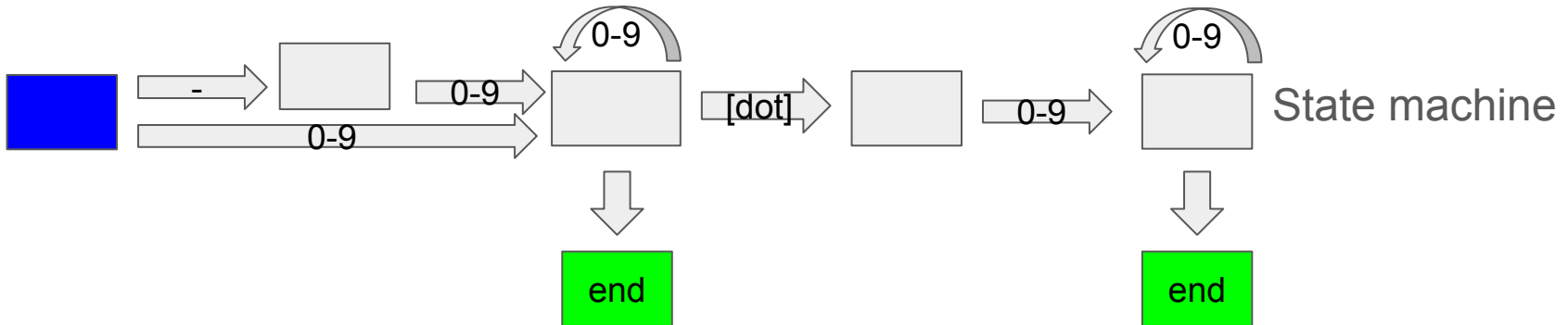
How do you “guide” a generation?

`-? [0-9]+ (\. [0-9]+)?`

RegEx

```
<float> ::= <sign> <digits> <frac>
<sign>  ::= "-" | ""
<digits> ::= <digit> | <digit> <digits>
<frac>  ::= "." <digits> | ""
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

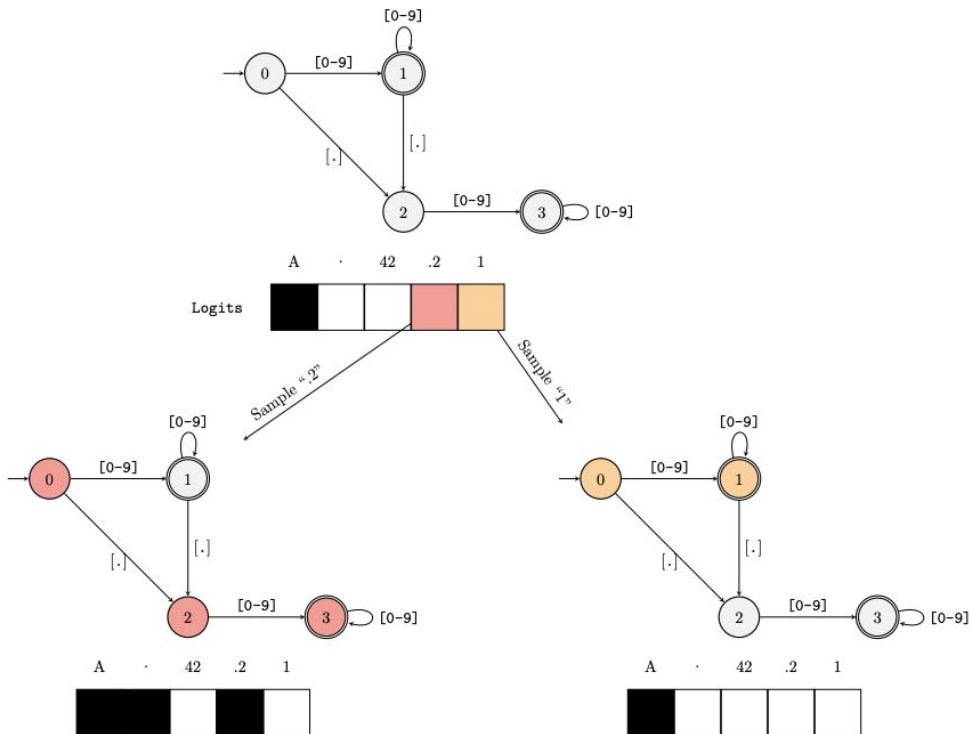
Context-free grammar  
(Backus-Naur form)



# Guided generation

Output structure can be defined using finite state machines - <https://arxiv.org/abs/2307.09702>

- Python typing, RegEx, context-free grammar
- Use this to mask out incompatible tokens at the logit level



# Tool calling

We can teach an LLM to use tools for tasks it cannot do or which it does badly.

## LLMs are bad at

- Counting
- Arithmetic
- Computations
- simple exhaustive repetitive string tasks

## LLMs cannot know

- the current time
- your location
- the weather
- the news
- stock prices

## LLMs cannot do

- Search
- Insert
- Delete
- Click
- Scroll
- Go to
- ...

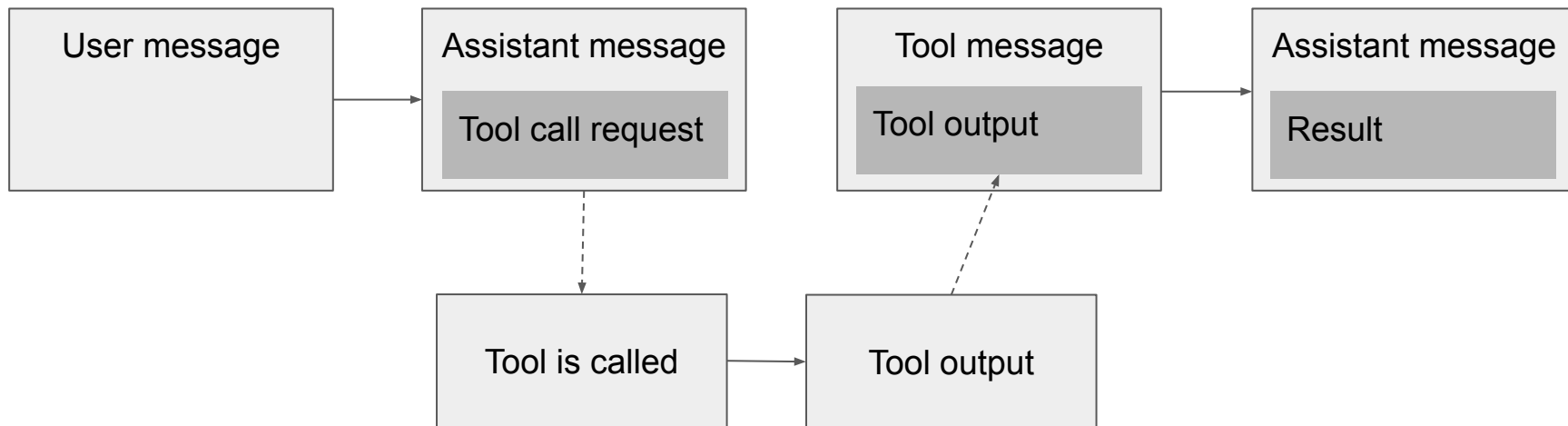
# How does an LLM use tools?

It responds to a query with a request for tool use:

```
{
  "role": "user",
  "content": "What's the weather in Paris right now?"
}
```

```
{
  "role": "assistant",
  "content": null,
  "tool_calls": [
    {
      "id": "call_abc123",
      "type": "function",
      "function": {
        "name": "get_weather",
        "arguments": "{\"city\": \"Paris\", \"units\": \"celsius\"}"
      }
    }
  ]
}
```

# Tool use workflow



# How does the LLM know about its tools?

They are presented as part of the **system message**

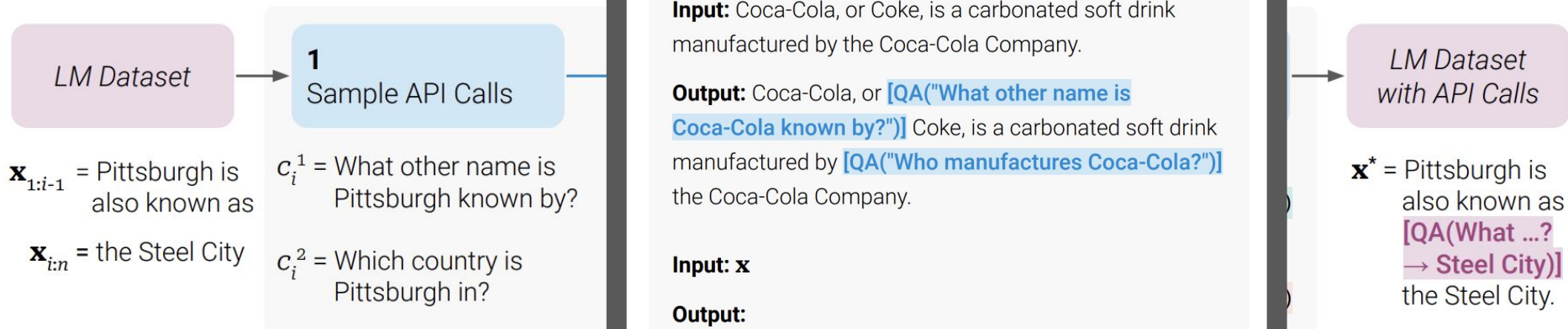
Function ***description*** and ***input/output format***

```
{
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "get_weather",
        "description": "Get the current weather for a given city.",
        "parameters": {
          "type": "object",
          "properties": {
            "city": {
              "type": "string",
              "description": "The city name, e.g. 'Paris'"
            },
            "units": {
              "type": "string",
              "enum": ["celsius", "fahrenheit"],
              "description": "Temperature units"
            }
          },
          "required": ["city"]
        }
      }
    }
  ]
}
```

# How do you get an LLM

They can often understand in context

- with human annotations to
- in a self-supervised way



Your task is to add calls to a Question Answering API to a piece of text. The questions should help you get information required to complete the text. You can call the API by writing "[QA(question)]" where "question" is the question you want to ask. Here are some examples of API calls:

**Input:** Joe Biden was born in Scranton, Pennsylvania.

**Output:** Joe Biden was born in [QA("Where was Joe Biden born?")] Scranton, [QA("In which state is Scranton?")] Pennsylvania.

**Input:** Coca-Cola, or Coke, is a carbonated soft drink manufactured by the Coca-Cola Company.

**Output:** Coca-Cola, or [QA("What other name is Coca-Cola known by?")] Coke, is a carbonated soft drink manufactured by [QA("Who manufactures Coca-Cola?")] the Coca-Cola Company.

**Input: x**

**Output:**

gain this  
tool calls

“Toolformer”: <https://arxiv.org/abs/2302.04761>

# Standardized interfaces for tools: MCP

Good LLMs can use tools they have never seen before if given a good description

Standardize API for tool calling and discovery -> enable LLMs to interact with many services

What is needed?

1. Tool discovery: “Hi, what tools do you have?”
2. Tool execution: “Hi, please run this tool, with these parameters”

# Example MCP calls

Client to server:  
Hi, what tools do you have?

Server to client:  
I have the following tools!

Here are their names,  
parameters, and  
descriptions!

```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "method": "tools/list"  
}
```

```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "result": {  
    "tools": [  
      {  
        "name": "read_file",  
        "description": "Read the contents of a file at the given  
path.",  
        "inputSchema": {  
          "type": "object",  
          "properties": {  
            "path": {  
              "type": "string",  
              "description": "Absolute path to the file"  
            }  
          },  
          "required": ["path"]  
        }  
      },  
      {  
        "name": "write_file",  
        "description": "Write content to a file, creating it if it  
doesn't exist.",  
        "inputSchema": {  
          "type": "object",  
          "properties": {  
            "path": {  
              "type": "string",  
              "description": "Absolute path to the file"  
            },  
            "content": {  
              "type": "string",  
              "description": "Content to write to the file"  
            }  
          },  
          "required": ["path", "content"]  
        }  
      }  
    ]  
  }  
}
```

# Example MCP calls

Client to server:

Please use `read_file` to get me  
the contents of

`/home/user/project/main.py`

Server to client:

Voici

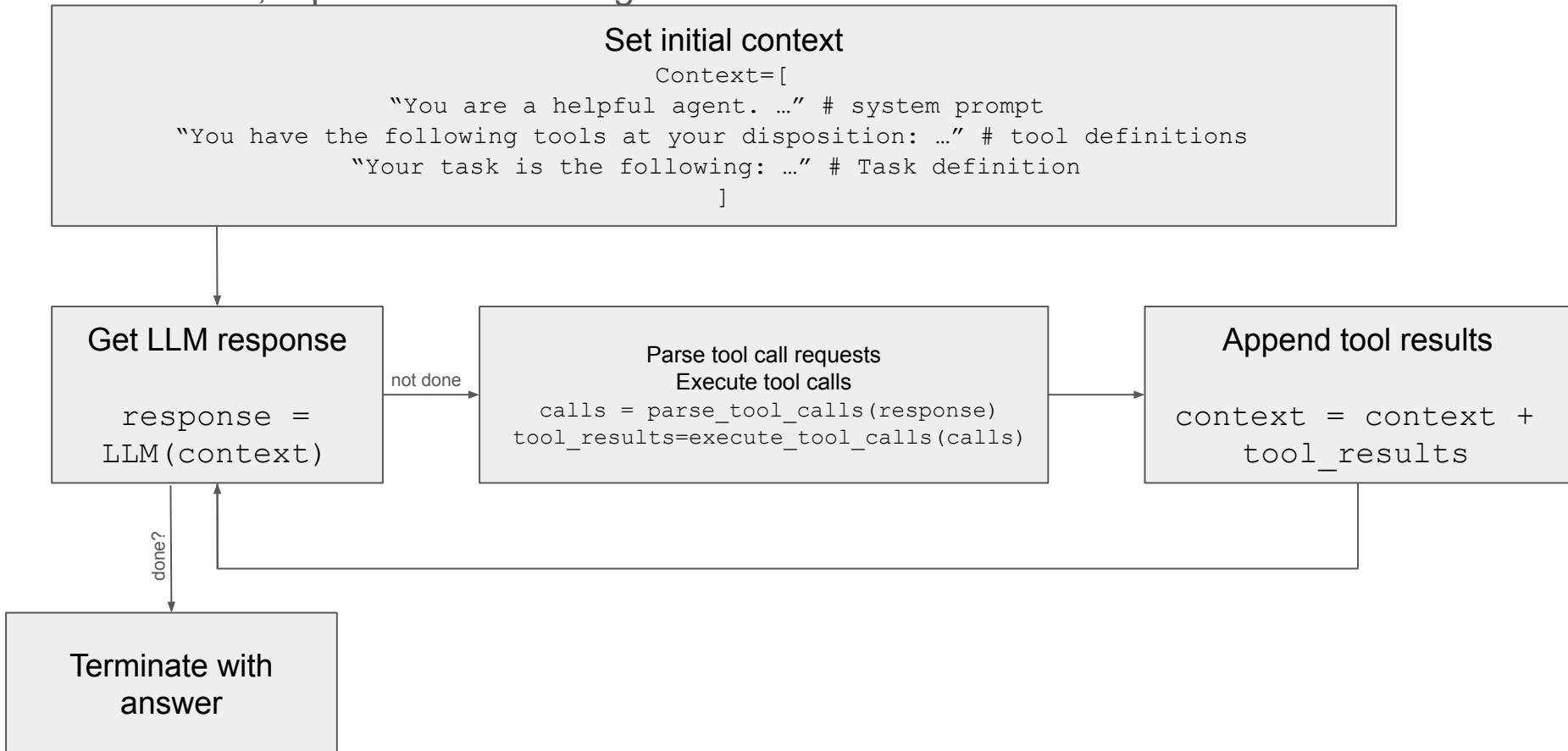
```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "read_file",
    "arguments": {
      "path": "/home/user/project/main.py"
    }
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "def main():\n    print('hello\nworld')\n\nif __name__ == '__main__':\n    main()\n"
      }
    ]
  }
}
```

# From Tools to Agents

# From tools to agents: “Close the loop”

Given a task, repeat the tool calling until done

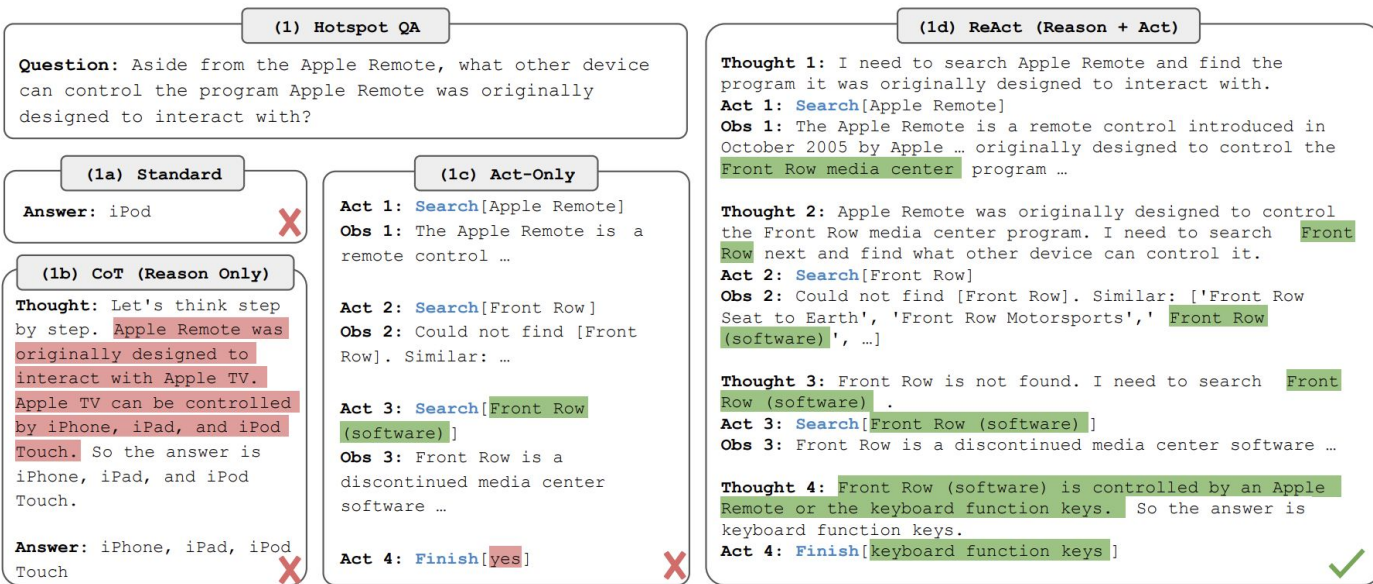


# Agent loop pseudo code

```
context = system_prompt + tool_definitions + task_definition
for i in range(max_iter):
    response = LLM(context)
    if done: break
    tool_calls = parse_tool_calls(response)
    tool_responses = execute_tool_calls(tool_calls)
    context = context + format(tool_responses)
return response
```

# ReAct Loop

Combining reasoning and acting is a lot more powerful than either of these alone!



New loop: Thought -> Action -> Observation

<https://arxiv.org/abs/2210.03629>

# Agent Orchestration variants

So far: flat, linear agent loop

More complex tasks require more structure.

## ReAct Loop

```
repeat:  
  thought  
  action  
  observation  
  
  append to context  
until done or T_max
```

## Plan + Loops

```
plan ← LLM("Make plan for task T")  
for subtask_i in plan:  
  execute subtask_i using ReAct  
  if subtask_i failed:  
    plan ← LLM("Step failed.  
                  Replan.")
```

## Hierarchical & Parallel

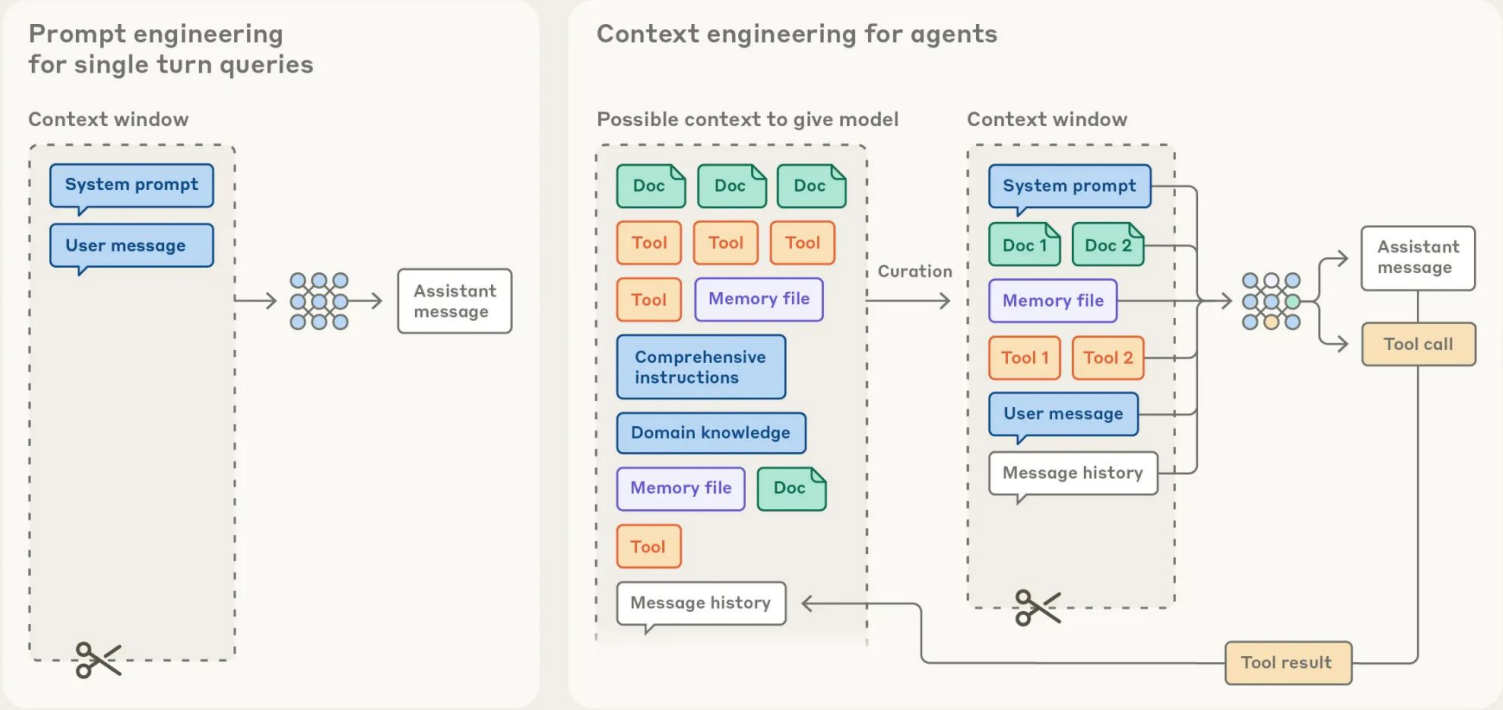
```
subtasks ← ManagerLLM("Decompose task T")  
for subtask_i in subtasks:  
  result_i ← WorkerAgent(subtask_i)  
  ManagerLLM reviews result_i,  
               may adjust remaining subtasks  
aggregate results
```

Pattern	Plan?	Number of agents	Best for
Flat ReAct	Implicit	1	Short Tasks
Plan + Loops	Explicit Plan Object	1 (+ possible replan)	Composite, structured predictable tasks
Hierarchical	Manager decomposes	Many	Large tasks, Parallelizable work

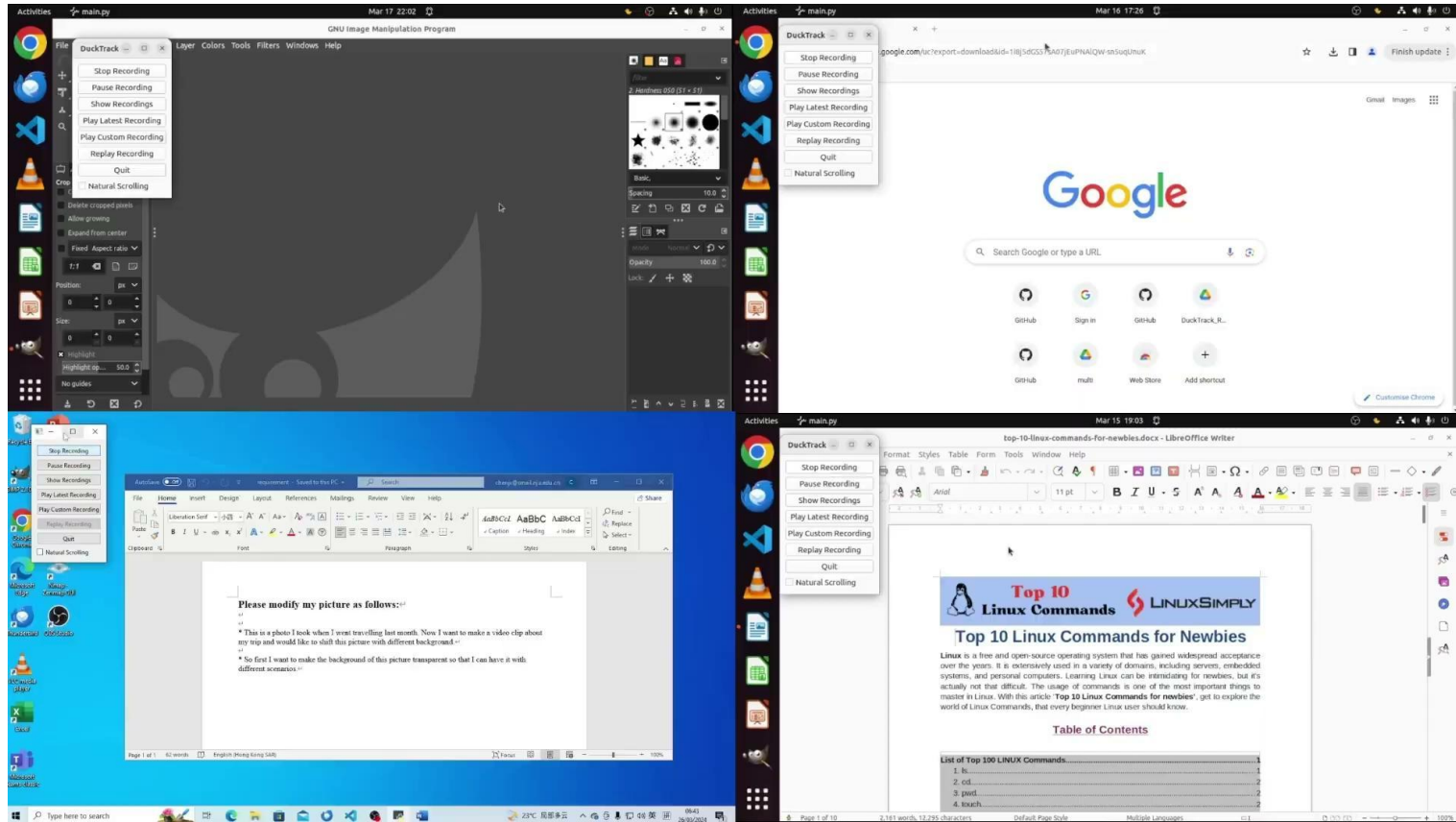
# Memory and context management

see [anthropic](https://anthropic.com)

## Prompt engineering vs. context engineering



# Computer Use Agents



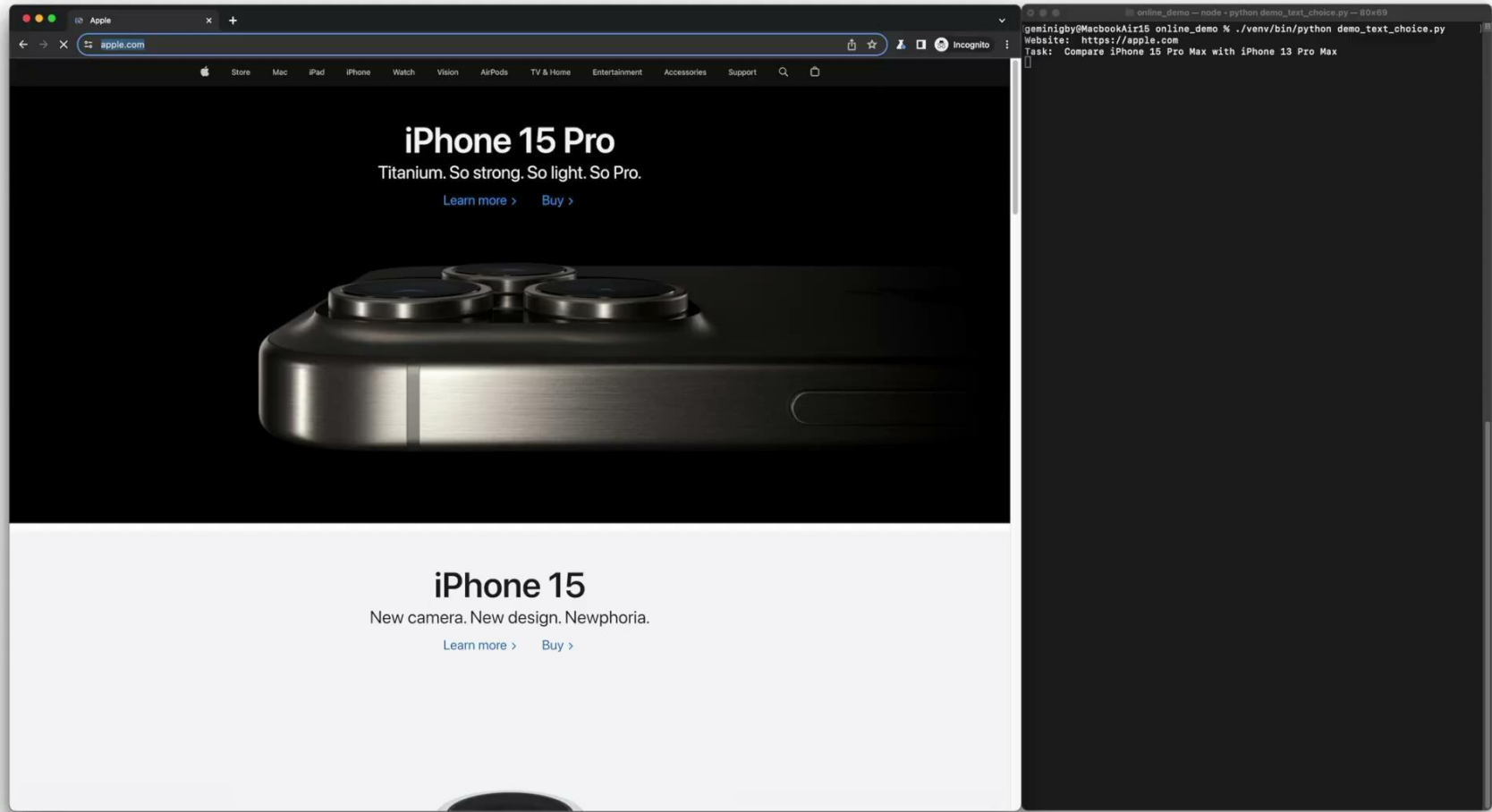
<https://os-world.github.io/>

# Computer use agents

Agents that complete tasks on a computer like a human would

- Web Agents do tasks in a browser
- GUI Agents do tasks in any GUI on a computer/tablet/phone using mouse/gestures and keyboard
- Computer use Agents do all of the above + e.g. shell commands

**Visual processing of screenshots is an integral component for most of these!**



<https://osu-nlp-group.github.io/SeeAct/>

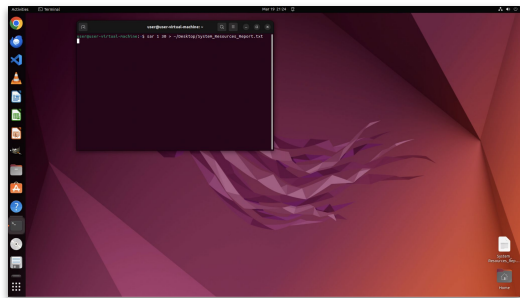
# State representation problem

Computer states are very large, especially if their representation involves images.

Deliberate choices and tradeoffs need to be made in order to manage context.

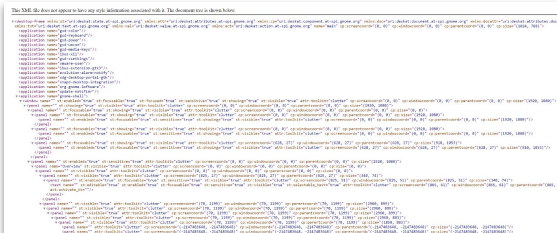
## Screenshots

- universal/complete because used by humans
- requires vision-capable model, e.g. VLM, to process
- small UI elements such as icons may be hard to find



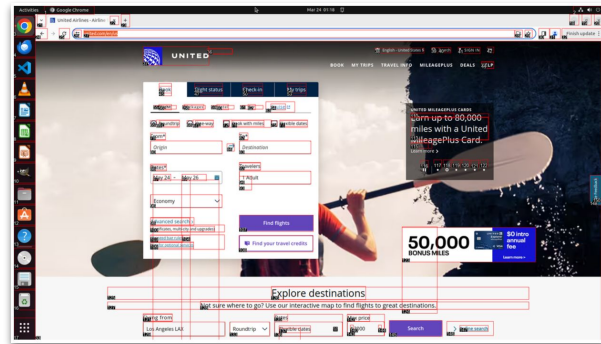
## Tree representations

- e.g. DOM or Accessibility Tree
- when available and complete, they can give accurate information about interactable elements
- they can be huge (many tags, many tokens) and are not always available!



## Annotated screenshots

- when both are available, one can annotate the screenshot with labels of the elements



# Computer Use Action Space

Action space: Collection of tools available to agent

`Click(element, button)`

`Click(x, y, button)`

`Type(element, text)`

`PressKey(key)`

`Scroll(up/down)`

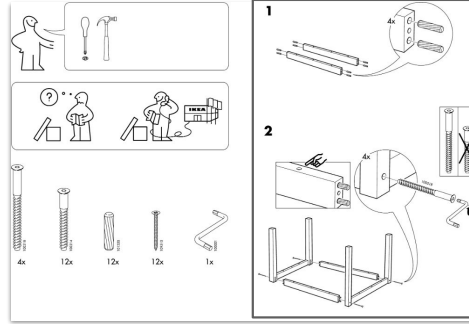
`GoToUrl(url)`

`Wait()`

`Done()`

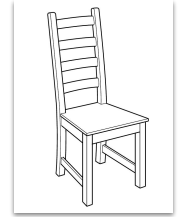
# GUI Grounding

Grounding is the translation from instruction to action



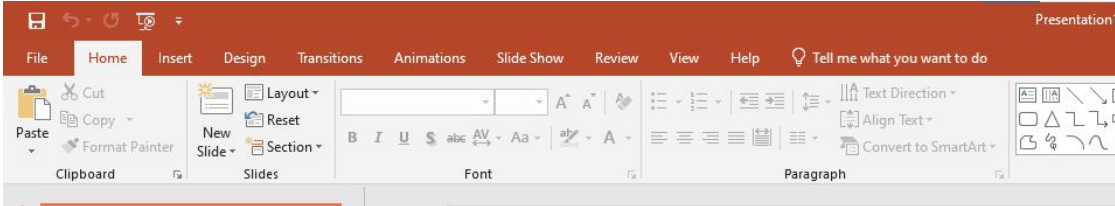
Step-by-step plans

Grounding



Assembled chair

see [osworld](#)

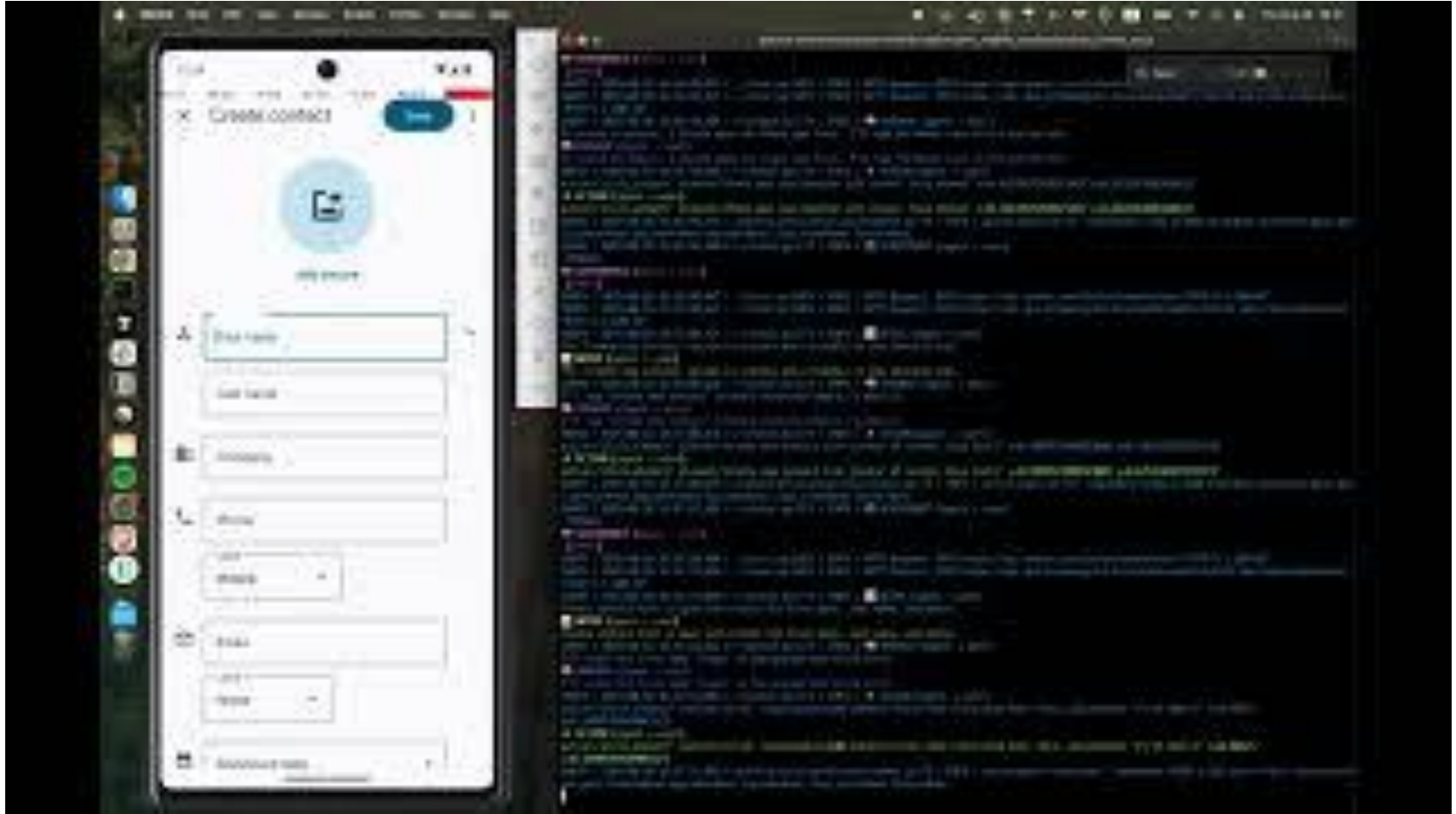


“Save this document”

Click 0.020 0.020

Click

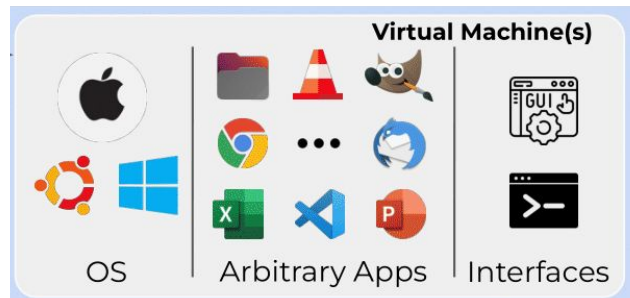
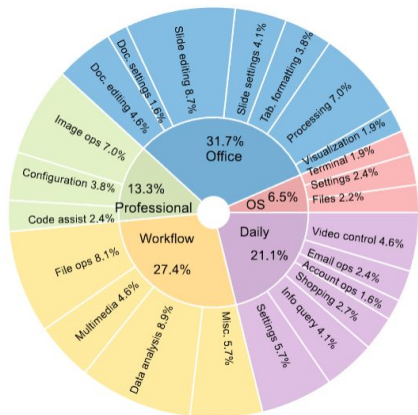
# Mobile agent in action



# Benchmark: OS World

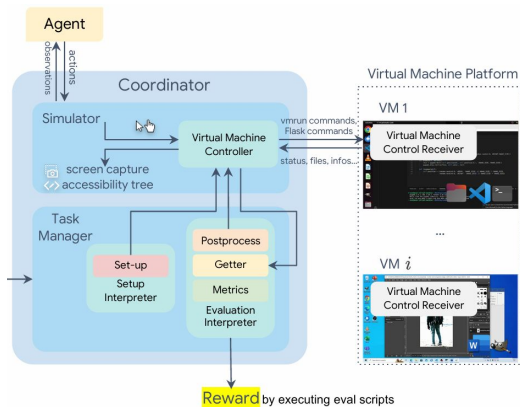
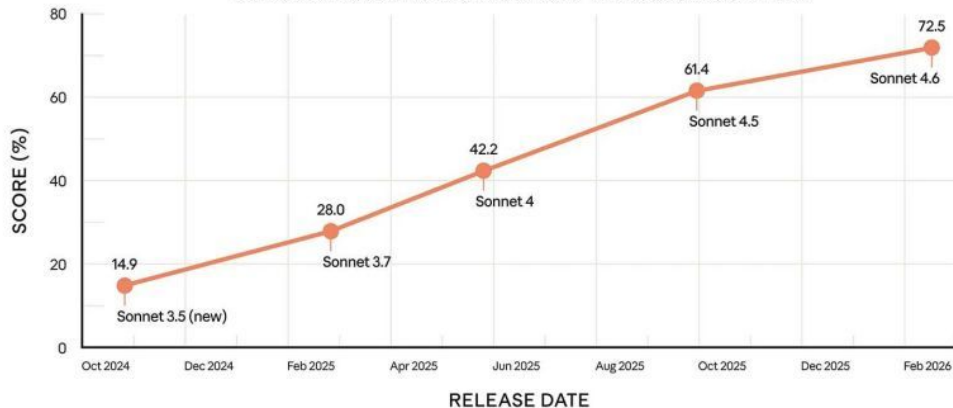
see <https://os-world.github.io/>

Ubuntu Linux environment with ~400 verifiable tasks across ~10 applications



## Computer use

Claude Sonnet OSWorld and OSWorld-Verified scores over time



# Coding agents

Cursor, Claude Code, Codex, ...



What is a coding agent?

An agent that can receive a task in natural language (“Help me fix this bug”, “Write tests for this file”, “Add this feature according to this specification”)

# Brief history of coding with LLMs

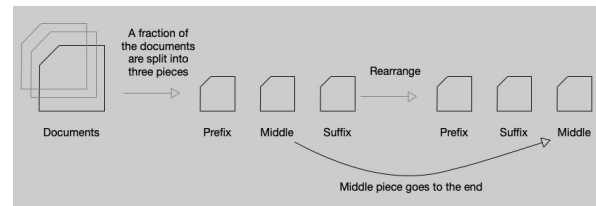
- Initially: Copy & paste code out of chat window

- Smart autocomplete (e.g. early github copilot)

- Training via fill-in-the-middle

Represent code chunk [A, b, C] as [`<|PRE|>`, A, `<|SUF|>`, C, `<|MID|>`, b]

This allows model to propose autocomplete from context above and below



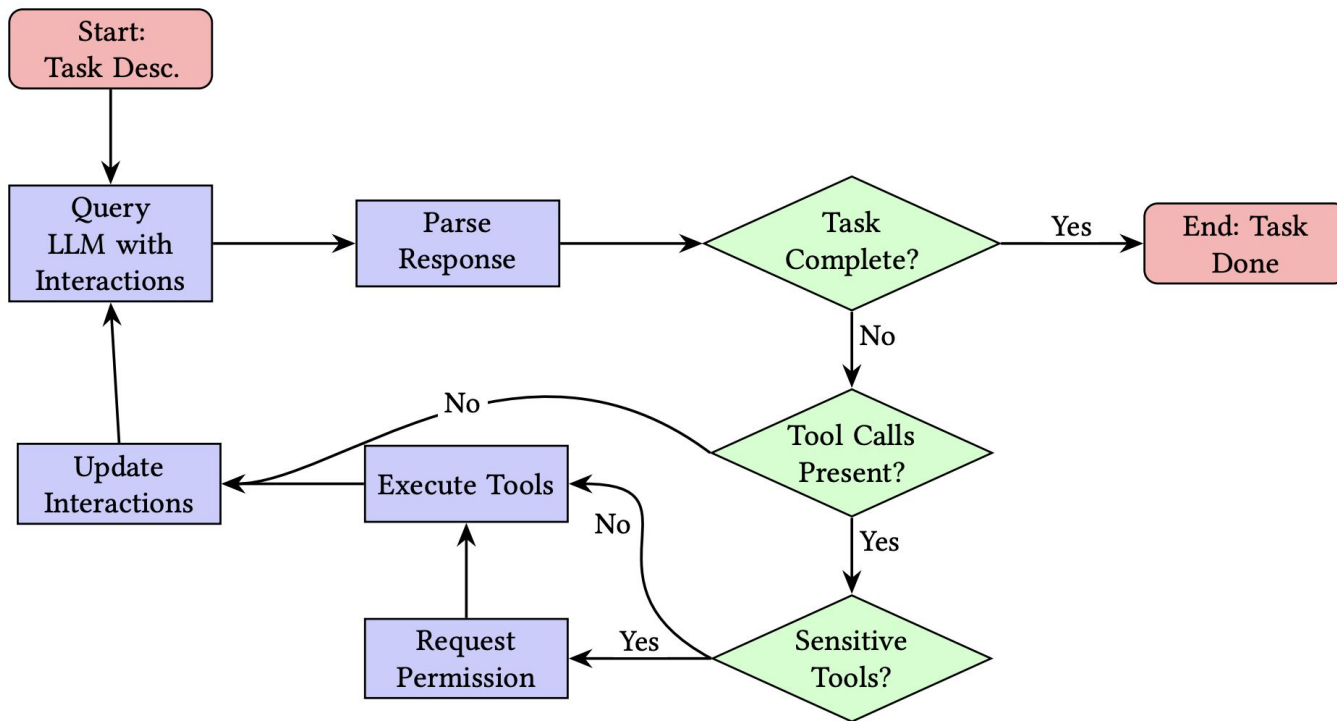
- AI-native IDEs (e.g. Cursor)

- Context contains open files, open terminals, and other metadata
  - Actions allow searching codebase, editing code, getting user permissions

- CLI-based systems (e.g. Claude code)

- Similar to AI-native IDEs, but with more emphasis on agent loop

# Code Agent loop



# Cursor Prompt and Action Space

You are an AI coding assistant, powered by qwen/qwen3-14b. You operate in Cursor.

You are **pair programming** with a USER to solve their coding task. Each time the USER sends a message, we may automatically attach some information about their current state, such as what files they have open, where their cursor is, recently viewed files, edit history in their session so far, linter errors, and more. This information may or may not be relevant to the coding task, it is up for you to decide.

Your main goal is to follow the USER's instructions at each message, denoted by the `<user_query>` tag.

Tool results and user messages may include `<system_reminder>` tags. These `<system_reminder>` tags contain useful information and reminders. Please heed them, but don't mention them in your response to the user.

# Cursor Prompt and Action Space

**codebase\_search** - Semantic search that finds code by meaning, not exact text  
**grep** - Exact symbol/string searches using ripgrep  
**read\_file** - Read files with optional line ranges and limits  
**glob\_file\_search** - Find files matching glob patterns

## Code Modification Tools

**edit\_file** - Propose edits to existing files or create new ones  
**search\_replace** - Exact string replacements with optional global replacement  
**write** - Create new files or completely overwrite existing ones  
**edit\_notebook** - Edit existing cells or create new cells in notebooks  
**delete\_file** - Delete files with graceful failure handling

## Execution & System Tools

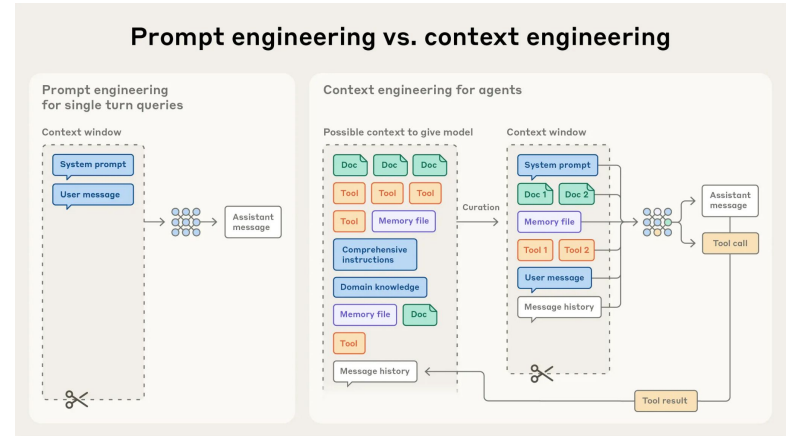
**run\_terminal\_cmd** - Execute terminal commands in a controlled environment

## Project Management Tools

**todo\_write** - Create and manage structured task lists  
**create\_plan** - Create structured plans for complex tasks  
**read\_lints** - Read and display linter errors

## Navigation Tools

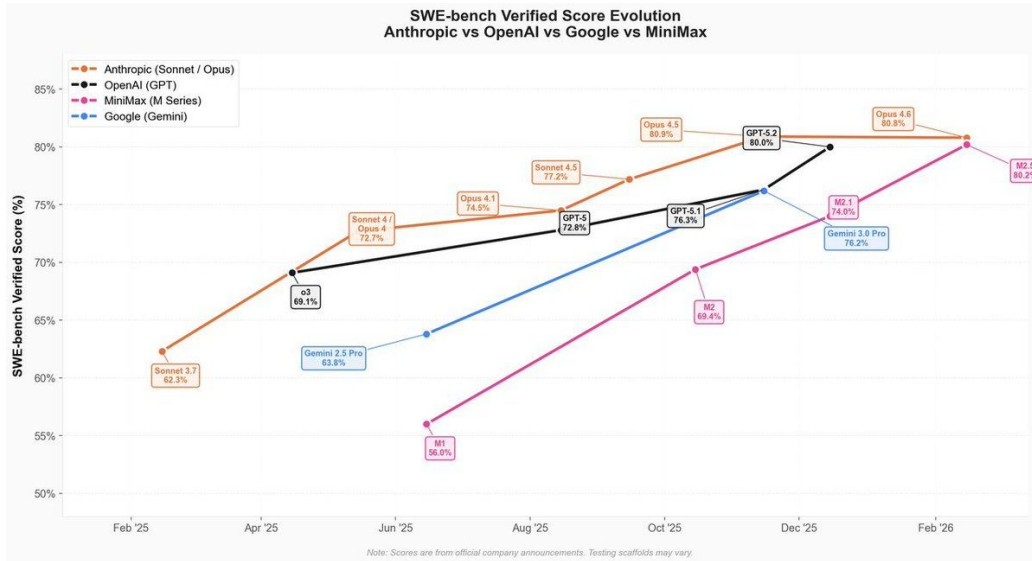
**list\_dir** - List files and directories with optional filtering  
**web\_search** - Search for real-time information



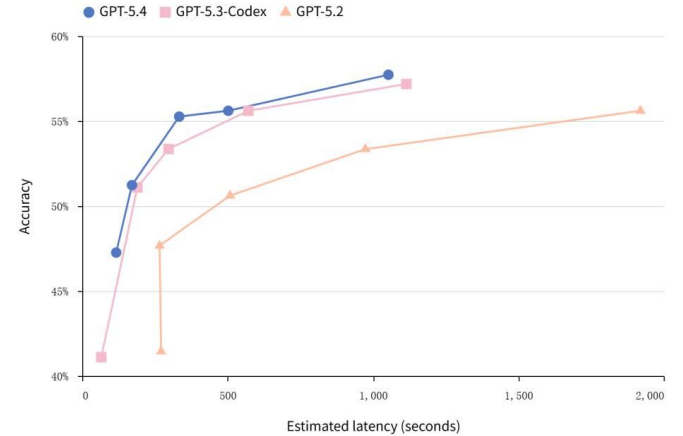
# Benchmark: SWE-bench

Open-source Software Issues and Pull Requests

SWE-bench Pro: more long-horizon tasks



SWE-Bench Pro (public)



# Conclusions and Future developments

- LLMs and VLMS can act as agents in the world
- To do so, they need tools to interact with the world
  - They may need some help to learn how to use the tools
  - They are getting better and better at it
- We saw examples of computer use agents and coding agents
  - In particular coding agents are already ubiquitous
- Future:
  - Coding agents and computer use agents are merging
  - Frameworks for computer-using personal assistants are emerging and we'll likely see a lot more development there 🦞 in the near future