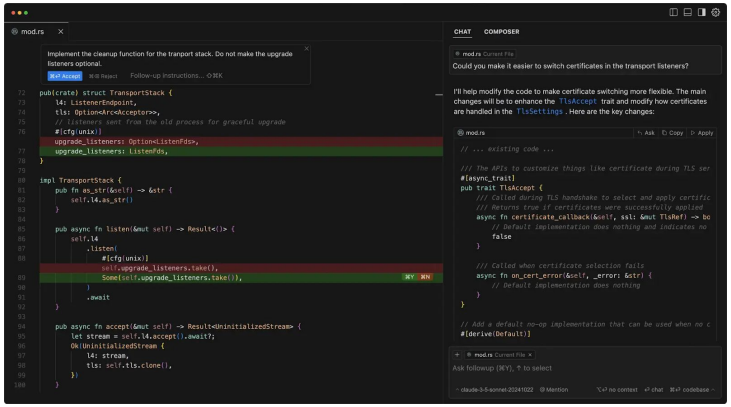
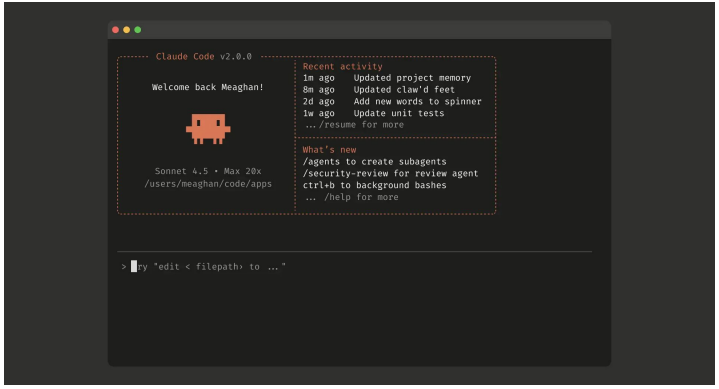
Agents

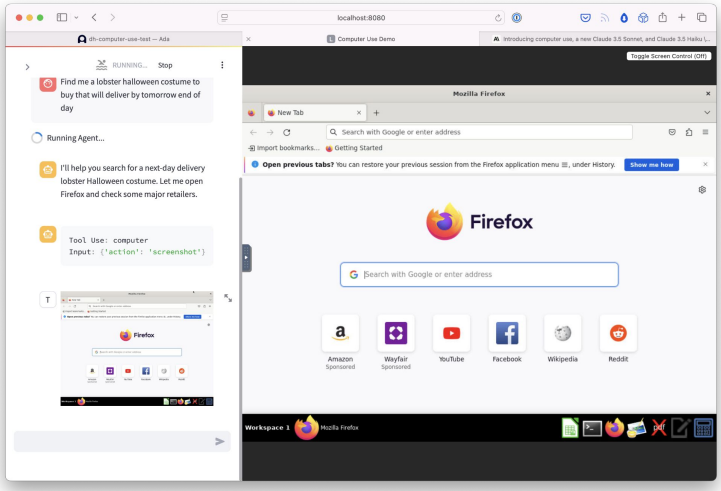
AI models interacting with the world

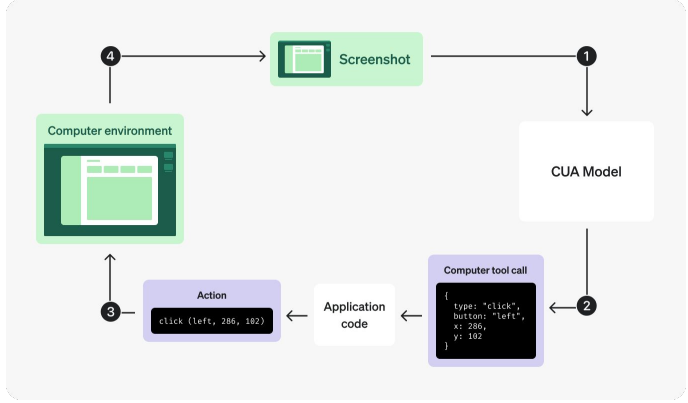
MVA class Training and Deploying Large-Scale Models 2026

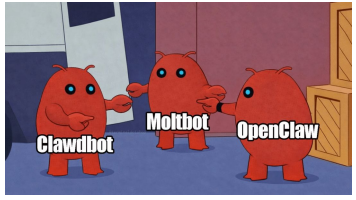
Michael Eickenberg


mod.rs
Implement the cleanup function for the tranport stack. Do not make the upgrade listeners optional.
CHAT
COMPOSER
Could you make it easier to switch certificates in the transport listeners?
I'll help modify the code to make certificate switching more flexible. The main changes will be to enhance the TlsAccept trait and modify how certificates are handled in the TlsSettings . Here are the key changes:


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
> Try "edit < filepath> to ..."


localhost:8080
Find me a lobster halloween costume to buy that will deliver by tomorrow end of day
Running Agent...
I'll help you search for a next-day delivery lobster Halloween costume. Let me open Firefox and check some major retailers.
Tool Use: computer
Input: {'action': 'screenshot'}
Mozilla Firefox
Firefox
Search with Google or enter address
Amazon
Wayfair
YouTube
Facebook
Wikipedia
Reddit
Workspace 1

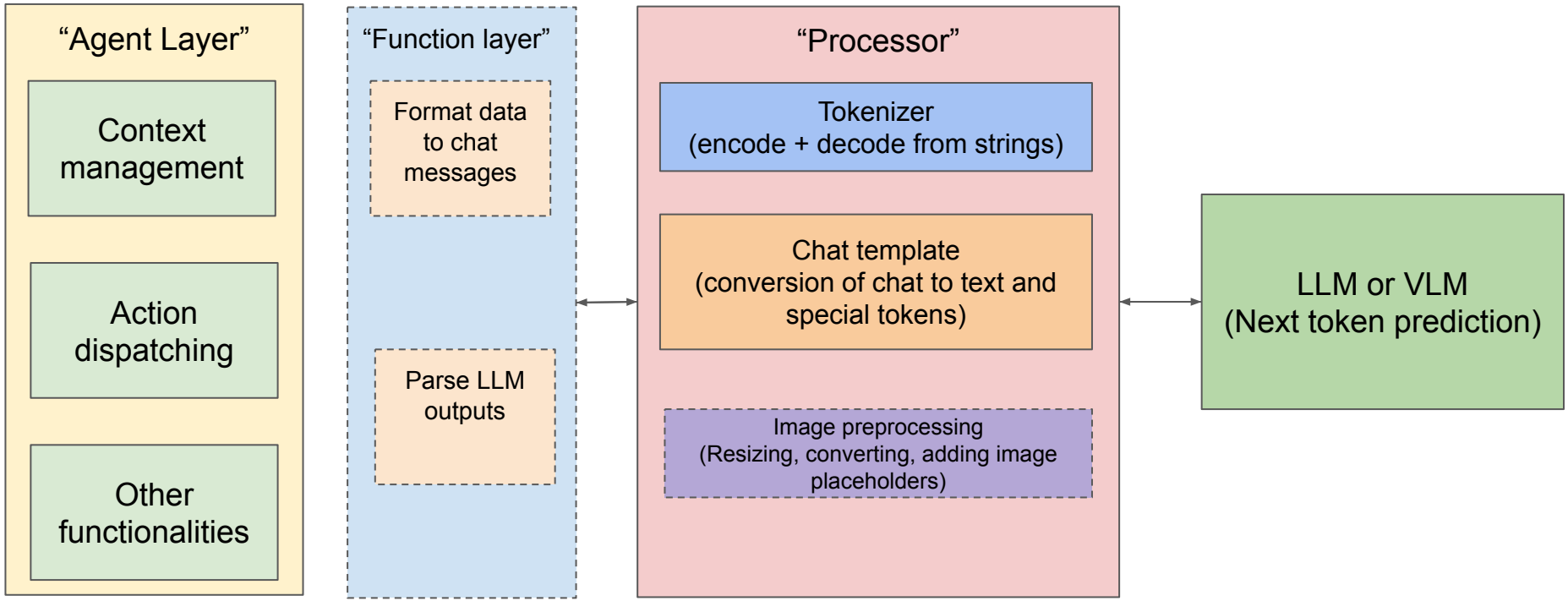

Screenshot
Computer environment
CUA Model
Computer tool call
{
type: "click",
button: "left",
x: 286,
y: 102
}
Application code
Action
click (left, 286, 102)
1
2
3
4


Clawdbot
Moltbot
OpenClaw

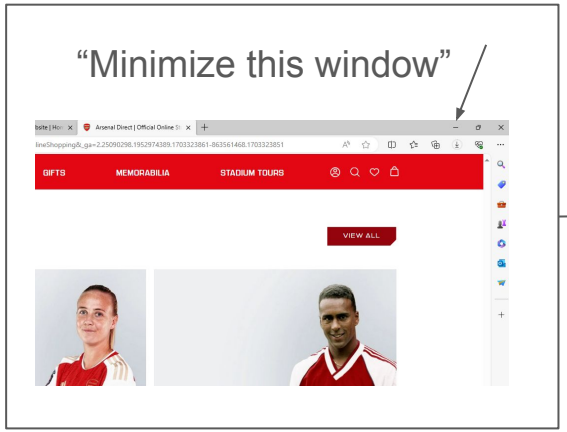
Contents

- 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



format

```
{“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”:“…”}
     {“type”: “text”,
      “text”: “Minimize this window”}
]}
```

Processor

VLM

Processor

parse

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

```
Location(x=0.888, y=0.025)
```

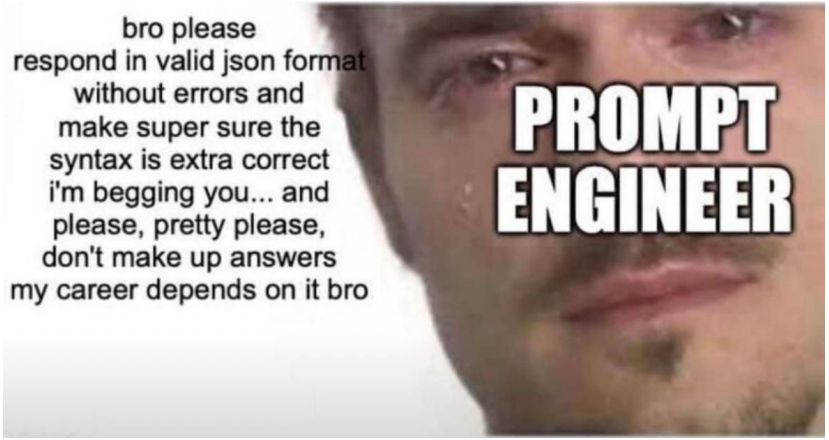
Interaction between LLM output and structure

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



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

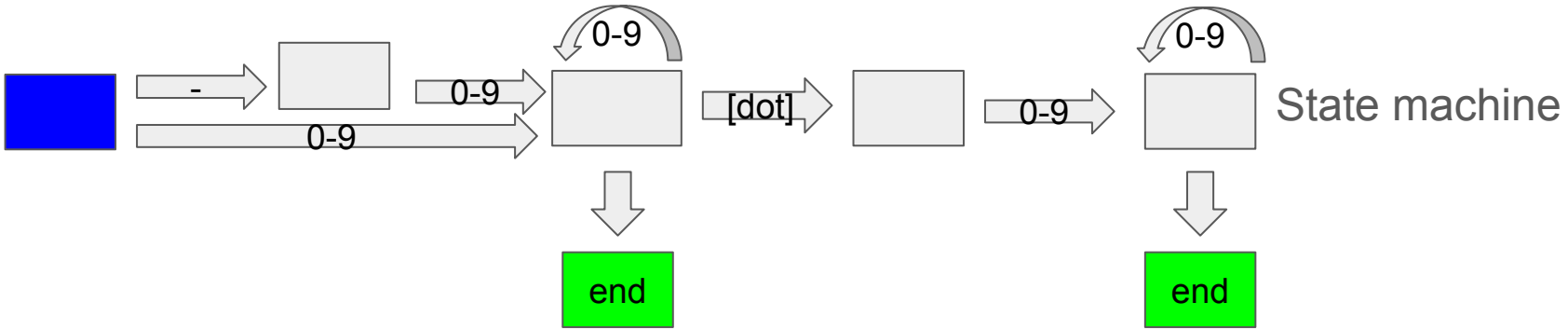
How do you “guide” a generation?

```
-?[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)

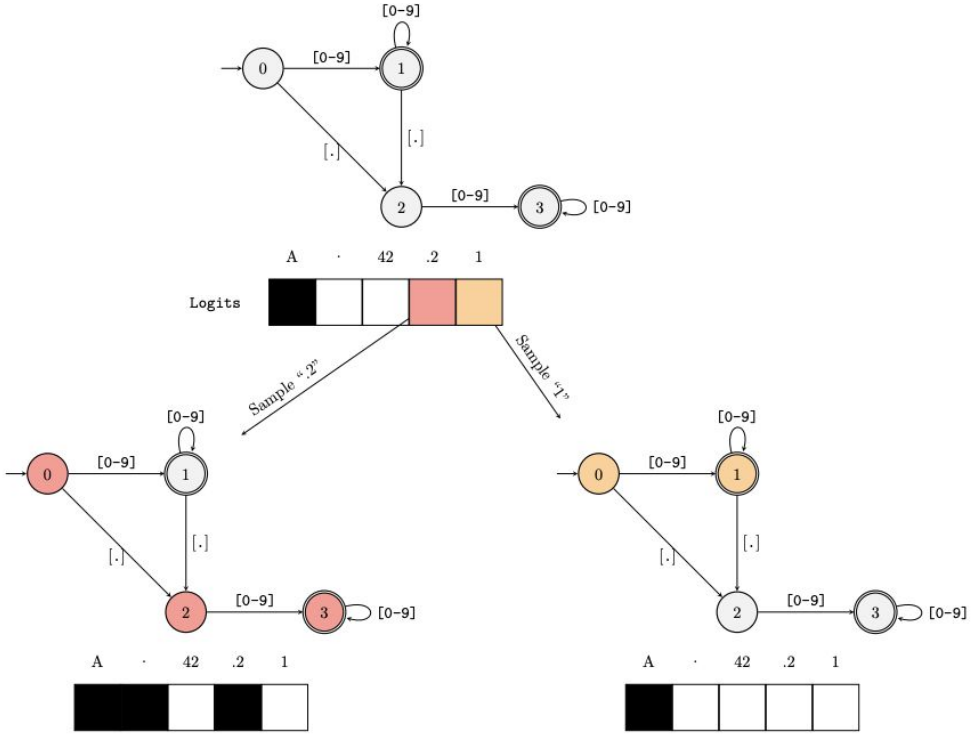


State machine

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
- …

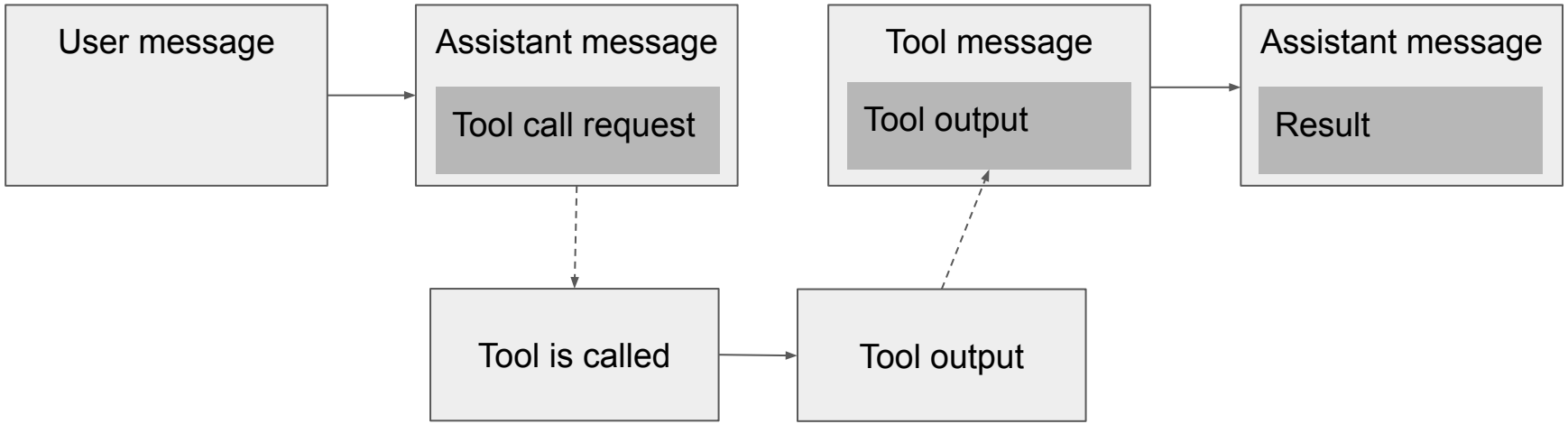
How does an LLM use tools?

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 LL

They can often understand in o ain this

- with human annotations to tool calls
- 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.

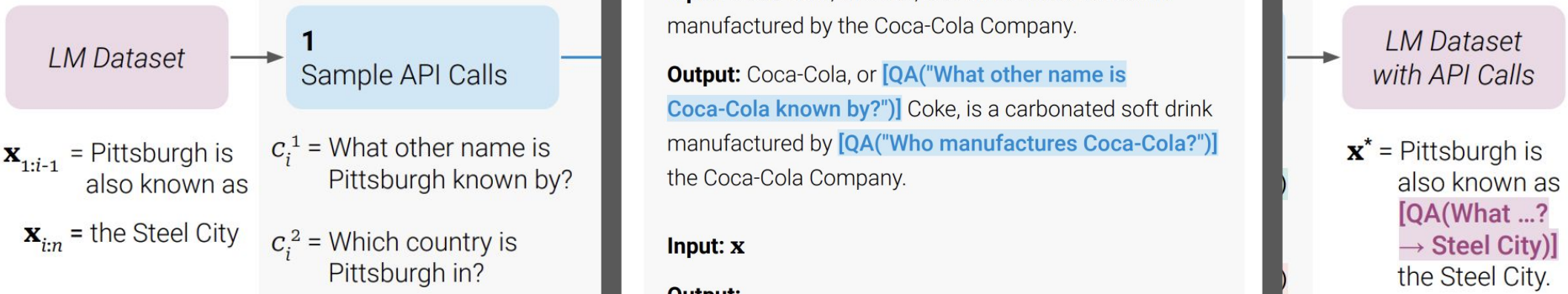
Output: Joe Biden was born in **[QA("Where was Joe Biden born?")]** Scranton, **[QA("In which state is Scranton?")]** Pennsylvania.

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: $\mathbf{x}$

Output:



"Toolformer": https://arxiv.org/ab

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?

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

Server to client:
I have the following tools!

Here are their names, parameters, and descriptions!

```
{
 "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.",
```

Example MCP calls

Client to server:
Please use `read_file` to get me the contents of `/home/user/project/main.py`

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

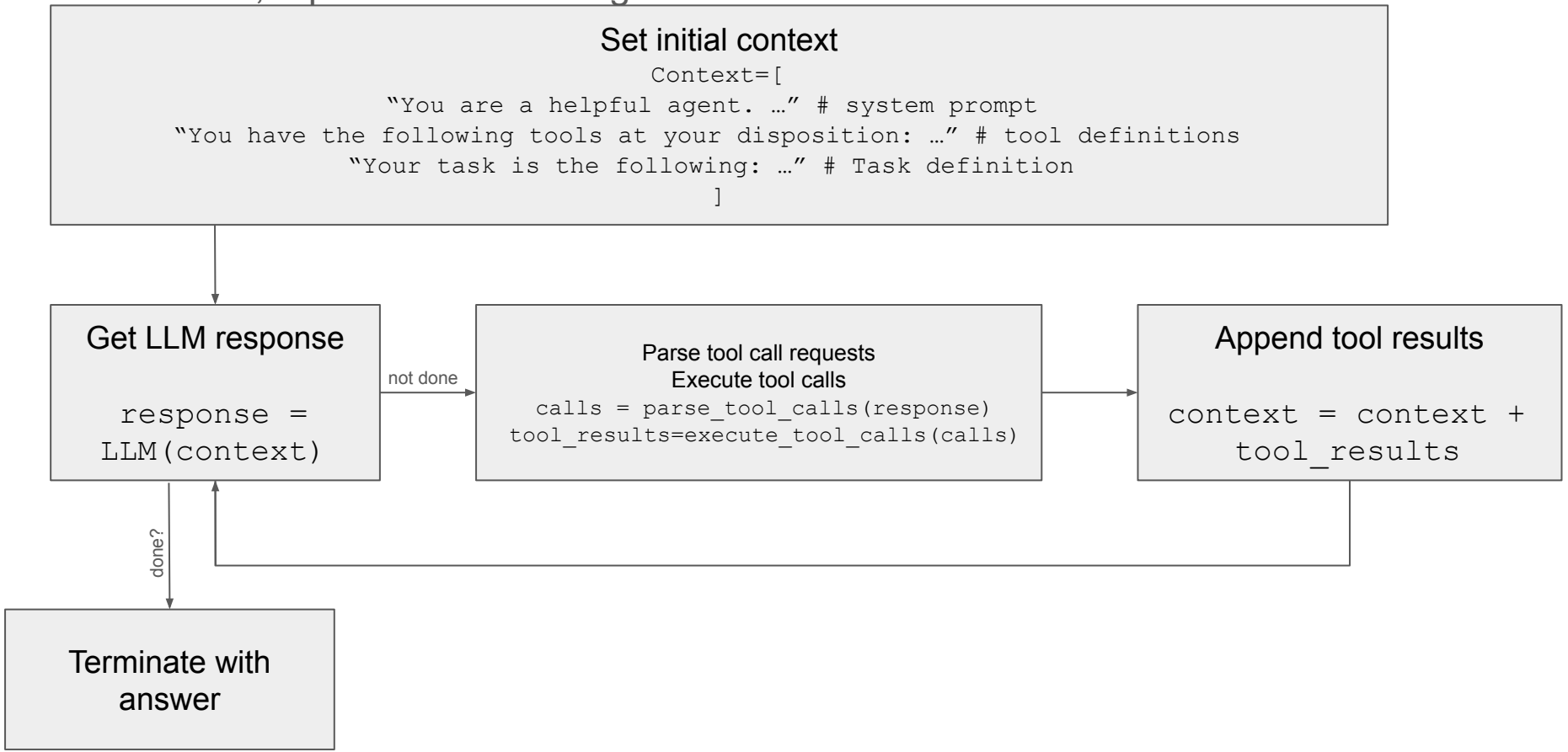
Server to client:
Voici

```
{
 "jsonrpc": "2.0",
 "id": 2,
 "result": {
   "content": [
     {
       "type": "text",
       "text": "def main():\n    print('hello
world')\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

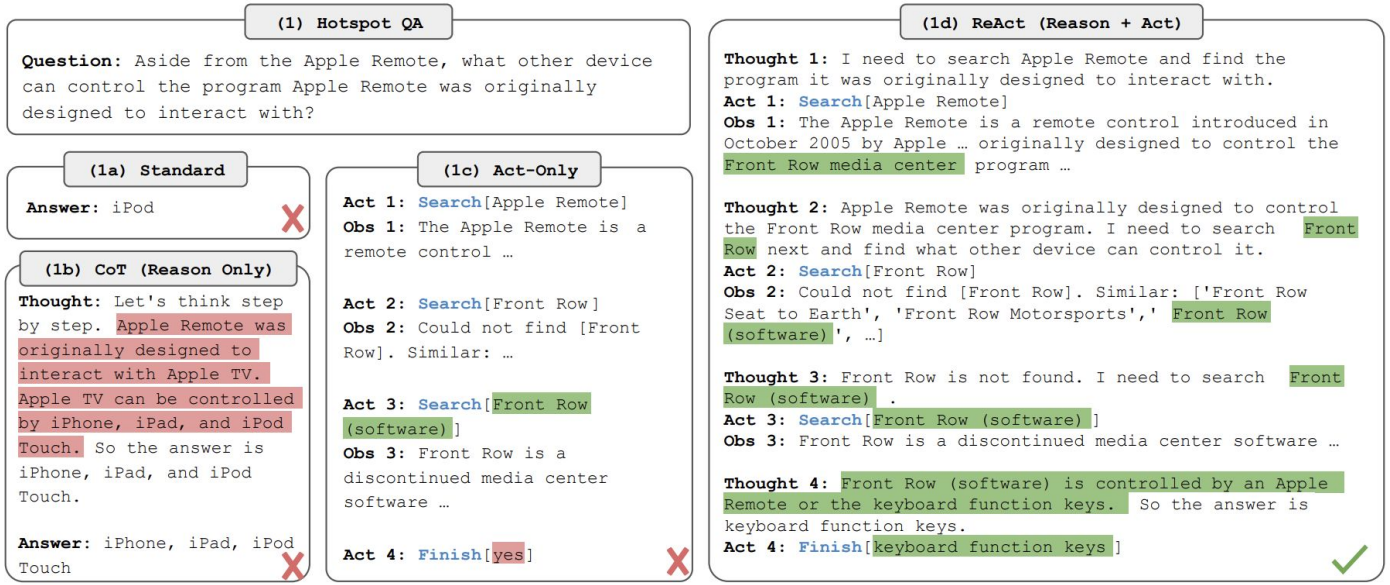


Agent loop pseudo code

```
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.")
```

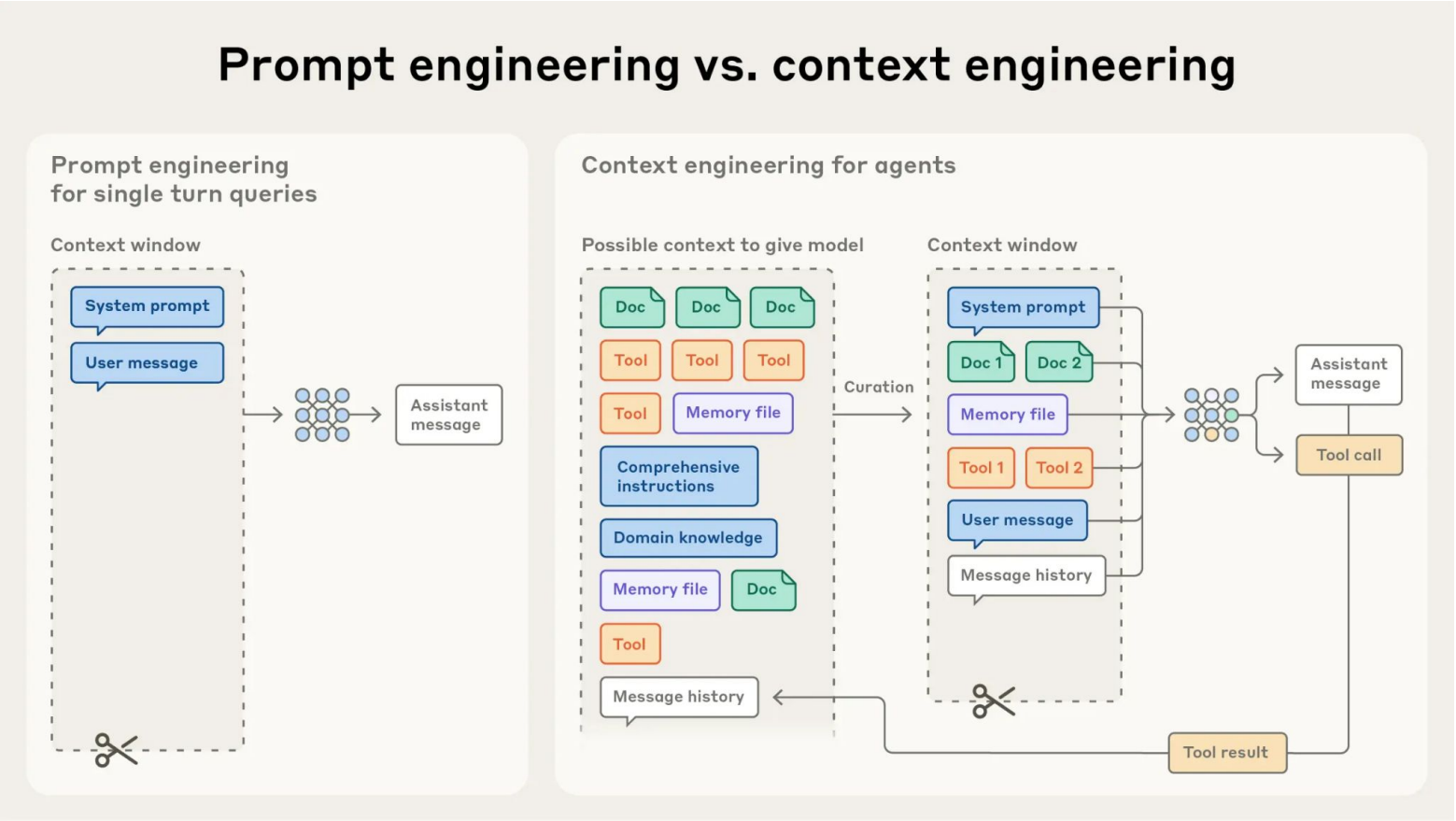
Hierarchical & Parallel

```
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
```

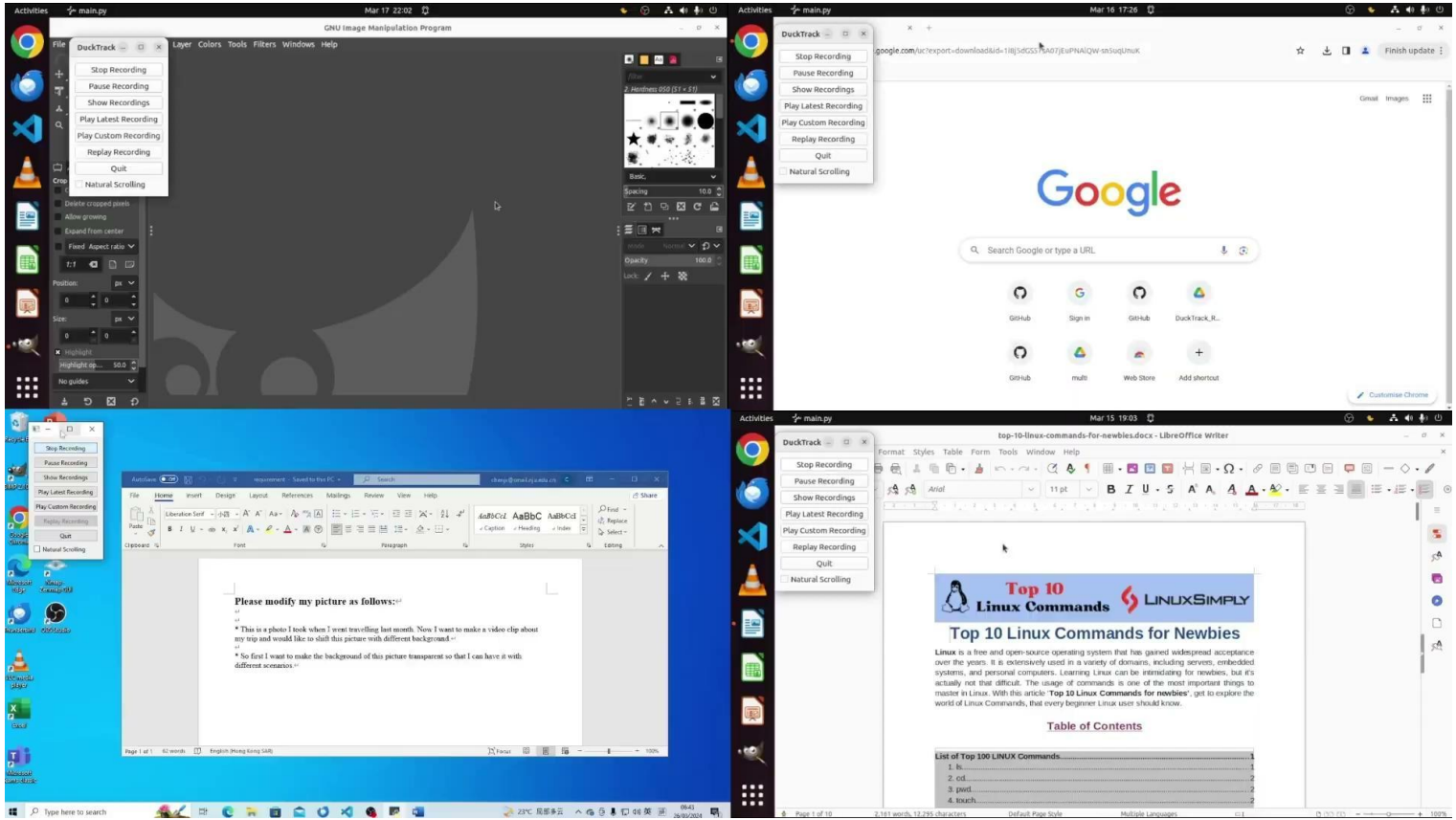
Pattern	Plan?	Number of agents	Best for
Flat ReAct	Implicit	1	Short Tasks
Plan + Loops	Explicit Plan Object	1 (+ possible replan)	Composite, structured predictable tasks
Hierarchical	Manager decomposes	Many	Large tasks, Parallelizable work

Memory and context management

see anthropic



Computer Use Agents



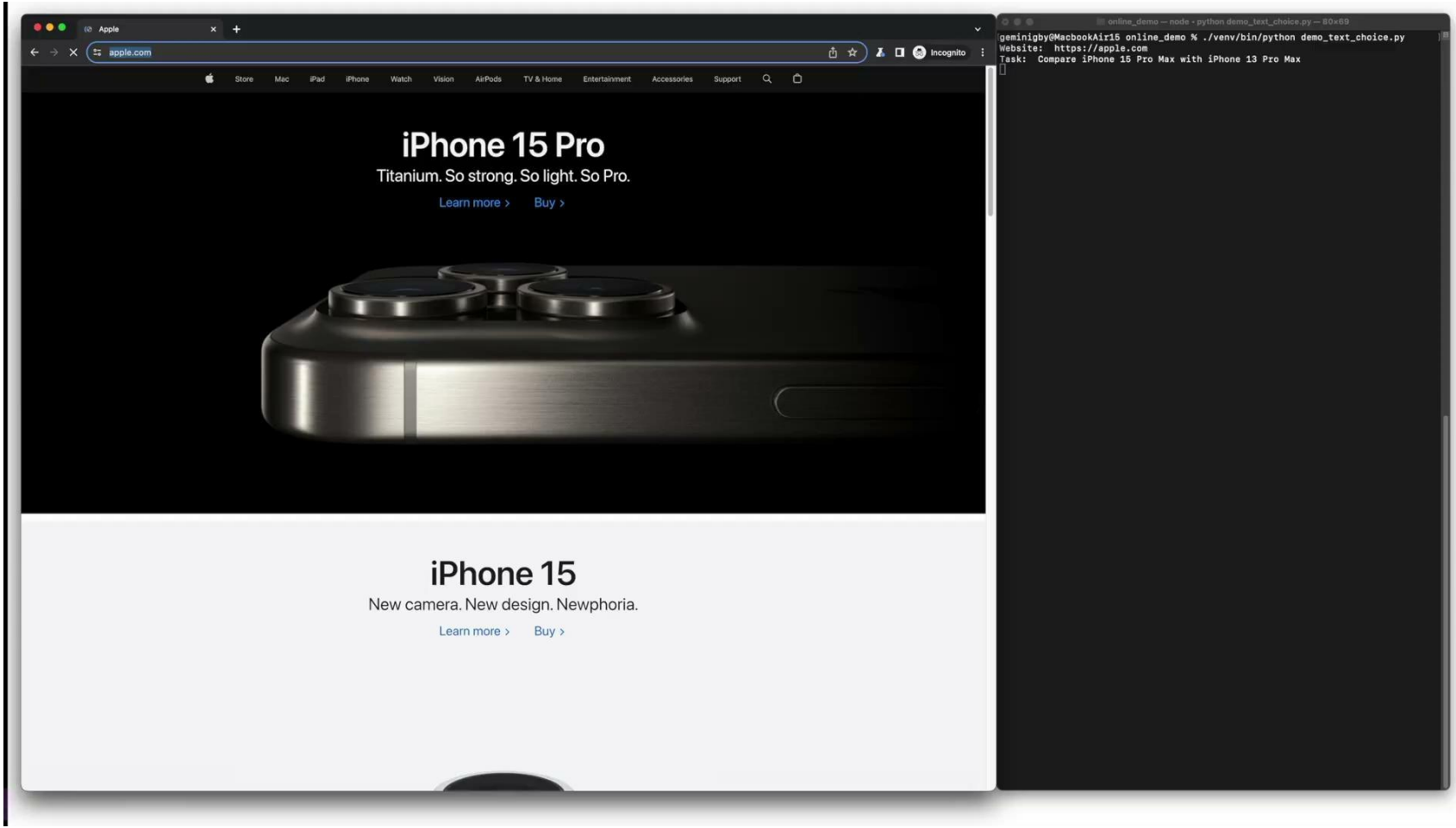
https://os-world.github.io/

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/

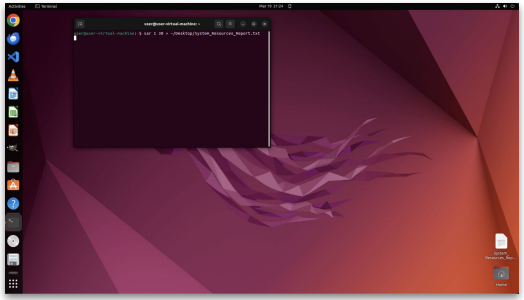
State representation problem

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

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

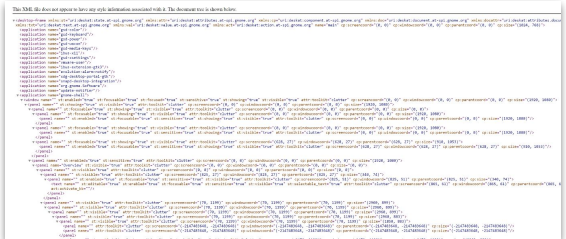
Screenshots

- 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



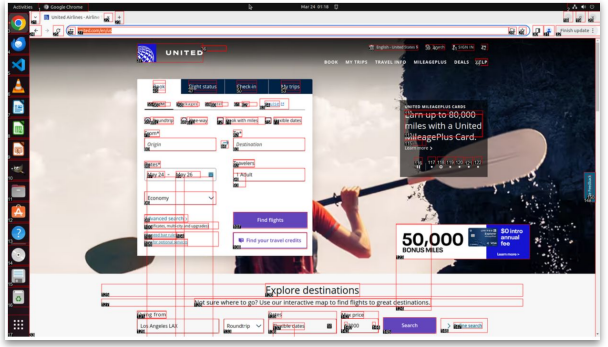
Tree representations

- 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



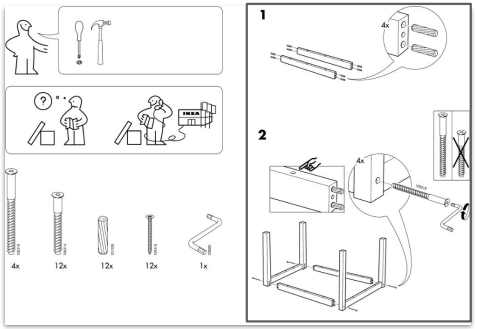
Computer Use Action Space

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()
```

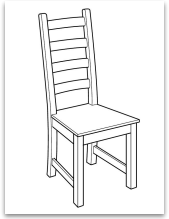
GUI Grounding

Grounding is the translation from instruction to action



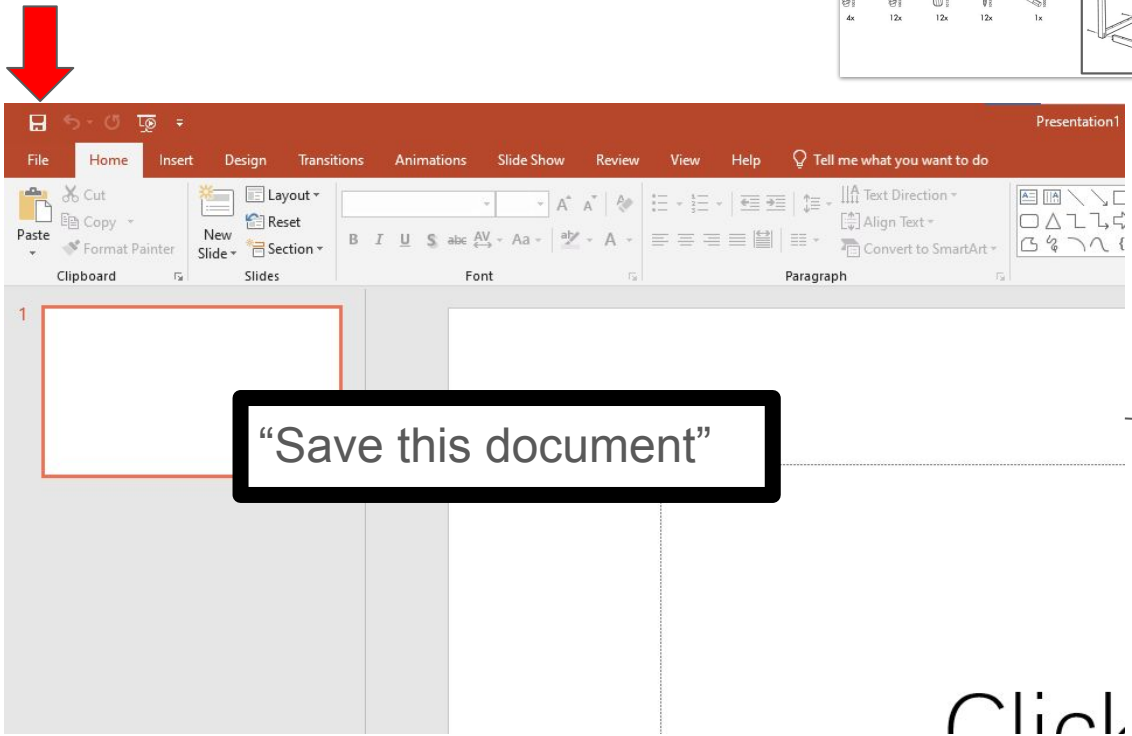
Step-by-step plans

Grounding



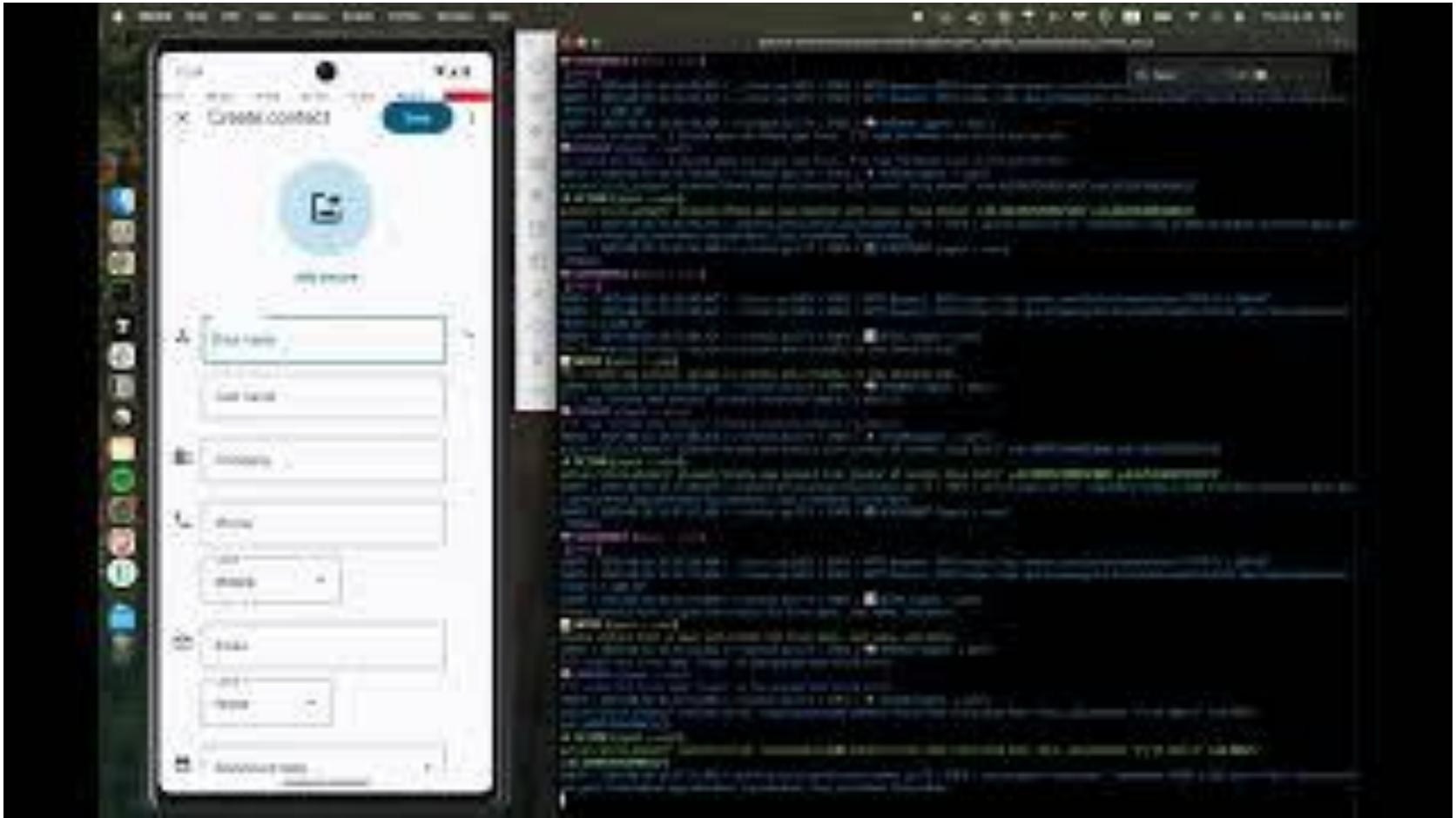
Assembled chair

see osworld




```
Click 0.020 0.020
```

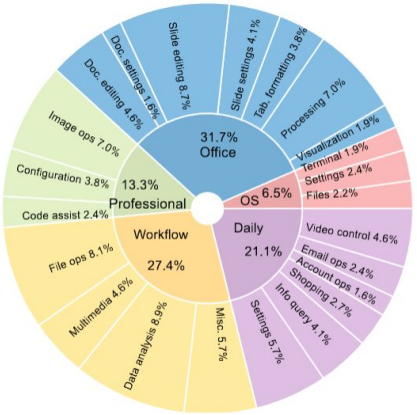
Mobile agent in action

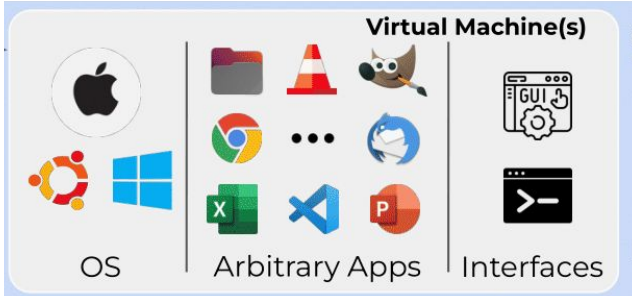


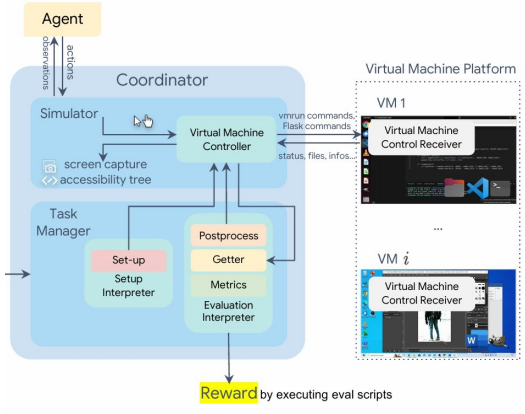
Benchmark: OS World

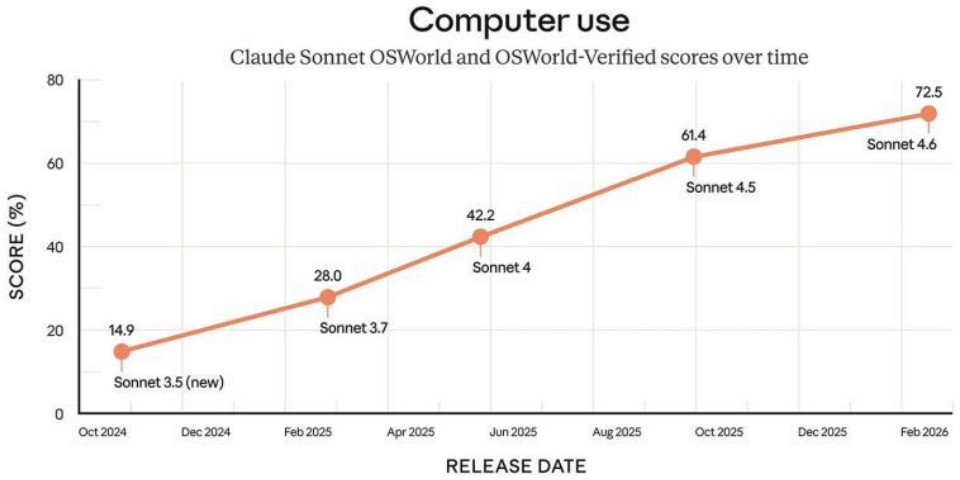
see https://os-world.github.io/

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









Coding agents

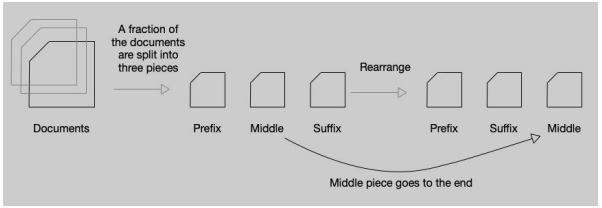
Cursor, Claude Code, Codex, …

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

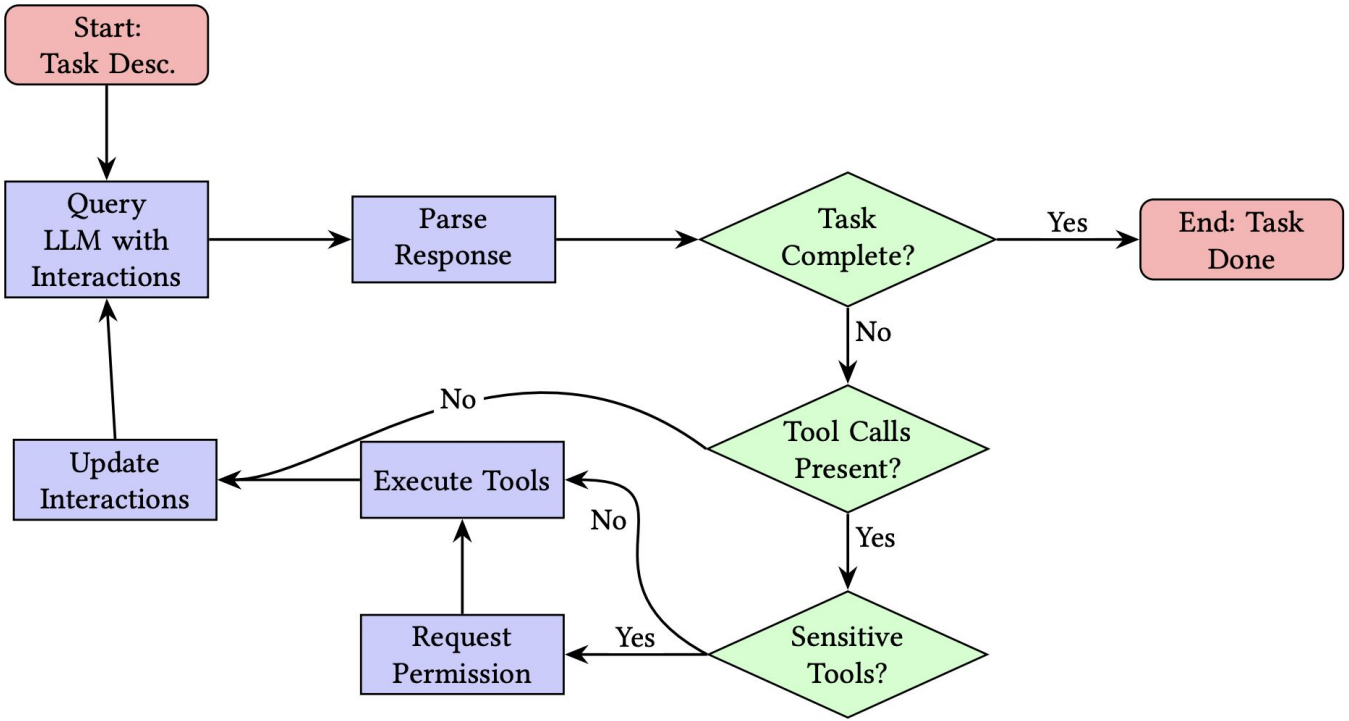


- Smart autocomplete (e.g. early github copilot)
  - Training via fill-in-the-middle
    Represent code chunk [A, b, C] as [<|PRE|>, A, <|SUF|>, C, <|MID|>, b]
    This allows model to propose autocomplete from context above and below

- 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



https://arxiv.org/abs/2601.18341

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

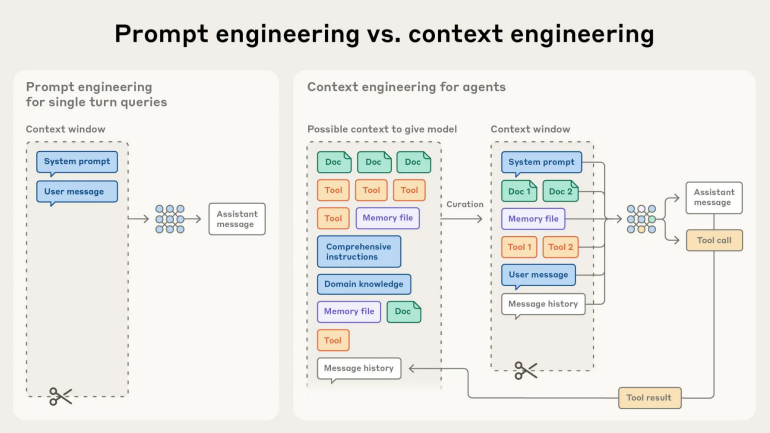
codebase_search - Semantic search that finds code by meaning, not exact text
grep -Exact symbol/string searches using ripgrep
read_file - Read files with optional line ranges and limits
glob_file_search -Find files matching glob patterns

✏️ 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

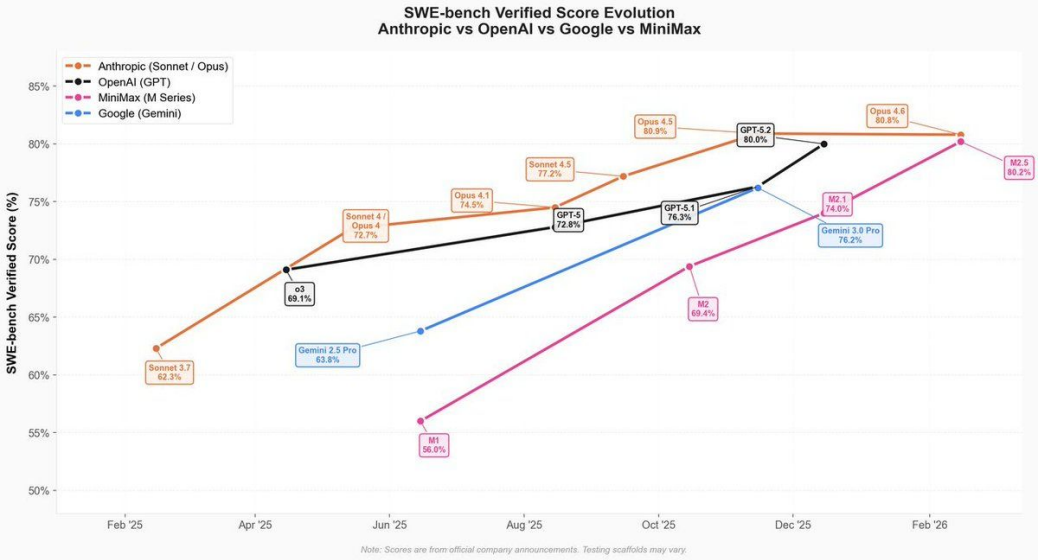
🧭 Navigation Tools
list_dir - List files and directories with optional filtering
web_search - Search for real-time information

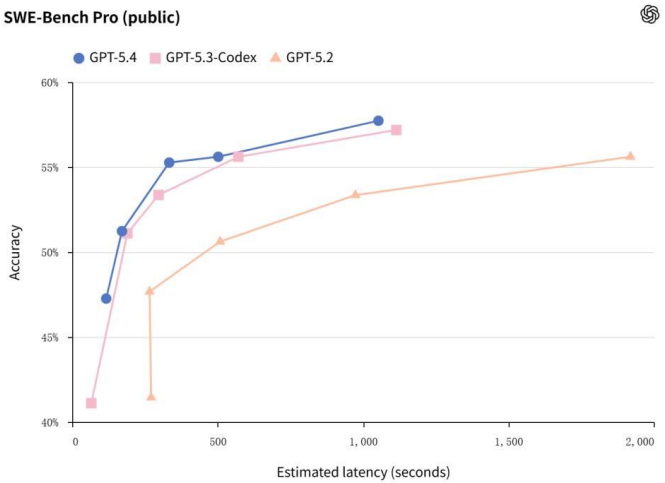


Benchmark: SWE-bench

Open-source Software Issues and Pull Requests

SWE-bench Pro: more long-horizon tasks





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