Structured Generation and the New Trend

Yixin Dong

Carnegie Mellon University

Nov 4, 2025

Context

Part 01
Introduction

Large Language Model Generation with Structures

Problem: LLM Generation with Structures

Code generation

Language Code

Function/tool calling

JSON Schema

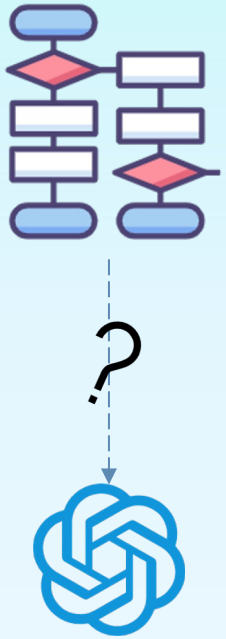
Embodied Agents

DSL (e.g. PDDL)

Structured Outputs

Fig: https://arxiv.org/pdf/2304.11477

Problem: LLM's Limited Ability with Complex Structures



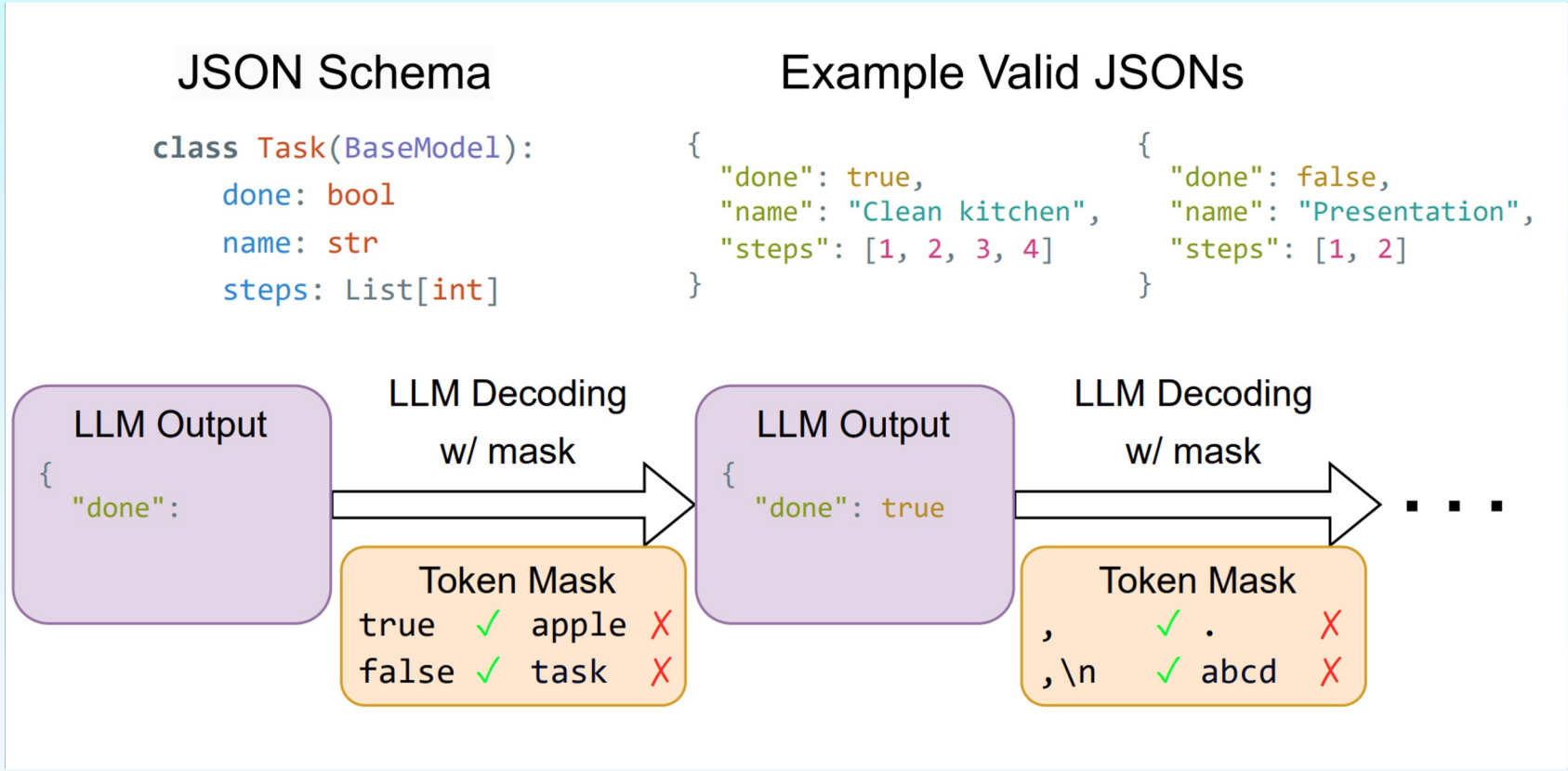
Structures are increasingly complex

- Advanced agents
- Complex tool calling
- DSLs unfamiliar to LLMs

LLM's generation ability is limited

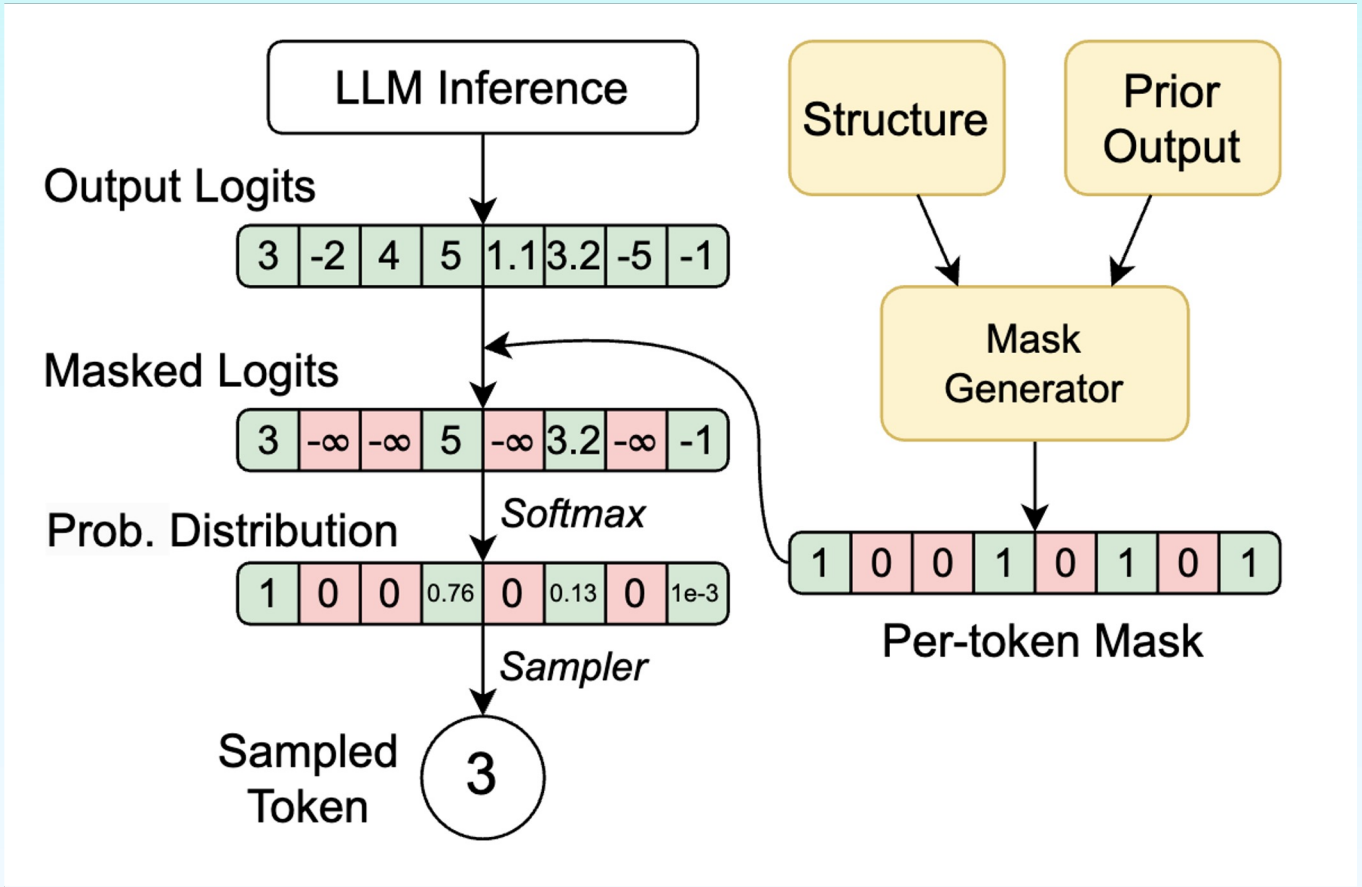
- On-device small LMs
- Compressed models

Background: Constrained Decoding



An example of constrained decoding with JSON Schema

Background: Constrained Decoding



Apply a per-token mask to prevent generating invalid tokens according to the structure

The overhead of the mask generator is crucial!

Integration with LLM Serving Frameworks

- XGrammar is designed for easy integration and cross-platform support (with C++, Python, and JavaScript APIs)
  - Its core is implemented in C++, so easy to port to other platforms
- XGrammar has already been integrated with vLLM, SGLang, MLC-LLM, WebLLM, VILA

XGrammar Open-Source Project

XGrammar has been adopted by these LLM serving engines:

...

And industrial collaborators:


...


Part 02

XGrammar v1

Flexible And Efficient Structured Generation Engine

XGrammar: Flexible and Efficient Structured Generation Engine

XGrammar is a structured generation library that features

Flexibility: Full support for context-free grammar

Efficiency: SOTA performance in constraint decoding

Zero-overhead JSON Schema generation

Integration: Easy to integrate with existing LLM serving frameworks

vLLM, MLC-LLM, SGLang, etc

More Powerful: Context-free Grammar

Context-free Grammar

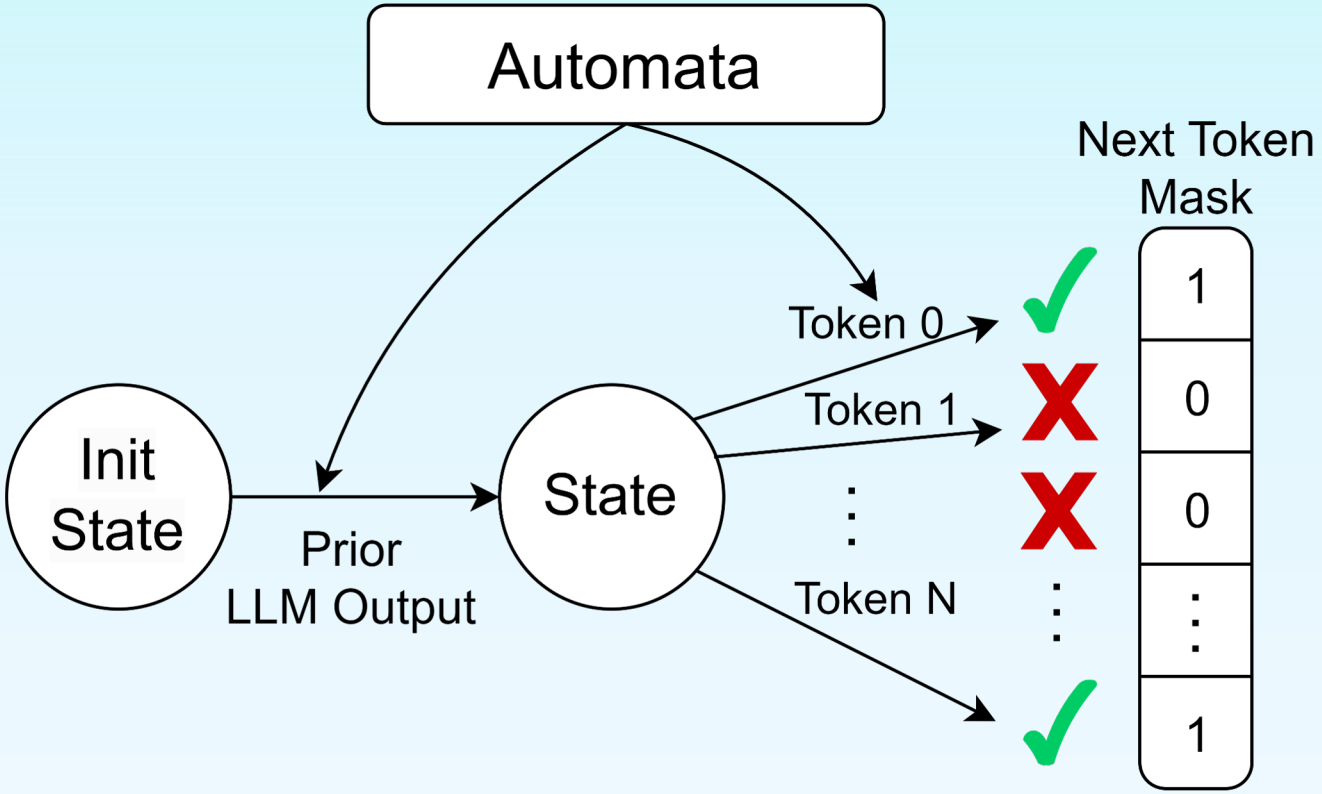
```
root ::= <array> | <str>

array ::=
  '[' (<str> | <array> ',')*
  <str> | <array> ']'

str ::= '"' [^"\]* '"'
```

Prior methods mainly support **regex** as input grammar

XGrammar support **the more powerful CFG**, therefore supporting

- Regex
- JSON, JSON Schema
- SQL
- Python (w/ additional state maintained)

Previous Mask Generation Method

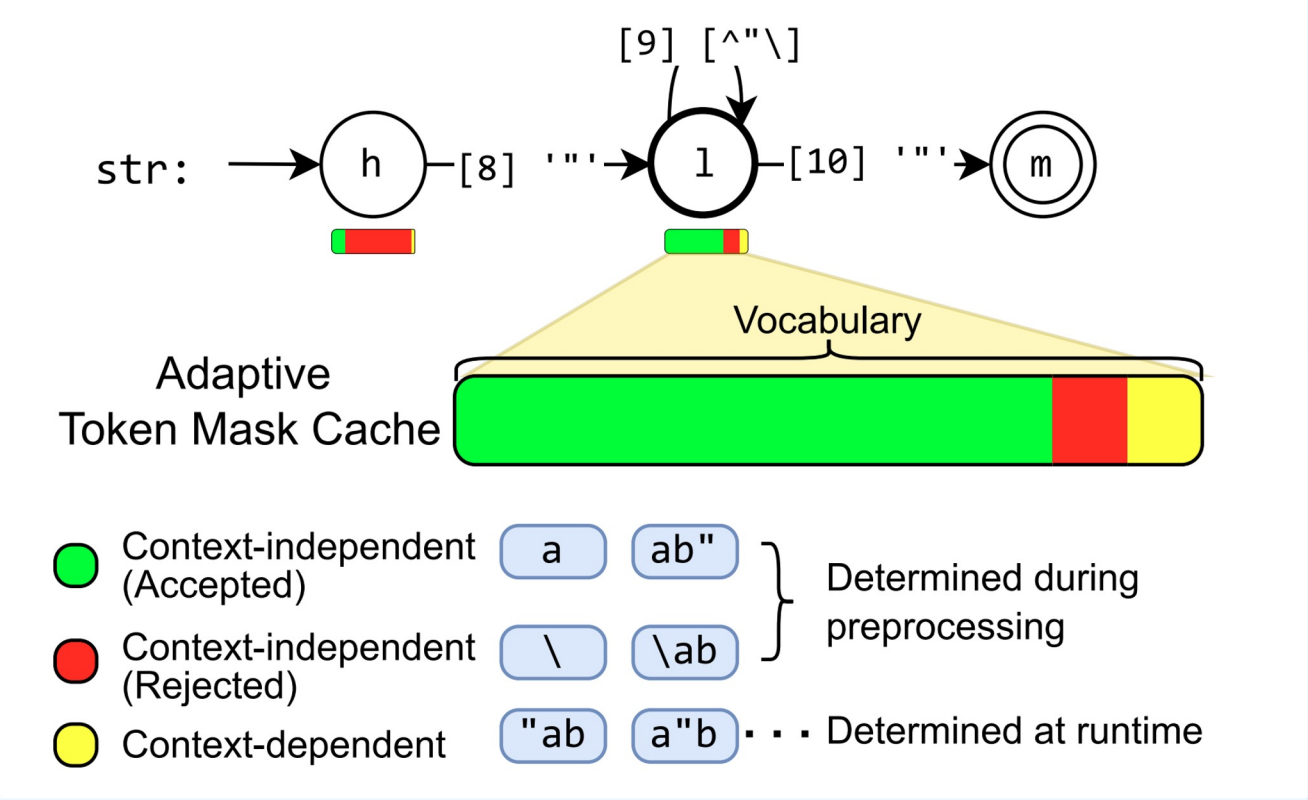


Check if each token matches, then generate the mask

Every token checking is very slow!

Our optimization: most token checking can be fast via preprocessing

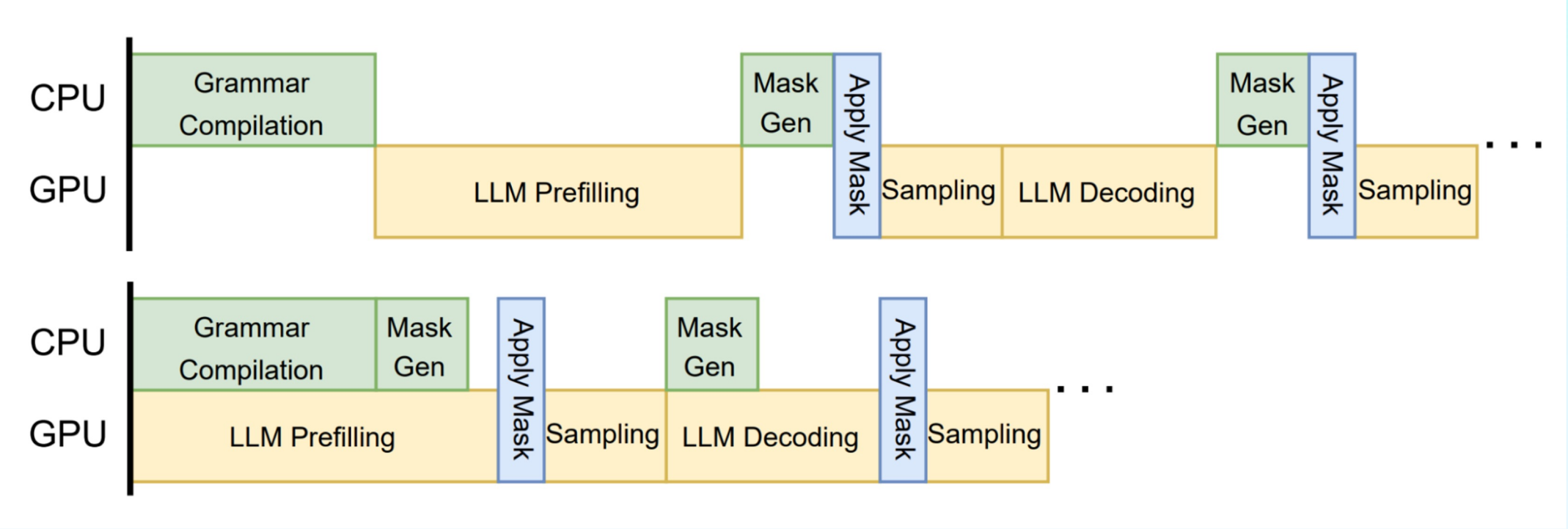
The Adaptive Token Mask Cache (Cont'd)



- Most tokens can be determined **ahead of time – context-independent tokens**
- Plus a minority of tokens that need to check **at runtime – context-dependent tokens**
- So before running, we can compile a **token mask cache** – for each node, calculate accept/reject for the context-independent tokens

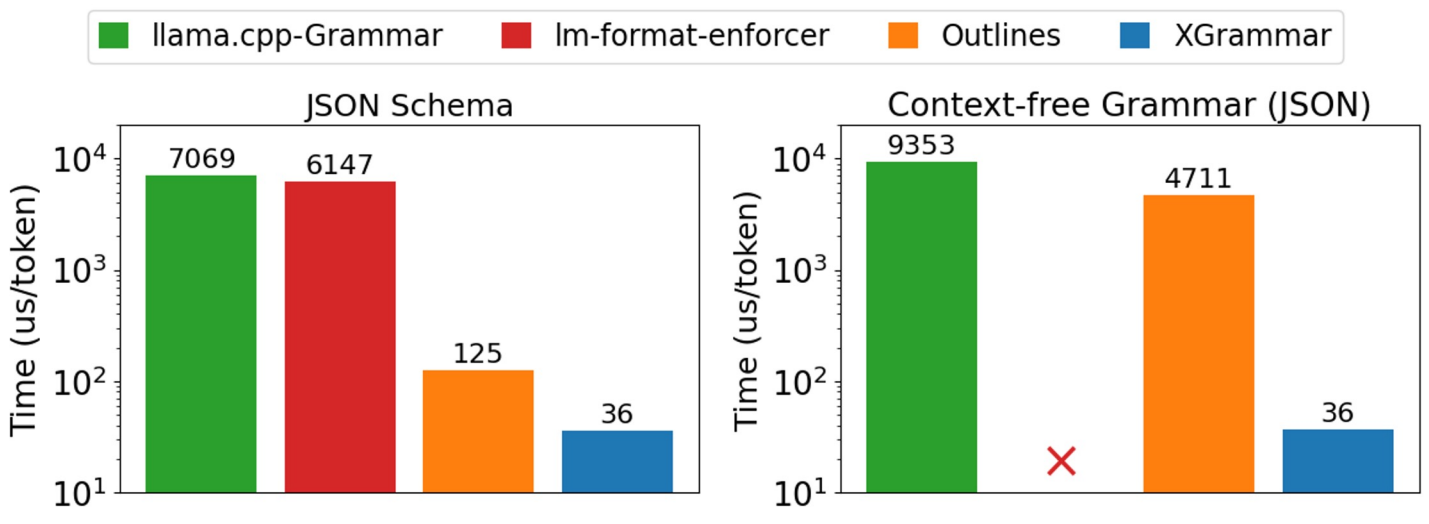
Context-dependent tokens: **less than 1% for Llama-3.1 w/ JSON grammar (1134 out of 128k)**

Overlapping Mask Generation and LLM Inference



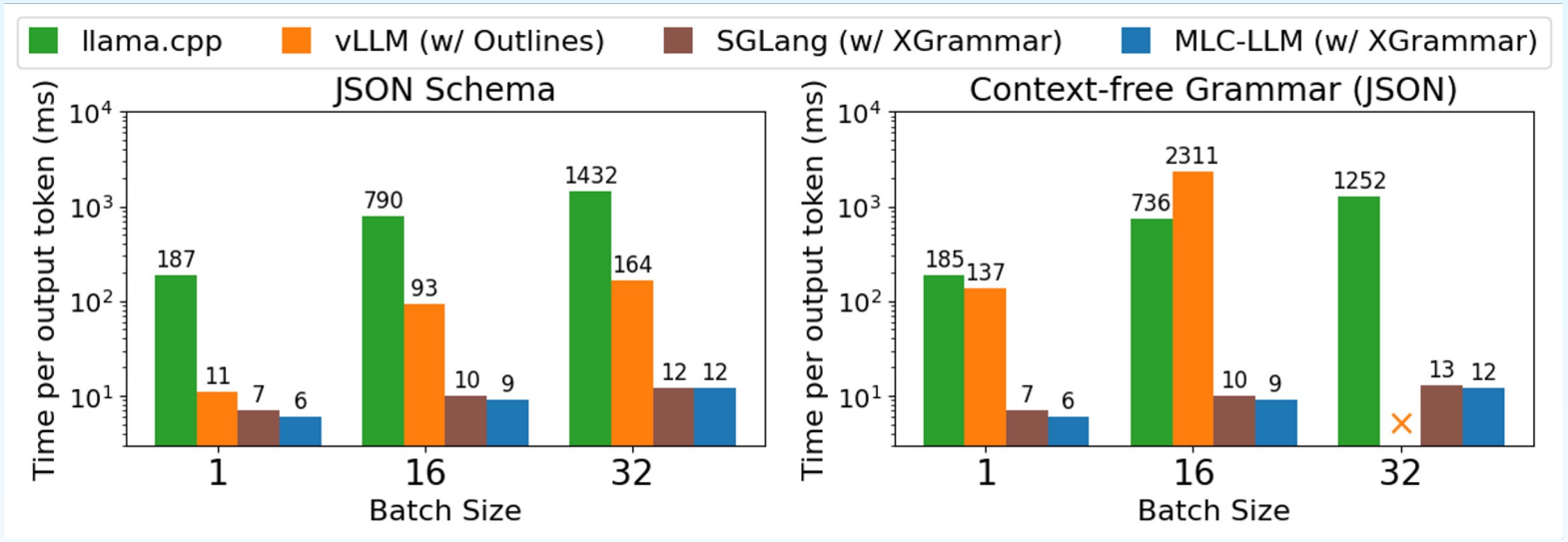
- Top: constrained decoding pipeline without overlapping
- Bottom: constrained decoding pipeline with overlapping

Evaluation



Overhead of masking logits. (Llama-3-8B, AMD 7950X CPU, RTX 4090)

Up to 3.5x on JSON schema
Up to 10x on CFG-guided



Time per output token for end-to-end LLM inference. (Llama-3-8B, AMD 7950X CPU, H100 GPU)

Up to 14x in JSON-schema
Up to 80x in CFG-guided

Multimodal Generation w/ VILA

XGrammar has been integrated into VILA to enable structural generation with multimodal input

Question

Schema

```
{
  "description": "The image ...",
  "objects": [
    "mountains made of meat",
    "valleys and rivers made of sliced ham",
    ...
  ],
  "setting": {
    "location": "a surreal landscape made of meat",
    "time_of_day": "daytime",
    "lighting": "bright and sunny"
  },
  "colors": [
    "pink and red for the mountains",
    ...
  ]
}
```

Part 03
Structural Tags

Flexible and Dynamic Structural Generation API

The Structural Tag

```
None
{"name": "function_name", "parameters": params}
```

- If LLM outputs {"name": "get_weather" ➔ We follow the schema of get_weather
- If LLM outputs {"name": "find_address" ➔ We follow the schema of find_address
- Do dispatches between grammars

The Structural Tag

- The Structural Tag is a JSON DSL
- The Structural Tag provides a convenient way to describe output structure
- E.g. Tool Calling

Function-Calling

```
structural_tag = TriggeredTagsFormat(
    triggers=["<function="],
    tags=[
        TagFormat(
            begin="<function=func1>",
            content=JSONSchemaFormat(json_schema=...),
            end="</function>",
        ),
        TagFormat(
            begin="<function=func2>",
            content=JSONSchemaFormat(json_schema=...),
            end="</function>",
        ),
    ],
    at_least_one=False,
    stop_after_first=False,
)
```

Allow any output until a trigger is encountered, then dispatch to the corresponding tag.

When the end tag is encountered, the grammar will allow any following output, until the next trigger is encountered.

Types of Structural Tags

- Basic Formats:
  - ConstStringFormat
  - JSONSchemaFormat
  - AnyTextFormat
  - GrammarFormat
  - RegexFormat
  - QwenXMLParameterFormat
- Combinatorial Formats:
  - SequenceFormat
  - OrFormat
  - TagFormat
  - **TriggeredTagsFormat**
  - TagsWithSeparatorFormat

Force Thinking

```
{
    "type": "structural_tag",
    "format": {
        "type": "sequence",
        "elements": [
            {
                "type": "tag",
                "begin": "<think>",
                "content": {"type": "any_text"},
                "end": "</think>",
            },
            {
                "type": "triggered_tags",
                "triggers": ["<function="],
                "tags": [
                    {
                        "begin": "<function=func1>",
                        "content": {"type": "json_schema", "json_schema": ...},
                        "end": "</function>",
                    },
                    {
                        "begin": "<function=func2>",
                        "content": {"type": "json_schema", "json_schema": ...},
                        "end": "</function>",
                    },
                ],
            },
        ],
    },
}
```

The output should start with a reasoning part (<think>...</think>), then can generate a mix of text and tool calls.

Force non-thinking mode

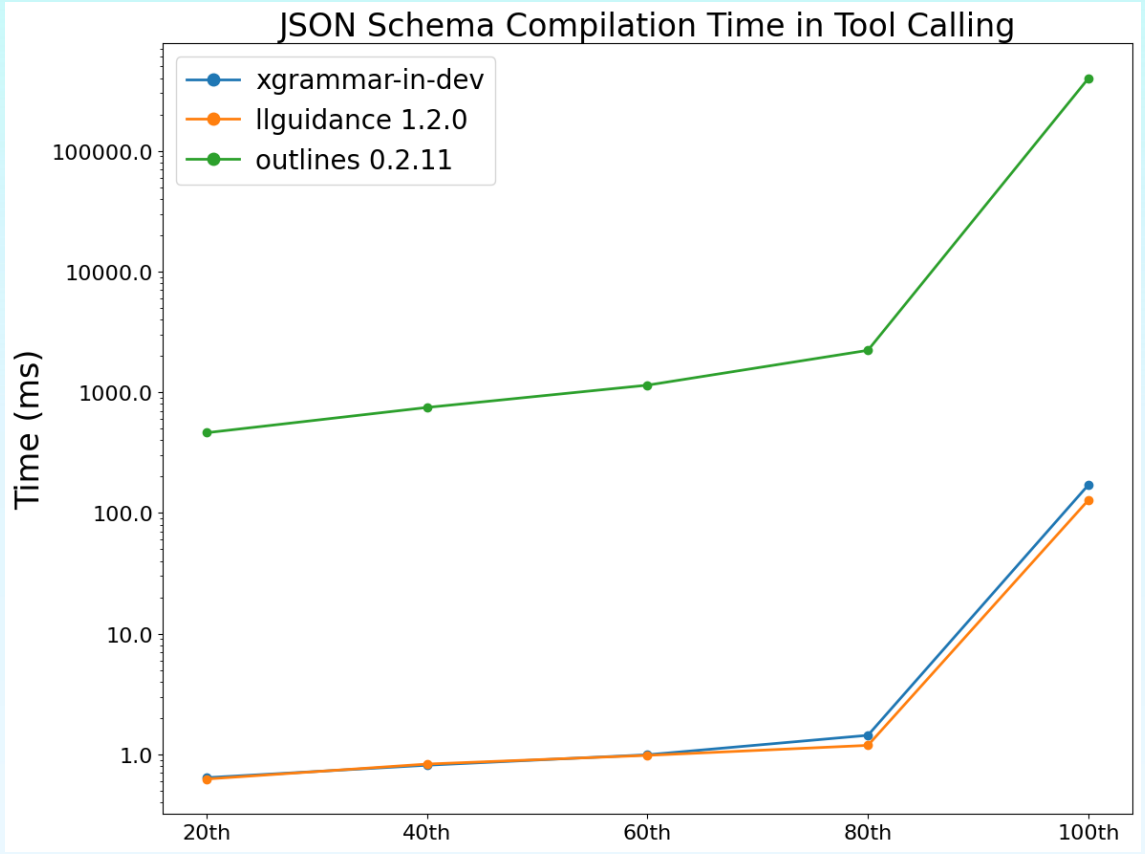
```
{
    "type": "structural_tag",
    "format": {
        "type": "sequence",
        "elements": [
            {
                "type": "const_string",
                "text": "<think></think>"
            },
            {
                "type": "triggered_tags",
                "triggers": ["<tool_call>"],
                "tags": [
                    {
                        "begin": "<tool_call>\n{\"name\": \"func1\", \"arguments\": ",
                        "content": {"type": "json_schema", "json_schema": ...},
                        "end": "}\n</tool_call>",
                    },
                    {
                        "begin": "<tool_call>\n{\"name\": \"func2\", \"arguments\": ",
                        "content": {"type": "json_schema", "json_schema": ...},
                        "end": "}\n</tool_call>",
                    },
                ],
            },
        ],
    },
}
```

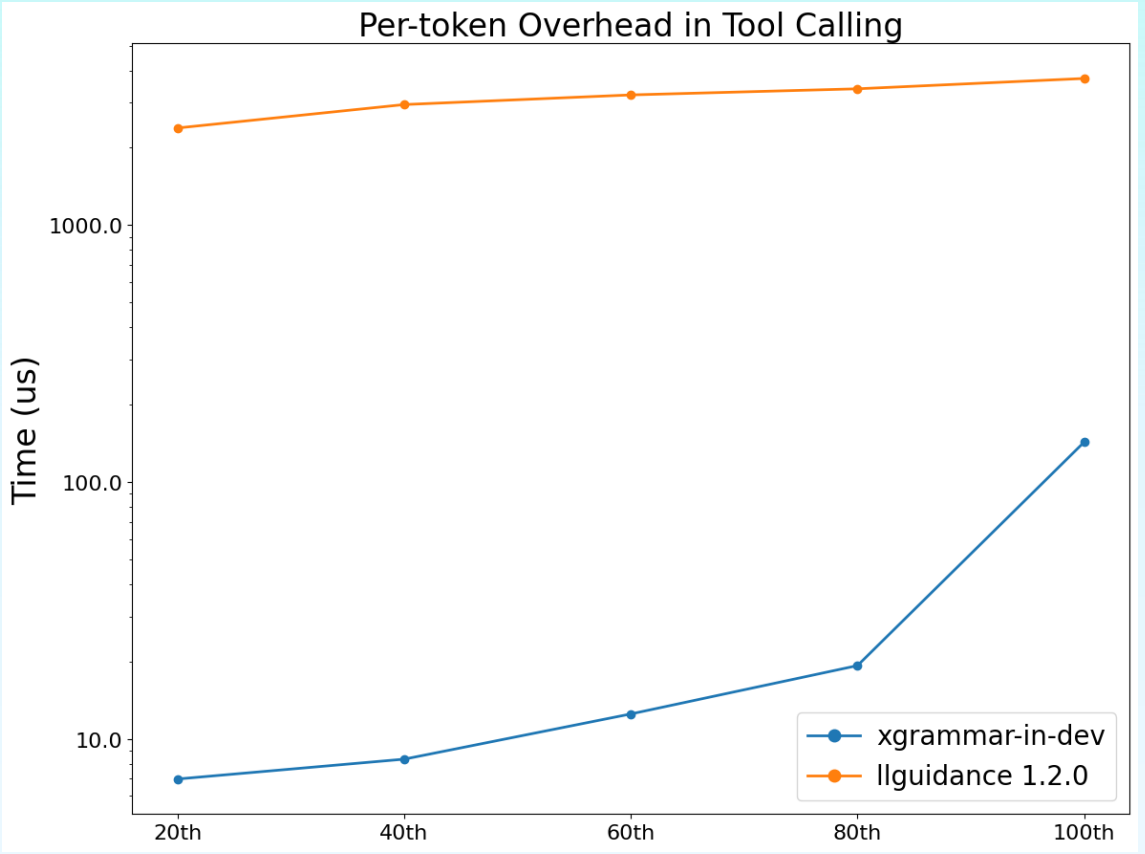
Qwen-3 has a hybrid thinking mode that allows switching between thinking and non-thinking mode. Thinking mode is the same as above, while in non-thinking mode, the output would start with a empty thinking part <think></think>, and then can generate any text.

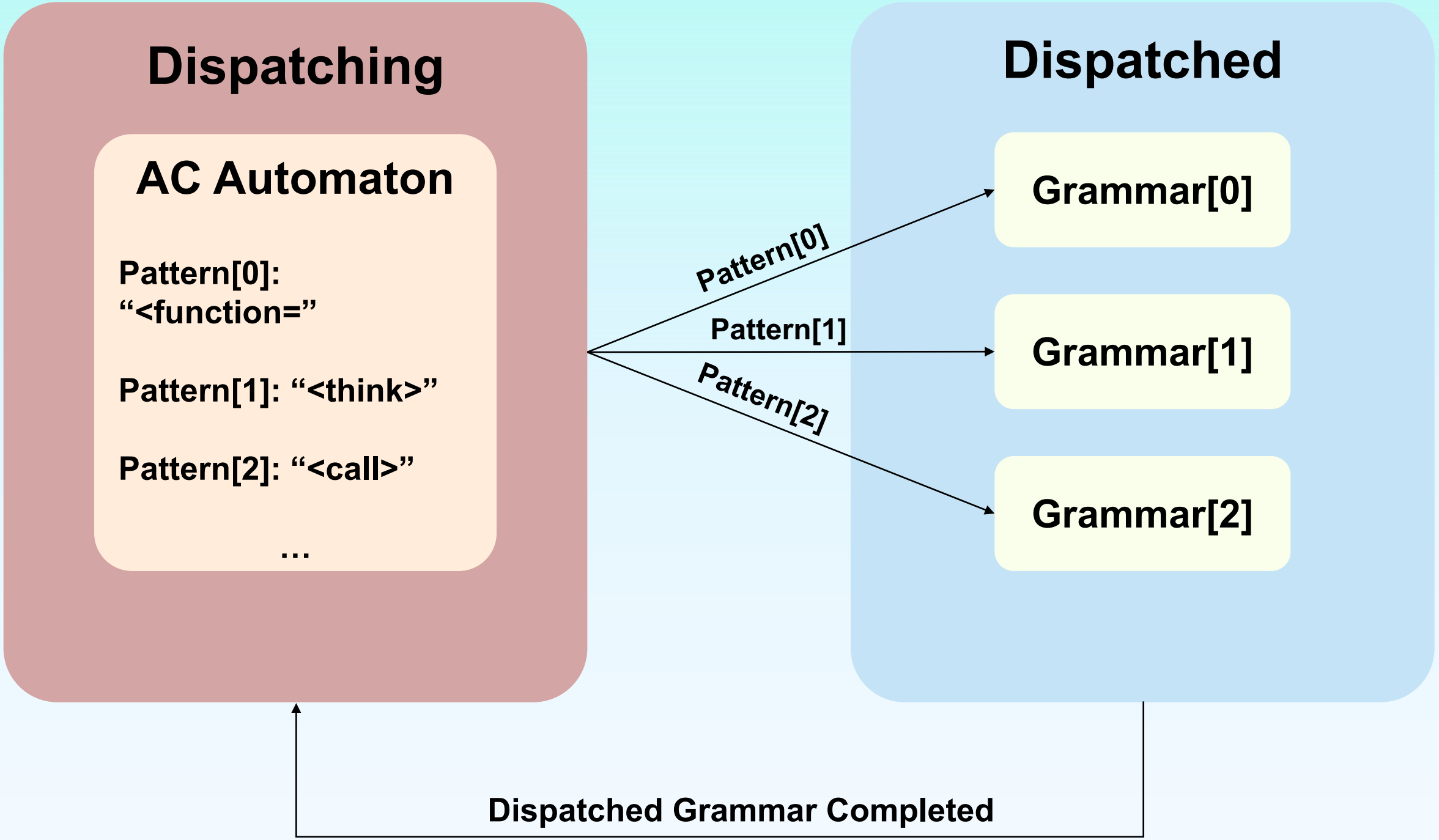
Part 04

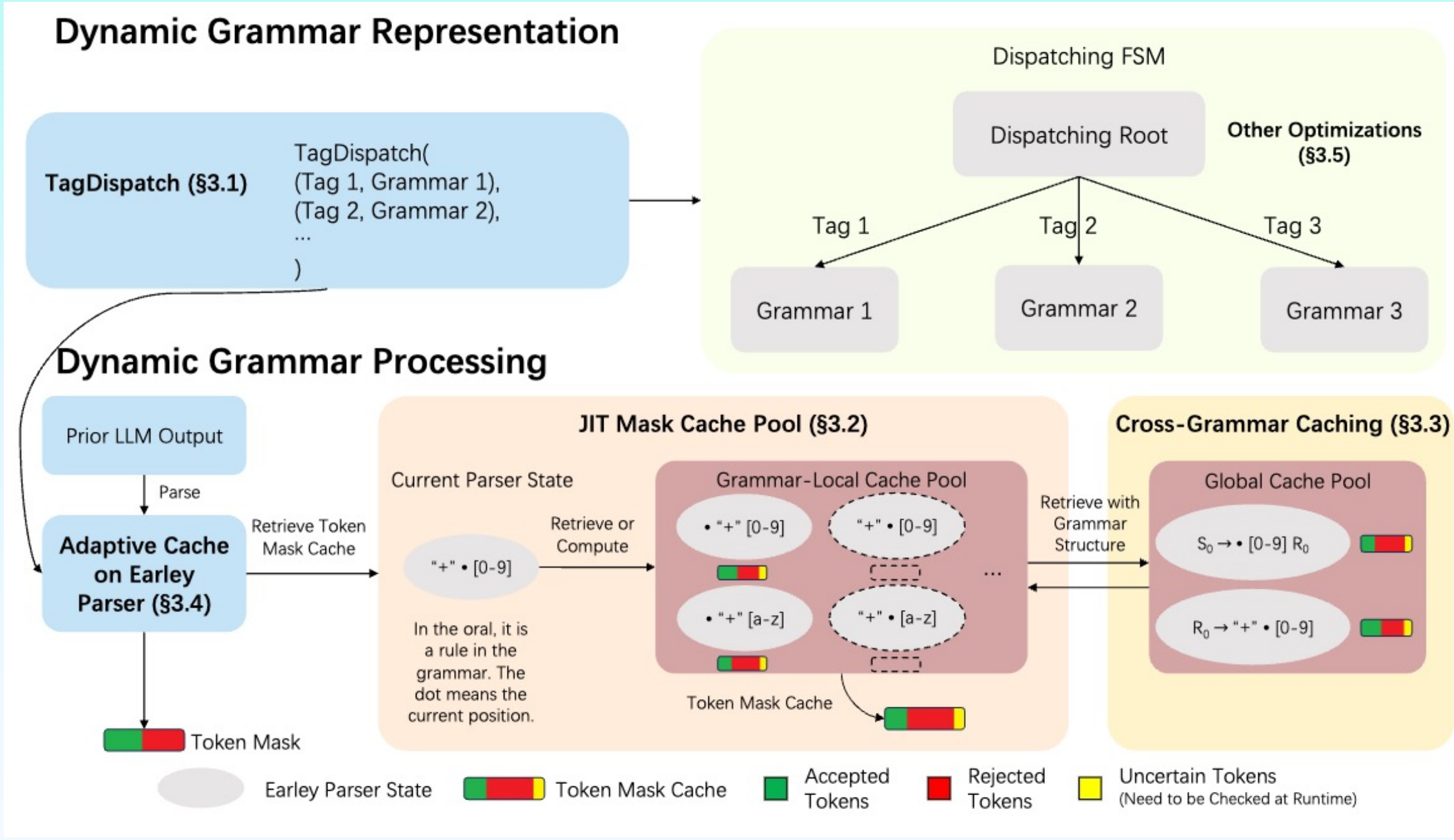
XGrammar v2

Zero Overhead Function Calling


JSON Schema Compilation Time in Tool Calling
xgrammar-in-dev
llguidance 1.2.0
outlines 0.2.11
Time (ms)
100000.0
10000.0
1000.0
100.0
10.0
1.0
20th
40th
60th
80th
100th

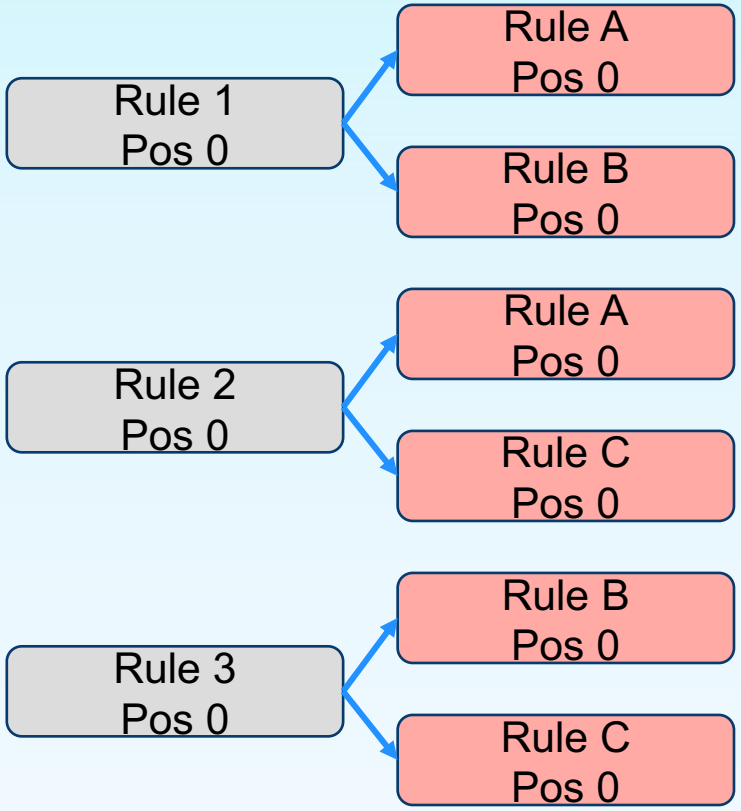

Per-token Overhead in Tool Calling
Time (us)
1000.0
100.0
10.0
xgrammar-in-dev
llguidance 1.2.0
20th
40th
60th
80th
100th


Dispatching
AC Automaton
Pattern[0]:
“<function=”
Pattern[1]: “<think>”
Pattern[2]: “<call>”
…
Dispatched
Grammar[0]
Grammar[1]
Grammar[2]
Pattern[0]
Pattern[1]
Pattern[2]
Dispatched Grammar Completed


Dynamic Grammar Representation
TagDispatch (§3.1)
TagDispatch(
(Tag 1, Grammar 1),
(Tag 2, Grammar 2),
...
)
Dispatching FSM
Dispatching Root
Other Optimizations (§3.5)
Tag 1
Tag 2
Tag 3
Grammar 1
Grammar 2
Grammar 3
Dynamic Grammar Processing
Prior LLM Output
Parse
Adaptive Cache on Earley Parser (§3.4)
Retrieve Token Mask Cache
JIT Mask Cache Pool (§3.2)
Current Parser State
"+" • [0-9]
In the oral, it is a rule in the grammar. The dot means the current position.
Retrieve or Compute
Grammar-Local Cache Pool
• "+" [0-9]
"+" • [0-9]
• "+" [a-z]
"+" • [a-z]
...
Token Mask Cache
Retrieve with Grammar Structure
Cross-Grammar Caching (§3.3)
Global Cache Pool
$S_0 \rightarrow$ • [0-9] R_0
$R_0 \rightarrow$ "+" • [0-9]
Token Mask
Earley Parser State
Token Mask Cache
Accepted Tokens
Rejected Tokens
Uncertain Tokens
(Need to be Checked at Runtime)

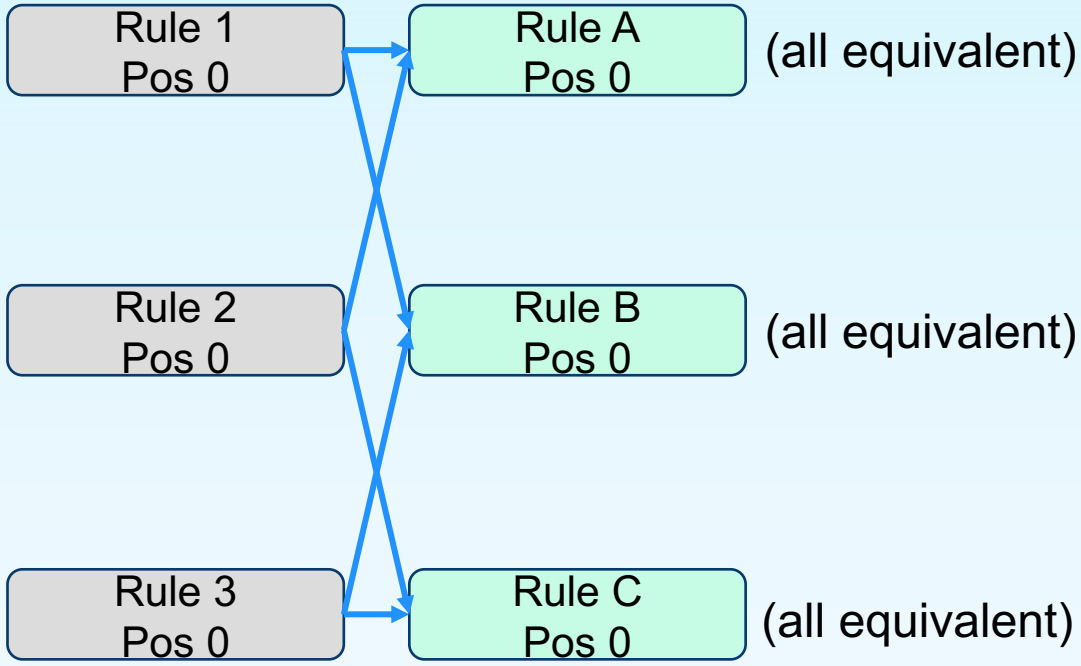
Earley Parser

PDA Parser



👎Worst
exponential states

Earley Parser



👍linear states

Thanks.

Repo: https://github.com/mlc-ai/xgrammar
Papers: https://arxiv.org/abs/2411.15100
Blog: https://blog.mlc.ai/2024/11/22/achieving-efficient-flexible-portable-structured-generation-with-xgrammar
Documentation: https://xgrammar.mlc.ai/docs/