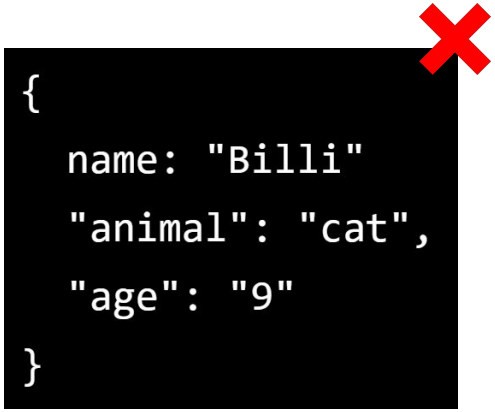
XGrammar

Aditya Ramesh

Motivation

Desired

Added key



Incorrect format

We want to specify a specific JSON format but LLMs are not guaranteed to do so correctly!

Improve Performance

1. Prompt engineering
    a. In context learning, multi-stage prompting
2. Finetuning
3. **Constrained Decoding**

Finite State Machines

Outlines
by dottxt.ai

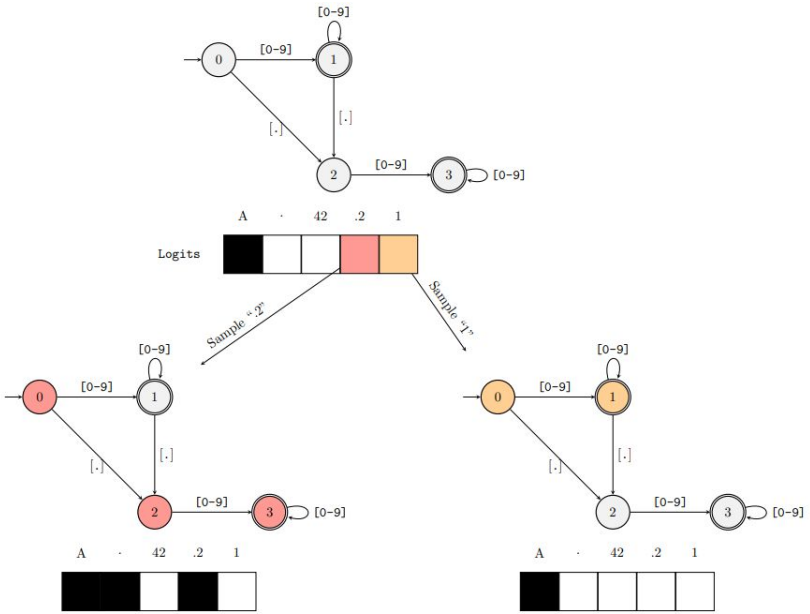


Figure 1: FSM masking for the regular expression `([0-9]*)?\.?[0-9]*`.

Use regex to construct a FSM that constrains the sampling of an LLM!

https://arxiv.org/pdf/2307.09702

Finite State Machines

Outlines
by dottxt.ai

Use a mask to constrain the LLM

$$\boldsymbol{\alpha} = \mathrm{LLM}(S_t, \boldsymbol{\theta})$$
$$s_{t+1} \sim \mathrm{Categorical}(\boldsymbol{\alpha})$$

$$\boldsymbol{\alpha} = \mathrm{LM}(\tilde{S}_t, \boldsymbol{\theta})$$
$$\tilde{\boldsymbol{\alpha}} = \mathrm{m}\left(\tilde{S}_t\right) \odot \boldsymbol{\alpha}$$
$$\tilde{s}_{t+1} \sim \mathrm{Categorical}(\tilde{\boldsymbol{\alpha}})$$

$\tilde{S}_t$: current sequence,
$m(\tilde{S}_t)$: token mask,
α : logits,
$\mathrm{Categorical}(\tilde{\alpha})$: sampling method.

https://arxiv.org/pdf/2307.09702

Outlines (2023)

1. Define regex
2. Convert regex into FSM
3. Precompute mask for each state in FSM
4. Store mask as hashmap
5. Use mask to sample

Algorithm 2 LLM token sampling with masking

```
function SAMPLE_TOKENS(L)
    s ← ()
    for i ← 1, L do
        α ← LLM(s, θ)
        Construct the mask m(s)        ← Precomputed before sampling
        α̃ ← m ⊙ α
        Sample s̃ ∼ Categorical(α̃)
        if s̃ = EOS then
            break
        end if
        s ← append(s, s̃)
    end for
    return s
end function
```

https://arxiv.org/pdf/2307.09702

Interleave Optimization (2023)

{guidance}

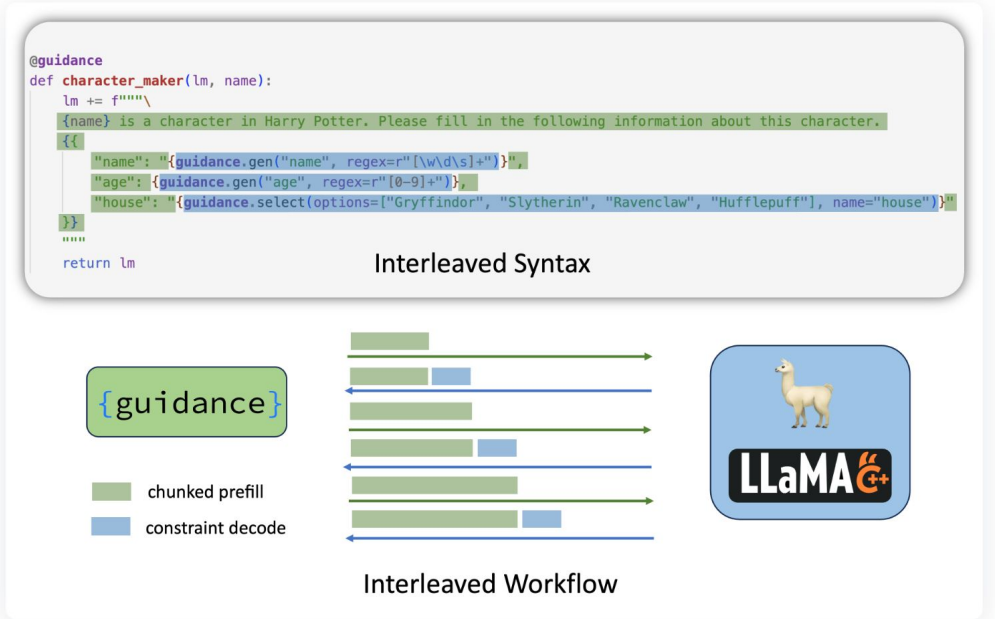


Figure from [LMSYSORG]

https://github.com/guidance-ai/guidance#features
https://www.lmsys.org/blog/2024-02-05-compressed-fsm

Jump-Forward Optimization (2023)

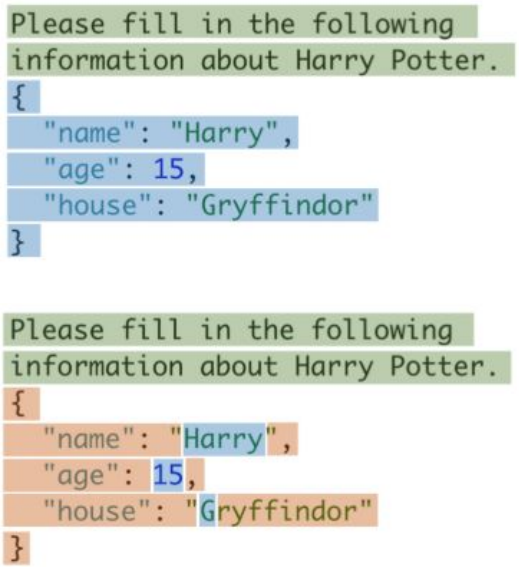


Figure from LMSYSORG

https://github.com/guidance-ai/guidance#features
https://www.lmsys.org/blog/2024-02-05-compressed-fsm

Is regex enough? No!

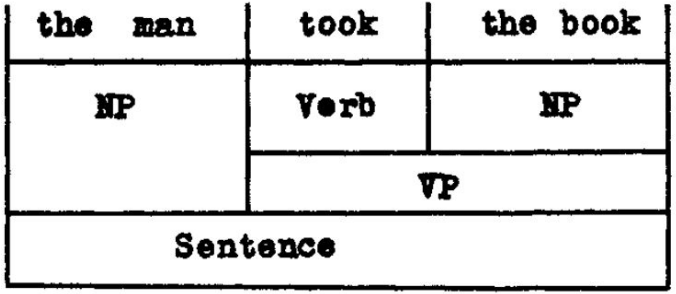
JSON and SQL have recursive structures

regex can not model **recursive structures** (Chomsky 1956)

JSON and SQL need to be modeled with **Context-free Grammar** (Chomsky 1956)

https://chomsky.info/wp-content/uploads/195609-.pdf

Context-Free Grammar (CFG)



(20) Σ : #^Sentence^#
F: Sentence → NP^VP
VP → Verb^NP
NP → the^man, the^book
Verb → took

(21) D_1: #^Sentence^#
#^NP^VP^#
#^NP^Verb^NP^#
#^the^man^Verb^NP^#
#^the^man^Verb^the^book^#
#^the^man^took^the^book^#

D_2: #^Sentence^#
#^NP^VP^#
#^the^man^VP^#
#^the^man^Verb^NP^#
#^the^man^took^NP^#
#^the^man^took^the^book^#

Introduced as **Phrase Structure** (Chomsky 1956)

https://chomsky.info/wp-content/uploads/195609-.pdf

JSON as CFG

```
   JSON-text = ws value ws

These are the six structural characters:

   begin-array     = ws %x5B ws  ; [ left square bracket

   begin-object    = ws %x7B ws  ; { left curly bracket

   end-array       = ws %x5D ws  ; ] right square bracket

   end-object      = ws %x7D ws  ; } right curly bracket

   name-separator  = ws %x3A ws  ; : colon

   value-separator = ws %x2C ws  ; , comma
```

```
ws = *(
        %x20 /              ; Space
        %x09 /              ; Horizontal tab
        %x0A /              ; Line feed or New line
        %x0D )              ; Carriage return
```

```
value = false / null / true / object / array / number / string

false = %x66.61.6c.73.65   ; false

null  = %x6e.75.6c.6c      ; null

true  = %x74.72.75.65      ; true
```

JSON is defined with Context-Free Grammar!

https://datatracker.ietf.org/doc/html/rfc8259#section-2

Pushdown Automata (PDA)

regex ⟷ FSM

CFG ⟷ PDA

Finite state machines can not model recursion!
(Chomsky 1956)

Pushdown (or stack) is needed to analyze nested formal languages left-to-right!
(Oettinger 1961)

Pushdown Automata creates a context-free language!
(Schützenberger 1963)

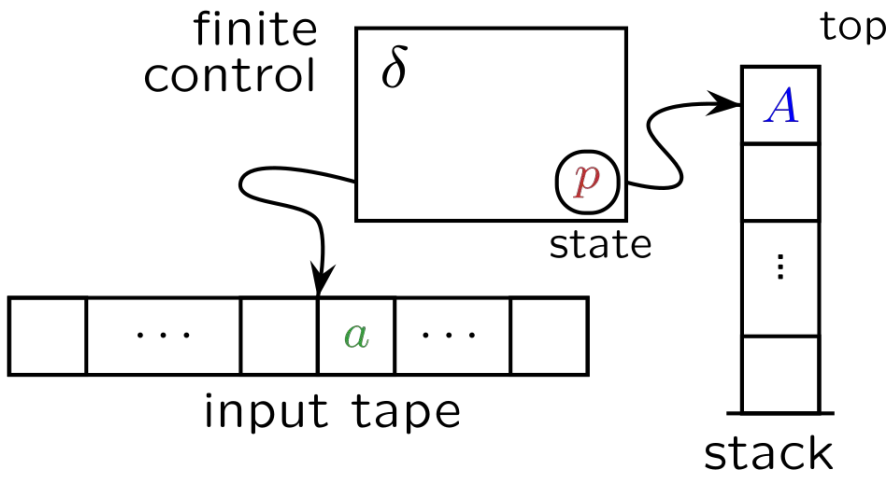
https://chomsky.info/wp-content/uploads/195609-.pdf
https://mt-archive.net/Oettinger-1961.pdf

Pushdown Automata (Informal)

A **pushdown automaton** is a finite-state machine plus a **stack-based memory**.

On transition we can

a) Transition along the FSM
b) Manipulate the stack



From Wikipedia

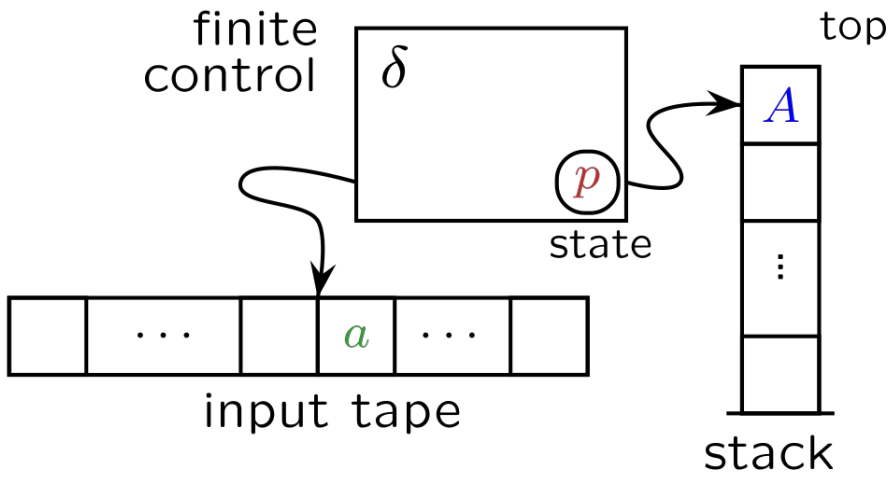
https://en.wikipedia.org/wiki/Pushdown_automaton
https://web.stanford.edu/class/archive/cs/cs103/cs103.1132/lectures/17/Small17.pdf

Pushdown Automata

Deterministic PDA (PDA) - For every input and stack symbol, there is only one transition

Non-deterministic PDA (NPDA) - if several actions are possible at a input and stack symbol

NPDA will maintain multiple stacks in parallel for each path



From Wikipedia

https://en.wikipedia.org/wiki/Pushdown_automaton
https://web.stanford.edu/class/archive/cs/cs103/cs103.1132/lectures/17/Small17.pdf

Pushdown Automata Example

$$L = \{a^n b^n \mid n \geq 0\}$$

Examples	Non-Examples
aabb	abb
aaabbb	b
ab	bbaa
	ababa

FSM alone can not keep track of how many '*a*' are present as we parse left to right. A stack is needed!

Pushdown Automata Example

$$L = \{a^n b^n \mid n \geq 0\}$$

Intuition

Read a: push A
Read b: pop A

Accept if input is finished and stack is empty

Input left	Stack
aaabbb	$
aabbb	A$
abbb	AA$
bbb	AAA$
bb	AA$
b	A$
ε	$
accept	

JSON

```
{
  "name": "Billi",
  "friends": [
    { "name": "Lea" },
    { "name": "Summer" },
    { "name": "Bunny" }
  ]
}
```

Input token	Action	Stack
{	push OBJECT	OBJECT
"name"	read object key	OBJECT
:	read colon	OBJECT
"Billi"	read string value	OBJECT
,	expect another pair	OBJECT
"friends"	read object key	OBJECT
:	read colon	OBJECT
[	push ARRAY	OBJECT ARRAY
{	push OBJECT	OBJECT ARRAY OBJECT
"name"	read object key	OBJECT ARRAY OBJECT
:	read colon	OBJECT ARRAY OBJECT
"Lea"	read string value	OBJECT ARRAY OBJECT
}	pop OBJECT	OBJECT ARRAY
,	expect another element	OBJECT ARRAY
{	push OBJECT	OBJECT ARRAY OBJECT
"name"	read object key	OBJECT ARRAY OBJECT
:	read colon	OBJECT ARRAY OBJECT
"Summer"	read string value	OBJECT ARRAY OBJECT
}	pop OBJECT	OBJECT ARRAY
,	expect another element	OBJECT ARRAY
{	push OBJECT	OBJECT ARRAY OBJECT
"name"	read object key	OBJECT ARRAY OBJECT
:	read colon	OBJECT ARRAY OBJECT
"Bunny"	read string value	OBJECT ARRAY OBJECT
}	pop OBJECT	OBJECT ARRAY
]	pop ARRAY	OBJECT
}	pop OBJECT	empty
end	accept	empty

XGrammar

Problem

PDA's unbounded stack length results in infinite number of states

Can't precompute token masks for all scenarios

Must compute mask during runtime

XGrammar

Preprocessing phase:

Generate adaptive token mask cache for ***context-independent*** tokens

Runtime phase:

Process ***context-dependent*** tokens with persistent execution stack

XGrammar - Token Context

During transition, we either

1) Expand into child rule (push onto stack)
2) Advance within current rule (FSM)
3) Return to parent rule (pop from stack)

For case (1) and (2), we only need the stack's top node - ***Context-independent!***

For case (3), we inspect the stack - ***Context-dependent!***

Experiment show >1% of tokens are context dependent!
(Llama-3.1 on JSON grammar)

XGrammar - Token Mask Cache

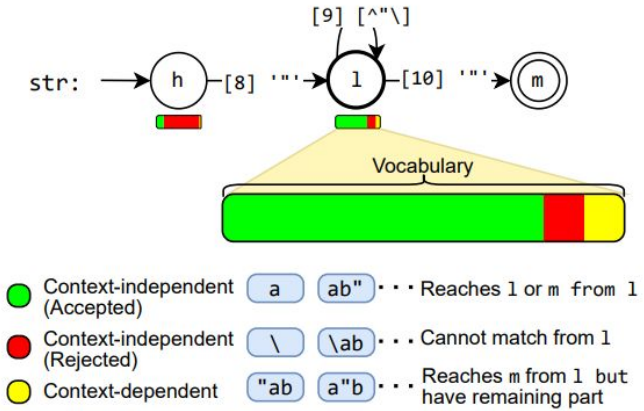


Figure 4. An example for the token mask cache. Tokens are categorized into three types: context-independent (accepted), context-independent (rejected), and context-dependent. The first two types can be directly determined for mask generation at runtime.

Figure from XGrammar paper

Each node has a set of context-independent and context-dependent tokens

XGrammar - Adaptive Storage Format

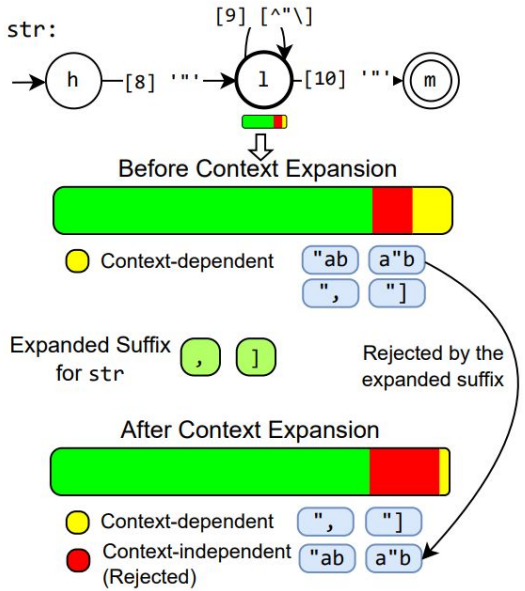
Figure 5. The adaptive storage format. In accept-heavy cases, we store the rejected tokens and context-dependent tokens. In reject-heavy cases, we store the accepted tokens and context-dependent tokens. In rare cases where two kinds of tokens are equal, we compress the accepted and rejected tokens into a bitset of the vocabulary size.

Figure from XGrammar paper

For Llama-3.1 on JSON grammar, reduced total memory from 160 MB to 0.46 MB!

XGrammar - Context Expansion

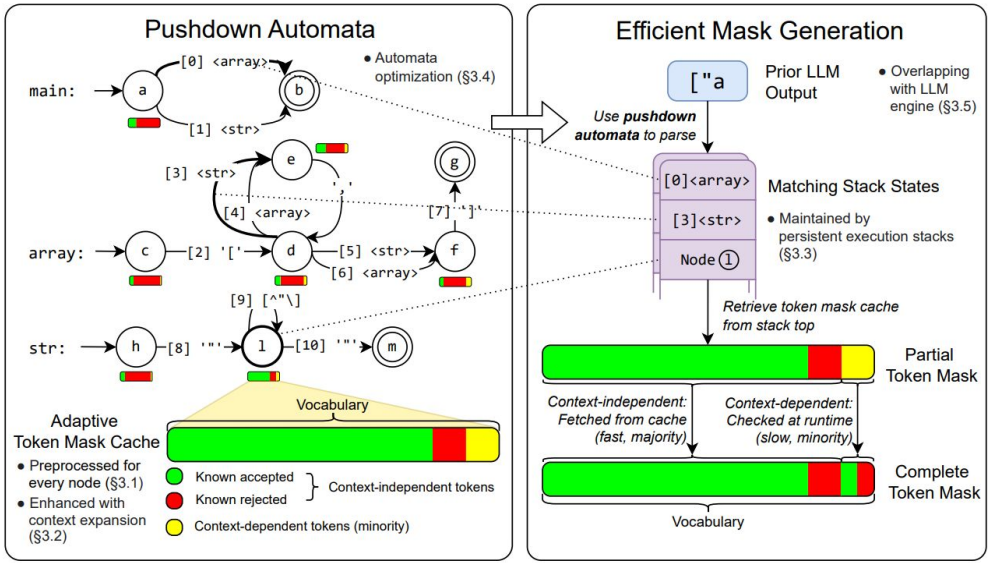
Checking context-dependent tokens still a bottleneck




```
rule: str
possible parent continuations:
    “,” …
    “]” …
    “}” …
    “:” …
```

Can be precomputed!

XGrammar - Persistent Execution Stack



The may be ambiguity in what the stack will be as we parse left-to-right

XGrammar - Persistent Execution Stack

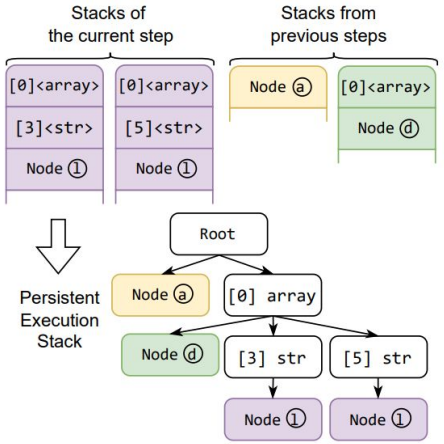


Figure 7. The persistent stack organizes multiple matching stacks from the current step, as well as stacks from previous steps, into a single tree. It reduces memory consumption and supports rolling the state back to previous steps.

Persistent Execution Stack for efficient state branching!

Allows for efficient rollback to check for valid states

XGrammar - PDA optimization

Rule inlining

Before

```
array -> "[" elements? "]"
elements -> value ("," value)*
```

After

```
array -> "[" value ("," value)* "]"
```

XGrammar - PDA optimization

Node Merging

```
state S
 ├── 'a' -> node X
 └── 'a' -> node Y
```

```
state S
 └── 'a' -> merged node XY
```

We can merge when all ways to reach the node are equivalent

XGrammar - Overlap

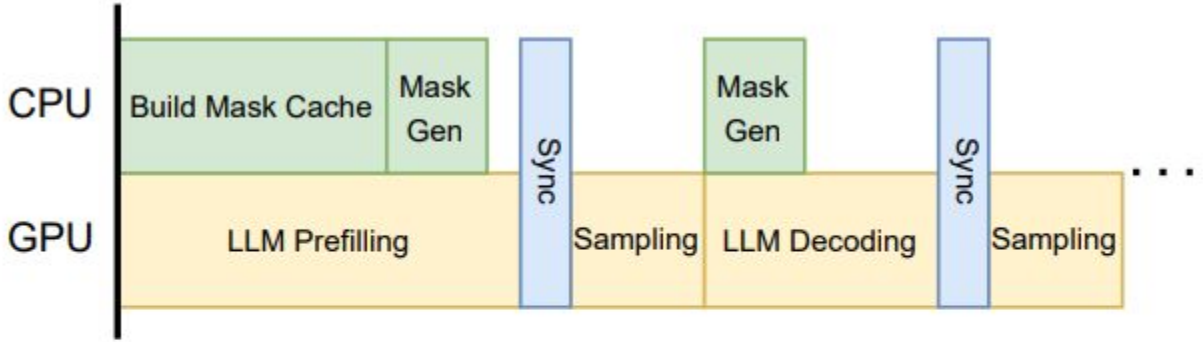


Figure 8. Overlapping building the mask cache with LLM prefilling, and mask generation with LLM decoding to minimize the overhead.

XGrammar - Results

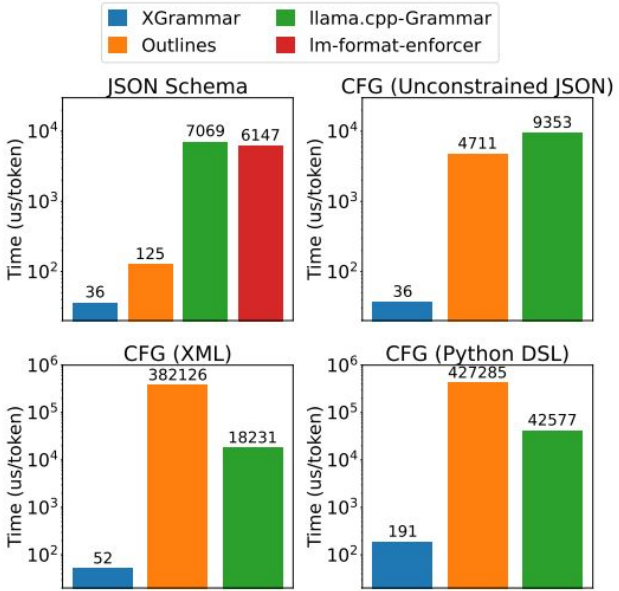


Figure 9. Per token mask generation latency. XGrammar consistently outperforms existing constrained decoding libraries.

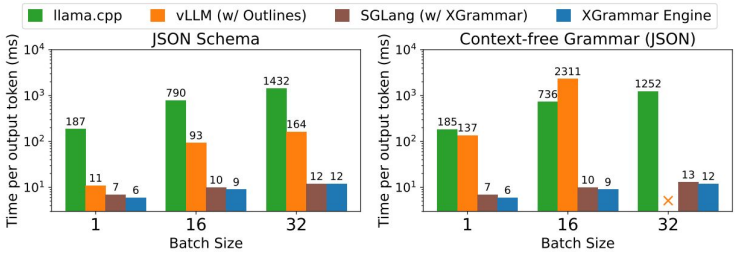


Figure 10. End-to-end evaluation on Llama 3.1 inference with structured constraints. Some results with a batch size of 32 are not reported because their API call time exceeded the API timeout limit of 600 seconds.

XGrammar 2

TagDispatch

Problem: Agent generates free form text then emit a tag (<function=get_weather>), then output follows JSON schema for function. Encoding as one EBNF grammar file is inefficient

Solution: TagDispatch compiles all candidate tag as on DFA and switches between dispatching and dispatched mode

Cross-Grammar Cache

Problem: Multiple EBNF structures may contain the same sub-FSM structure leading to redundant computing of mask

Solution: Cross Grammar cache reuse.

XGrammar 2

Earley-based Adaptive Token Mask Cache

Problem: NPDA can blow up, growing exponentially.

Solution: Adapt the token mask to use Earley states

JIT Compilation of Token Mask Cache

Problem: In dynamic agents, a request may have hundreds of unique tool calls and grammar, but only a few is used. Compiling all increases TTFT.

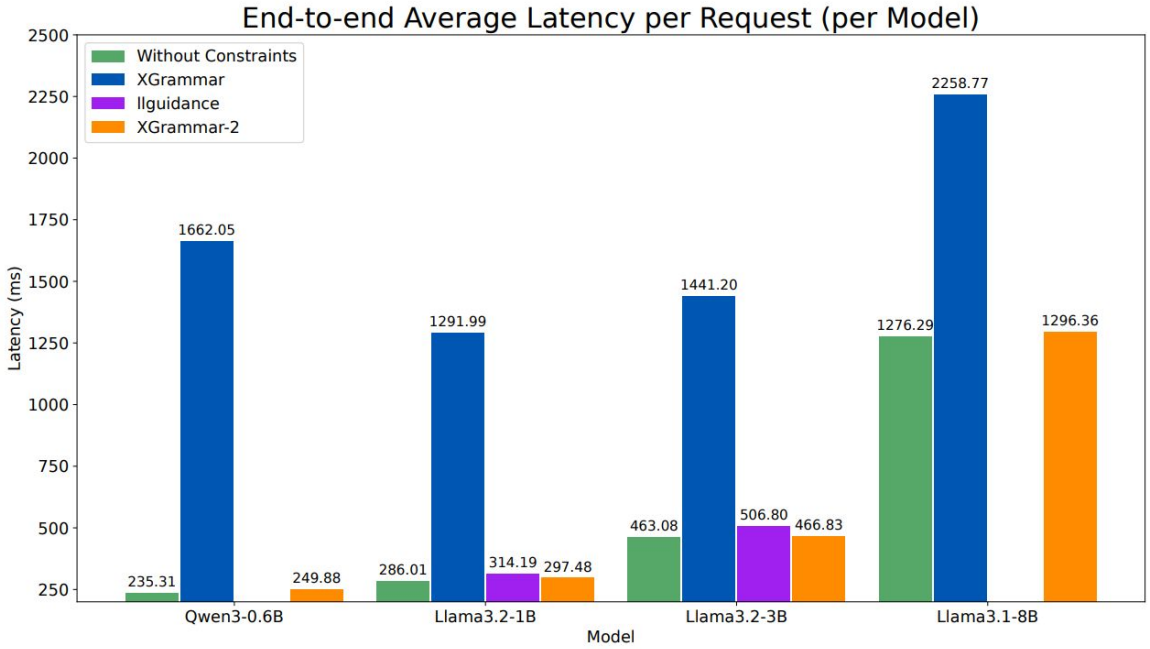
Solution: Only compile when used (JIT)

Repetition State Compression

Problem: Grammar might have bounded repetition, expanding this naively would lead to redundant computation of token-mask caches.

Solution: Introduce repetition construct

XGrammar 2 Performance



- ➔ Fast JSON Decoding with FSM blog
- ➔ Outlines
- ➔ Coalescence
- ➔ Three Models for the Description of Language
- ➔ On Context-Free Languages and Push-Down Automata
- ➔ XGrammar
- ➔ XGrammar-2
- ➔ LLGuidance: Making Structured Outputs Go Brrr