Yale University

LLM Inference, Tool Use, and Agents

S&DS 265 · Introductory Machine Learning · Lecture 24

Zhuoran Yang

Fall 2026

AI usage: figures were generated, and these slides polished, with the help of AI tools.

Outline

1. Why Agents
2. Model Serving
3. Prompting: Changing Behaviour without Training
4. Tool Calls
5. The Agent Loop

Learning Objectives

In this lecture you will learn:

1. What separates an agent from an instruct model;
2. The key ingredients of serving a model;
3. How a language model calls a tool, and how repeating that gives an agent loop.

Outline

OpenClaw Is the Fastest-Growing Repo on GitHub

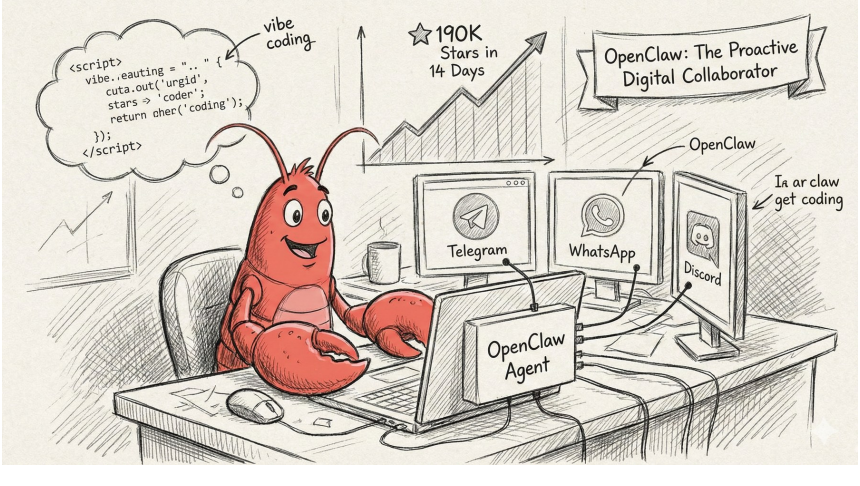
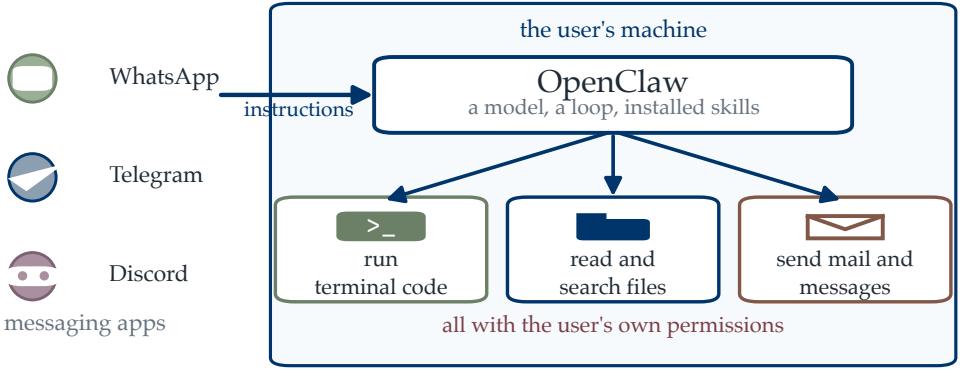


Illustration from *OpenClaw: The Open-Source AI Agent That Grew 190K GitHub Stars in 14 Days*, Faisal haque, `ai.plainenglish.io`, 2026.

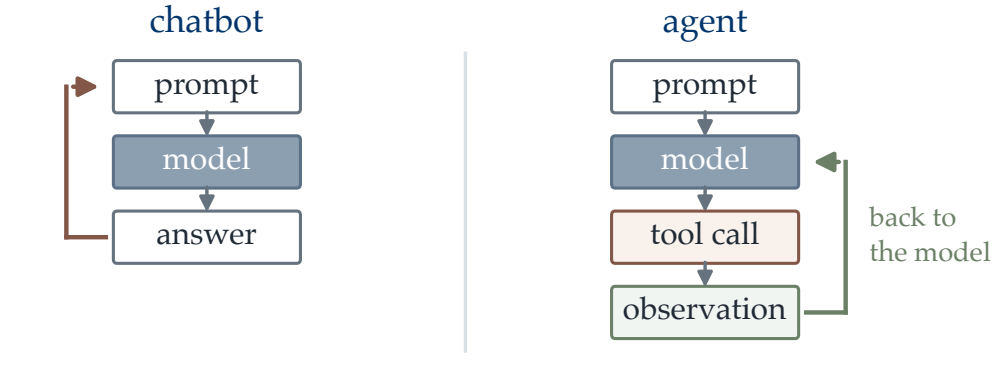
What OpenClaw Is



OpenClaw can run terminal code, read and search files, and send emails and text messages. It connects an LLM backend to the external world.

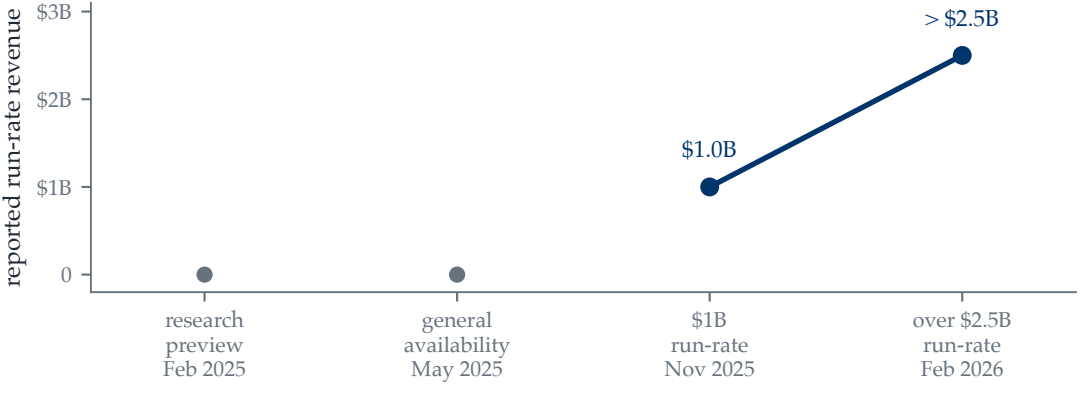
Architecture as described in Bitdefender, *Technical Advisory: OpenClaw Exploitation in Enterprise Networks*, 2026.

Chatbot and Agent



The same weights, arranged two ways. In an agent the model decides what data it will see next, which is what lets it recover from a surprise.

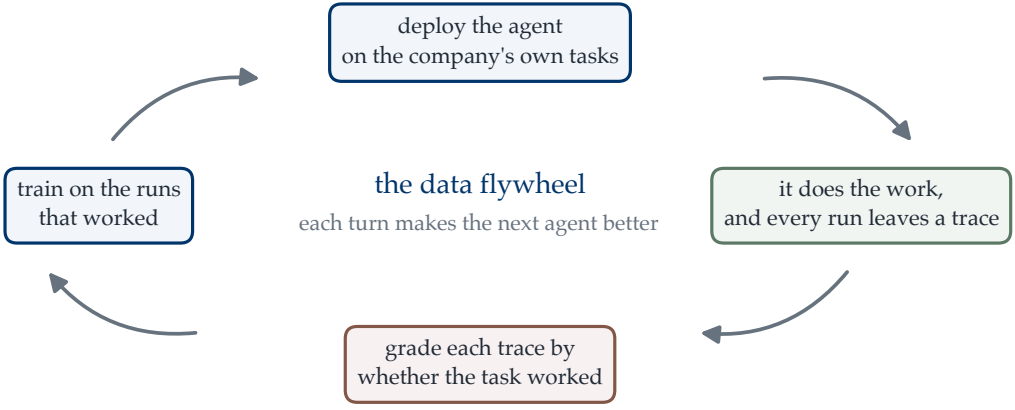
Anthropic's Revenue Skyrocketed after Claude Code



One product's reported run-rate revenue at four dated milestones. These are the company's own disclosures, made in its own announcements.

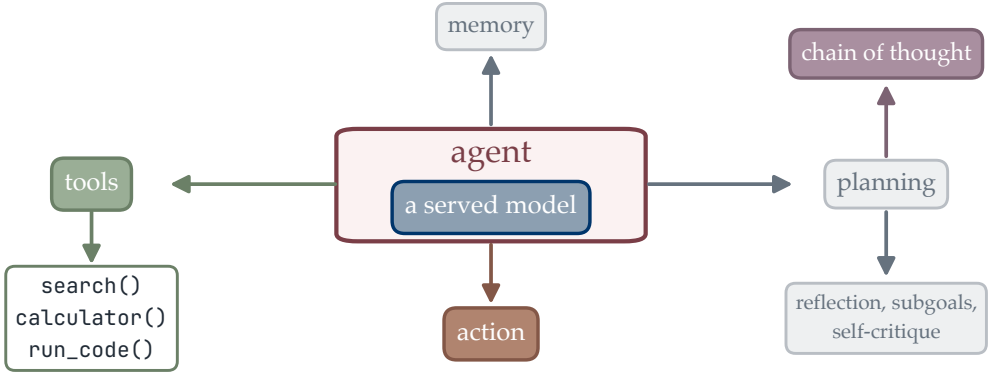
Anthropic announcements of 3 December 2025 and 12 February 2026; figures as stated there.

The Data Flywheel: Why Enterprises All Build Agents



Deploy an agent, collect graded traces, train on the runs that worked, and deploy again. Each turn of the loop makes the next agent better at the company's own tasks.

What an Agent Is Made Of



The four in colour are this lecture: a served model, tools, actions, and chain of thought.

After Lilian Weng, *LLM Powered Autonomous Agents*, 2023, `https://lilianweng.github.io/posts/2023-06-23-agent/`.

What This Lecture Covers

The weights are fixed throughout. Everything here is the system around them:

1. **Model serving**: what one request contains, and what it costs to answer;
2. **Prompting**: examples, chains of thought, and sampled votes;
3. **Tool use**: how a call is emitted, and how the result comes back;
4. The **agent loop** that turns an LLM into an agent.

What This Lecture Covers

The weights are fixed throughout. Everything here is the system around them:

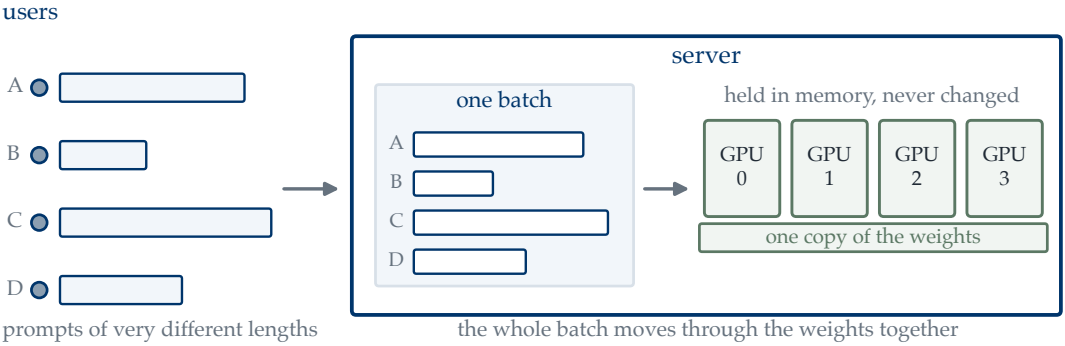
1. **Model serving**: what one request contains, and what it costs to answer;
2. **Prompting**: examples, chains of thought, and sampled votes;
3. **Tool use**: how a call is emitted, and how the result comes back;
4. The **agent loop** that turns an LLM into an agent.

Nothing in this lecture involves training. The LLM is fixed throughout.

Outline

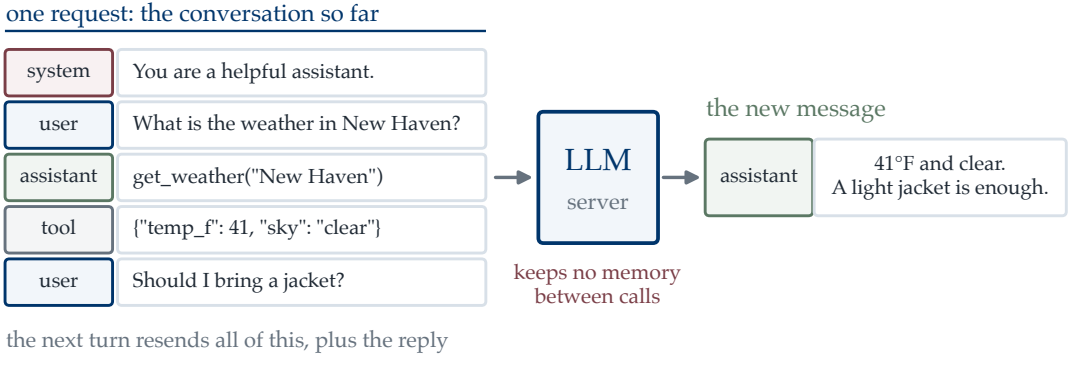
1. Why Agents
2. Model Serving
3. Prompting: Changing Behaviour without Training
4. Tool Calls
5. The Agent Loop

What Serving a Model Looks Like



Many users, prompts of very different lengths, and one copy of the weights spread over several GPUs. The server collects whatever has arrived into a batch, because throughput comes from sharing a forward pass.

What One Request Contains



The whole conversation travels in every call, tagged by role — and a tool result is a message like any other. The server holds no state between calls: "the model remembers" is a property of the client, which resends the history.

The Request an Application Sends

A list of role–content pairs, plus decoding settings:

```
{ "messages": [
   {"role": "system", "content": "You are a helpful assistant."},
   {"role": "user", "content": "What is the weather in New Haven?"},
   {"role": "assistant", "tool_calls": [
      {"name": "get_weather", "arguments": {"city": "New Haven"}}]},
   {"role": "tool", "name": "get_weather", "content": "41F, clear"},
   {"role": "user", "content": "Should I bring a jacket?"}
 ],
 "temperature": 0.2, "max_tokens": 300 }
```

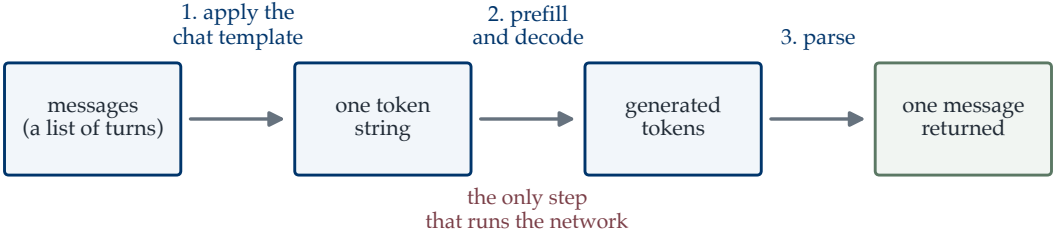
The Request an Application Sends

A list of role–content pairs, plus decoding settings:

```
{ "messages": [
   {"role": "system", "content": "You are a helpful assistant."},
   {"role": "user", "content": "What is the weather in New Haven?"},
   {"role": "assistant", "tool_calls": [
      {"name": "get_weather", "arguments": {"city": "New Haven"}}]},
   {"role": "tool", "name": "get_weather", "content": "41F, clear"},
   {"role": "user", "content": "Should I bring a jacket?"}
 ],
 "temperature": 0.2, "max_tokens": 300 }
```

system carries the developer's instructions, user the authenticated person's request, assistant earlier model output, and tool what a program returned. All four arrive as tokens in one string, so nothing in the format enforces the system message.

What the Server Does with It



steps 1 and 3 are string handling

A message list goes in and a message comes back. Only the middle step runs the network; the two on either side of it are string handling.

The Three Steps

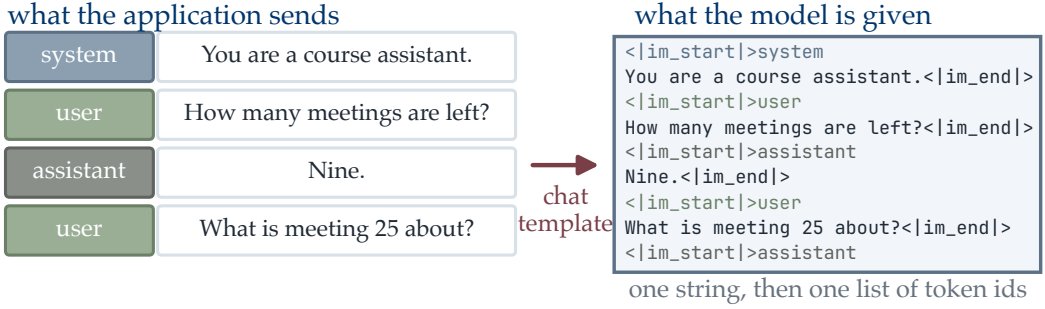
1. Apply the **chat template** of Lecture 23, turning the message list into one token string;
2. **Prefill** that string, then **decode** until the stop token;
3. **Parse** the generated tokens back into a message.

The Three Steps

1. Apply the **chat template** of Lecture 23, turning the message list into one token string;
2. **Prefill** that string, then **decode** until the stop token;
3. **Parse** the generated tokens back into a message.

Step 2 is where the cost is, and most of what follows is about it. Steps 1 and 3 are what make the model usable as a chatbot.

Step 1: the Chat Template

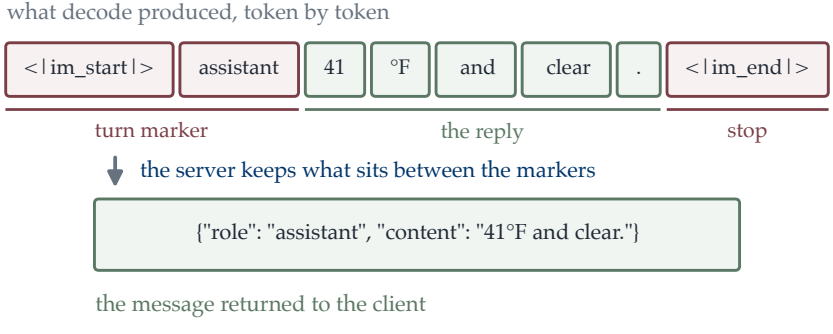


the roles are a convention of the "template", not a feature of the network

the server flattens; the model only ever sees the right-hand side

Special tokens mark the turns. Posttraining taught the model to stop at the end marker, which is how the server knows a reply has finished.

Step 3: Recovering the Message from the Tokens



The same markers that framed the prompt now delimit the reply, so a plain string operation recovers the message. Nothing here inspects the model: the format alone carries it.

Prefill and Decode Stages of a Single Request

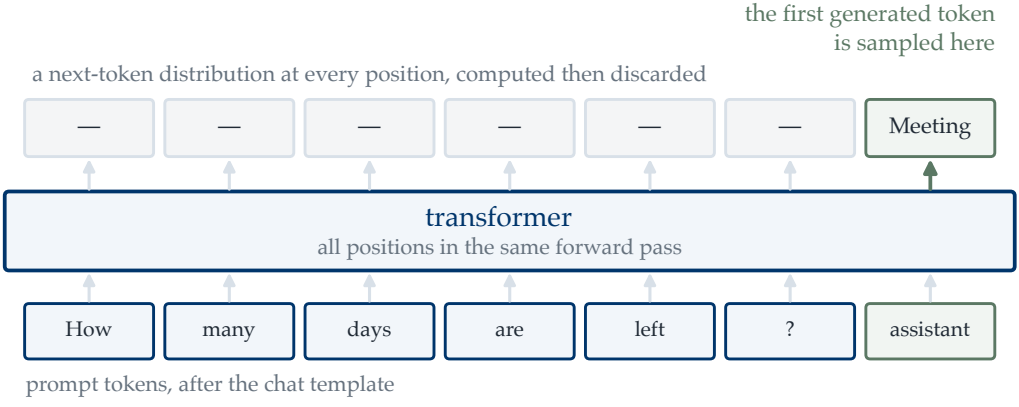
- **Prefill**: run the whole prompt through the network once. Every token position is processed in parallel.
- **Decode**: produce tokens one at a time. Each needs a full forward pass, and cannot start until the previous token is chosen.

Prefill and Decode Stages of a Single Request

- **Prefill**: run the whole prompt through the network once. Every token position is processed in parallel.
- **Decode**: produce tokens one at a time. Each needs a full forward pass, and cannot start until the previous token is chosen.

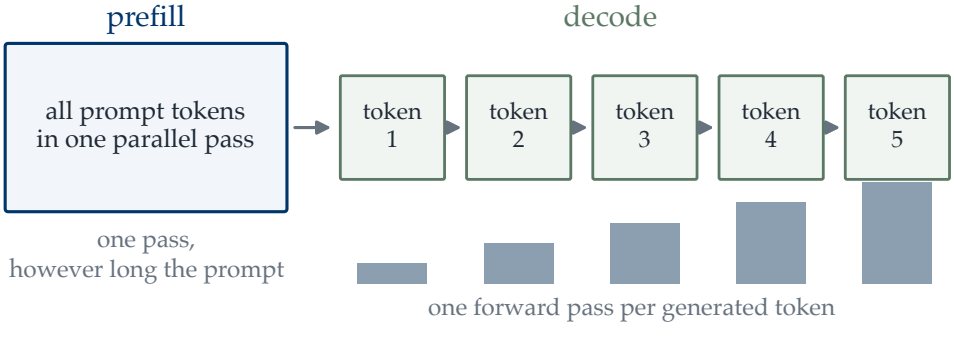
A 2000-token prompt costs one pass; a 500-token answer costs 500. Output length is what users wait for.

Why Prefill Is One Dense Pass



The causal mask of Lecture 22 lets every position be computed at the same time, so a long prompt is one wide matrix multiply. Only the distribution at the last position is used — that is where decoding starts.

Prefill and Decode



The same weights, run two different ways. Prefill does all positions at once; decode does one token per pass, and cannot be parallelized. The two phases stress the hardware so differently that large deployments run them on separate machines: search PD disaggregation.

Techniques for Efficient Model Serving

Decode is the expensive half, and three ideas carry almost all of the practical gain:

- The **KV cache**: reuse what prefill already computed;
- **Paged attention**: hand out cache memory in fixed-size blocks;
- **Continuous batching**: keep the batch full as requests finish.

Techniques for Efficient Model Serving

Decode is the expensive half, and three ideas carry almost all of the practical gain:

- The **KV cache**: reuse what prefill already computed;
- **Paged attention**: hand out cache memory in fixed-size blocks;
- **Continuous batching**: keep the batch full as requests finish.

All three are systems decisions: they change what serving costs, and leave the output distribution untouched.

Source: Paged attention and continuous batching are from Kwon et al. (2023), *vLLM*, arXiv:2309.06180.

The KV Cache

Attention at position t compares the query q_t against the key of every earlier position, and averages those positions' values:

$$\text{Attn}(q_t) = \sum_{s \le t} \alpha_{ts} \cdot v_s, \qquad \alpha_{ts} \propto \exp\{q_t^\top k_s / \sqrt{d}\}.$$

The KV Cache

Attention at position t compares the query q_t against the key of every earlier position, and averages those positions' values:

$$\mathrm{Attn}(q_t) = \sum_{s \le t} \alpha_{ts} \cdot v_s, \qquad \alpha_{ts} \propto \exp\{q_t^\top k_s / \sqrt{d}\}.$$

The pairs (k_s, v_s) do not depend on t: computed once, read back at every later step. They are the **KV cache**.

The KV Cache

Attention at position t compares the query q_t against the key of every earlier position, and averages those positions' values:

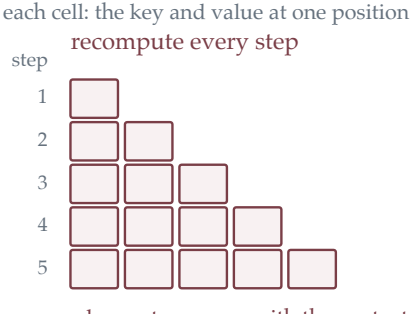
$$\mathrm{Attn}(q_t) = \sum_{s \le t} \alpha_{ts} \cdot v_s, \qquad \alpha_{ts} \propto \exp\{q_t^\top k_s / \sqrt{d}\}.$$

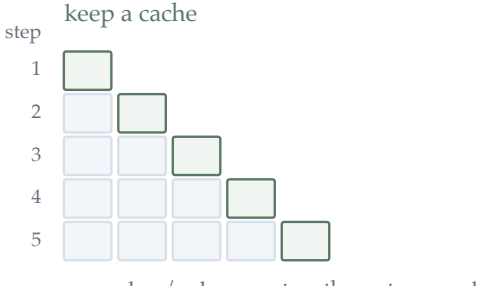
The pairs (k_s, v_s) do not depend on t: computed once, read back at every later step. They are the **KV cache**.

$$\underbrace{2}_{K,V} \times (\text{layers}) \times (\text{width}) \times (\text{bytes}) \;=\; 128\ \text{KiB per token}$$

For Llama-3 8B that is 1 GiB per 8k-token conversation. Against 16 GiB of weights, the cache is what limits how many users fit at once.

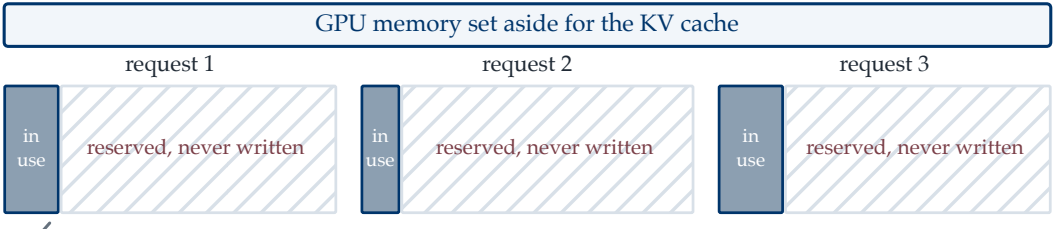
What the KV Cache Saves





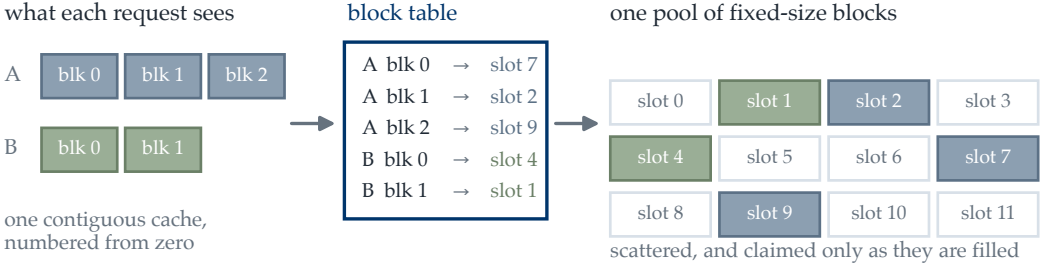
Without a cache, every decode step recomputes the keys and values of the whole context. Storing them turns that into one new pair per step, and the cost of a step stops growing with the conversation.

The Problem: Static Memory Allocation



A request's cache grows one token at a time and stops somewhere unknown, so the simple scheme reserves its maximum length up front. Most of that run is then reserved and never written, and being contiguous, the free part cannot be given to anyone else.

Better Memory Allocation

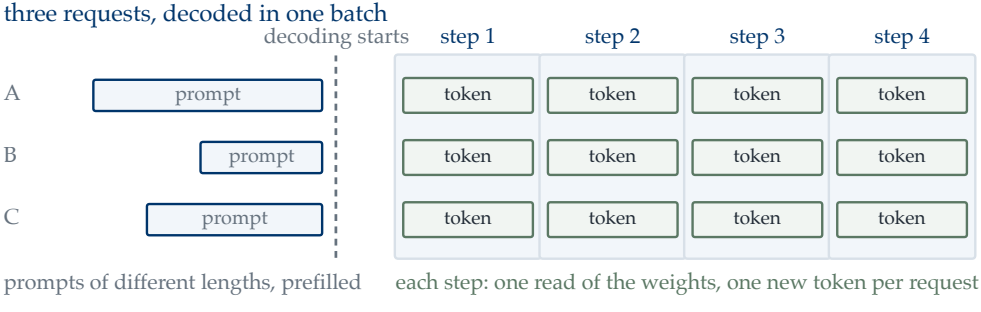


the same indirection as virtual memory: no request reserves space for a length it may never reach

Paged attention moves to non-contiguous allocation. Each request still sees one contiguous cache, while a block table maps its blocks onto fixed-size slots anywhere in a shared pool — the same indirection that virtual memory uses.

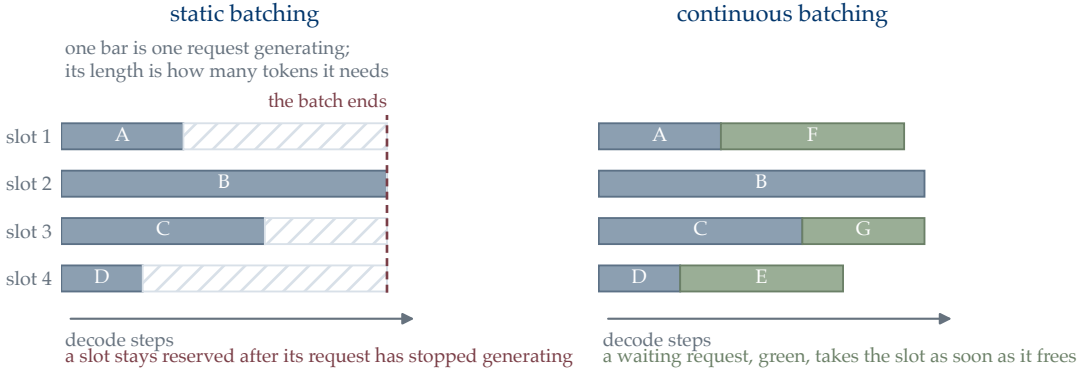
Kwon et al. (2023), *Efficient Memory Management for Large Language Model Serving with PagedAttention*, arXiv:2309.06180.

Decoding a Batch Together



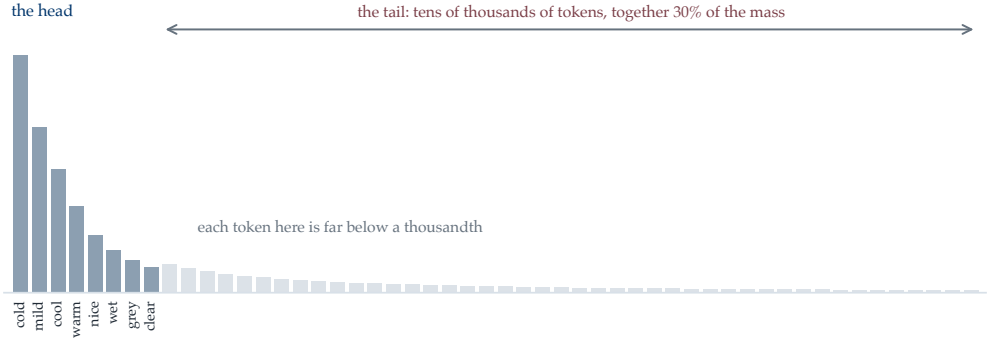
Decode reads every weight in the model to produce one token, so a single request wastes almost all of that read. Prompts of different lengths are prefilled, then the batch advances in lockstep: one forward pass per step, one new token for every request in it.

Continuous Batching



Static batching fixes the membership at the start: four requests enter together, and no new one may join until every slot is done. Since requests stop generating at different steps, most slots sit idle. Continuous batching lets a finished request leave and a waiting one take its slot.

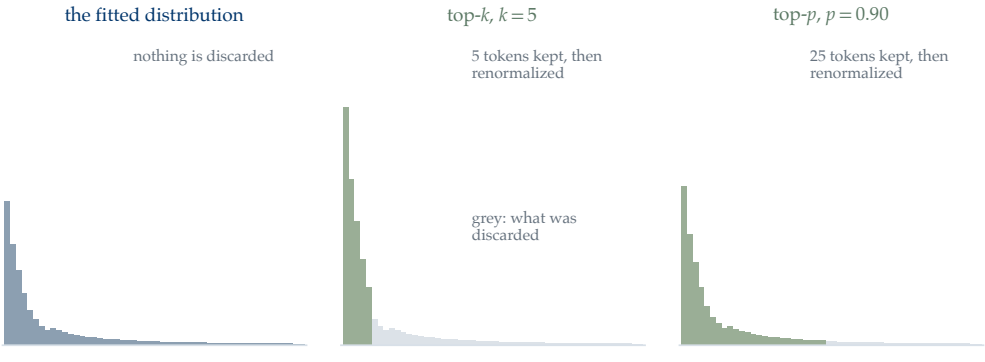
Why We Do Not Want the Tail



A model's next-token distribution is heavy-tailed: a few plausible continuations, and tens of thousands of tokens each carrying a little mass. Sampling faithfully draws from that tail every so often, and the model then conditions on what it drew — so everything after continues from a sentence it would never have written.

Illustrative distribution: fixed head probabilities and a power-law tail, exactly as written in the figure script.

Top-k and Top-p Sampling

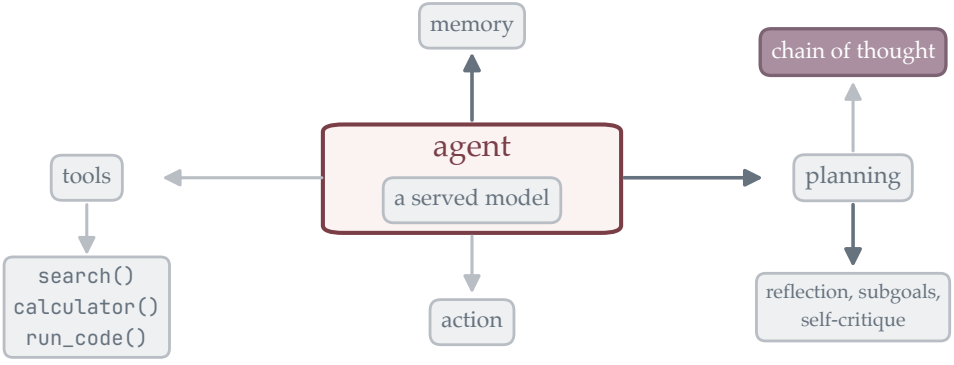


Top-k keeps the k most probable tokens, **top-p** the smallest set whose probability reaches p, and both then renormalize so the survivors sum to one. Here $k = 5$ keeps 5 and $p = 0.90$ keeps 25: a fixed count and a fixed mass need not agree. Temperature rescales the logits first; Appendix A.1 and A.2 state all of it.

Exact renormalization of the illustrative distribution on the previous slide.

Outline

Prompting: What Goes into the Context



The weights are fixed and the request is one token string, so everything an application controls is what is in that string. We are now in the chain of thought box: the text a model writes before it answers.

After Lilian Weng, *LLM Powered Autonomous Agents*, 2023, `https://lilianweng.github.io/posts/2023-06-23-agent/`.

Prompting Is an Art

The same weights answer well or badly depending on how the request is phrased. Practitioners collect phrases that happen to work — “take a deep breath”, “let’s solve this step by step”, “you are an expert” — and swap them in and out.

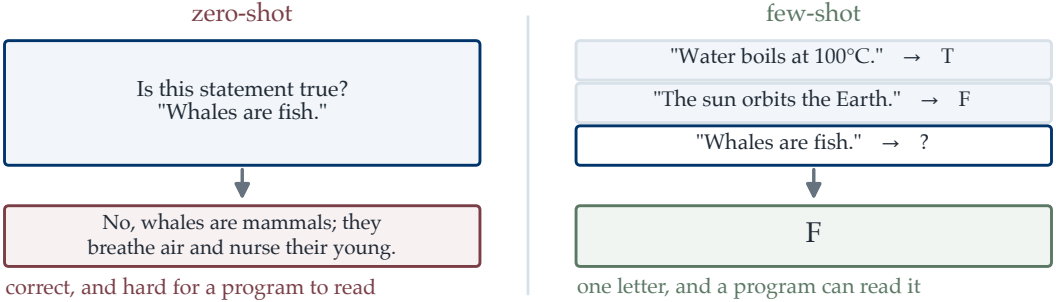
Prompting Is an Art

The same weights answer well or badly depending on how the request is phrased. Practitioners collect phrases that happen to work — "take a deep breath", "let's solve this step by step", "you are an expert" — and swap them in and out.

We introduce a few principled methods:

- **Few-shot prompting**: put solved examples in the context;
- **Chain of thought**: ask for the reasoning before the answer;
- **Variants of chain of thought**: search or vote over several of them.

Few-Shot Prompting

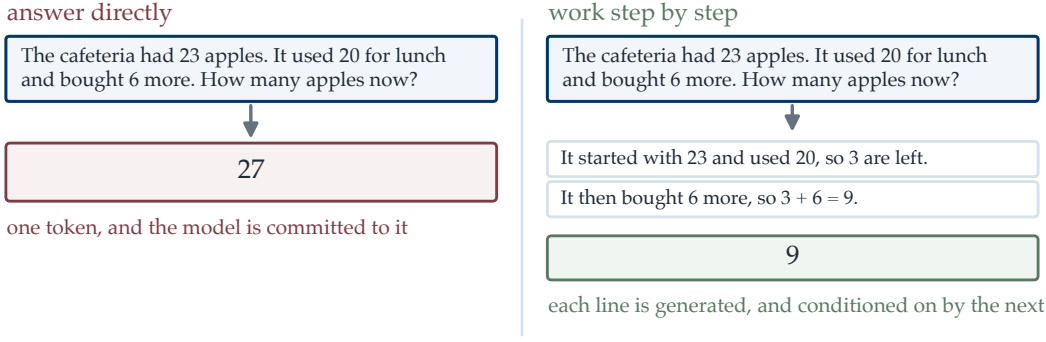


the examples name the task and fix the answer format: T, in place of “true” or a sentence

Giving a few solved examples in the prompt makes the model understand the task you want. They also fix the format of the answer, e.g., “true/false” or “T/F”.

Brown et al. (2020), *Language Models are Few-Shot Learners*, arXiv:2005.14165.

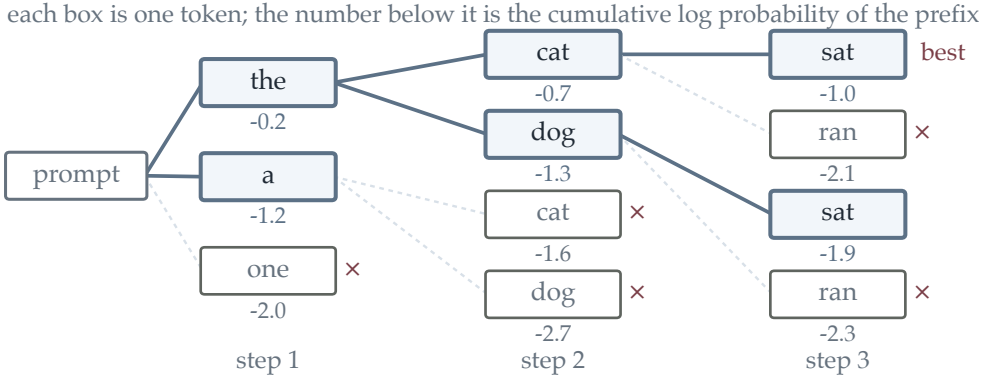
Chain of Thought



Chain-of-thought prompting makes the model generate intermediate reasoning steps before the answer, which helps on hard problems. It combines with few-shot prompting: write the reasoning traces into the examples, and the model follows them.

Wei et al. (2022), *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*, arXiv:2201.11903.

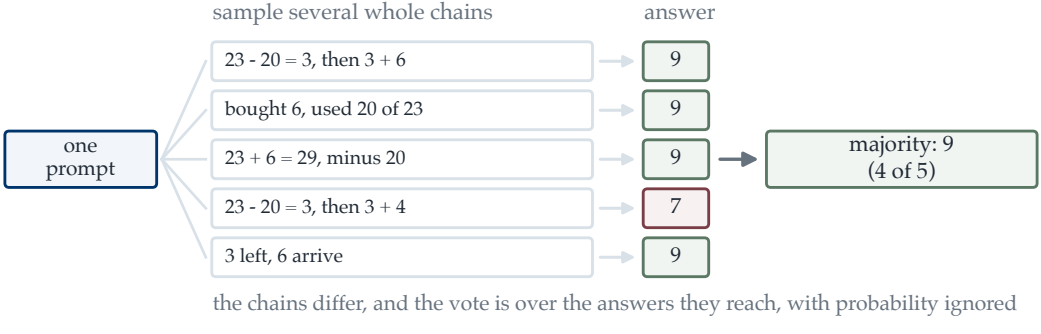
Variant One: Beam Search over Reasoning Steps



Keep the k best partial chains at each step, extend every one, and prune back to k. The model itself does the ranking. This is **Tree of Thoughts** in its breadth-first form.

Yao et al. (2023), *Tree of Thoughts*, arXiv:2305.10601. Appendix A.4 has beam search as a decoding rule.

Variant Two: Self-Consistency



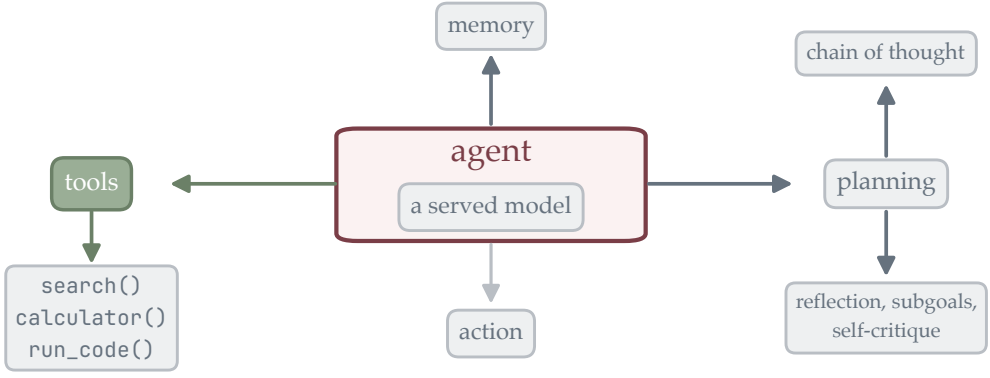
Sample n complete chains, read the final answer off each, and take the majority. This searches for a stable conclusion: different routes that arrive at the same place are better evidence than one route the model found likely.

Wang et al. (2023), *Self-Consistency Improves Chain of Thought Reasoning*, arXiv:2203.11171.

Outline

1. Why Agents
2. Model Serving
3. Prompting: Changing Behaviour without Training
4. Tool Calls
5. The Agent Loop

LLM Generates Tool Calls



Prompting spends more tokens inside a closed system. To check a fact, run a calculation, or read a file, the model has to reach outside it — and the only thing it can emit is text.

After Lilian Weng, *LLM Powered Autonomous Agents*, 2023.

Structure of a Tool Call

Continuing the request we saw earlier. The model emits a call:

```
{"name": "get_weather", "arguments": {"city": "New Haven"}}
```

Structure of a Tool Call

Continuing the request we saw earlier. The model emits a call:

```
{"name": "get_weather", "arguments": {"city": "New Haven"}}
```

The program runs it and returns a response:

```
{"role": "tool", "name": "get_weather", "content": "41F, clear"}
```

Structure of a Tool Call

Continuing the request we saw earlier. The model emits a call:

```
{"name": "get_weather", "arguments": {"city": "New Haven"}}
```

The program runs it and returns a response:

```
{"role": "tool", "name": "get_weather", "content": "41F, clear"}
```

A **call** names a function and its arguments. A **response** comes back under a new role, so the model can recognize that it is a tool response.

Nothing new happens in the LLM. These are tokens.

Specify Tools and Format in System Prompt

System prompt specifies what tools are available and how to use them.

system
```
tools: get_weather(city)
reply with <tool_call>name {args}</tool_call>
```

user
```
What is the weather in New Haven?
```

assistant
```
<tool_call>get_weather {city: "New Haven"}</tool_call>
```

tool
```
<tool_result>41F, clear</tool_result>
```

Tool Calls in Chat Template

The chat template wraps each turn in reserved tokens, and generation stops at the closing tag:

```
◁im_start|>assistant
<tool_call>get_weather {"city": "New Haven"}</tool_call>◁im_end|>
```

Tool Calls in Chat Template

The chat template wraps each turn in reserved tokens, and generation stops at the closing tag:

```
◁im_start|>assistant
<tool_call>get_weather {"city": "New Haven"}</tool_call>◁im_end|>
```

So the program knows a call was requested before any text reaches the user, and can run it and continue the same conversation.

Tool Calls in Chat Template

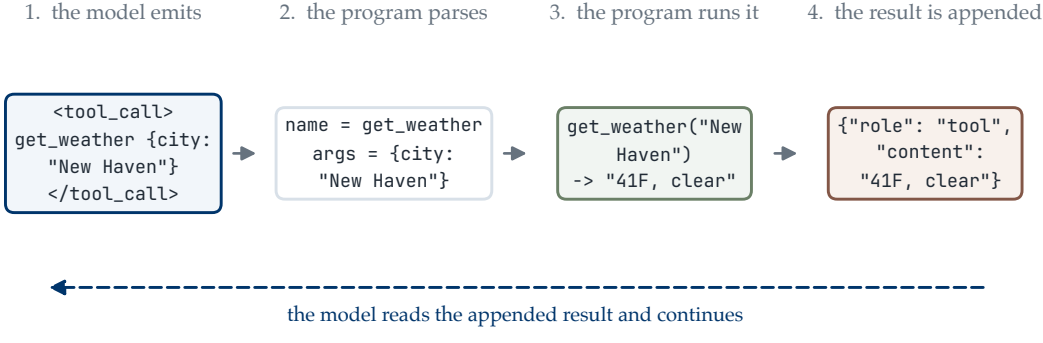
The chat template wraps each turn in reserved tokens, and generation stops at the closing tag:

```
◁im_start|>assistant
<tool_call>get_weather {"city": "New Haven"}</tool_call>◁im_end|>
```

So the program knows a call was requested before any text reaches the user, and can run it and continue the same conversation.

The model produces this shape because posttraining data contains it: SFT conversations in which a schema is shown and the correct response is a call.

One Round Trip, at the Token Level



The model performs only the first step — token generation. Parsing, executing, and appending are ordinary program code — after which the model reads the appended result as ordinary context and carries on the next turn.

Outline

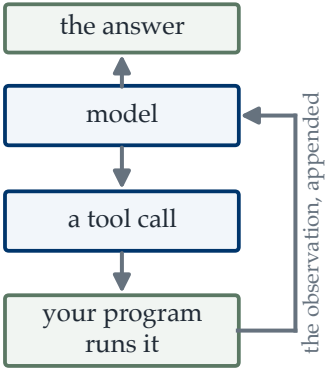
1. Why Agents
2. Model Serving
3. Prompting: Changing Behaviour without Training
4. Tool Calls
5. The Agent Loop

An Agent Loop Is a While Loop with Tool Calls

a while loop with tool calls

```
messages = [system_prompt, task]

while steps_left:
    reply = model(messages)
    if reply.is_answer:
        return reply.answer
    obs = run(reply.tool_call)
    messages += [reply, obs]
```



A simple agent loop: the model generates tool calls, the tools are executed, the observation is appended, and the loop goes round — until the answer is found.

Toy Example of Prompt of a Research Agent

The agent loop is generic. This prompt turns it into a research agent:

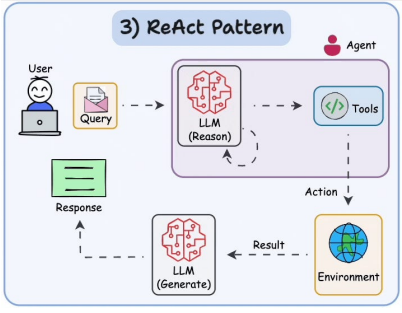
```
You answer questions using only the tools below.
Work in a loop. At each step emit exactly one JSON object:

  {"thought": "...", "action": "search", "input": "..."}
  {"thought": "...", "action": "read",   "input": "<doc_id>"}
  {"thought": "...", "action": "calc",   "input": "25 - 16"}
  {"thought": "...", "action": "answer", "input": "..."}

Rules:
- Never state a fact you have not read in an observation.
- Cite the doc_id that supports each claim.
- Prefer "answer" once you can cite every claim.
```

This prompt, plus an implementation of those four tools, is the agent.

ReAct: Thought, Action, Observation, Repeated



(1) Hotspot QA

Question: Aside from the Apple Remote, what other device can control the program Apple Remote was originally designed to interact with?

(1d) ReAct (Reason + Act)

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.
Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a remote control introduced in October 2005 by Apple … originally designed to control the Front Row media center program …

Thought 2: Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.
Act 2: Search[Front Row]
Obs 2: Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports',' Front Row (software) ', …]

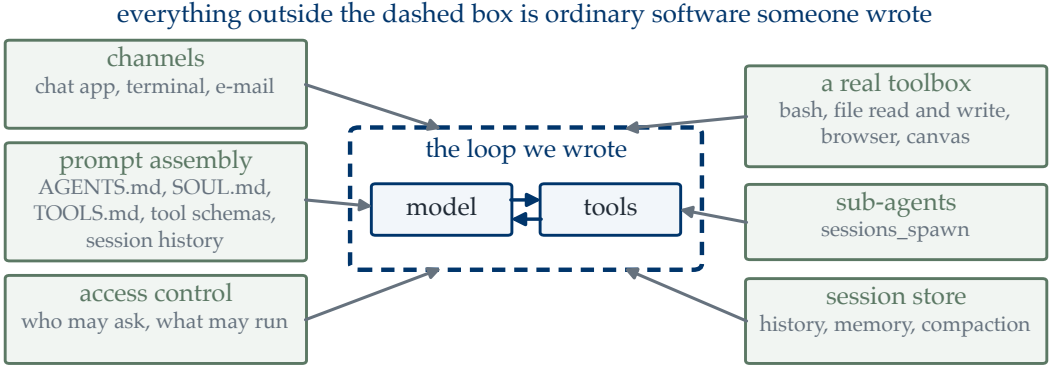
Thought 3: Front Row is not found. I need to search Front Row (software) .
Act 3: Search[Front Row (software)]
Obs 3: Front Row is a discontinued media center software …

Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.
Act 4: Finish[keyboard function keys]

The model itself writes `Thought` and `Act`; `Obs` is pasted in by the program.

Source: Right: Yao et al. (2023), *ReAct*, arXiv:2210.03629, Figure 1(d).

What a Practical Agent (e.g., OpenClaw) Adds



A shipped agent runs the loop we just wrote. Around it sit blocks that decide who may ask, what goes into the prompt, what may run, and what is remembered — and not one of them is a model.

After the OpenClaw architecture overview, ppaolo.substack.com.

Summary: Model Serving

A request becomes one token string through the chat template, then

$$\underbrace{\text{prefill}}_{\text{one dense pass}} \longrightarrow \underbrace{\text{decode}}_{\text{one pass per token}} .$$

Summary: Model Serving

A request becomes one token string through the chat template, then

$$\underbrace{\text{prefill}}_{\text{one dense pass}} \longrightarrow \underbrace{\text{decode}}_{\text{one pass per token}} .$$

- The **KV cache** makes decode linear rather than quadratic, and then dominates memory.
- **Paged** allocation and **continuous batching** keep that memory and the batch full.
- **Top-k** and **top-p** truncate the tail before sampling.

Summary: Tool Calls and the ReAct Loop

A tool call is text: the schema goes in the system prompt, the model emits a call in the chat template.

Then program parses the tool call, validates it, executes it, and pastes the result back as an observation.

Summary: Tool Calls and the ReAct Loop

A tool call is text: the schema goes in the system prompt, the model emits a call in the chat template.

Then program parses the tool call, validates it, executes it, and pastes the result back as an observation. Repeating that exchange is the ReAct loop,

$$\text{Thought} \rightarrow \text{Action} \rightarrow \text{Observation} \rightarrow \text{Thought} \rightarrow \cdots$$

Appendix

Three groups of notes skipped in the lecture hour.

A. **Truncated sampling.** Top-k and top-p written out exactly, and the step on which the two rules disagree.

B. **Beam search.** A token-level search that scores prefixes by cumulative log probability, with the corrections implementations make.

C. **References and further reading.**

A.1 Top-k: a Fixed Number of Candidates

Let $z \in \mathbb{R}^V$ be the logits at one position and

$$p(w) = \frac{\exp\{z_w/\tau\}}{\sum_{w'} \exp\{z_{w'}/\tau\}}.$$

Order the vocabulary so that $p_{(1)} \geq p_{(2)} \geq \cdots \geq p_{(V)}$, and keep the first k:

$$\mathcal{A}_k = \{w_{(1)}, \ldots, w_{(k)}\}, \qquad \tilde{p}(w) = \frac{p(w) \cdot \mathbf{1}\{w \in \mathcal{A}_k\}}{\sum_{w' \in \mathcal{A}_k} p(w')}.$$

k is fixed before the step is seen, so the same count survives however certain the model was.

A.2 Top-p: a Fixed Mass of Probability

With the same ordering, take the shortest prefix whose mass reaches p:

$$m = \min\Big\{j : \sum_{i \le j} p_{(i)} \ \ge \ p\Big\}, \qquad \mathcal{A}_p = \{w_{(1)}, \dots, w_{(m)}\},$$

and renormalize over $\mathcal{A}_p$ exactly as before.

A.2 Top-p: a Fixed Mass of Probability

With the same ordering, take the shortest prefix whose mass reaches p:

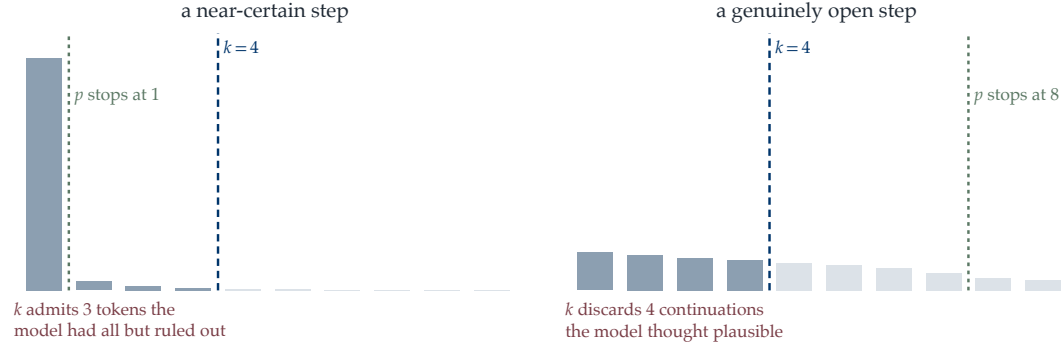
$$m = \min\Big\{j : \sum_{i \le j} p_{(i)} \ \ge \ p\Big\}, \qquad \mathcal{A}_p = \{w_{(1)}, \dots, w_{(m)}\},$$

and renormalize over $\mathcal{A}_p$ exactly as before.

- m is a function of the step: near 1 where the model is confident, and large where it is not.
- Servers apply temperature, then top-k, then top-p: the survivors are $\mathcal{A}_k \cap \mathcal{A}_p$.

Source: Holtzman et al. (2020), *The Curious Case of Neural Text Degeneration*, arXiv:1904.09751.

A.3 The Same Two Steps Under Both Rules



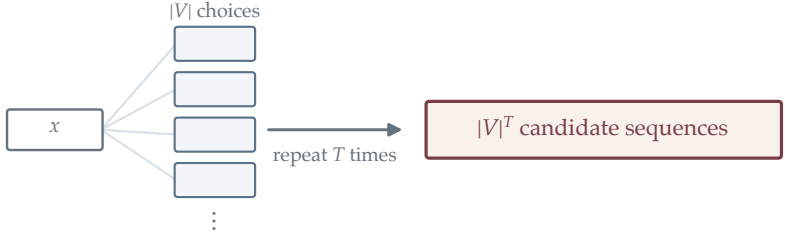
A fixed k errs in opposite directions: too permissive where the model is nearly certain, too strict where it is uncertain.

Illustrative distributions written into the figure script; the counts shown are exact for them.

B.1 The Decoding Problem

Find the token sequence that the model assigns the highest probability:

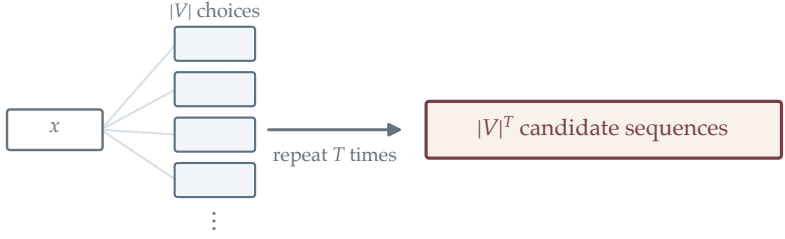
$$y^\star = \arg\max_y \log p_\theta(y \mid x) = \arg\max_y \sum_t \log p_\theta(y_t \mid x, y_{<t})$$



B.1 The Decoding Problem

Find the token sequence that the model assigns the highest probability:

$$y^\star = \arg\max_y \log p_\theta(y \mid x) = \arg\max_y \sum_t \log p_\theta(y_t \mid x, y_{<t})$$

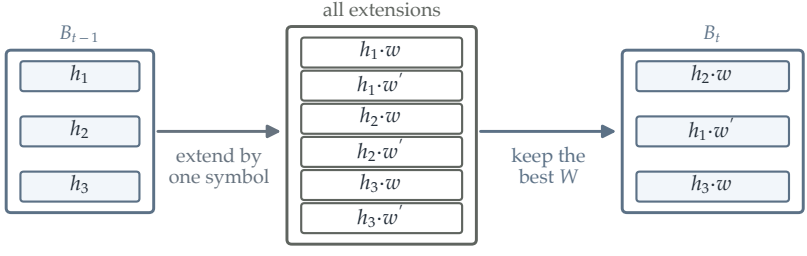


With $|V| \approx 10^5$ and a hundred tokens, that is about 10^{500} sequences. Exact maximization is out of reach.

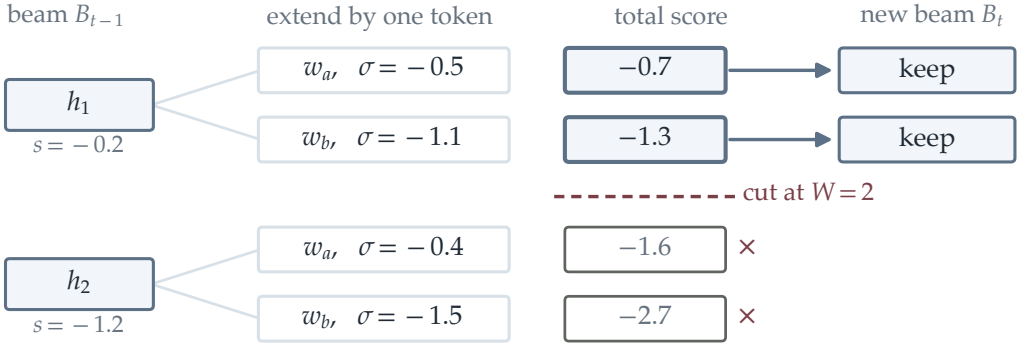
B.2 Beam Search, for Any Score

Let h be a **hypothesis** and $h \cdot w$ its extension by one symbol. Assume any score that accumulates, and keep the best W:

$$\text{score}(h \cdot w) = s(h) + \sigma(w \mid h), \qquad B_t = \text{top-}W_{\,h \in B_{t-1},\, w \in V}\ \text{score}(h \cdot w).$$



B.3 One Recursion Step



$$\text{score}(h \cdot w) = s(h) + \sigma(w \mid h)$$

Each hypothesis carries a history score $s(h)$; each extension adds a step score σ. The four totals are ranked and cut at W, which is the only place the beam gets smaller.

Original course illustration; the arithmetic is exact for the values shown.

B.4 Instantiated for a Language Model

The two pieces are read off the model directly:

- One-token score: $\sigma(w \mid h) = \log p_\theta(w \mid x, h)$;
- Hypothesis score: $s(h) = \sum_i \log p_\theta(y_i \mid x, y_{<i}) = \log p_\theta(h \mid x)$.

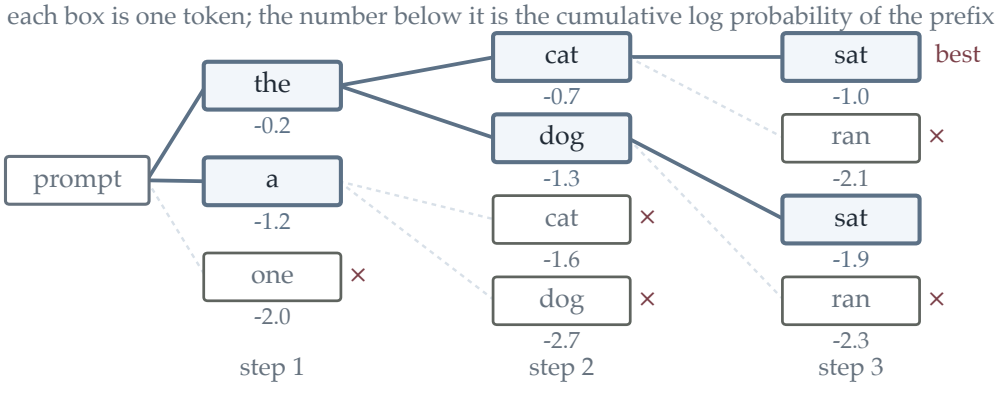
B.4 Instantiated for a Language Model

The two pieces are read off the model directly:

- One-token score: $\sigma(w \mid h) = \log p_\theta(w \mid x, h)$;
- Hypothesis score: $s(h) = \sum_i \log p_\theta(y_i \mid x, y_{<i}) = \log p_\theta(h \mid x)$.

Greedy decoding is the case $W = 1$. A step runs W forward passes and ranks $W \cdot |V|$ candidates, so beam search costs W times greedy.

B.5 Three Steps at the Token Level



Width $W = 2$ on real tokens. Each box is one token and the number below it is the cumulative log probability of the prefix; the unit of search is a token.

Original course illustration; the edge weights and totals are exact for the values shown.

B.6 Two Corrections in Every Implementation

Every extra token adds a negative term to s, so the raw score prefers short sequences. Implementations divide by a power of the length:

$$\tilde{s}(y_{1:T}) = \frac{s(y_{1:T})}{T^{\alpha}}, \qquad \alpha \in [0.6, 1.0].$$

A sequence that emits the stop token leaves the beam and is held aside; the search ends when W have finished.

Beam search maximizes probability. That suits translation; on open-ended text the most likely continuation is generic and repetitive.

Source: Wu et al. (2016), arXiv:1609.08144, for the length penalty; Holtzman et al. (2020), arXiv:1904.09751, for the degeneration it causes.

C.1 References

- Brown et al. (2020), *Language Models are Few-Shot Learners* — arXiv:2005.14165.
- Lewis et al. (2020), *Retrieval-Augmented Generation* — arXiv:2005.11401.
- Wei et al. (2022), *Chain-of-Thought Prompting* — arXiv:2201.11903.
- Wang et al. (2023), *Self-Consistency Improves Chain of Thought Reasoning* — arXiv:2203.11171.
- Yao et al. (2023), *ReAct: Synergizing Reasoning and Acting* — arXiv:2210.03629.
- Schick et al. (2023), *Toolformer* — arXiv:2302.04761.
- Kwon et al. (2023), *Efficient Memory Management for LLM Serving with PagedAttention* — vLLM, arXiv:2309.06180.
- Greshake et al. (2023), *Not What You've Signed Up For* — indirect prompt injection, arXiv:2302.12173.

C.2 Further Reading

- Weng, *LLM Powered Autonomous Agents* — `https://lilianweng.github.io/posts/2023-06-23-agent/`
- Anthropic, *Building Effective Agents* — `https://www.anthropic.com/engineering/building-effective-agents`
- vLLM documentation, continuous batching and paged attention — `https://docs.vllm.ai`
- Elshafie, *Paged Attention from First Principles: A View Inside vLLM* — `https://hamzaelshafie.bearblog.dev/paged-attention-from-first-principles-a-view-inside-vllm/`
- Hugging Face, *Chat Templates* and tool calling — `https://huggingface.co/docs/transformers/chat_templating`
- OWASP, *Top 10 for LLM Applications* — `https://owasp.org/www-project-top-10-for-large-language-model-applications/`