LLM post training

Topics

- Speculative decoding
- Guided decoding
- Long-context LLM

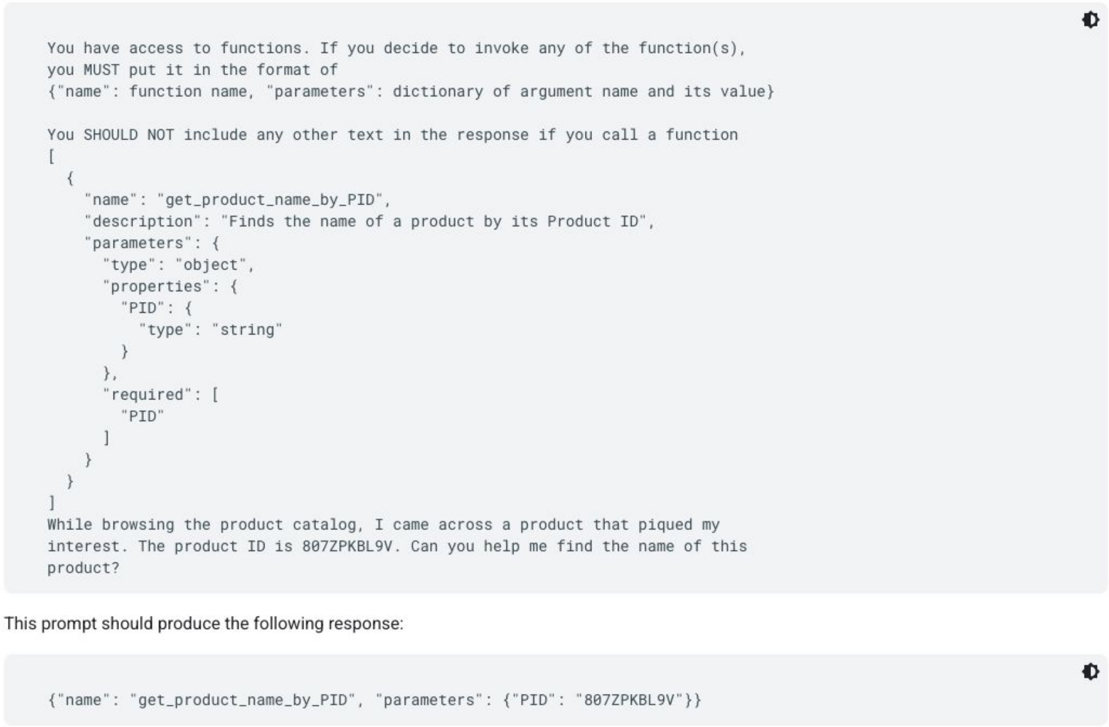
Post training

- Trade-off
    - Complexity and Speed
    - Open-ended and Close-ended
    - Accuracy and Speed

Why ?

- Function calling
- Text-to-SQL
- Long-context
- Low latency.

Function calling

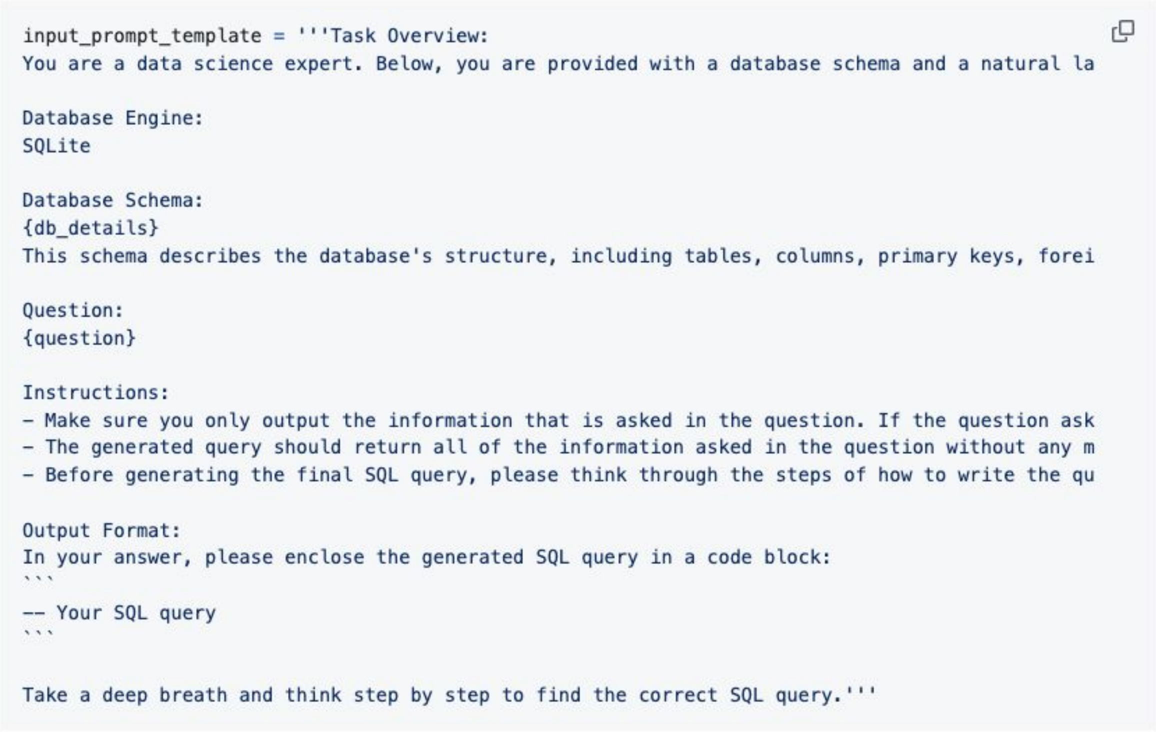


```
You have access to functions. If you decide to invoke any of the function(s),
you MUST put it in the format of
{"name": function name, "parameters": dictionary of argument name and its value}

You SHOULD NOT include any other text in the response if you call a function
[
  {
    "name": "get_product_name_by_PID",
    "description": "Finds the name of a product by its Product ID",
    "parameters": {
      "type": "object",
      "properties": {
        "PID": {
          "type": "string"
        }
      },
      "required": [
        "PID"
      ]
    }
  }
]
While browsing the product catalog, I came across a product that piqued my
interest. The product ID is 807ZPKBL9V. Can you help me find the name of this
product?
```

This prompt should produce the following response:

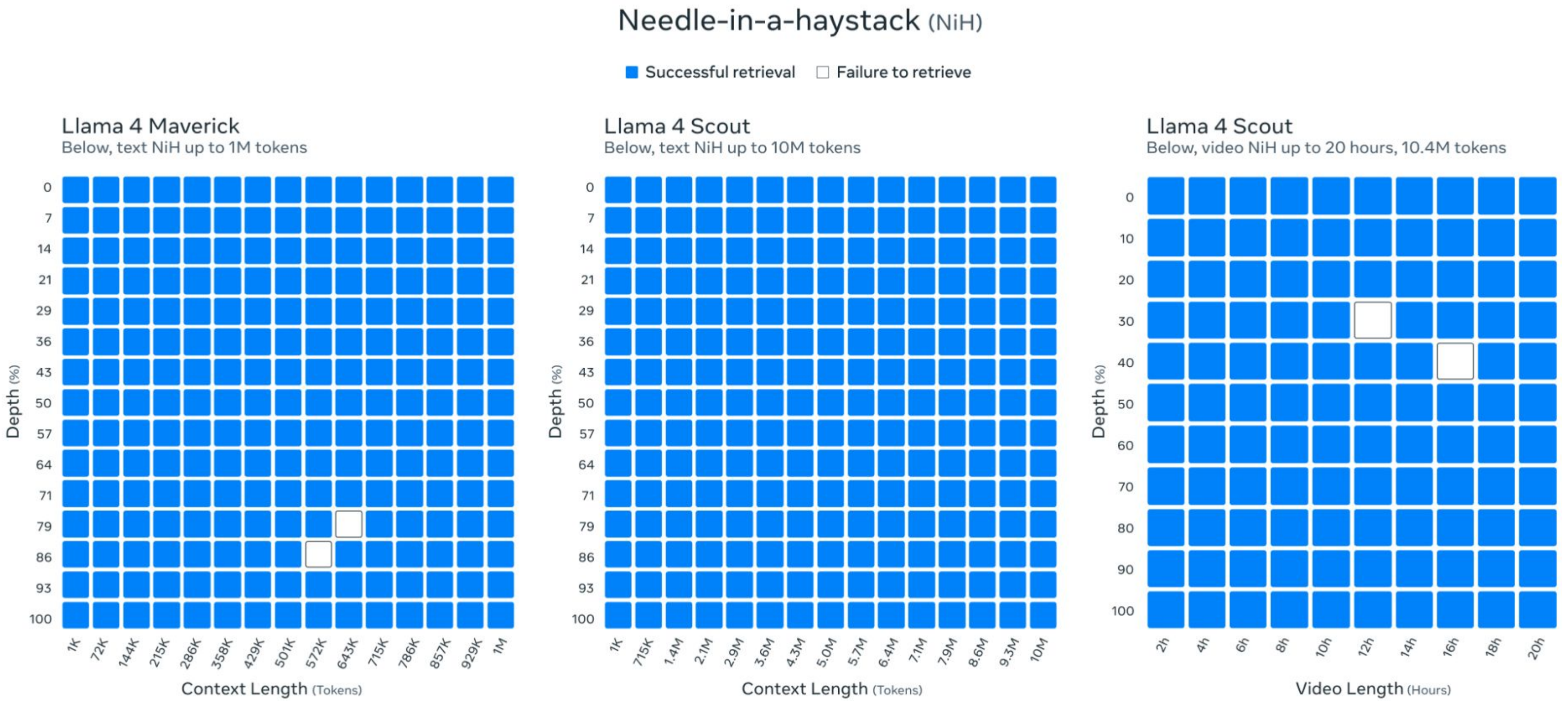
```
{"name": "get_product_name_by_PID", "parameters": {"PID": "807ZPKBL9V"}}
```

Text-to-SQL



~~~
input_prompt_template = '''Task Overview:
You are a data science expert. Below, you are provided with a database schema and a natural la

Database Engine:
SQLite

Database Schema:
{db_details}
This schema describes the database's structure, including tables, columns, primary keys, forei

Question:
{question}

Instructions:
- Make sure you only output the information that is asked in the question. If the question ask
- The generated query should return all of the information asked in the question without any m
- Before generating the final SQL query, please think through the steps of how to write the qu

Output Format:
In your answer, please enclose the generated SQL query in a code block:
```
-- Your SQL query
```

Take a deep breath and think step by step to find the correct SQL query.'''
~~~

Long-context



Needle-in-a-haystack (NiH)

Llama 4 Maverick

Below, text NiH up to 1M tokens

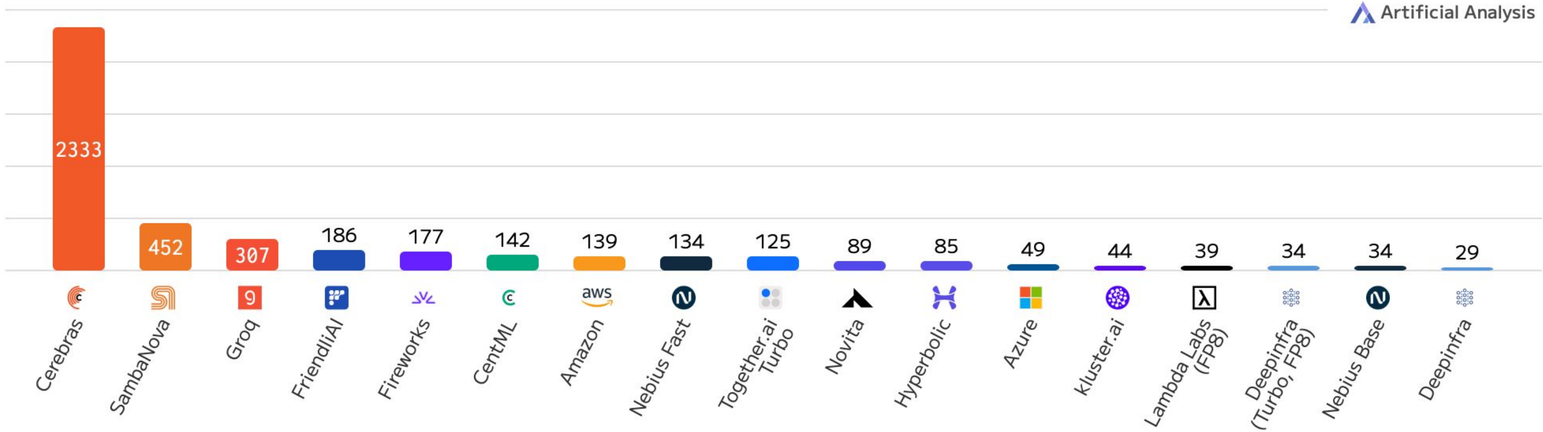
Llama 4 Scout

Below, text NiH up to 10M tokens

Llama 4 Scout

Below, video NiH up to 20 hours, 10.4M tokens

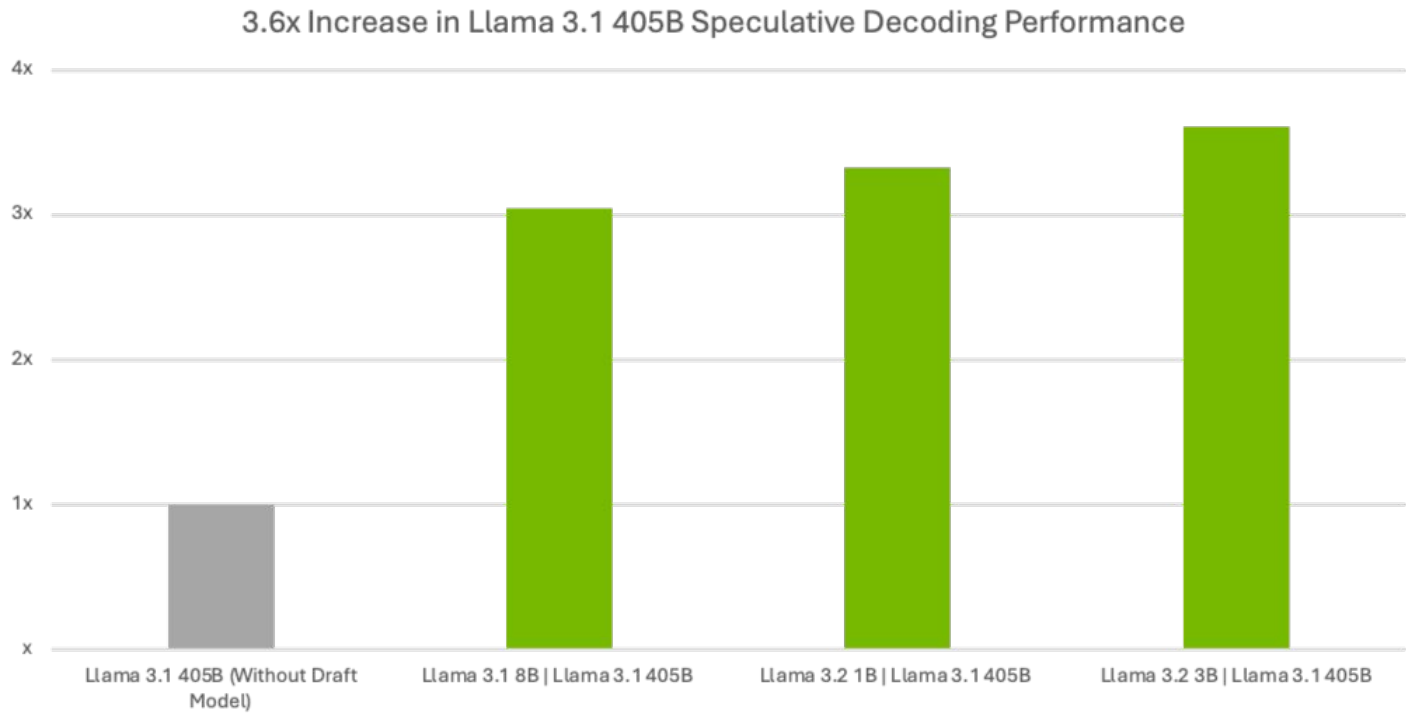
Low latency



Output Speed: Llama 3.3 70B Providers

Output Tokens per Second; Higher is better

Speculative decoding (1)



Speculative decoding (2)

- Model only
- Draft - Target model
  - Same family
  - Re-training

Speculative decoding (3)

Model only

(a) ข้าวผัดกะเพ __

(b) กรุง __

Speculative decoding (4)

Model only

Input : *“สั่งผัดกะเพราหมูแต่ได้ผัดถั่วฝักยาวมาแทน หรือแม้แต่กว่าจะตักเข้าปากได้แต่ละคำก็ต้องใช้เวลาเขี่ยพริกควานหาหมูสับอยู่นาน แต่มื้อนี้ไม่ต้องแล้วค่ะ เพราะเราจะมาทำข้าวผัดกะเพราหมูสับแบบของแท้ กะเพราจริง ๆ ไม่อิงผักใด ๆ ด้วยวิธีง่าย ๆ แต่ทำให้อร่อยได้ทุกคำที่ตักเข้าปาก รับประทานกับไข่ดาวกรอบ ๆ”*

ข้อความดังกล่าวเป็นเมนูอะไร

Output : ข้าวผัดกะเพ __

Speculative decoding (5)

Model only

N-gram decoding

- LlamaPromptLookupDecoding (llamacpp)
- Prompt lookup decoding (vllm, tensorrt-llm)

Speculative decoding (6)

Model only

Output : ข้าวผัดกะเพ __

Speculative decoding (7)

Model only

Iteration#1

input (without lookup) : ข้าวผัดกะเพ

output (without lookup) : ข้าวผัดกะเพรา

Iteration#2

input (without lookup) : ข้าวผัดกะเพรา

output (without lookup) : ข้าวผัดกะเพรา<eos>

Accpect

input (with lookup) : ข้าวผัดกะเพรา

output (with lookup) : ข้าวผัดกะเพรา<eos>

Reject

input (with lookup) : กรุงเทพ

output (with lookup) : กรุงศรีอยุธยา

Speculative decoding (8)

Same familty

กรุงเทพมหานคร อมรรัตนโกสินทร์ มหินทรายุธยา มหาดิลก

ภพ นพรัตนราชธานีบูรีรมย์

Speculative decoding (9)

Same familty

กรุงเทพมหา   อมรรัตนโกสินทร์ มหินทรายุธยา มหาดิลก

ภพ นพรัตนราชธานีบูรีรมย์

กรุงเทพมหานคร อมรรัตนโกสินทร์   ยุธยา มหาดิลก

ภพ นพรัตนราชธานีบูรีรมย์

Speculative decoding (10)

Same familty

model-1b

Iteration#1

input : กรุงเทพมหา

output : กรุงเทพมหานคร

Iteration#2

input : กรุงเทพมหานคร

output : กรุงเทพมหานคร <eos>

model-70b

Iteration#1

input : กรุงเทพมหานคร <eos>

output : กรุงเทพมหานคร อมร

Verify

input : กรุงเทพมหานคร <eos>

output : กรุงเทพมหานคร อมร

model-1b

Iteration#3

input : กรุงเทพมหานคร อมร

output : กรุงเทพมหานคร อมรรัตน

Speculative decoding (11)

Same familty

Model size	x1	x2	x3	x4	x5
	กรุง	เทพ	มหา	นคร	<eos>
1b probability	0.5	0.6	0.8	0.5	0.9
70b probability	0.7	0.8	0.85	0.6	0.3

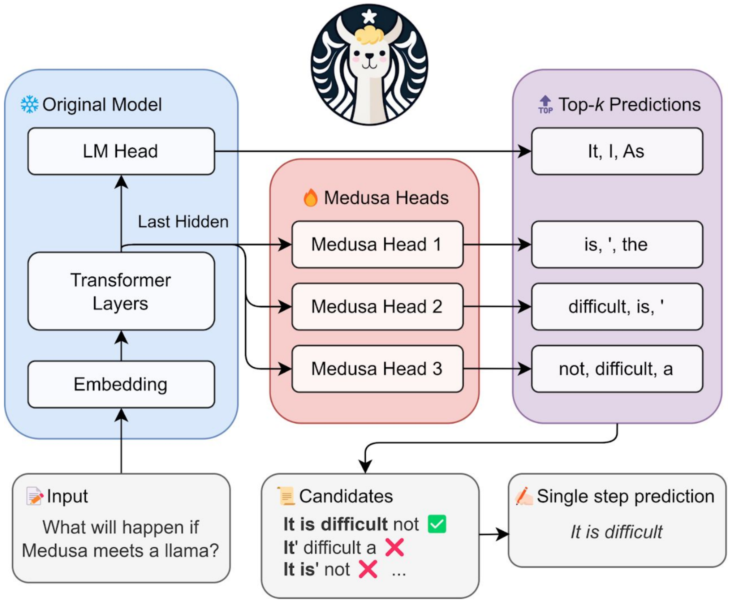
Speculative decoding (12)

Re-training

- Medusa
- Eagle

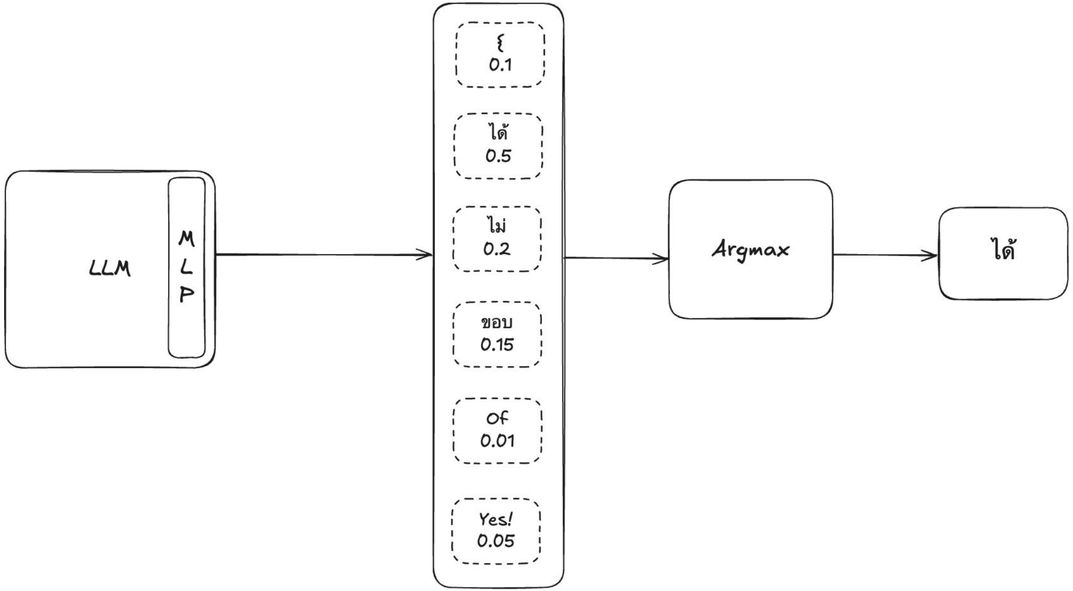
Speculative decoding (13)

Medusa



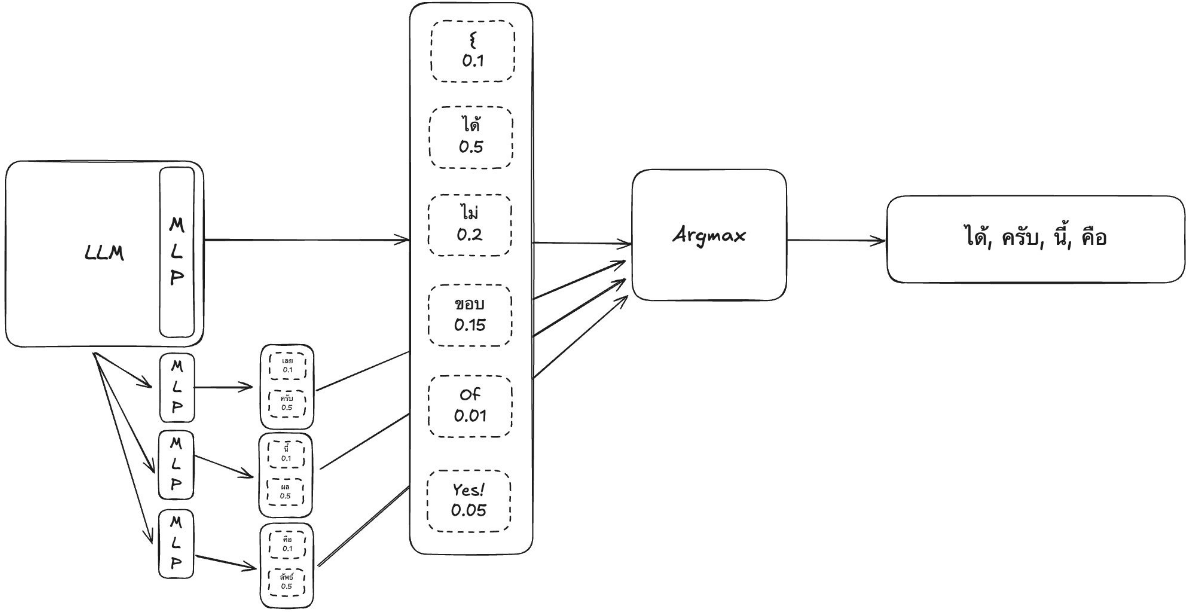
Speculative decoding (14)

Medusa



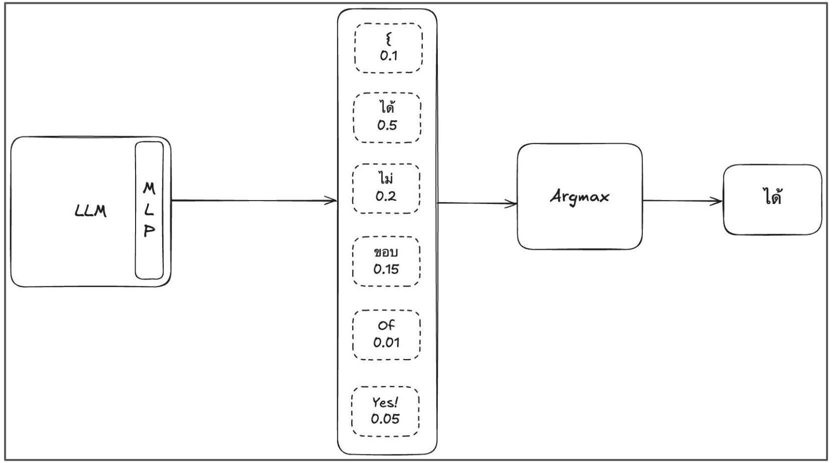
Speculative decoding (15)

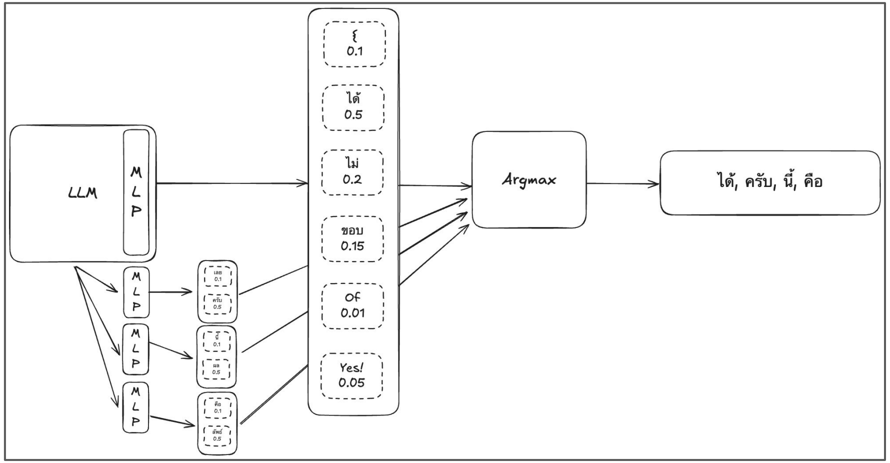
Medusa



Speculative decoding (16)

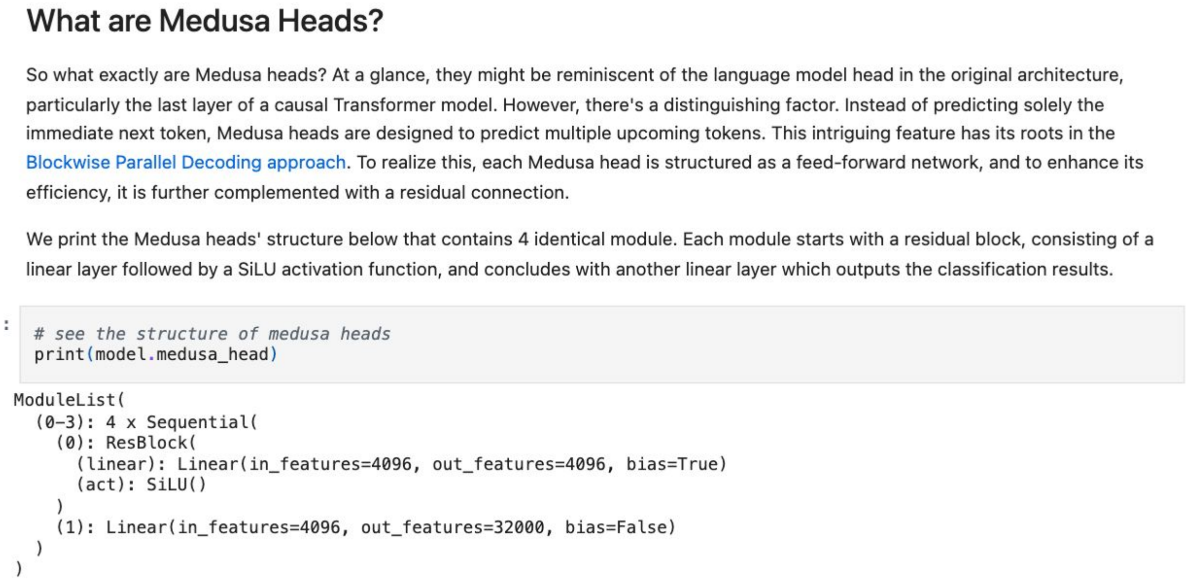
Medusa





Speculative decoding (17)

Medusa



What are Medusa Heads?

So what exactly are Medusa heads? At a glance, they might be reminiscent of the language model head in the original architecture, particularly the last layer of a causal Transformer model. However, there's a distinguishing factor. Instead of predicting solely the immediate next token, Medusa heads are designed to predict multiple upcoming tokens. This intriguing feature has its roots in the Blockwise Parallel Decoding approach. To realize this, each Medusa head is structured as a feed-forward network, and to enhance its efficiency, it is further complemented with a residual connection.

We print the Medusa heads' structure below that contains 4 identical module. Each module starts with a residual block, consisting of a linear layer followed by a SiLU activation function, and concludes with another linear layer which outputs the classification results.

```
# see the structure of medusa heads
print(model.medusa_head)
```

```
ModuleList(
  (0-3): 4 x Sequential(
    (0): ResBlock(
      (linear): Linear(in_features=4096, out_features=4096, bias=True)
      (act): SiLU()
    )
    (1): Linear(in_features=4096, out_features=32000, bias=False)
  )
)
```

Guided decoding (0)

Function calling and Text-to-SQL problem

- Correct format
- Right choice

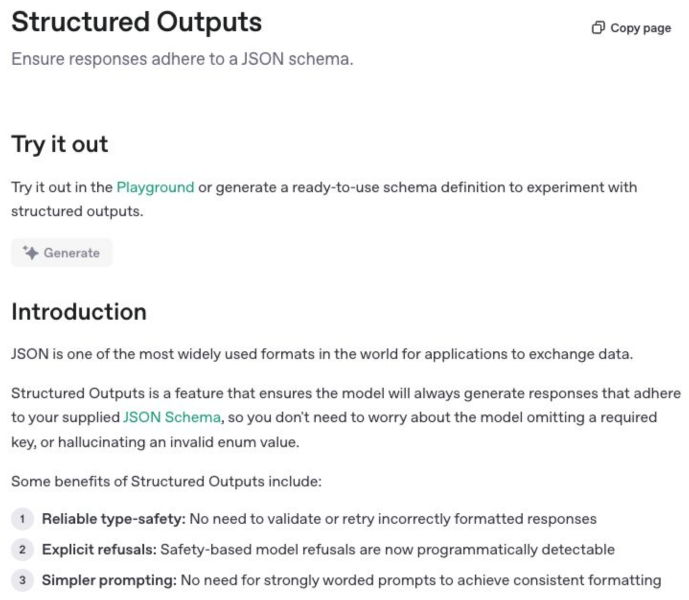
Guided decoding (1)

Function calling and Text-to-SQL problem

- Correct format (Solved with Guided decoding)
- Right choice

Guided decoding (2)

JSON mode, Bias, Grammar-free



Structured Outputs

Copy page

Ensure responses adhere to a JSON schema.

Try it out

Try it out in the Playground or generate a ready-to-use schema definition to experiment with structured outputs.

Generate

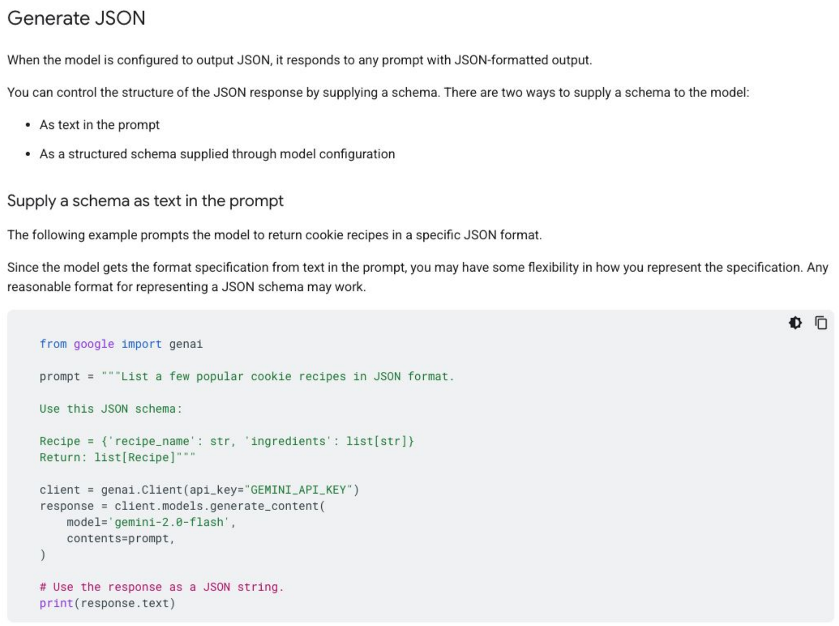
Introduction

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.

Some benefits of Structured Outputs include:

1. **Reliable type-safety:** No need to validate or retry incorrectly formatted responses
2. **Explicit refusals:** Safety-based model refusals are now programmatically detectable
3. **Simpler prompting:** No need for strongly worded prompts to achieve consistent formatting



Generate JSON

When the model is configured to output JSON, it responds to any prompt with JSON-formatted output.

You can control the structure of the JSON response by supplying a schema. There are two ways to supply a schema to the model:

- As text in the prompt
- As a structured schema supplied through model configuration

Supply a schema as text in the prompt

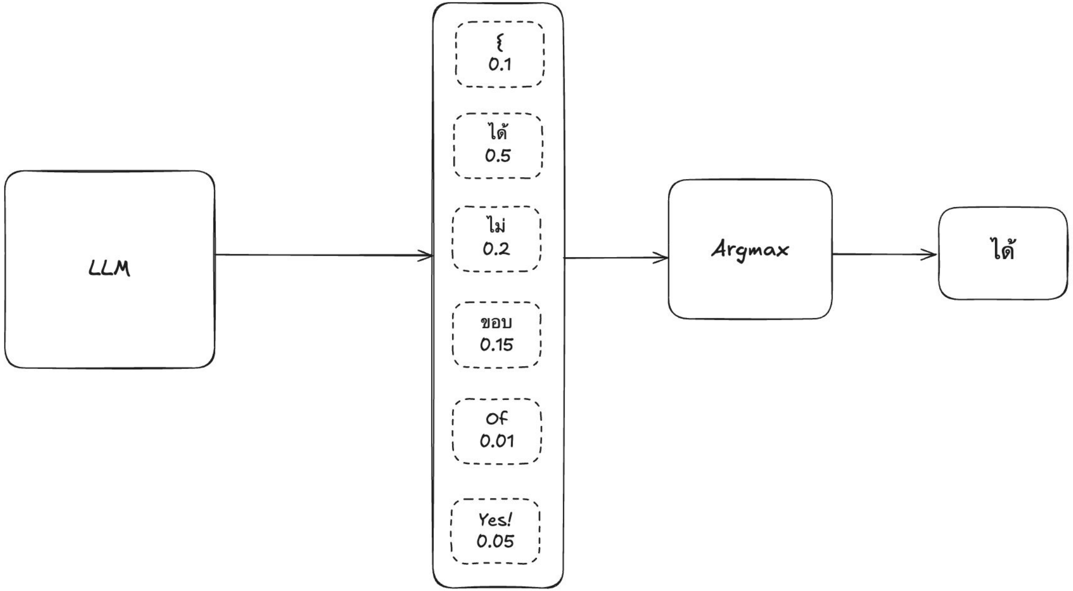
The following example prompts the model to return cookie recipes in a specific JSON format.

Since the model gets the format specification from text in the prompt, you may have some flexibility in how you represent the specification. Any reasonable format for representing a JSON schema may work.

```
from google import genai

prompt = """List a few popular cookie recipes in JSON format.

Use this JSON schema:

Recipe = {'recipe_name': str, 'ingredients': list[str]}
Return: list[Recipe]"""

client = genai.Client(api_key="GEMINI_API_KEY")
response = client.models.generate_content(
    model='gemini-2.0-flash',
    contents=prompt,
)

# Use the response as a JSON string.
print(response.text)
```

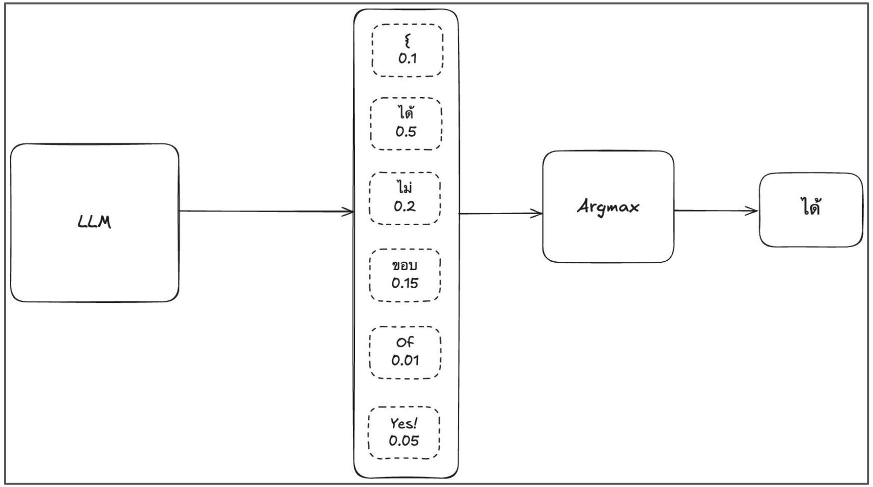
Guided decoding (3)

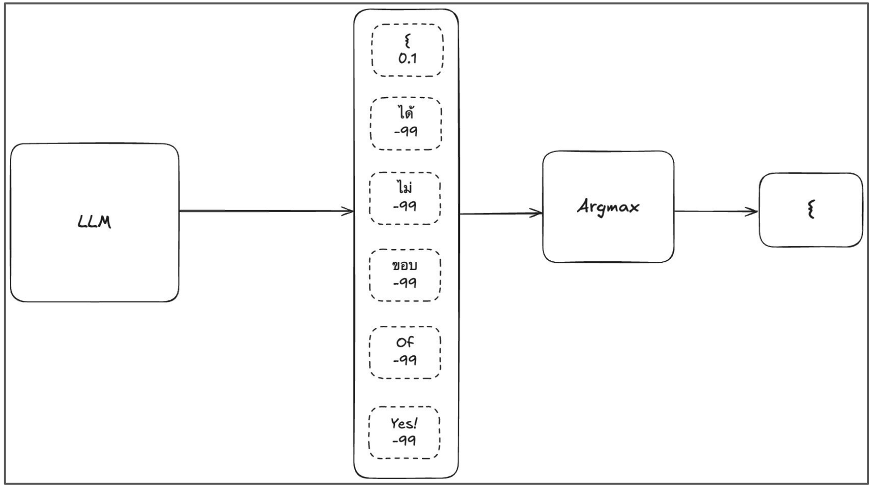
Original LLM



Guided decoding (4)

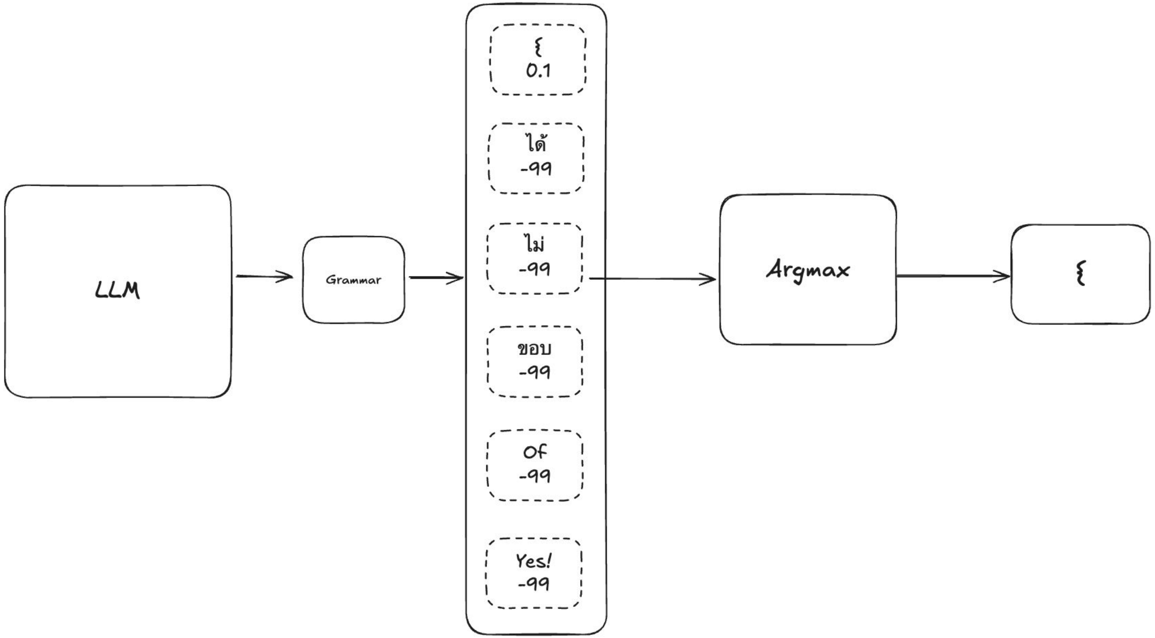
Manipulate probability





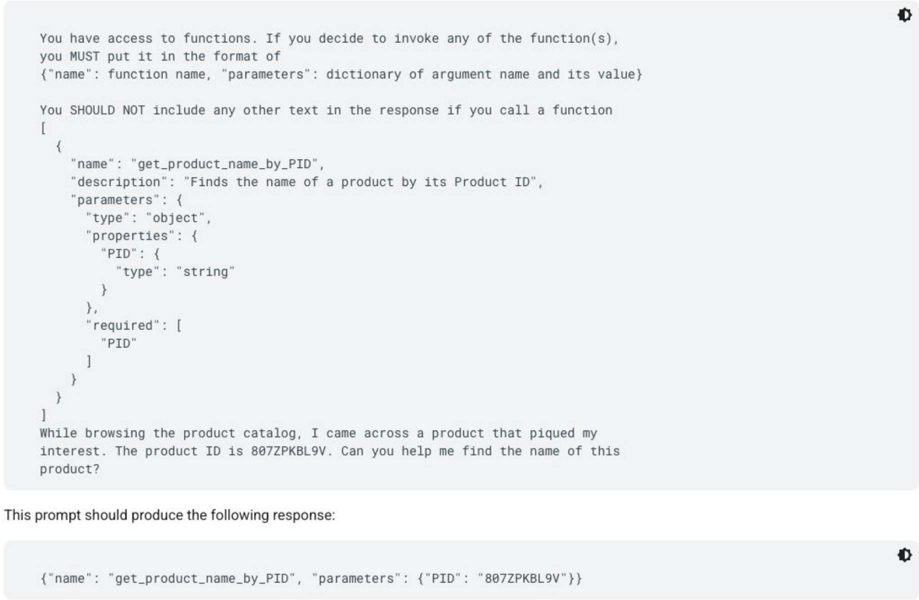
Guided decoding (5)

Grammar



Guided decoding (6)

Function calling



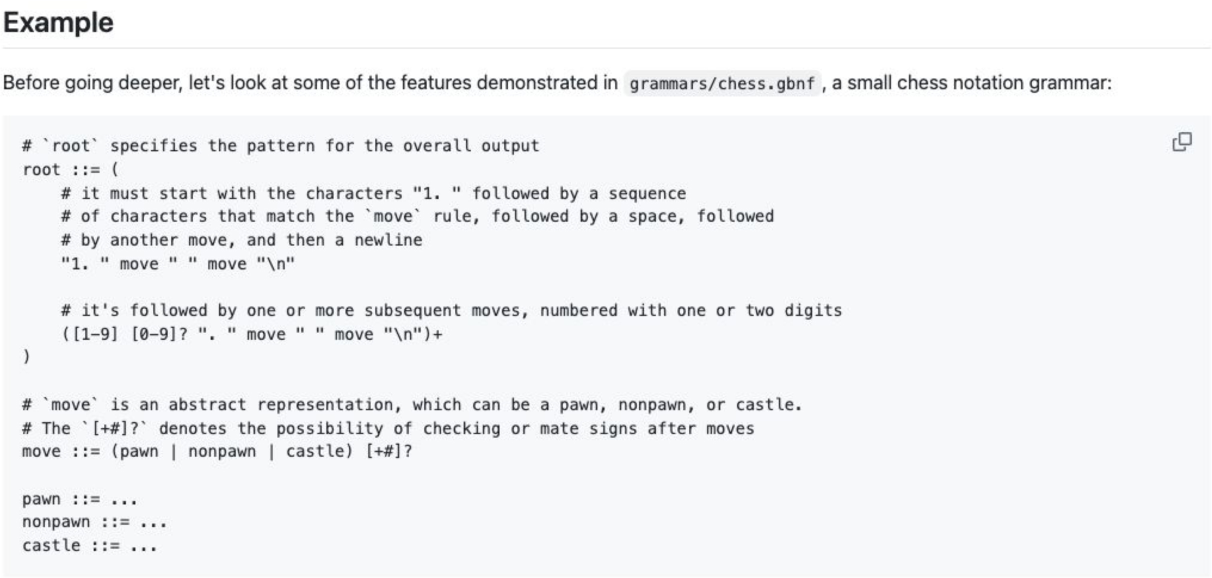
```
You have access to functions. If you decide to invoke any of the function(s),
you MUST put it in the format of
{"name": function name, "parameters": dictionary of argument name and its value}

You SHOULD NOT include any other text in the response if you call a function
[
  {
    "name": "get_product_name_by_PID",
    "description": "Finds the name of a product by its Product ID",
    "parameters": {
      "type": "object",
      "properties": {
        "PID": {
          "type": "string"
        }
      },
      "required": [
        "PID"
      ]
    }
  }
]
While browsing the product catalog, I came across a product that piqued my
interest. The product ID is 807ZPKBL9V. Can you help me find the name of this
product?
```

This prompt should produce the following response:

```
{"name": "get_product_name_by_PID", "parameters": {"PID": "807ZPKBL9V"}}
```

Guided decoding (7)

Function calling (llamacpp)



Example

Before going deeper, let's look at some of the features demonstrated in `grammars/chess.gbnf`, a small chess notation grammar:

```
# `root` specifies the pattern for the overall output
root ::= (
    # it must start with the characters "1. " followed by a sequence
    # of characters that match the `move` rule, followed by a space, followed
    # by another move, and then a newline
    "1. " move " " move "\n"

    # it's followed by one or more subsequent moves, numbered with one or two digits
    ([1-9] [0-9]? ". " move " " move "\n")+
)

# `move` is an abstract representation, which can be a pawn, nonpawn, or castle.
# The `[+#]?` denotes the possibility of checking or mate signs after moves
move ::= (pawn | nonpawn | castle) [+#]?

pawn ::= ...
nonpawn ::= ...
castle ::= ...
```

Guided decoding (8)

Function calling (llamacpp)

Select one of the following.

```
root := (“กะเพรา”|“ข้าวมันไก่”|“ข้าวไข่เจียว”)
```

Allow only Thai and English language.

```
root := [a-zA-Zก-๙0-9\s]+
```

Guided decoding (9)

Function calling (llamacpp)

SQL statement.

```
root ::= (stopgen|gen)
stopgen ::= ("NONE")
gen ::= ("```sql\nSELECT " attr " FROM monthly_sale_record " wherecond "```")
attr ::= ("*"|"month"|"material_code"|"revenue") +
wherecond ::= (""|"WHERE "tablerow" "whereops" '"alphanum"'")
tablerow ::= ("month"|"material_code"|"revenue")
alphanum ::= [a-zA-Z0-9_]+
whereops ::= ("="|"!="|"<"|"<="|">"|">=")
```

Guided decoding (10)

Library

- xgrammar
- lm-format-enforcer
- outlines
- guidance

Extra.

Reduce End-to-End latency !!!

Let’s say we need to inference 3 times.

Original

1 (tools select, 30 token) -> 2 (execute tools, 50 token) -> 3 (refine output, 200 token)

Extra.

Reduce End-to-End latency !!!

Step 1

Boost base throughput with speculative decoding and quantization.

H100, 32b, 30 token per sec -> H100, 32b, 120 token per sec

Step 2

Reduce token needed with grammar.

Reduce about 33% - 50% with vanilla model.

Extra.

Reduce End-to-End latency !!!

Let’s say we need to inference 3 times.

Original

1 (tools select, 30 token) -> 2 (execute tools, 50 token) -> 3 (refine output, 200 token)

inference time = 1 sec + 2 sec + 7 sec = 10 sec

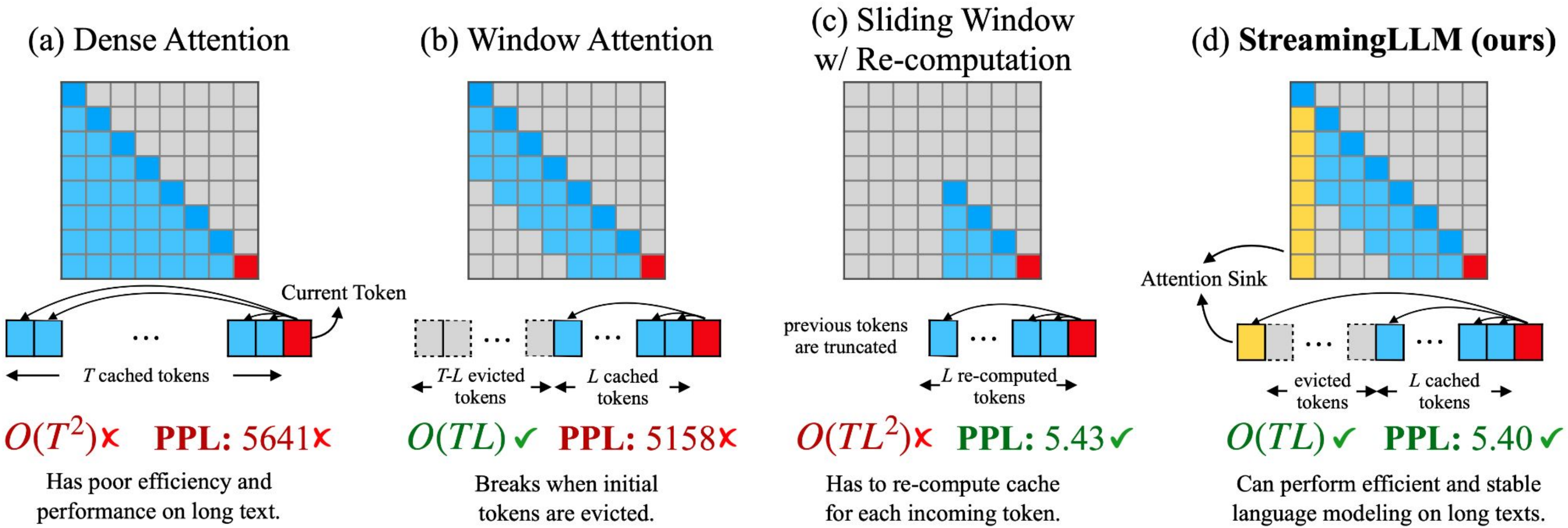
Refined

1 (tools select, 10 token) -> 2 (execute tools, 30 token) -> 3 (refine output, 200 token)

inference time = 0.1 sec + 0.3 sec + 2 sec = 2.4 sec

Long-context LLM (1)

Streaming LLM (Attention sink)


(a) Dense Attention
Current Token
...
T cached tokens
$O(T^2)$ ✗ PPL: 5641 ✗
Has poor efficiency and performance on long text.
(b) Window Attention
...
T-L evicted tokens
L cached tokens
$O(TL)$ ✓ PPL: 5158 ✗
Breaks when initial tokens are evicted.
(c) Sliding Window w/ Re-computation
previous tokens are truncated
...
L re-computed tokens
$O(TL^2)$ ✗ PPL: 5.43 ✓
Has to re-compute cache for each incoming token.
(d) **StreamingLLM (ours)**
Attention Sink
...
evicted tokens
L cached tokens
$O(TL)$ ✓ PPL: 5.40 ✓
Can perform efficient and stable language modeling on long texts.

Long-context LLM (2)

KV cache size (Llama2-70b)

bs = 1, seq = 512, vram = 1.25 GB
bs = 1, seq = 4096, vram = 10 GB
bs = 16, seq = 4096, vram = 160 GB

batch_size * layer * kv-heads * n_emd * length * K&V * FP16

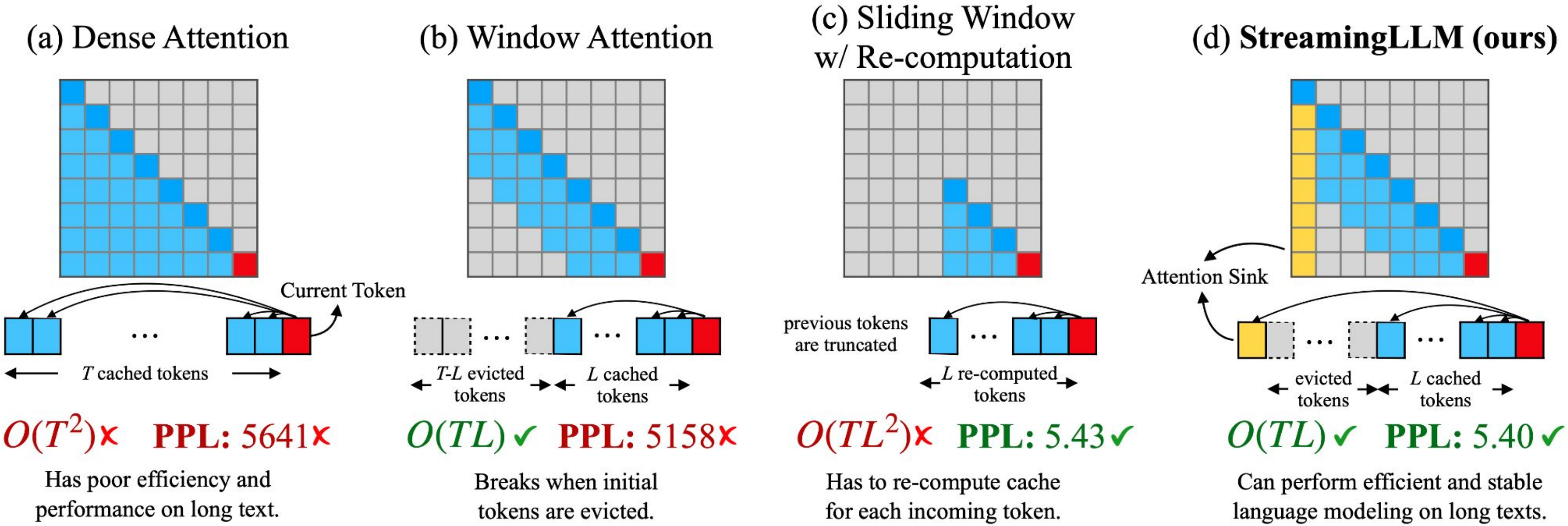
Long-context LLM (3)

Streaming LLM (Attention sink)

Deploying Large Language Models (LLMs) in streaming applications such as multi-round dialogue, where long interactions are expected, is urgently needed but poses two major challenges. Firstly, during the decoding stage, caching previous tokens' Key and Value states (KV) consumes extensive memory. Secondly, popular LLMs cannot generalize to longer texts than the training sequence length. Window attention, where only the most recent KVs are cached, is a natural approach — but we show that it fails when the text length surpasses the cache size. We observe an interesting phenomenon, namely attention sink, that keeping the KV of initial tokens will largely recover the performance of window attention. In this paper, we first demonstrate that the emergence of attention sink is due to the strong attention scores towards initial tokens as a "sink" even if they are not semantically important. Based on the above analysis, we introduce StreamingLLM, an efficient framework that enables LLMs trained with a finite length attention window to generalize to infinite sequence length without any fine-tuning. We show that StreamingLLM can enable Llama-2, MPT, Falcon, and Pythia to perform stable and efficient language modeling with up to 4 million tokens and more. In addition, we discover that adding a placeholder token as a dedicated attention sink during pre-training can further improve streaming deployment. In streaming settings, StreamingLLM outperforms the sliding window recomputation baseline by up to 22.2× speedup. Code and datasets are provided in the link.

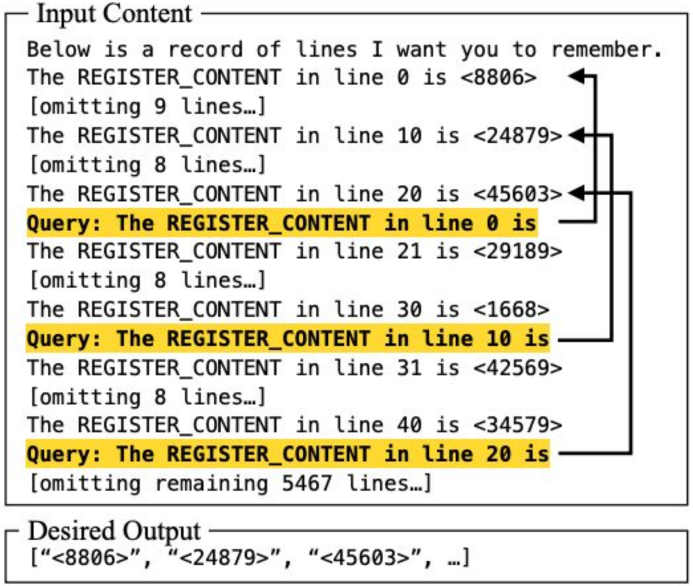
Long-context LLM (4)

Streaming LLM (Attention sink)



Long-context LLM (5)

Scenario



Input Content

```
Below is a record of lines I want you to remember.
The REGISTER_CONTENT in line 0 is <8806>
[omitting 9 lines…]
The REGISTER_CONTENT in line 10 is <24879>
[omitting 8 lines…]
The REGISTER_CONTENT in line 20 is <45603>
Query: The REGISTER_CONTENT in line 0 is
The REGISTER_CONTENT in line 21 is <29189>
[omitting 8 lines…]
The REGISTER_CONTENT in line 30 is <1668>
Query: The REGISTER_CONTENT in line 10 is
The REGISTER_CONTENT in line 31 is <42569>
[omitting 8 lines…]
The REGISTER_CONTENT in line 40 is <34579>
Query: The REGISTER_CONTENT in line 20 is
[omitting remaining 5467 lines…]
```

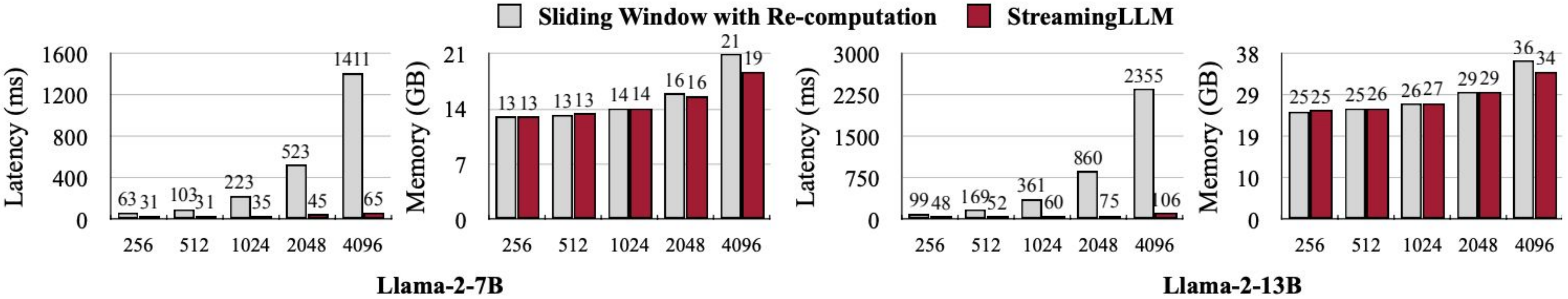
Desired Output

```
["<8806>", "<24879>", "<45603>", …]
```

Figure 8: The first sample in StreamEval.

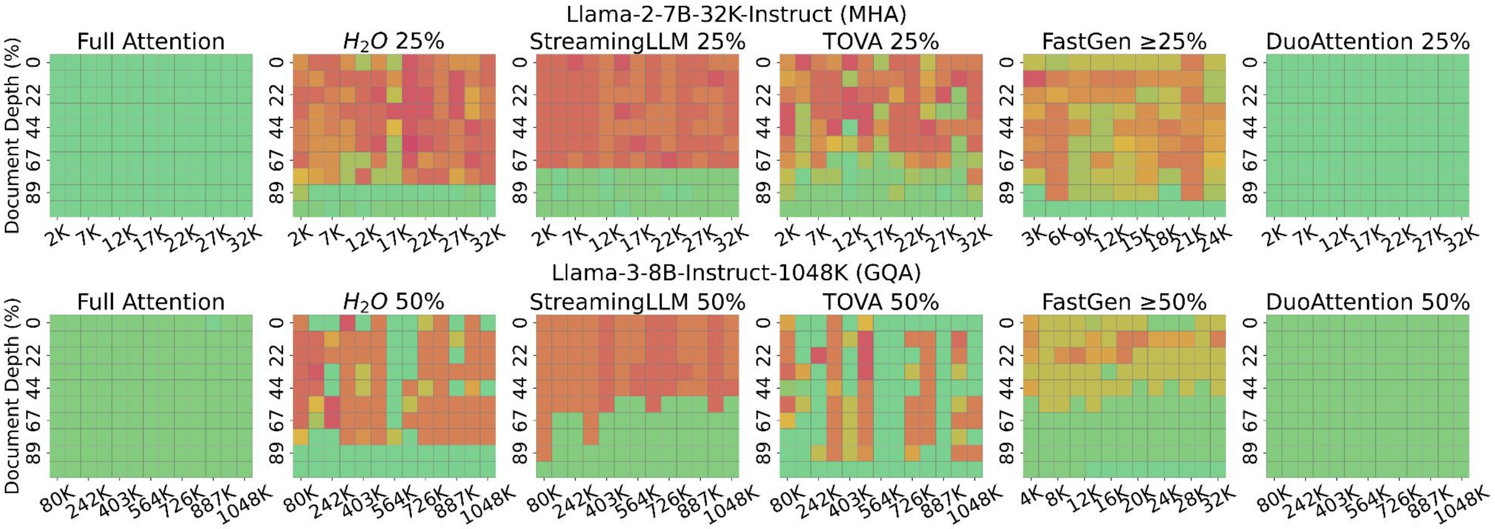
Long-context LLM (6)

Performance



Long-context LLM (7)

Duo attention



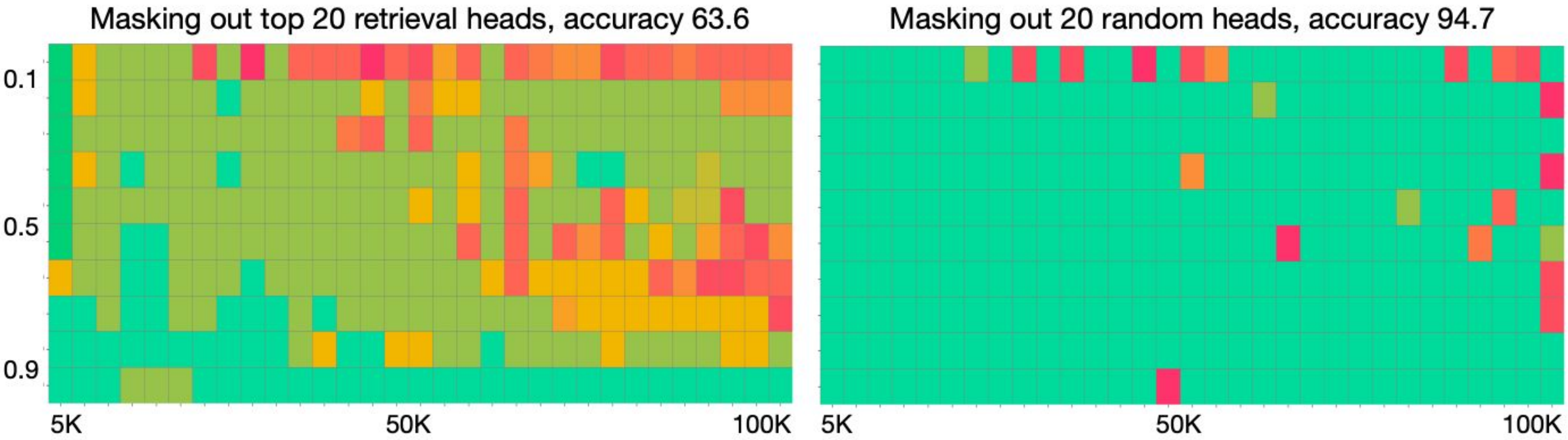
Long-context LLM (8)

Duo attention

Deploying long-context large language models (LLMs) is essential but poses significant computational and memory challenges. Caching all Key and Value (KV) states across all attention heads consumes substantial memory. Existing KV cache pruning methods either damage the long-context capabilities of LLMs or offer only limited efficiency improvements. In this paper, we identify that only a fraction of attention heads, a.k.a, Retrieval Heads, are critical for processing long contexts and require full attention across all tokens. In contrast, all other heads, which primarily focus on recent tokens and attention sinks–referred to as Streaming Heads–do not require full attention. Based on this insight, we introduce DuoAttention, a framework that only applies a full KV cache to retrieval heads while using a light-weight, constant-length KV cache for streaming heads, which reduces both LLM's decoding and pre-filling memory and latency without compromising its long-context abilities. DuoAttention uses a lightweight, optimization-based algorithm with synthetic data to identify retrieval heads accurately.

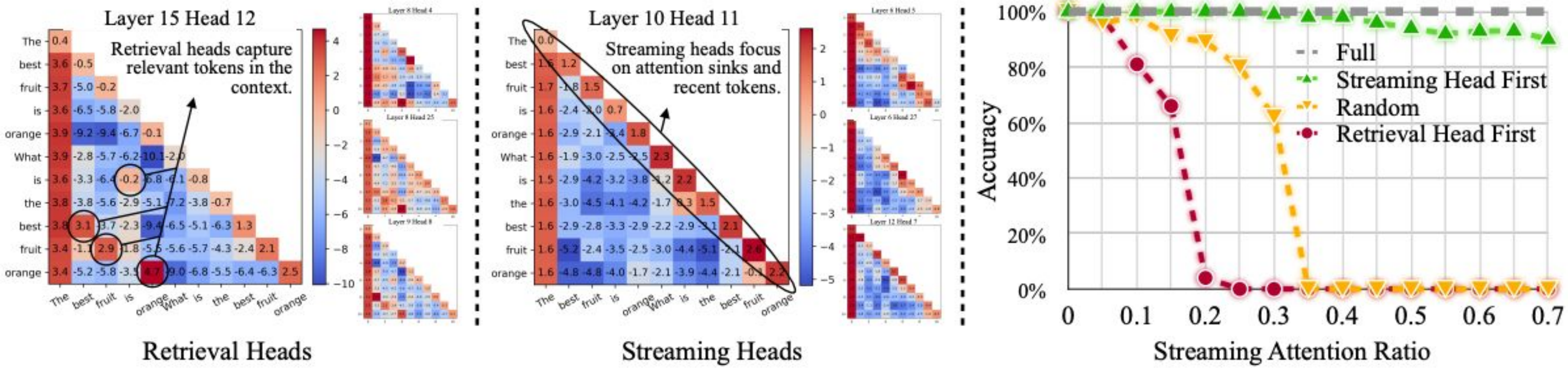
Long-context LLM (9)

Retrieval heads



Long-context LLM (10)

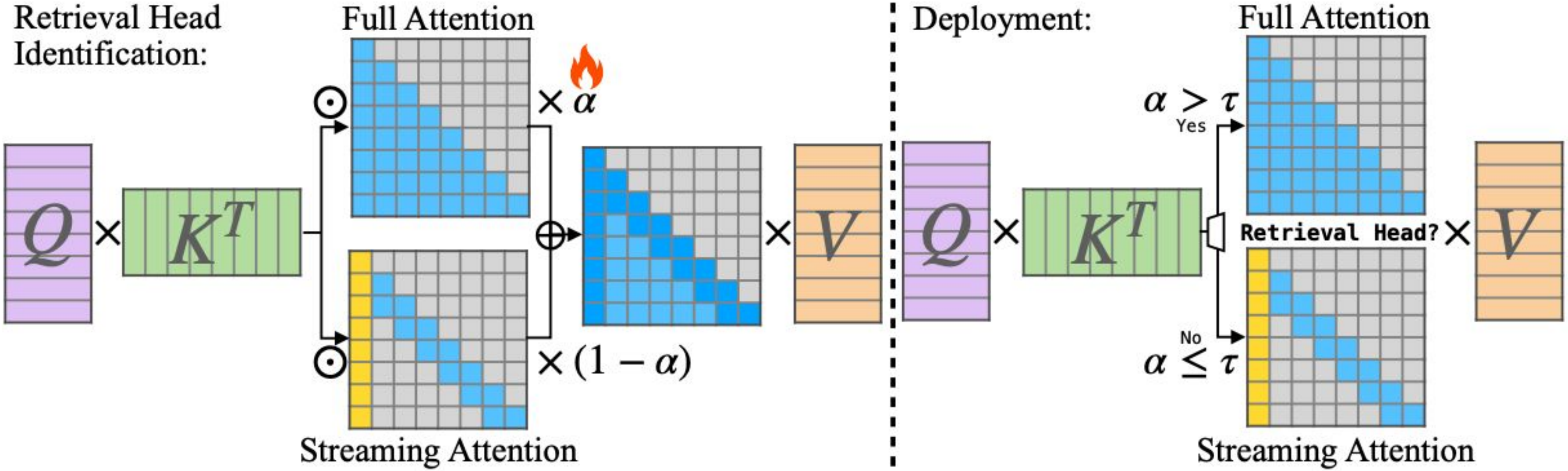
Duo attention


Retrieval heads capture relevant tokens in the context.

Streaming heads focus on attention sinks and recent tokens.

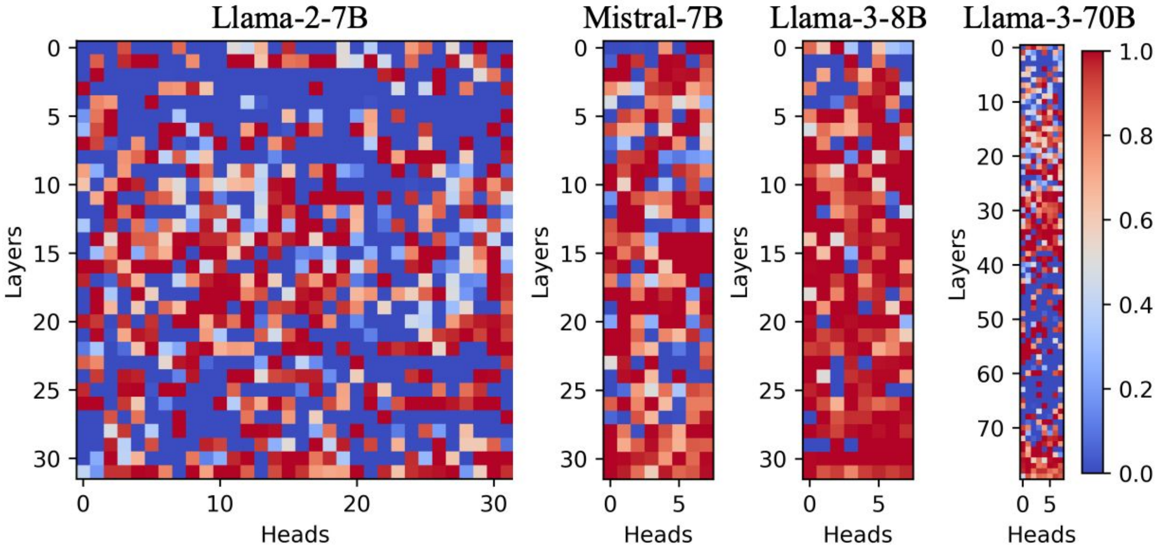
Long-context LLM (11)

Duo attention

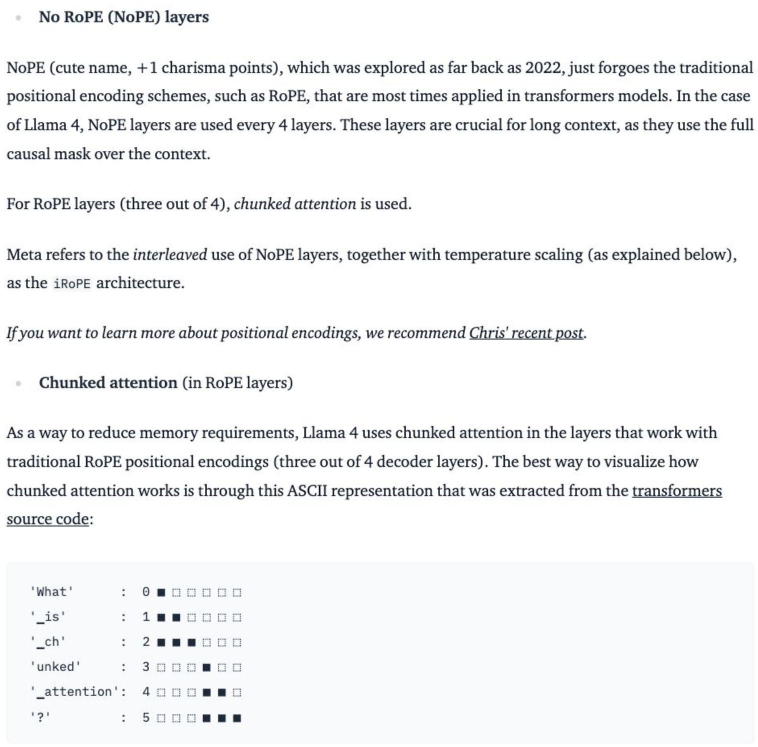


Long-context LLM (12)

Retrieval heads



Extra Llama 4



- **No RoPE (NoPE) layers**

NoPE (cute name, +1 charisma points), which was explored as far back as 2022, just forgoes the traditional positional encoding schemes, such as RoPE, that are most times applied in transformers models. In the case of Llama 4, NoPE layers are used every 4 layers. These layers are crucial for long context, as they use the full causal mask over the context.

For RoPE layers (three out of 4), *chunked attention* is used.

Meta refers to the *interleaved* use of NoPE layers, together with temperature scaling (as explained below), as the `iRoPE` architecture.

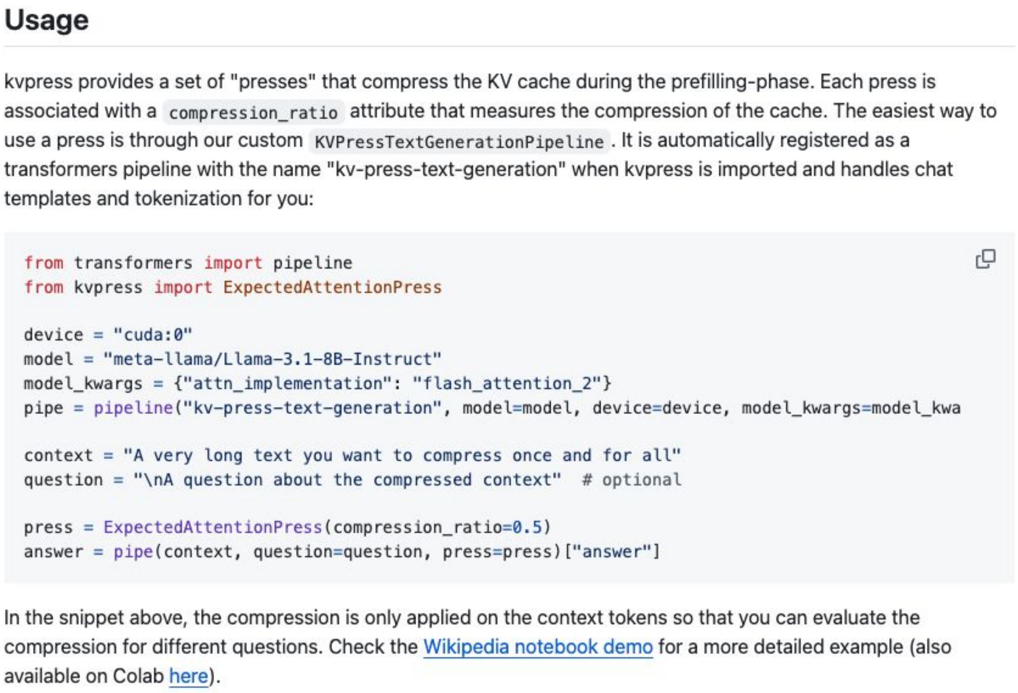
If you want to learn more about positional encodings, we recommend Chris' recent post.

- **Chunked attention** (in RoPE layers)

As a way to reduce memory requirements, Llama 4 uses chunked attention in the layers that work with traditional RoPE positional encodings (three out of 4 decoder layers). The best way to visualize how chunked attention works is through this ASCII representation that was extracted from the transformers source code:

```
'What'      :  0 ■ ⬚ ⬚ ⬚ ⬚ ⬚
'_is'       :  1 ■ ■ ⬚ ⬚ ⬚ ⬚
'_ch'       :  2 ■ ■ ■ ⬚ ⬚ ⬚
'unked'     :  3 ⬚ ⬚ ⬚ ■ ⬚ ⬚
'_attention':  4 ⬚ ⬚ ⬚ ■ ■ ⬚
'?'         :  5 ⬚ ⬚ ⬚ ■ ■ ■
```

Library (kvpress)



Usage

kvpress provides a set of "presses" that compress the KV cache during the prefilling-phase. Each press is associated with a `compression_ratio` attribute that measures the compression of the cache. The easiest way to use a press is through our custom `KVPressTextGenerationPipeline`. It is automatically registered as a transformers pipeline with the name "kv-press-text-generation" when kvpress is imported and handles chat templates and tokenization for you:

```
from transformers import pipeline
from kvpress import ExpectedAttentionPress

device = "cuda:0"
model = "meta-llama/Llama-3.1-8B-Instruct"
model_kwargs = {"attn_implementation": "flash_attention_2"}
pipe = pipeline("kv-press-text-generation", model=model, device=device, model_kwargs=model_kwa

context = "A very long text you want to compress once and for all"
question = "\nA question about the compressed context"  # optional

press = ExpectedAttentionPress(compression_ratio=0.5)
answer = pipe(context, question=question, press=press)["answer"]
```

In the snippet above, the compression is only applied on the context tokens so that you can evaluate the compression for different questions. Check the Wikipedia notebook demo for a more detailed example (also available on Colab here).