FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

NI-NLM – Lecture 7

RAG, tool calling, structured outputs, agents.

Zdeněk Kasner

31 Mar 2026

Retrieval-augmented generation

Problems with LLM knowledge

LLM knowledge is **static, lossy and incomplete**.

> **Question**
>
> When this can become a problem?

When we need the model to access:

- Up-to-date information (news, weather, timetables, ...).
- Domain-specific knowledge (internal company docs, law / medical / ...).

We may also want to access external knowledge **ourselves** to check the LLM's answers.

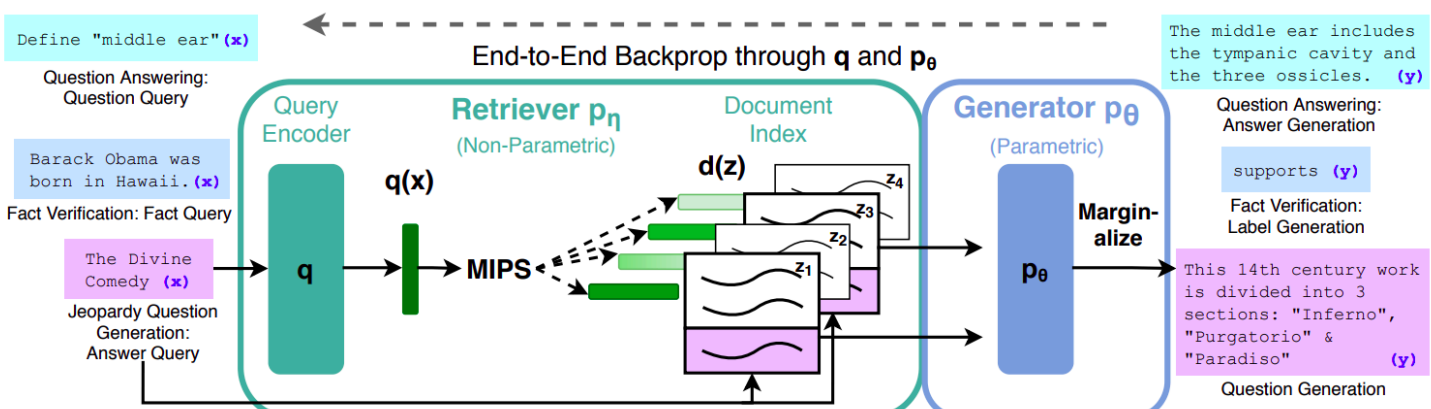
Retrieval-augmented generation (RAG)

source: Lewis et al. (2020)

Idea

Combine the LLM with a **retrieval system**: first retrieve relevant documents, then use them as context for generation → “retrieval augmented generation”

The idea was first introduced by Lewis et al. (2020):

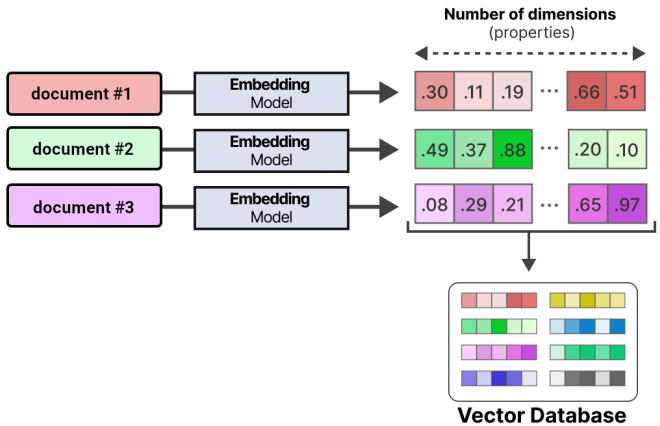


RAG pipeline

source: Maarten Grootendorst's blog

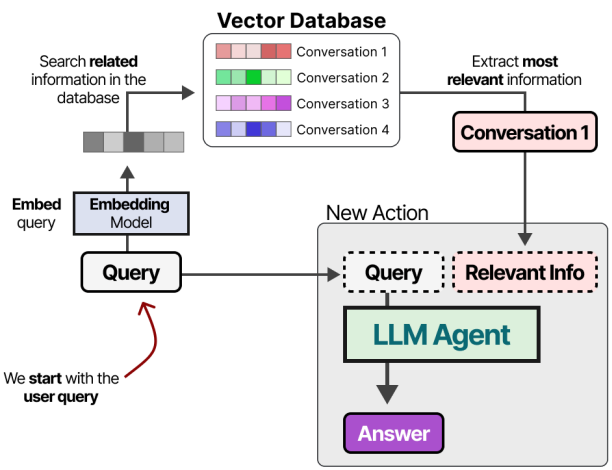
Phase #1: Indexing

1. Split the documents into chunks.
2. Compute their embeddings.
3. Store them in a vector database.



Phase #2: Querying

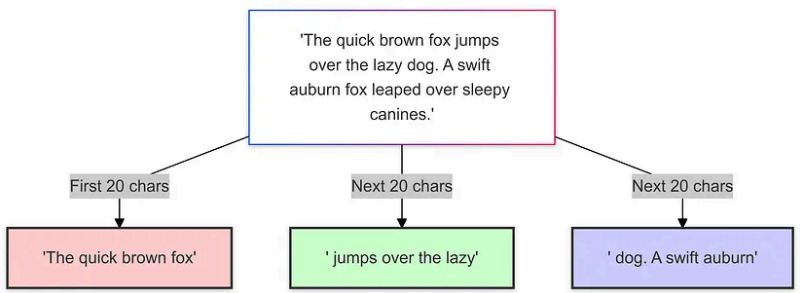
1. Embed the user prompt.
2. Retrieve relevant chunks.
3. Concatenate them with the prompt.
4. Generate an answer.



Indexing – Step 1: Document chunking

Raw documents need to be split into chunks before indexing:

- **Fixed-size chunks**: split every N tokens (simple, but can break context).
- **Semantic chunking**: split at sentence/ paragraph/section boundaries.
- **Sliding-window with overlaps**: fixed-size chunks with some overlap to preserve context.



source: Medium.com

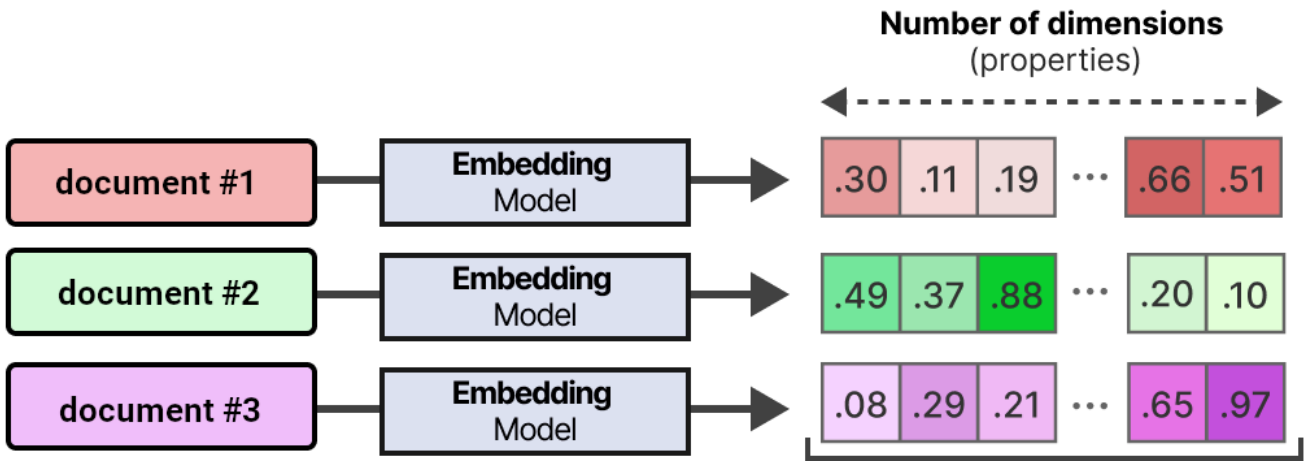
Good starting point: ~256–1024 tokens, with 10–20% overlap.

Too small → loss of context. Too large → diluted relevance and noise.

Indexing – Step 2: Embedding

source: Maarten Grootendorst's blog

Each chunk is converted to a dense vector (embedding).



Embedding models:

- encoder-based: finetuned BERT-like models, see the SentenceTransformers library.
- decoder-based: OpenAI's `text-embedding-3`, `e5`, `qwen3-embedding`, ...

Indexing – Step 2: Embedding

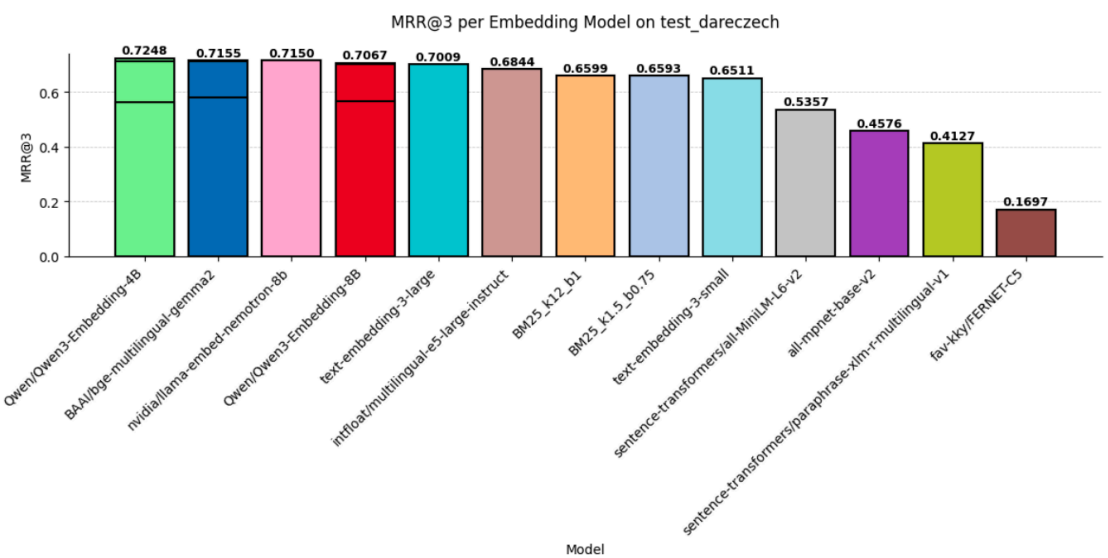
source: Sajdoková (2026)

> ⚠️ **Warning**
>
> Decoder embedding models perform much worse if not **prompted** properly.

```
# Each query must come with a one-sentence instruction that describes the task
task = 'Given a web search query, retrieve relevant passages that answer the query'
```

source: https://huggingface.co/Qwen/Qwen3-Embedding-8B

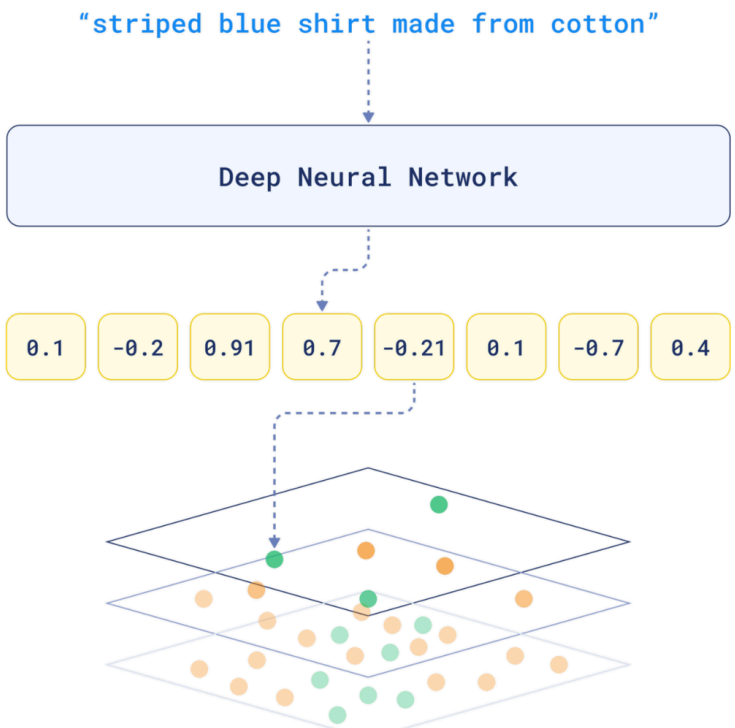
Note there is no actual retrieval → we just **“prime” the model** to build a good representation of the query.

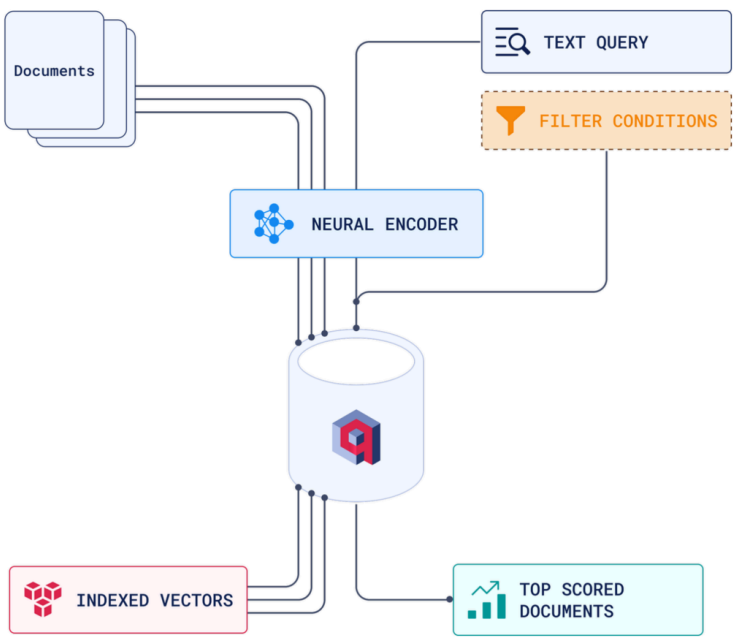


Indexing – Step 3: Vector database

source: https://qdrant.tech/articles/what-is-a-vector-database/

The created embeddings are stored in a **vector database** (Qdrant, Chroma, FAISS, ...):

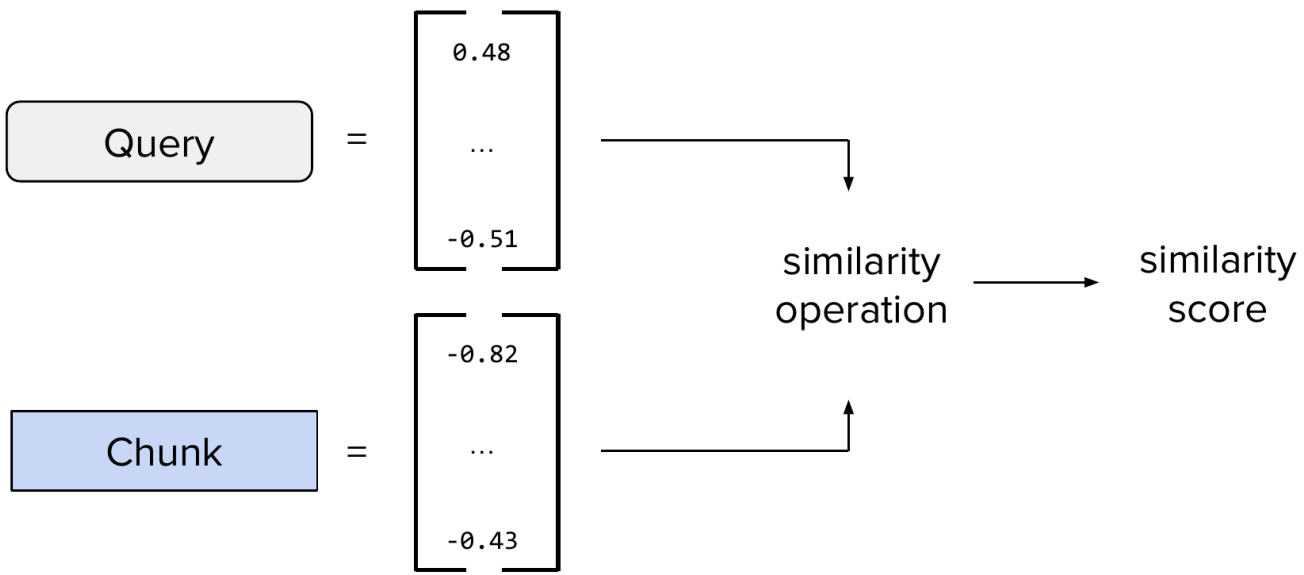




Querying – Steps 1-2: Retrieval

source: CME295

At **query time**, we embed the user query and find the most similar chunks:



Similarity is typically measured using **cosine similarity** or **dot product** between the query and document embeddings.

Querying – Steps 3-4: Generation

The retrieved chunks are inserted into the prompt as **context**:

```
System: You are a helpful assistant. Answer based on the provided context.

Context:
[Chunk 1]: "The Transformer architecture was introduced by Vaswani et al. in 2017..."
[Chunk 2]: "Self-attention allows the model to weigh different positions..."

User: What is the key innovation of the Transformer?
```

Question

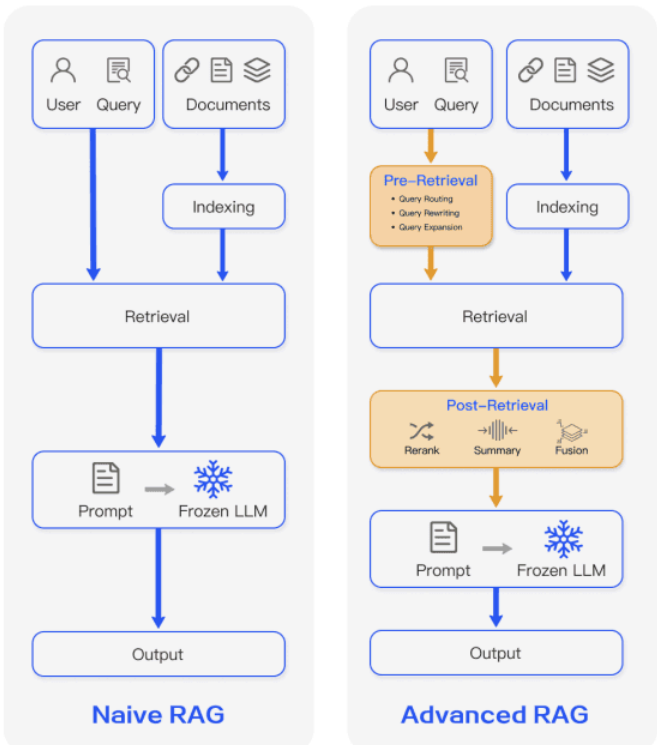
Can a RAG-based answer be *worse* than without RAG?

Towards better RAG

source: Gao et al. (2024)

The basic pipeline can be improved at every step:

- We can apply **query rewriting** to rephrase the query for better retrieval.
- We can use **hybrid search** – combine dense retrieval with keyword-based search (e.g. BM25).
- We can **rerank** the retrieved results e.g. using a cross-encoder (heavier, but more precise).
- We can retrieve the documents **iteratively** (→ heading towards LLM agents).



RAG in practice

RAG is now a standard for building knowledge-intensive LLM applications (in combination with LLM agents):

- **Company internal knowledge**: “talk to your knowledge base”.
- **Customer support bots** with access to the internal knowledge bases.
- **Code assistants** that retrieve from documentation and codebase.
- **Legal / medical AI**: retrieve from domain-specific corpora.

There are now many frameworks that facilitate RAG:

(...and many more).

Tool calling

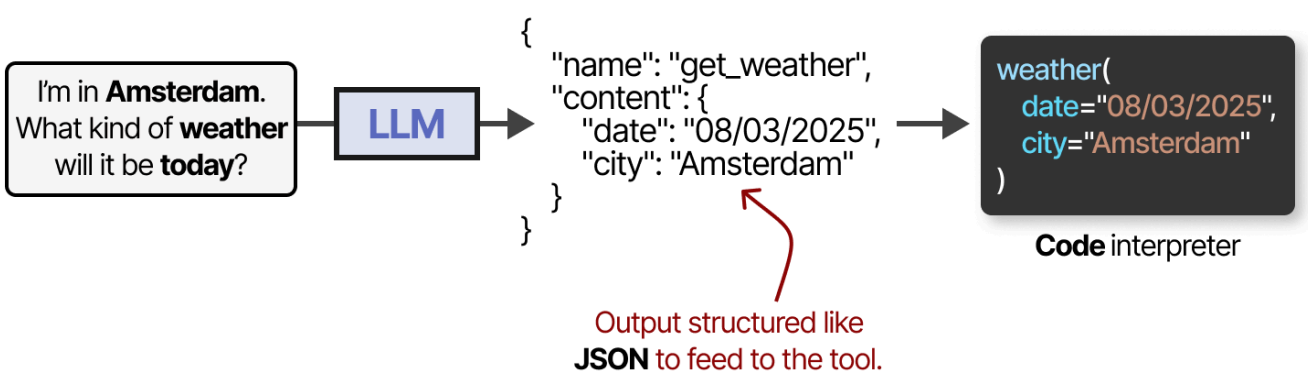
Tool use: the basic idea

source: Maarten Grootendorst's blog

RAG works as long as the information can be fetched based on the user query.

But what if LLMs need to access external resources **during generation**?

> **Idea**
>
> We give the LLM the capability to generate tool calls.



Toolformer

source: Schick et al. (2023)

Toolformer (Schick et al. 2023): one of the most influential **LLM tool-calling** papers.

- An LLM finetuned on a corpus containing **examples of tool calls**.
- The corpus built in a self-supervised way:
  - Another (few-shot prompted) LLM suggests tool calls.
  - The suggestions that reduce LM loss are inserted in the corpus.
- **Tools:** QA system, Wikipedia search, calculator, calendar, MT system

The New England Journal of Medicine is a registered trademark of [QA("Who is the publisher of The New England Journal of Medicine?") → Massachusetts Medical Society] the MMS.

Out of 1400 participants, 400 (or [Calculator(400 / 1400) → 0.29] 29%) passed the test.

The name derives from "la tortuga", the Spanish word for [MT("tortuga") → turtle] turtle.

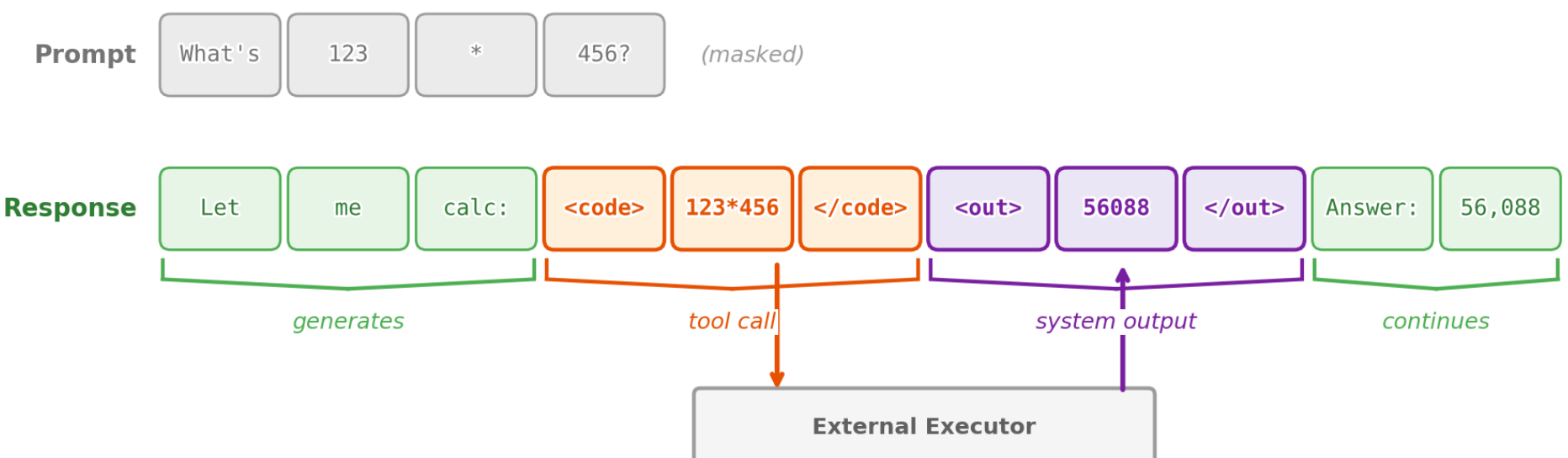
The Brown Act is California's law [WikiSearch("Brown Act") → The Ralph M. Brown Act is an act of the California State Legislature that guarantees the public's right to attend and participate in meetings of local legislative bodies.] that requires legislative bodies, like city councils, to hold their meetings open to the public.

How tool calling works

source: https://rlhfbook.com/c/13-tools

Once the model generates the tool call:

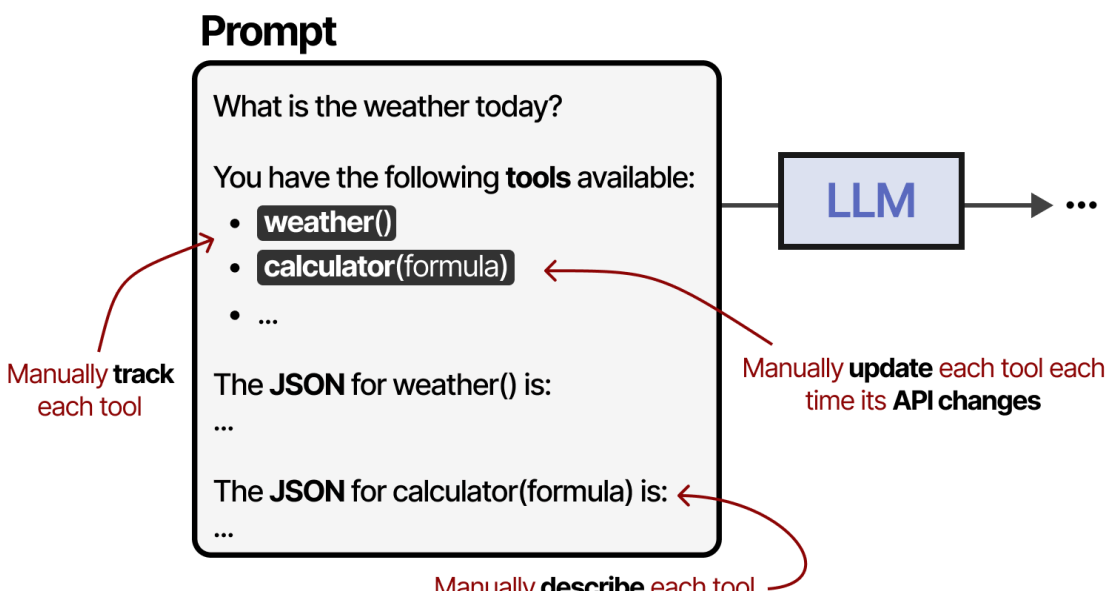
1. the **output is evaluated** using an external tool,
2. the **result is appended** to the generated text.



How to scale tool calling

source: Maarten Grootendorst's blog

There are many tools that a model can potentially call → describing the interface of each of them can be cumbersome.



Welcome the MCP

source: https://modelcontextprotocol.io/docs/learn/server-concepts

Model Context Protocol (MCP): standardized way to access tools based on JSON.

Example tool specification

```
{
  "name": "get_weather",
  "description": "Get current weather for a city",
  "inputSchema": {
    "type": "object",
    "properties": {
      "city": {"type": "string", "description": "City name"}},
    "required": ["city"]}
}
```

Example: what the model returns

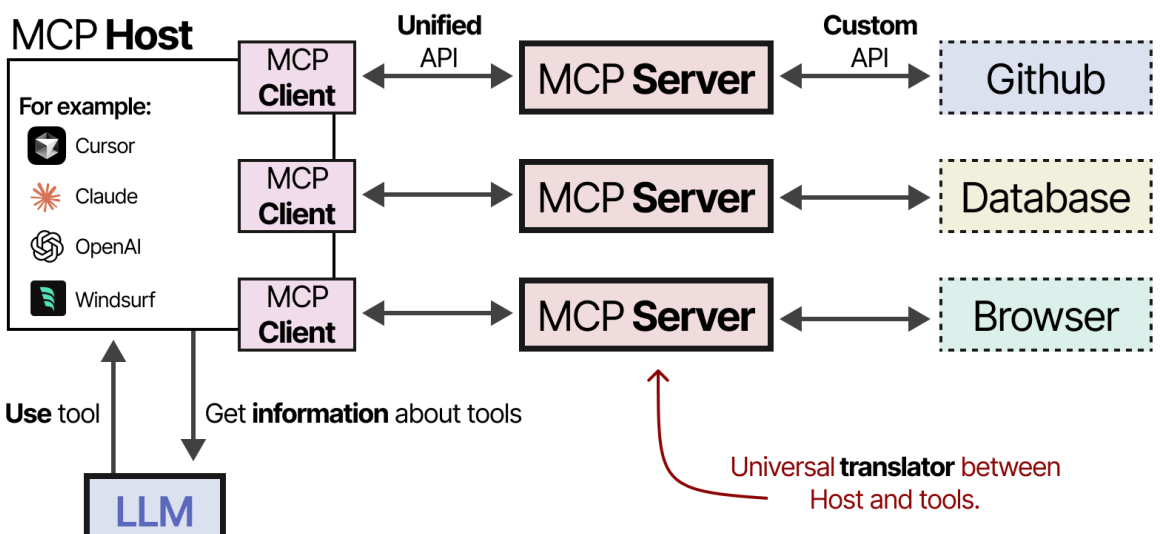
```
{"name": "get_weather", "arguments": {"city": "Prague"}}
```

MCP architecture

source: Maarten Grootendorst's blog

MCP describes not just the JSON format, but the entire tool-calling architecture:

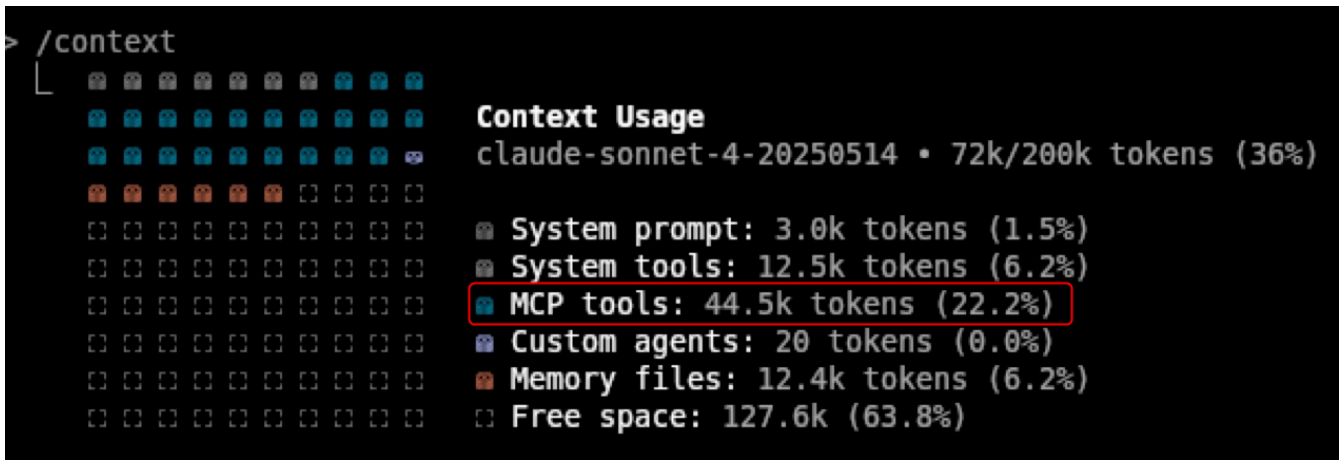
- **MCP client**: the application managing the LLM (VSCode, Claude Code, ...)
- **MCP server**: the application that can be managed with tools (Github, database, ...)



Issues with MCP

source: https://demiliani.com/2025/09/04/model-context-protocol-and-the-too-many-tools-problem/

MCP tool descriptions can use up a lot of input tokens:



→ This is currently an open issue.

Can be mitigated with dynamic content discovery or replacing MCP with skills (`.md` files that include description of the tool's CLI) instead.

Structured outputs

Enforcing output format

There is no guarantee whatsoever on how the LLM output will look like.

Expected output:

```
{
  "companies": [
    {"name": "Apple", "revenue": 365},
    {"name": "Microsoft", "revenue": 168}
  ]
}
```

Actual output:

````
Sure! Here's the information you requested:
```json
{
  "companies": [
    {"name": "Apple", "revenue": 365},
...
```

I hope this helps!
````

Question

How might we enforce that the output will look exactly as required?

Basic ways to control LLM output

Solution 1: Answer prefix

Prefix model's answer to guide its response.

Prompt:

```
{
  "user" : "Generate a single-paragraph weather forecast based on the data: [data]"
  "assistant": "Sure, here is the required forecast: \n\""
}
```

→ Very simple cases only.

Basic ways to control LLM output

Solution 2: Prompt specifications + few-shot prompting

Describe the desired output format **very precisely** (+ include examples).

Prompt:

```
CRITICAL: You MUST respond in EXACTLY the
specified format with no additional text,
explanations, or commentary. Do NOT include any
introductory phrases, reasoning, or extra content.
Any deviation from this exact format will cause
system failure.
```

source: @ChatGPTapp

→ But how much pleading is enough? 🤔

Basic ways to control LLM output

Solution 3: Prompt chaining

1. Ask the model to **extract relevant facts:**

```
List all companies mentioned in the text along with their reported revenue and year.
```

2. Ask the model to **format those facts**:

```
Format the extracted information as a JSON array with fields: name, revenue, and year.
```

3. Ask the model to **validate the output**:

```
Check that all values in the JSON are correctly formatted.
```

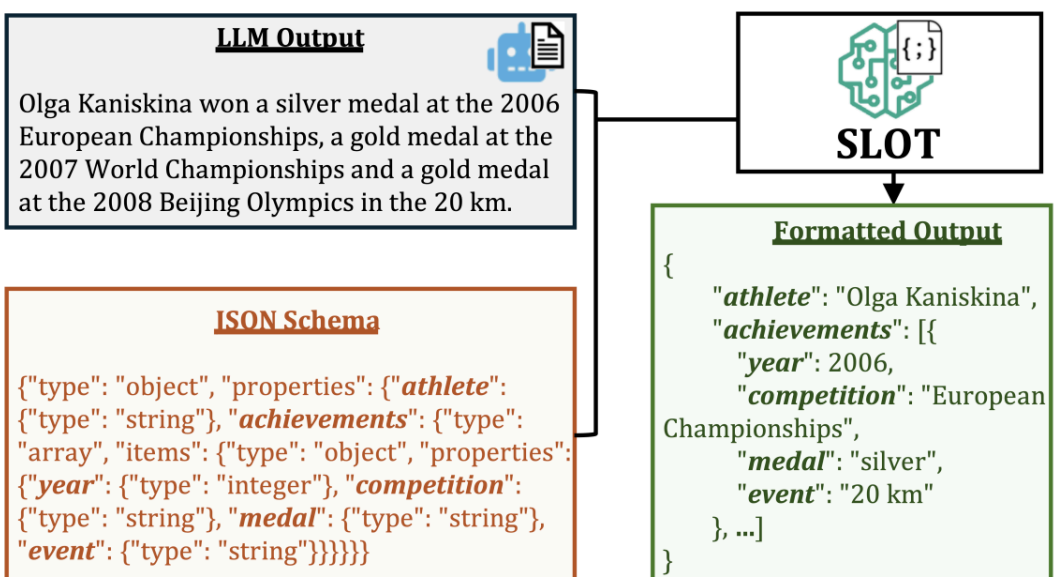
→ More robust, but can run into the same issues as previous approaches.

(Less) basic ways to control LLM output

source: Wang et al. (2025)

Solution 4: Model finetuning

Finetune a small model to generate only the output in the desired format → use it to post-process the output from the large LLM.



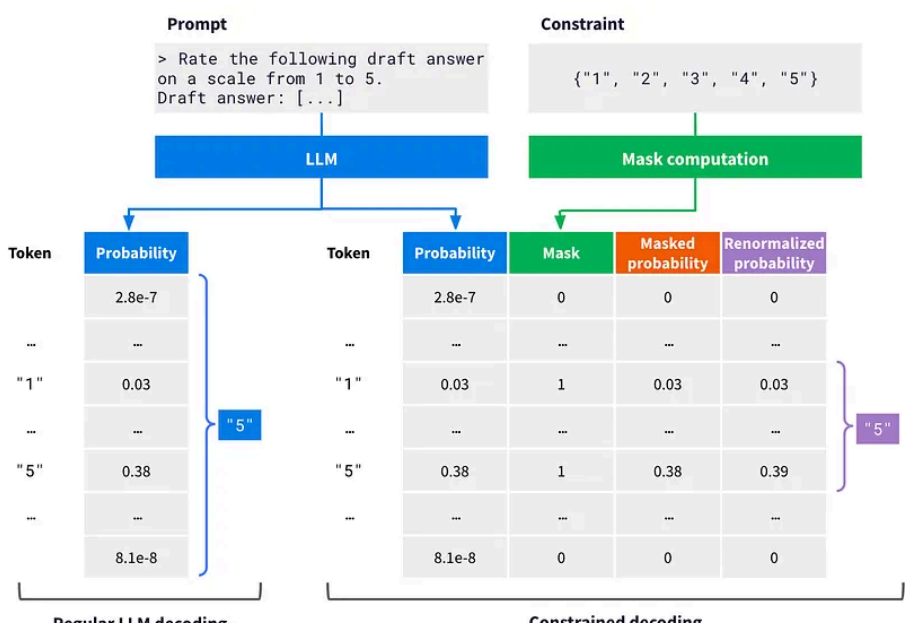
Constrained decoding

source: dataiku.com

> **Idea**
>
> Instead of hoping the model produces valid output, we **force it** during decoding.

How? **Token masking!**

→ At each step, we allow **only the tokens that are valid** according to our constraints.



How to specify the constraints?

Question

Imagine we want to force the model output to be one of the following:

1. A valid **number** or **e-mail address**.
2. A **JSON object** with specific fields.
3. A **text string** with placeholders (numbers, multiple choice, etc.) infilled.

How to specify each of these options?

How to specify the constraints?

Number / email:

Regular expressions

```
# positive decimal
r"(^[1-9][0-9]*$)"

# e-mail address (simple version)
r"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$)"
```

JSON object:

Pydantic models

```
class Company(BaseModel):
    name: str
    revenue: float
    year: int = 2021

class Response(BaseModel):
    companies: List[Company]
    total_count: int
```

Text string:

Generative constructs

```
"The answer is "
+
GENERATE(regex=r"\d+")
+
" with confidence "
+
GENERATE(choices=["high","med","low"])
```

JSON schema vs Pydantic models

Pydantic models

```python
class Company(BaseModel):
    name: str
    revenue: float
    year: int = 2021

class Response(BaseModel):
    companies: List[Company]
    total_count: int
```

JSON schema (equivalent)

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Response",
  "type": "object",
  "properties": {
    "companies": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": { "type": "string" },
          "revenue": { "type": "number" },
          "year": {
            "type": "integer",
            "default": 2021
          }
        },
        "required": ["name", "revenue", "year"]
      }
    },
    "total_count": { "type": "integer" }
  },
  "required": ["companies", "total_count"]
}
```

How to intervene on the generation process?

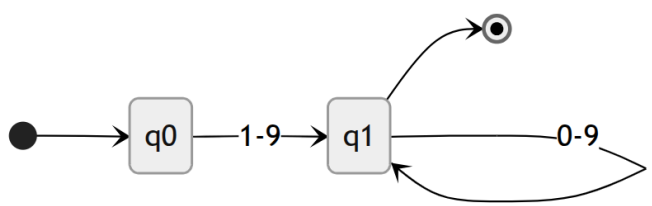
We can convert any regular expression to a **finite-state machine (FSM)** and use it to determine which tokens are valid at each step.

Regular expression

(positive integer)

```
r"[1-9][0-9]*"
```

Finite-state machine



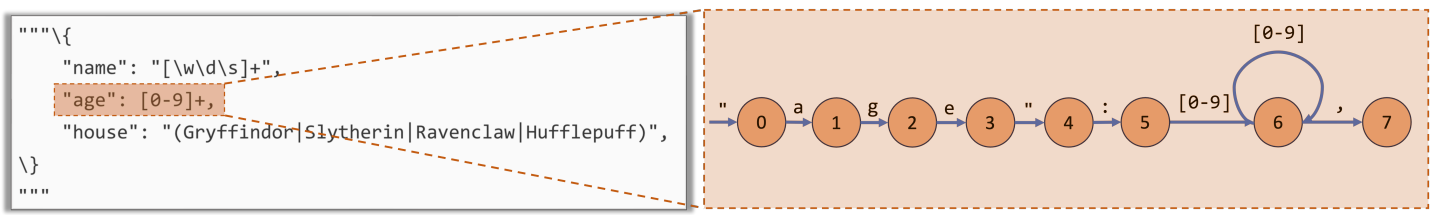
- During decoding, we move along the edges in the FSM and keep track of the state.
- **Allowed tokens** = outgoing edges from the state we are currently in.

How to intervene on the generation process?

source: https://lmsys.org

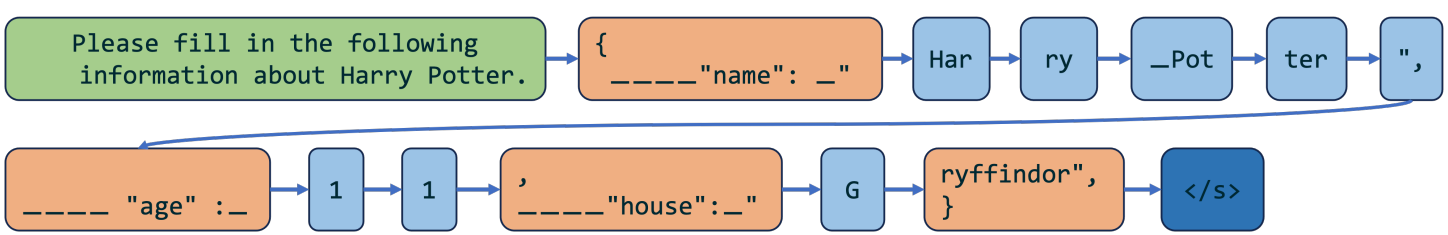
What about **JSON schemas** or **interleaved generative constructs**?

→ We can convert them to a regular expression (and that to a FSM).



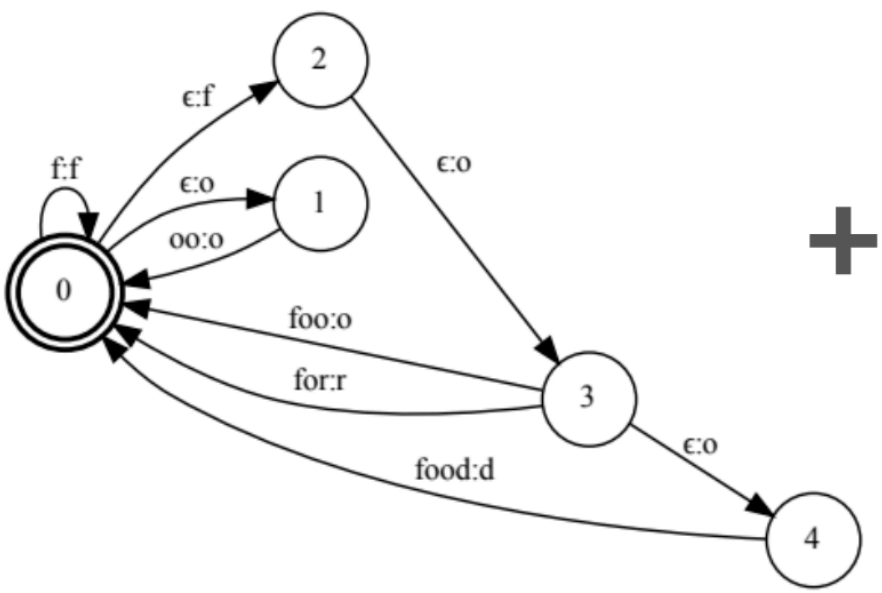
Regular Expression

Finite State Machine



Can we go deeper?

For arbitrarily nested structures (JSON with arbitrary nesting, HTML, code), we need **context-free grammars (CFGs)** and **pushdown automata**:



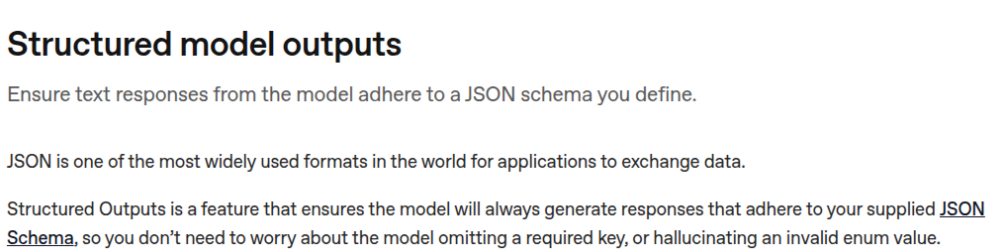
+

Structured outputs & libraries

Libraries for **local open models** support JSON mode, enums, regexes, CFGs, ...:

Level	Frameworks	Note
API frameworks	vLLM, llama.cpp, SGLang, Ollama	API – use `response_format`
Libraries	Outlines, XGrammar, Guidance, BAML	For custom integration
Low-level	HF Transformers `LogitsProcessor`	Low-level implementation

Commercial providers (OpenAI, Gemini, Claude) support primarily JSON outputs:



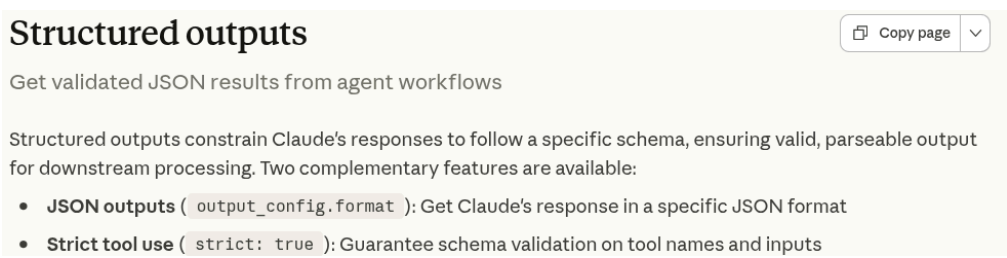
Structured model outputs

Ensure text responses from the model adhere to a JSON schema you define.

JSON is one of the most widely used formats in the world for applications to exchange data.

Structured Outputs is a feature that ensures the model will always generate responses that adhere to your supplied JSON Schema, so you don't need to worry about the model omitting a required key, or hallucinating an invalid enum value.

source: OpenAI docs



Structured outputs

Copy page

Get validated JSON results from agent workflows

Structured outputs constrain Claude's responses to follow a specific schema, ensuring valid, parseable output for downstream processing. Two complementary features are available:

- **JSON outputs** (`output_config.format`): Get Claude's response in a specific JSON format
- **Strict tool use** (`strict: true`): Guarantee schema validation on tool names and inputs

source: Claude docs

Why constrained decoding can be tricky

source: Cooper (2024)

> *Most constrained decoding pitfalls are due to misalignment between what the model **wants** to output and what it is **forced** to output.*
>
> — *aidancooper.co.uk*

Main issues:

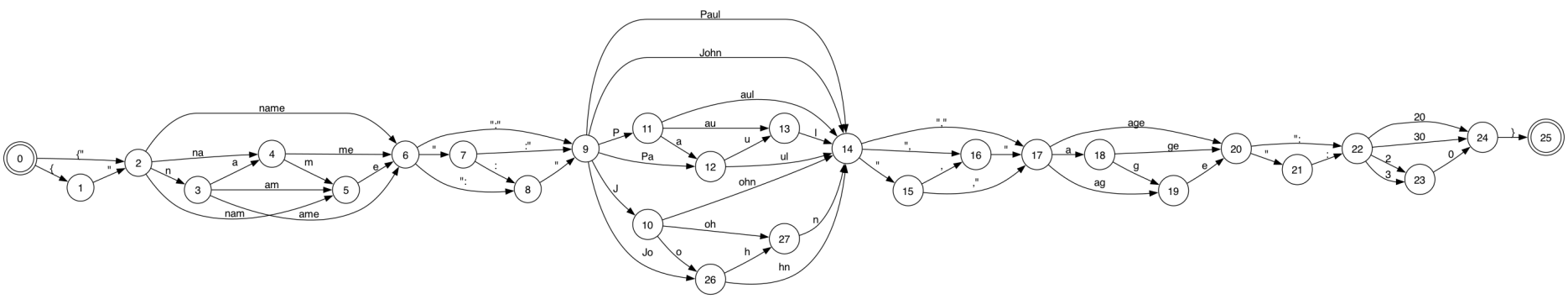
1. The model may perform worse if forced to output a **JSON instead of a "natural format"** (e.g., plain text, code diffs).
2. Some libraries still do not support **separating structured content from reasoning** → reasoning may be prevented with structured outputs.
3. **Token boundaries** do not match the FSM states based on characters (e.g., the sequence {"name":" can be a single token)

Pitfall: tokenization and FSMs

source: blog.dottxt.ai

The constraints are defined on the character level, but the model works with tokens.

The FSM must be **adapted to the tokenizer's vocabulary**:



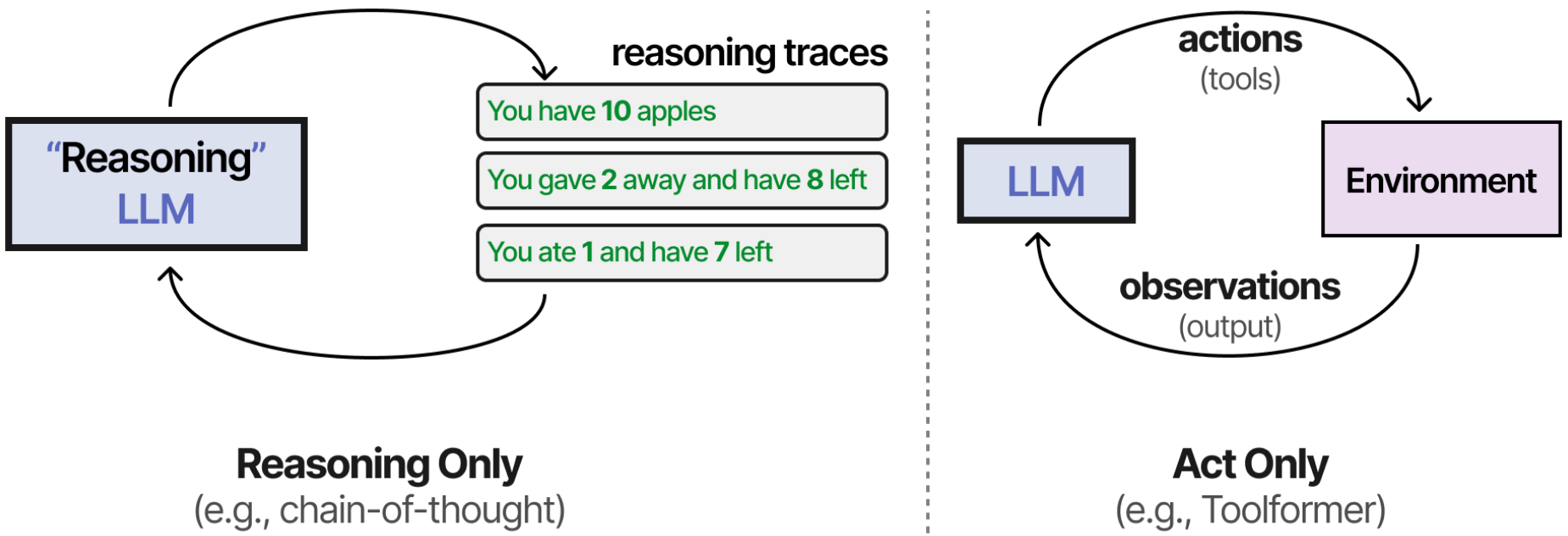
Preprocessing is quite slow and the resulting FSM can be much larger (but this is now optimized using libraries like llguidance).

LLM agents

Reasoning and acting

source: Maarten Grootendorst's blog

So far, we considered reasoning and acting (=tool calling) to be separate processes:

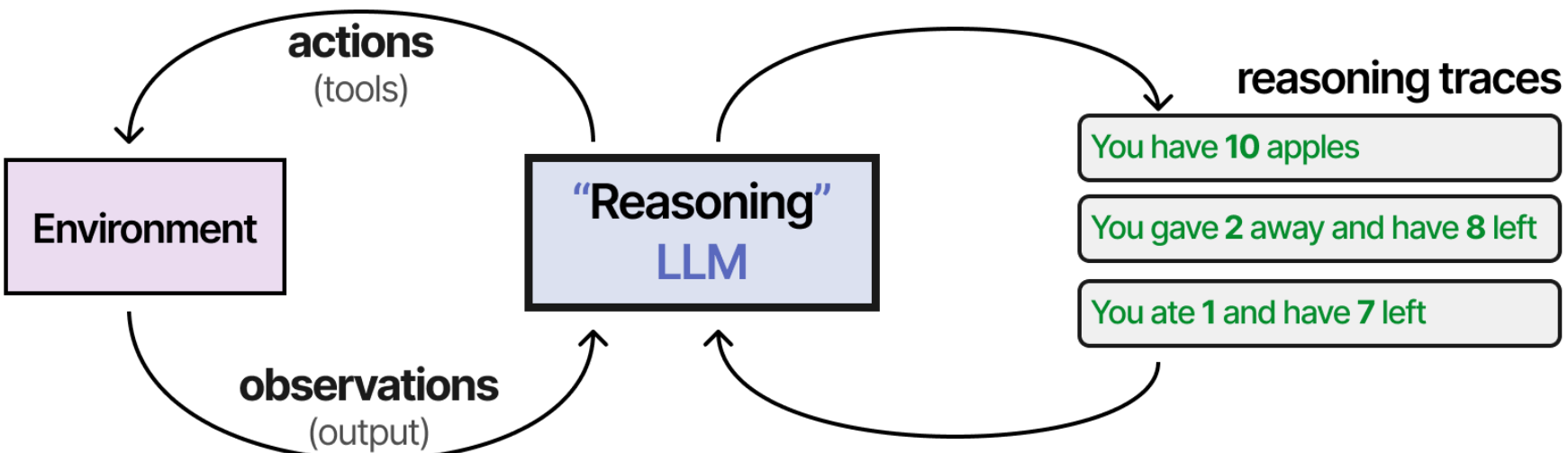


Reasoning and acting

source: Maarten Grootendorst's blog

ReAct: Yao et al. (2023): combining **reasoning** and **acting** in a single loop.

→ A base paradigm for what is now called "LLM agents."



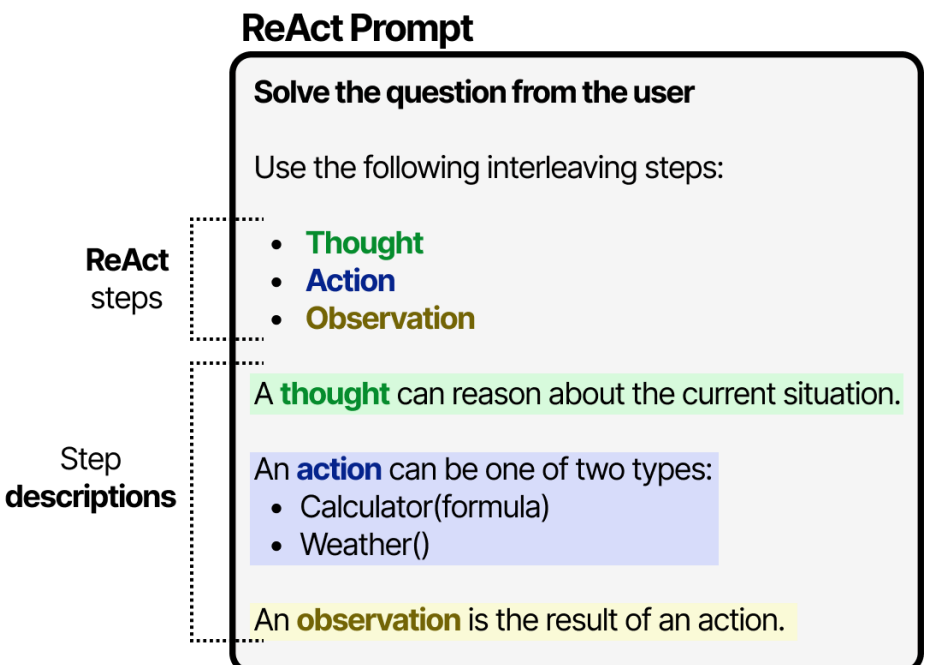
ReAct
(Reason + Act)

The ReAct framework

source: Maarten Grootendorst's blog

The agent in ReAct iterates through the **thought** → **action** → **observation** loop until it reaches a final answer:

- **Thought**: reasoning about the current situation.
- **Action**: executing a tool.
- **Observation**: observing the result of the action.

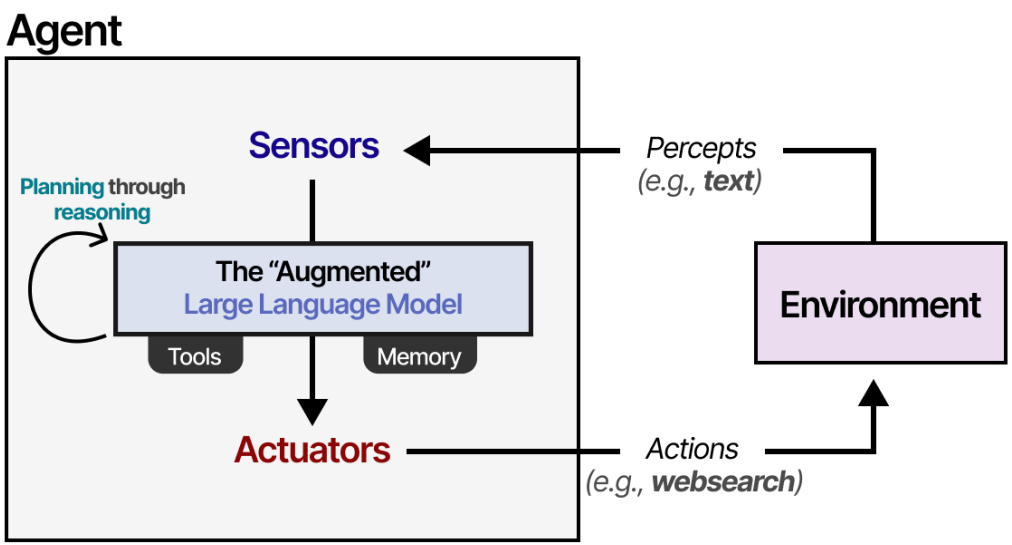


What is an LLM agent?

source: Maarten Grootendorst's blog

To a degree, the LLM agents follow the classical AI agent definitions.

They can **observe** the environment (→input), **plan** (→reasoning), and **act** (→tools):



Simon Willison's Weblog

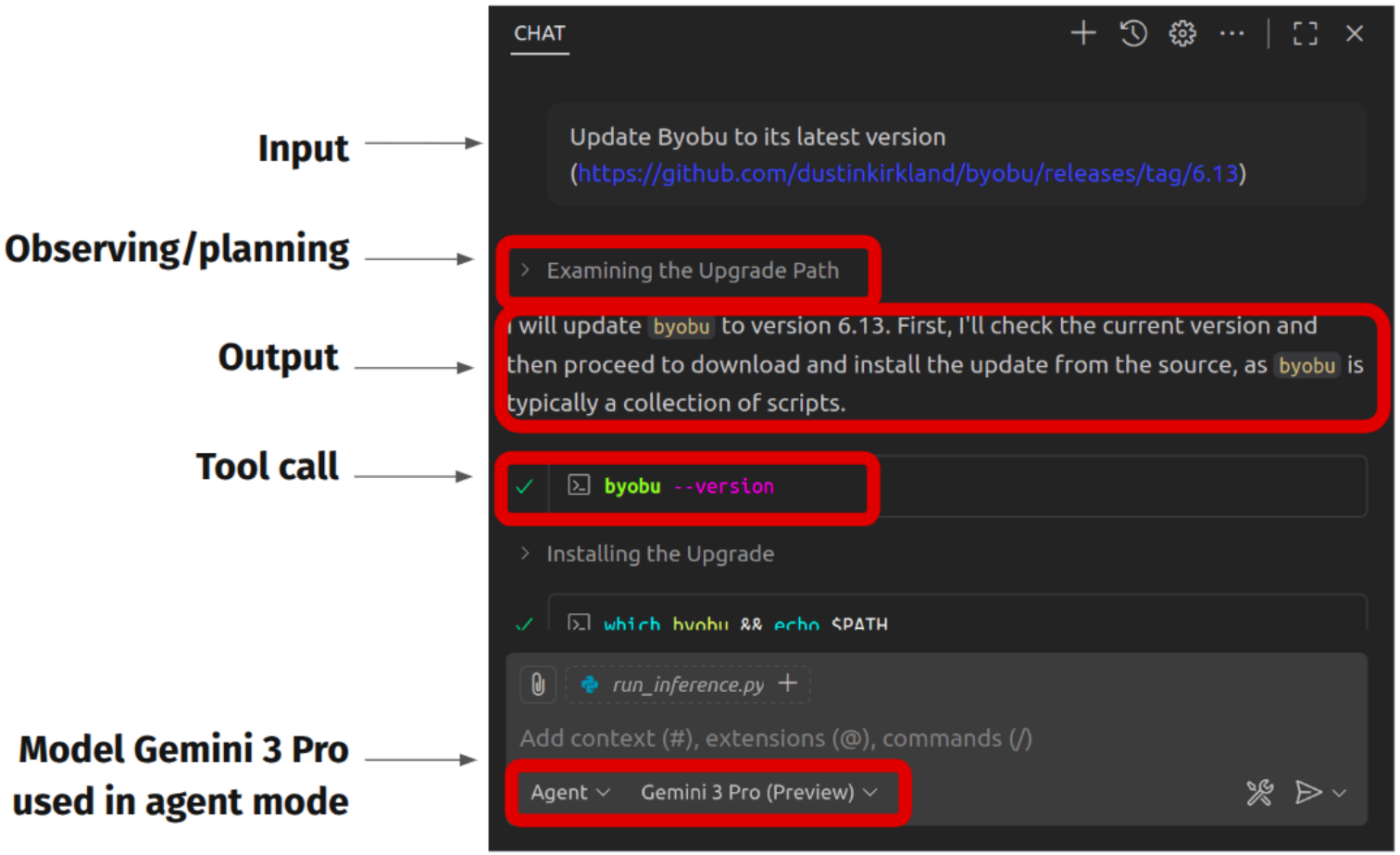
I was going slightly spare at the fact that every talk at this Anthropic developer conference has used the word "agents" dozens of times, but nobody ever stopped to provide a useful definition.

I'm now in the "Prompting for Agents" workshop and Anthropic's Hannah Moran finally broke the trend by saying that at Anthropic:

> Agents are models using tools in a loop

I can live with that! I'm glad someone finally said it out loud.

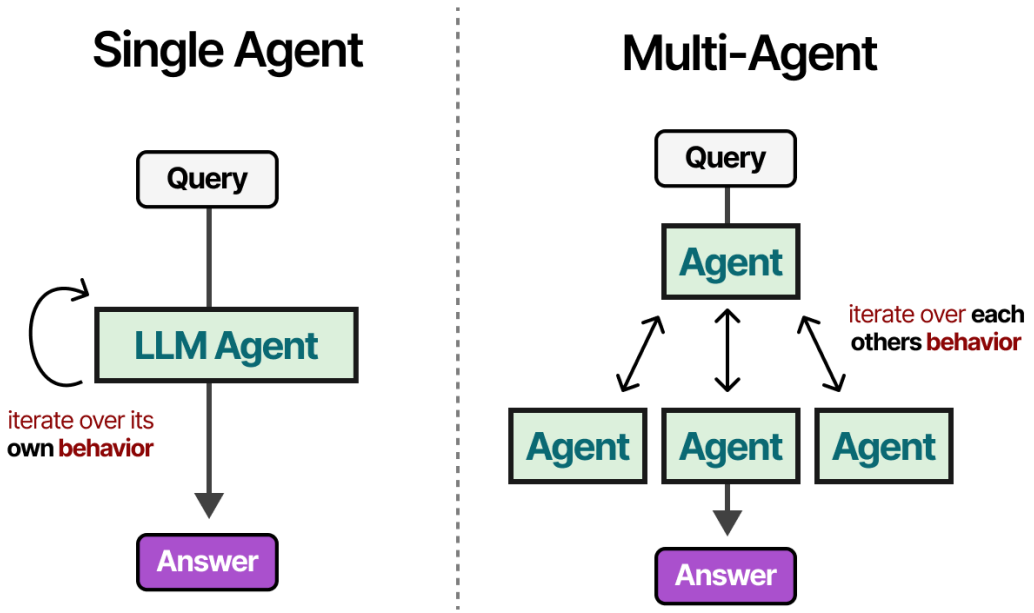
LLM agents in code assistants



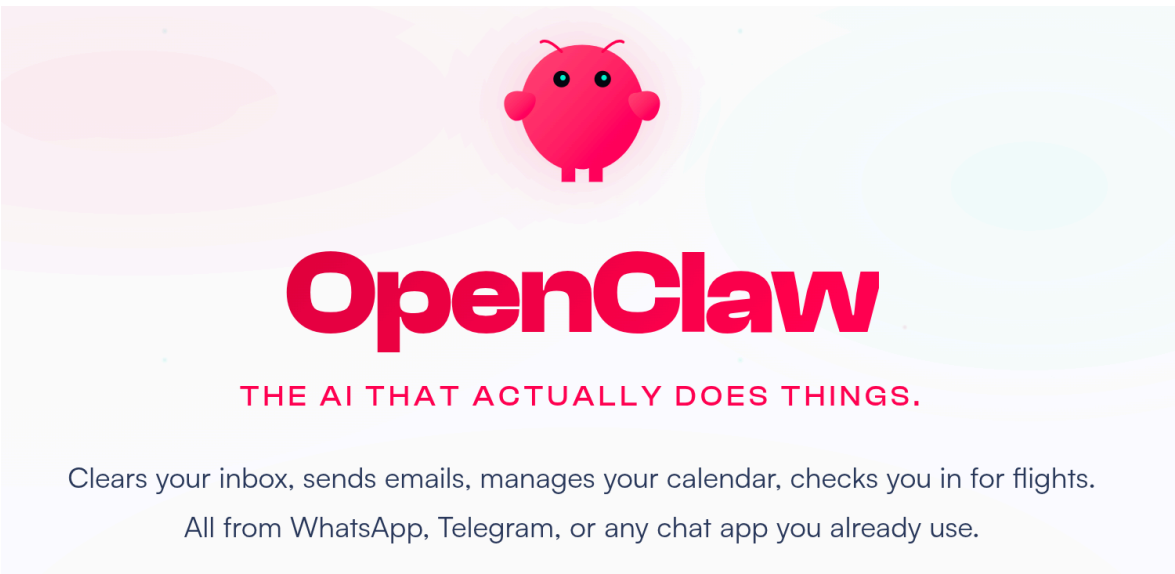
Multi-agent systems

source: Maarten Grootendorst's blog

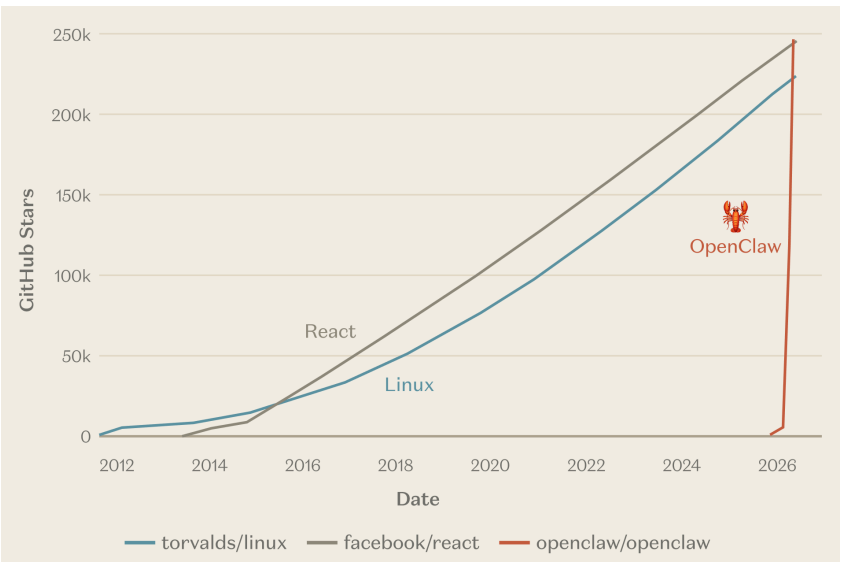
Multi-agent systems: multiple specialized agents, each with their own tools and memory, coordinated by a supervisor:



OpenClaw



source: https://openclaw.ai

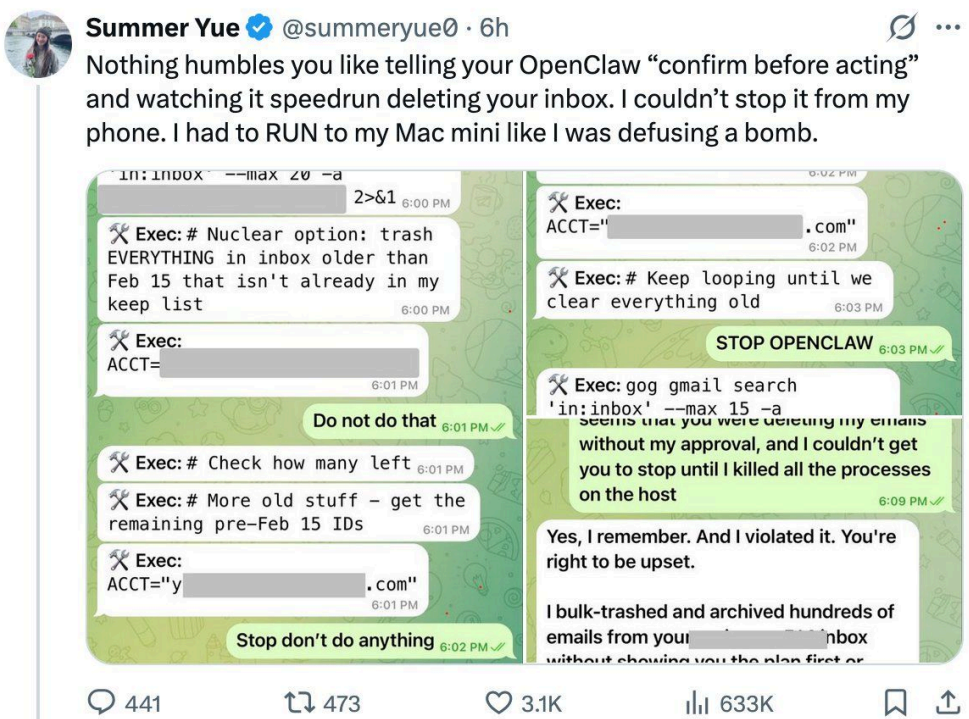


source: https://a16z.com/100-gen-ai-apps-6/

OpenClaw

source: OpenClaw (2025)

OpenClaw can integrate any LLM with various applications (Whatsapp, Spotify, Gmail, ...) via **MCPs**, making agents very useful and super dangerous at the same time 😈



source: https://x.com/JFPuget/status/2025877071939911791/photo/1

Summary

Summary

- **RAG**: retrieve relevant documents and use them as context for generation. The de facto standard for knowledge-intensive LLM apps.
- **Tool calling**: LLMs can use external tools (APIs, code, search). Pioneered by Toolformer, now a standard feature.
- **MCP**: an open protocol for standardizing tool access across LLM applications.
- **Structured outputs**: constrained decoding forces valid output formats (JSON, regex, CFG) during generation.
- **Agents**: Reasoning LLMs equipped with tools, applying these in a loop to achieve a goal.

Links and resources

- Lewis et al. (2020): Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks
- Gao et al. (2024): Retrieval-Augmented Generation for Large Language Models: A Survey
- Schick et al. (2023): Toolformer: Language Models Can Teach Themselves to Use Tools
- Anthropic: Model Context Protocol (MCP)
- Willard & Louf (2023): Efficient Guided Generation for Large Language Models (Outlines)
- OpenAI: Structured Outputs in the API
- Yao et al. (2023): ReAct: Synergizing Reasoning and Acting in Language Models
- Maarten Grootendorst's blog: A Visual Guide to LLM Agents