XGrammar: Flexible And Efficient Structured Generation Engine for Large Language Models

Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, Tianqi Chen

Dec 21, 2024

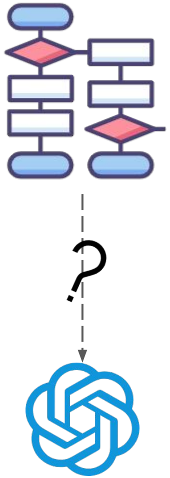
Problem: LLM Generation with Structures

Code generation	Function/tool calling	Embodied Agents
GitHub Copilot	LangChain	
Language Code	JSON Schema	DSL (e.g. PDDL)

Structured Outputs

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

Problem: LLM's Limited Ability with Complex Structures



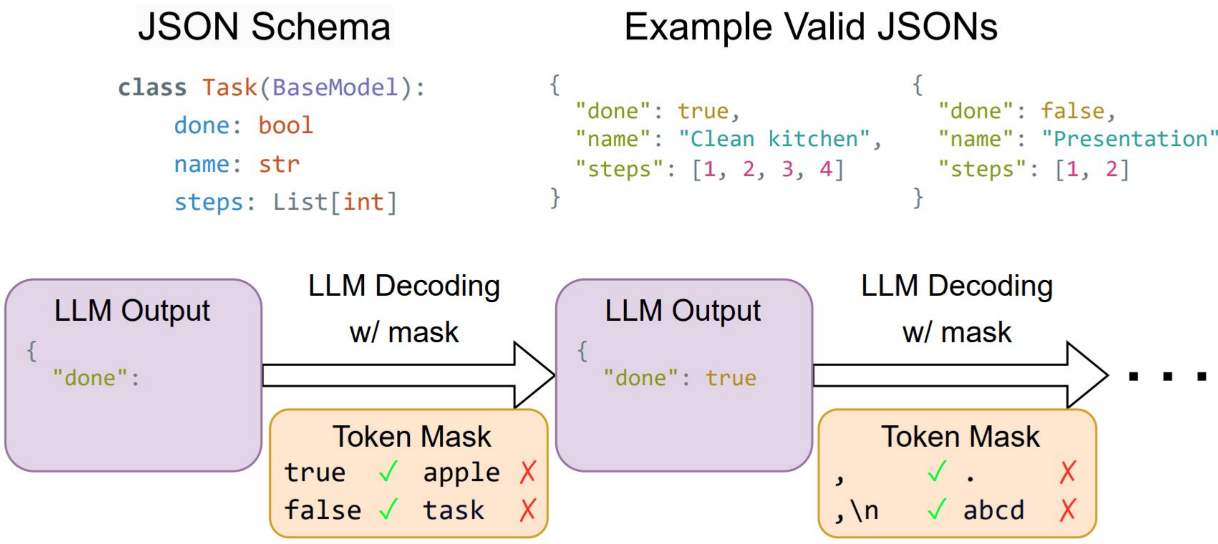
Structures are increasingly complex

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

LLM's generation ability is limited

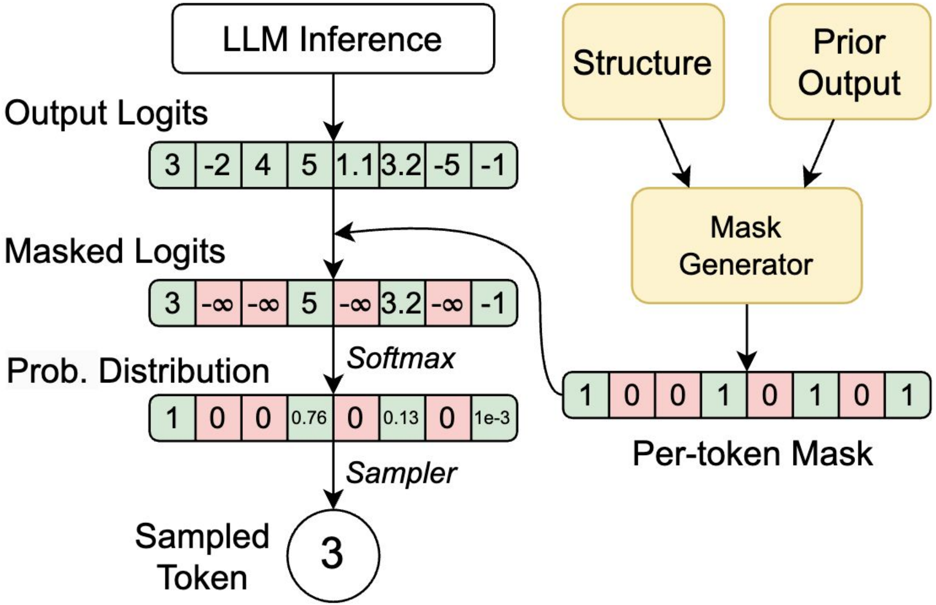
- On-device small LMs
- Compressed models

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!

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

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

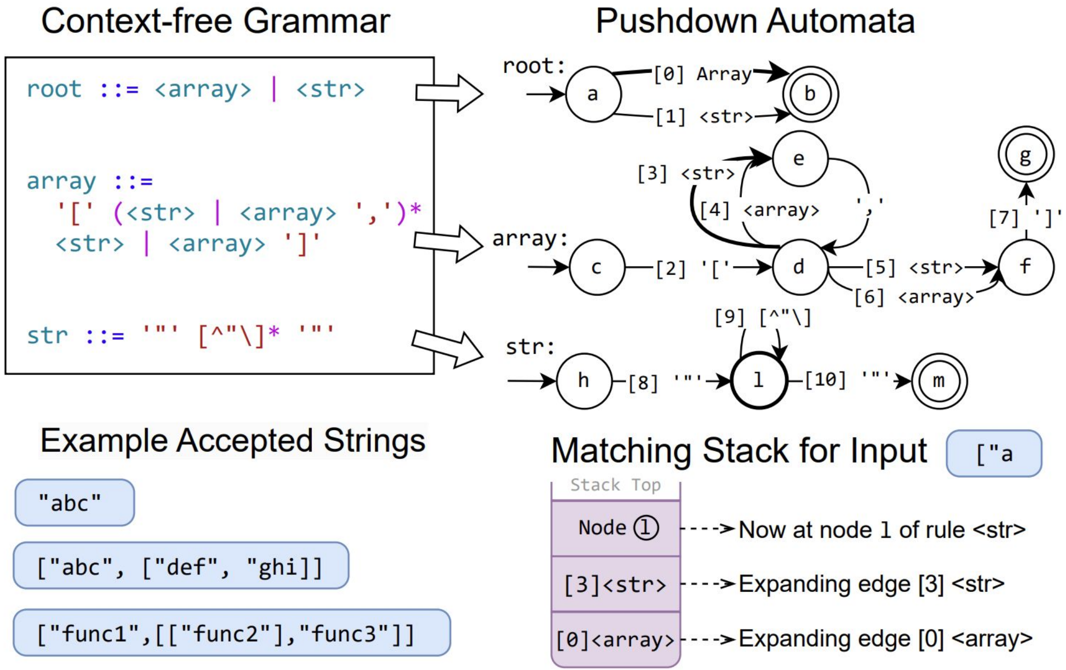
```
root ::= <array> | <str>


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)

More Powerful: Pushdown Automata



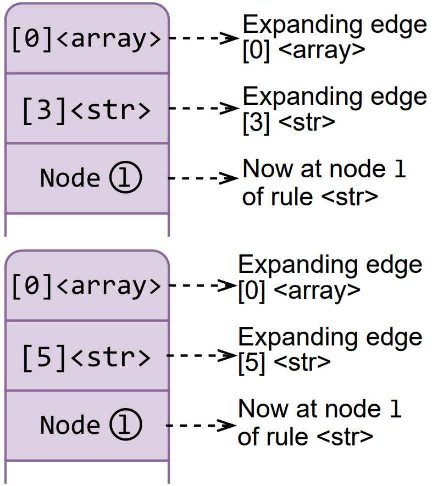
- XGrammar utilizes pushdown automata to match string to CFG
- The matching process is **a recursive expansion of rules**

Pushdown Automata (Cont'd)

Context-free Grammar

```
main ::= <array> | <str>
array ::= '[' (<str> | <array> ',')*
          <str> | <array> ']'
str ::= '"' [^"\]* '"'
```

Matching Stacks for Input `["a`



- It is possible that a string can map to multiples stacks due to ambiguity

XGrammar: Flexible and Efficient Structured Generation Engine

XGrammar is a structured generation library that features

Flexibility: Full support for context-free grammar

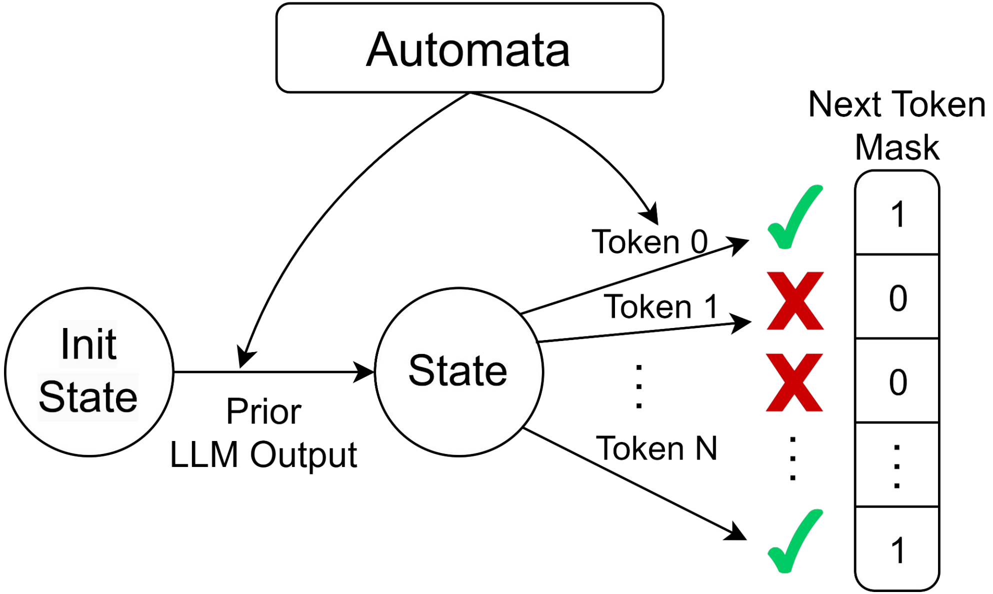
Efficiency: SOTA performance in constraint decoding

Zero-overhead JSON Schema generation

Integration: Easy to integrate with existing LLM serving frameworks

vLLM, MLC-LLM, SGLang, etc

Previous Mask Generation Method

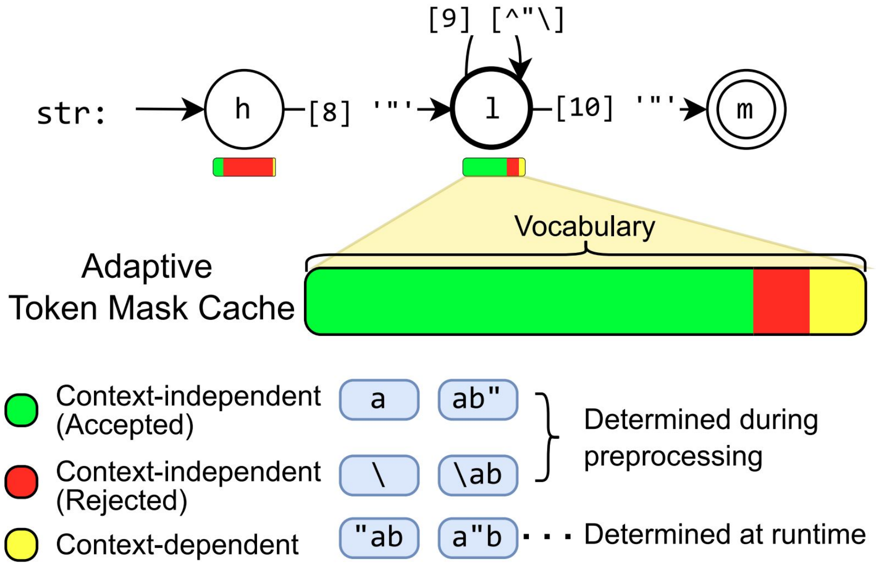


Check if each token matches, then generate the mask

Every token checking is very slow!

Our optimization: most token checking can be fast via preprocessing

Optimization #1. The Local Token Mask Cache



- 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

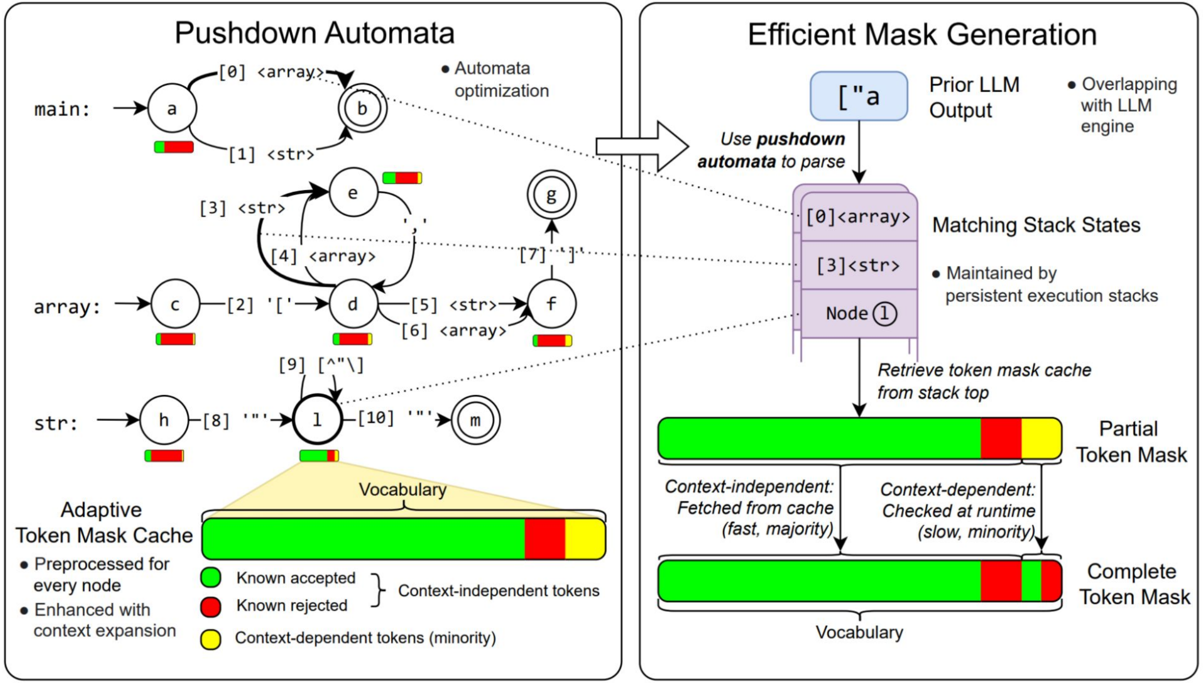
Context-dependent tokens: **less than 1% for Llama-3.1 w/ JSON grammar (1134 out of 128k)**

Thanks

Questions are welcome!

- **Paper**: https://arxiv.org/abs/2411.15100
- **Blog**: https://blog.mlc.ai/2024/11/22/achieving-efficient-flexible-portable-structured-generation-with-xgrammar
- **Code**: https://github.com/mlc-ai/xgrammar
- **Documentation**: https://xgrammar.mlc.ai/docs/
- **Wechat**: ubospica, **Email**: yixind@andrew.cmu.edu

The Local Token Mask Cache (Cont'd)



- At runtime, given the node we are at, retrieve the pre-computed token mask
- Then compute the context-dependent token's validity by checking the stack

The Local Token Mask Cache (Storage)

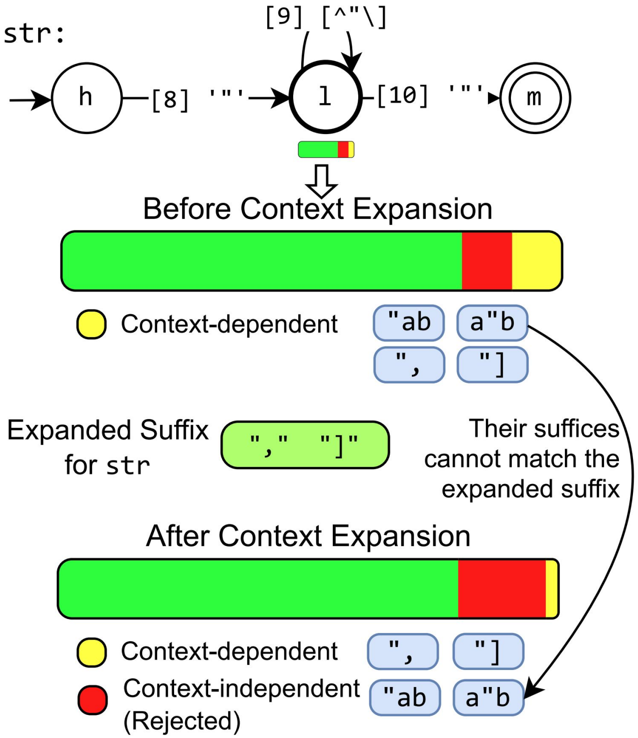
- For every stack top node, we store accepted/rejected/uncertain tokens.
- Accepted + Rejected + Uncertain = Vocabulary!
- We want the size of the pre-computed mask cache to be small!
- Three storage paradigms:
  a) **#accept** is large → [Rejected list], [Uncertain list]
  b) **#reject** is large → [Accepted list], [Uncertain list]
  c) #accept and #reject similar → <Accepted_bitset>, <Uncertain_bitset>
- Store the one with least memory consumption!

On Llama-3.1 w/ JSON grammar, reduces total memory usage to **0.2%** (from 160 MB to 0.46 MB)

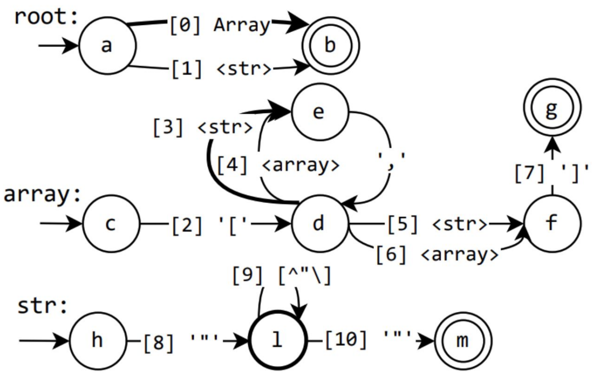
The Challenge of Llama-3

- Vocab_size: 32k → 128k
- More common tokens
- #(uncertain tokens): 100 → 1.5k
  - Means 15x check at runtime
- Further reduction of uncertain tokens is needed!

Optimization #2. Context Expansion

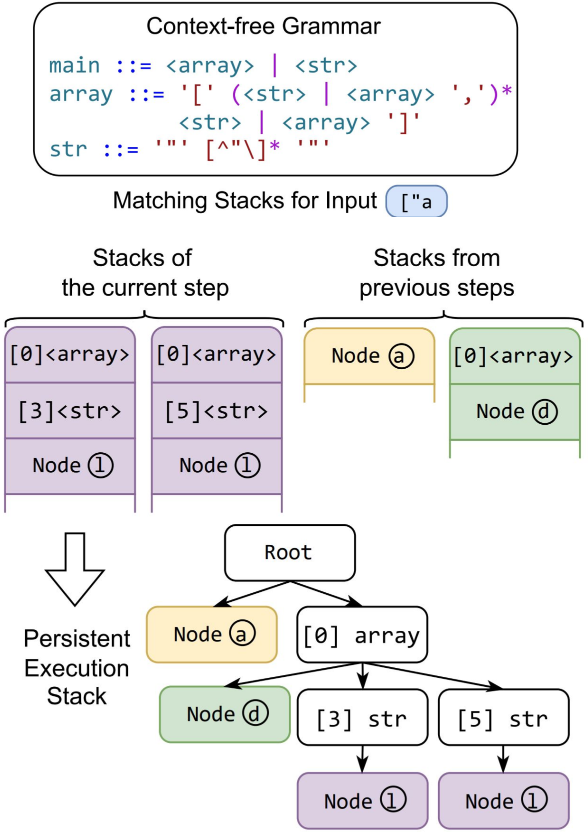


- For every rule, find all its references
- Collect all possible suffix of that rule
- If an uncertain token does not fit into any of the suffix, it is rejected



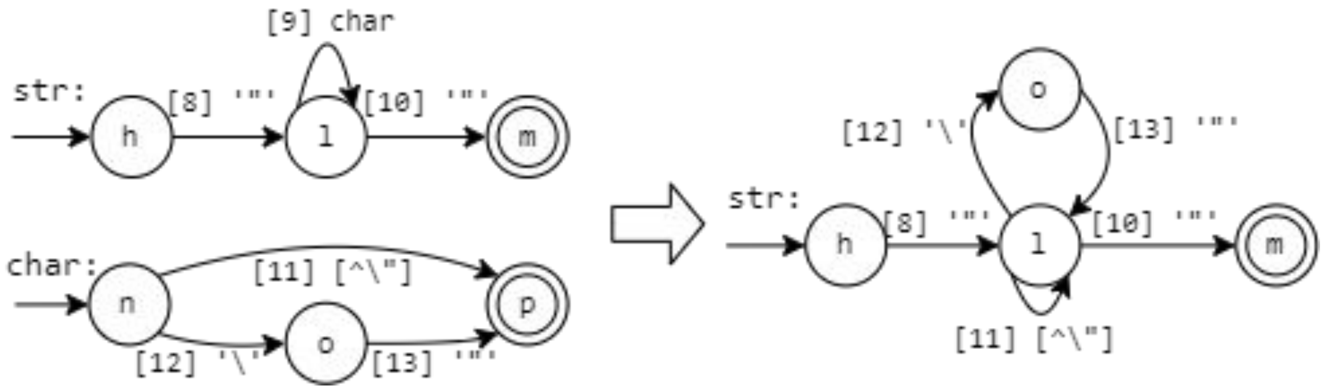
Reduce context-dependent tokens by **90%** on Llama-3.1 w/ JSON grammar

Optimization #3. Persistent Execution Stack



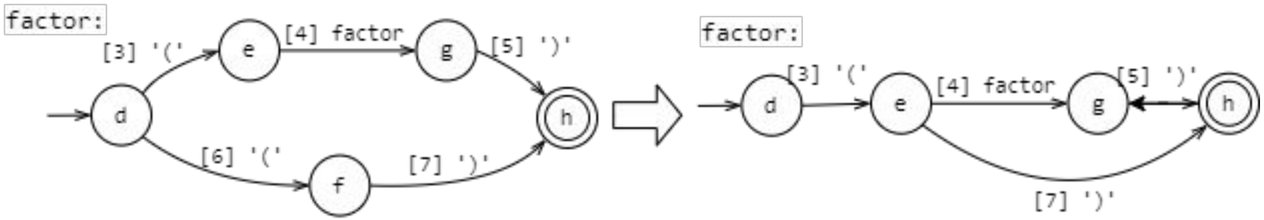
- Despite having most tokens pre-computed, we still need to compute **context-dependent tokens** efficiently
- As mentioned earlier, we can have multiple possible stacks due to the ambiguity
- We represent the stacks as a tree → **avoids memory redundancy** for storing multiple stacks
- Instead of copying the stack, we only split the branch
- Also supports rolling the state back

Optimizations #4. Optimization Passes (Inlining)



- Inline small, fragmented rules into large rules
- Benefits:
    - Reducing recursion overhead
    - Providing more rule-local information for the local token mask cache
        - Fragmented rules lack rule-local information

Optimizations #4. Optimization Passes (Path merging)



- Merges paths with common prefix (when safe)
- Reduces number of possible parsing stacks
  - Need to be handled one by one when generating masks
  - In the recursive cases, #(parsing stacks) may explode exponentially!

XGrammar: Flexible and Efficient Structured Generation Engine

XGrammar is a structured generation library that features

Flexibility: Full support for context-free grammar

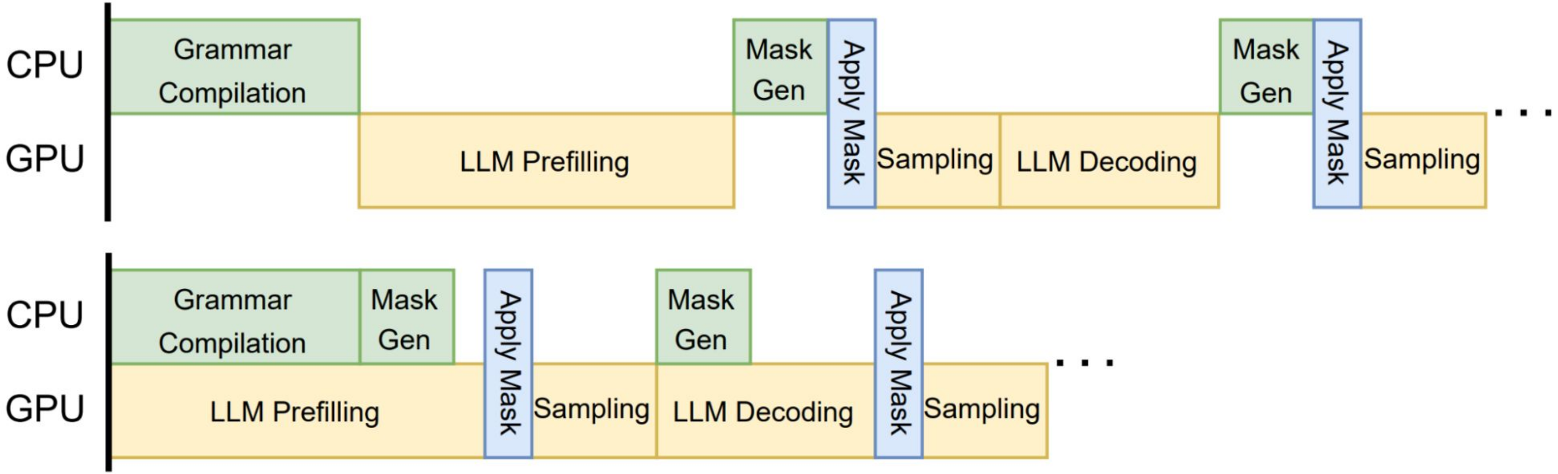
Efficiency: SOTA performance in constraint decoding

Zero-overhead JSON Schema generation

Integration: Easy to integrate with existing LLM serving frameworks

vLLM, MLC-LLM, SGLang, etc

Overlapping Mask Generation and LLM Inference

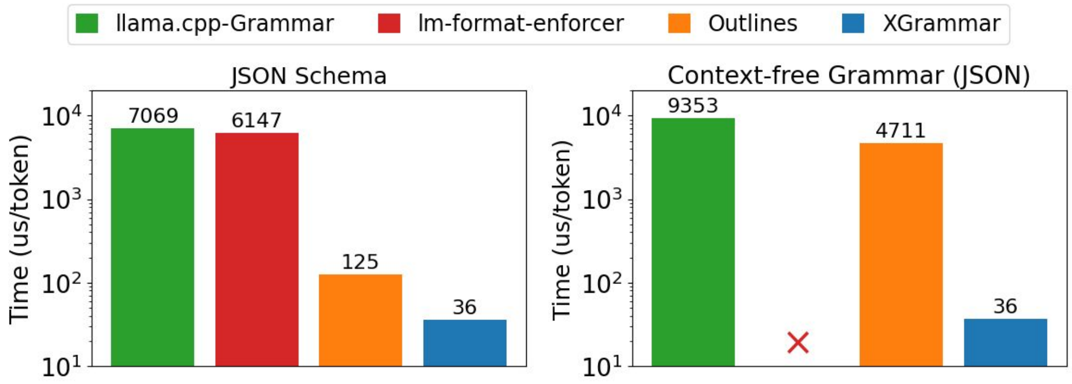


- Top: constrained decoding pipeline without overlapping
- Bottom: constrained decoding pipeline with overlapping

Integration with LLM serving frameworks

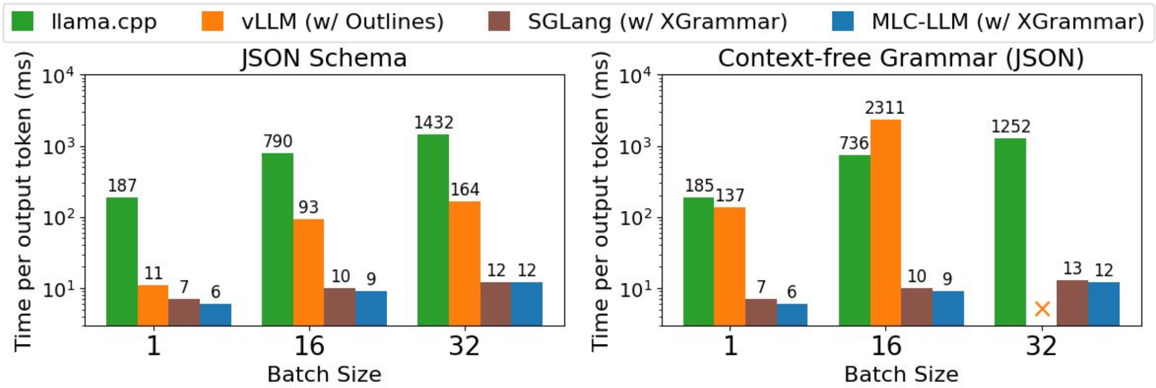
- XGrammar is designed for easy integration and cross-platform support (with C++, Python, and JavaScript APIs)
  - Its core is implemented in C++, so easy to port to other platforms
- XGrammar has already been integrated with vLLM, SGLang, MLC-LLM, WebLLM

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

Try it out on WebLLM

Run structured generation completely on your web browser with great efficiency!

Model

Llama-3.1-8B-Instruct-q4f16_1-MLC

Grammar

Custom Grammar

Custom EBNF Grammar

The custom grammar is described in the [extended Backus-Naur form (EBNF)](). Below is an example of JSON grammar in EBNF. Please follow this example when writing new grammars.

```
main    ::= "1. " move " " move "\n" ([1-9] [0-9]? ". " move "
" move "\n")+
move    ::= (pawn | nonpawn | castle) [+#]?
# piece type, optional file/rank, optional capture, dest file &
rank
nonpawn ::= [NBKQR] [a-h]? [1-8]? "x"? [a-h] [1-8]
# optional file & capture, dest file & rank, optional promotion
pawn    ::= ([a-h] "x")? [a-h] [1-8] ("=" [NBKQR])?
castle  ::= "O-O" "-O"?
```

Prompt

```
main    ::= "1. " move " " move "\n" ([1-9] [0-9]? ". " move "
" move "\n")+
move    ::= (pawn | nonpawn | castle) [+#]?
# piece type, optional file/rank, optional capture, dest file &
rank
nonpawn ::= [NBKQR] [a-h]? [1-8]? "x"? [a-h] [1-8]
# optional file & capture, dest file & rank, optional promotion
pawn    ::= ([a-h] "x")? [a-h] [1-8] ("=" [NBKQR])?
castle  ::= "O-O" "-O"?
```

Generate

Output

1. e4 e5 2. Nf3 Nc6 3. Bc4 Nf6 4. d3 d6 5. O-O O-O 6. b4 a5 7. b5 Nxb5 8. a4 Na6 9. Qe2 Qe7 10. a5 Nc5 11. a6 b6 12. a7 b7 13. a8=Q a8=Q 14. Nb1 Nb1 15. Nc3 Nc3 16. Na4 Na4 17. b5 b5 18.

Prefill Speed: 39.4 tok/s, Decode Speed: 19.8 tok/s, Time to First Token: 50 ms, Time Per Output Token: 4698 ms, Grammar Per-token Overhead: 0.07 ms

https://huggingface.co/spaces/mlc-ai/WebLLM-Structured-Generation-Playground