XGRAMMAR: FLEXIBLE AND EFFICIENT STRUCTURED GENERATION ENGINE FOR LARGE LANGUAGE MODELS

Yixin Dong [1] **Charlie F. Ruan** [1] **Yaxing Cai** [2] **Ruihang Lai** [1] **Ziyi Xu** [3] **Yilong Zhao** [4] **Tianqi Chen** [1 2]

ABSTRACT

The applications of LLM Agents are becoming increasingly complex and diverse, leading to a high demand for structured outputs that can be parsed into code, structured function calls, and embodied agent commands. These developments bring significant demands for structured generation in LLM inference. Context-free grammar is a flexible approach to enable structured generation via constrained decoding. However, executing context-free grammar requires going through several stack states over all tokens in vocabulary during runtime, bringing non-negligible overhead for structured generation. In this paper, we propose XGrammar, a flexible and efficient structure generation engine for large language models. XGrammar accelerates context-free grammar execution by dividing the vocabulary into context-independent tokens that can be prechecked and context-dependent tokens that need to be interpreted during runtime. We further build transformations to expand the grammar context and reduce the number of context-independent tokens. Additionally, we build an efficient persistent stack to accelerate the context-dependent token checks. Finally, we co-design the grammar engine with LLM inference engine to overlap grammar computation with GPU executions. Evaluation results show that XGrammar can achieve up to 100x speedup over existing solutions. Combined with an LLM inference engine, it can generate near-zero overhead structure generation in end-to-end low-LLM serving.

1 INTRODUCTION

Recent advancements in large language models (LLMs) have created new possibilities for complex applications such as code generation (Chen et al., 2021; Wang et al., 2021), debugging (Pearce et al., 2022; Mozannar et al., 2024), external tool invocation through function calling (OpenAI, 2024; LangChain, 2024), and robotic control (Liu et al., 2023). These applications bring great demand for the *structured generation* problem, which requires the output of LLMs conforms to specific formats or grammars, such as JSON, SQL or other formats tailored to the task. The downstream applications can then organically consume the structured outputs to perform followup interactions with the system.

Constrained decoding (Deutsch et al., 2019; Kuchnik et al., 2023) is a commonly adopted method for structured generation. It guarantees the output of LLMs adheres to the specified structure through only allowing tokens that conform to the structure to be generated at each decoding step. At each step, constrained decoding first scans the entire vocabulary to identify invalid tokens and sets their probabilities to zero, thereby preventing them from being generated. To support the rich structure formats arising in diverse applications, a flexible mechanism is needed to specify and check the constraints. Context-free grammar (CFG) (Chomsky, 1956; Poesia et al., 2022; Scholak et al., 2021) provides a general approach for defining structures through a set of rules. Each rule contains a sequence of characters or other rules, allowing recursive composition to represent complex structures. Compared to alternative formats such as regular expressions, CFGs offer greater flexibility by allowing recursive structures, making them suitable for describing common languages such as JSON, SQL, and domain-specific languages (DSLs).

However, naively applying CFG to constrained decoding is not efficient because of its flexible nature. First, each decoding step needs to interpret CFG for every possible token in the vocabulary, which can be as large as 128k in Llama 3.1 (Dubey et al., 2024a). Additionally, CFG interpretation requires a stack state that tracks the recursive rules matched so far, making it impossible to precompute and cache all combinatorial combinations of stack patterns ahead of time. Finally, each token in the LLM generation comprises multiple characters, which may cross the boundaries of grammar elements and cause further recursion or stack pop during runtime execution. The misaligned boundaries bring the need to handle them carefully during grammar execution.

In this paper, we introduce XGrammar, a flexible and effi-

[1]Carnegie Mellon University [2]NVIDIA [3]Shanghai Jiao Tong University [4]University of California, Berkeley. Correspondence to: Yixin Dong <yixind@andrew.cmu.edu>.

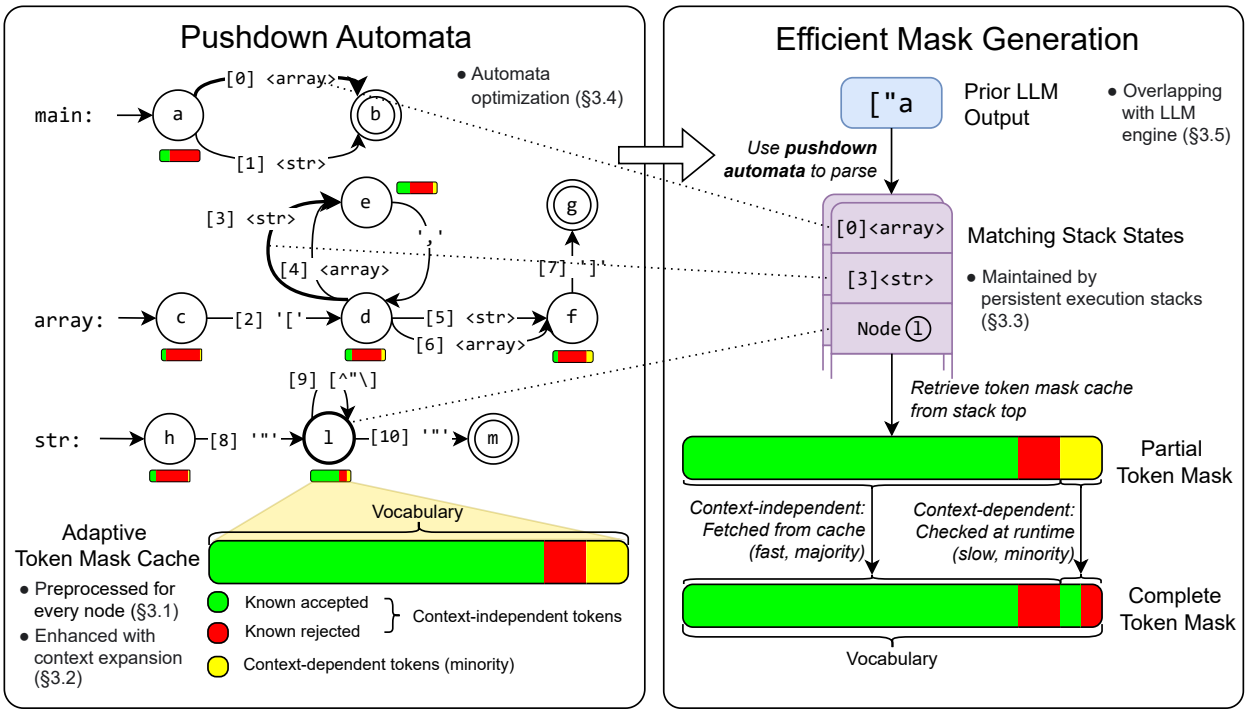

Figure 1. Overview of our approach. XGrammar first uses a pushdown automaton to parse the prior LLM output, flexibly supporting diverse grammars and producing the matching stack states. It then uses the stack top to index into the adaptive token mask cache—our key optimization—to retrieve a partial mask. Most of the partial mask consists of context-independent tokens and is determined during preprocessing. A small portion, however, is context-dependent and resolved at runtime. This yields the complete token mask, thus enabling efficient constraint decoding.

cient structured generation engine for large language models to address the above challenges. XGrammar builds a byte-level pushdown automaton to represent context-free grammars (CFGs). Our main insight(shown in Figure 1) is to categorize the tokens into **context-independent** tokens that can be decided only from the local context of automata and **context-dependent** tokens that require the entire stack state. We precompute the token correctness for all context-independent tokens and store them in an adaptive token mask cache with specific storage formats tailored to each automata location. We also build algorithms to expand the context of each local rule and reduce the number of context-dependent tokens. Additionally, we build a persistent stack-based system to enable rapid state branching and rollback, expediting context-dependent token checks and cache preprocessing. Finally, we co-designed the grammar engine with LLM inference engines to overlap the grammar computations with GPU computations, bringing minimal overhead for structured generation.

The main contribution of this paper is as follows:

- We introduce an adaptive token mask cache that leverages context-independent tokens and significantly reduces mask generation overhead.
- We design a persistent execution stack that enables fast rollback operations, rapid state branching, and rollback, expediting context-dependent token processing.
- We built an efficient grammar engine co-designed with the LLM serving framework to achieve minimal structured generation overhead.

Evaluation shows that XGrammar can achieve up to 100x reduction in per-token latency for context-free grammar compared to current state-of-the-art methods. Additionally, the XGrammar-integrated LLM serving engine for Llama-3.1 models achieves up to an 80x speedup in end-to-end LLM serving with structured output on the H100 GPU. We are open-sourcing XGrammar and integrating it into major open-source LLM frameworks.

2 Background

2.1 LLM Constrained Generation

Large Language Models (LLMs) like GPT-4 (OpenAI et al., 2024), Llama (Dubey et al., 2024a), and Mistral (Jiang et al.,

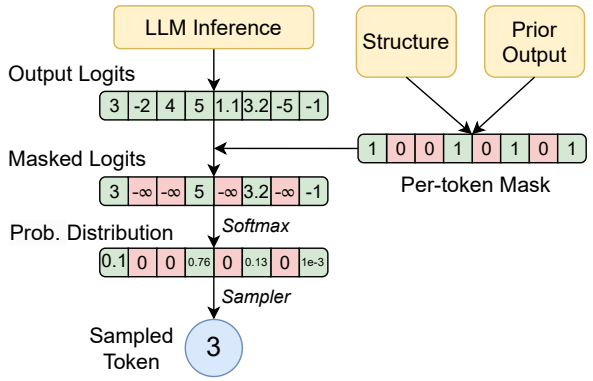

Figure 2. Constrained decoding with per-token mask. The per-token mask prevents LLM from generating tokens that would be invalid according to the structure at that step.

2023) generate text in an auto-regressive manner, predicting one token at a time based on preceding sequence of tokens. The process starts with an initial prompt and continues as the model iteratively appends tokens until the response is complete. In LLMs, tokens serve as the basic input and output units. Each token represents a fixed string but may not correspond to a complete semantic unit or may break a Unicode character (Wang et al., 2019), creating challenges for structured text generation. At each step, the model produces a logits vector across its vocabulary, which is then converted into a probability distribution using the softmax function (Bridle, 1989). A sampler then selects the next token from this distribution.

Constrained decoding guides the structure of LLM-generated text by restricting available tokens at each step, as illustrated in Figure 2. At each step, tokens that would violate the required structure are identified as invalid. Their logits are set to $-\infty$, effectively assigning them zero probability after the softmax operation and preserving the relative probabilities of other valid tokens. This ensures that only valid tokens are sampled. Efficiently identifying and masking invalid tokens is essential, as it directly impacts generation speed.

2.2 Context-free Grammar and Pushdown Automata

Context-free grammar (CFG) (Chomsky, 1956) is widely used to define structures in structured generation. With an example shown in Figure 3, CFG contains multiple rules, each including characters or references to other rules, allowing recursive composition to define complex structures. This makes CFG suitable for languages such as JSON, SQL, and various domain-specific languages. CFG's recursive nature provides greater expressive power than simpler patterns, such as regular expressions, which are also frequently

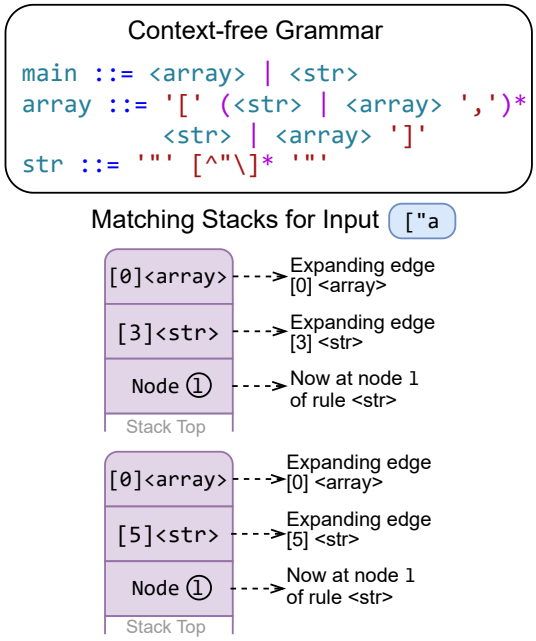

Figure 3. Up: A context-free grammar for arrays and strings that can be recursively composed. This CFG is converted into the pushdown automata in Figure 1. `[^"\]` denotes every character except `"` and `\`. Down: Two possible matching stacks for matching the string `["a` to the CFG. Each stack represents a possible expansion of the rules in the CFG. The edges and nodes in the stack correspond to the transitions and states in the PDA in Figure 1.

applied in LLM structured generation.

Pushdown automata (PDA) (Schützenberger, 1963; Evey, 1963) are typically used to recognize languages generated by CFGs, as they employ a stack to manage nested structures. In this paper, we use a definition of PDA that is equivalent to the original one, but more conducive to explaining the algorithm. An example of PDA is shown in Figure 1, and its stacks are shown in detail in Figure 3. A PDA consists of multiple finite state automata (FSA), each representing a grammar rule, with the stack handling recursive rule expansions. The transitions in the FSA include two types: character edges, which accept specific characters, and rule reference edges, which allow recursive entry into other rules. A formal definition of the PDA is provided in Appendix A. To match a string, the PDA begins with the main rule, recursively expanding child rules by pushing rule-reference edges onto the stack; once a rule is fully matched, it pops the stack to return to the previous rule. The top of the stack holds the current node reached. If the grammar is non-deterministic, meaning there can be multiple possible transitions in the

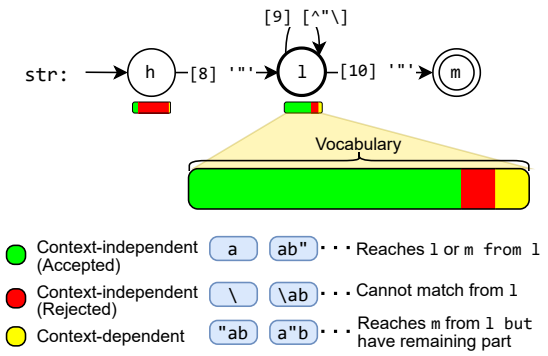

Figure 4. An example for the token mask cache. Tokens are categorized into three types: context-independent (accepted), context-independent (rejected), and context-dependent. The first two types can be directly determined for mask generation at runtime.

PDA for the same input character, the PDA can maintain multiple parallel stacks for each path, ensuring flexibility. However, the unbounded stack length results in an infinite number of possible states, making it impractical to precompute token masks for all scenarios, thus posing challenges for efficient constrained decoding.

3 XGrammar

As shown in Figure 1, XGrammar utilizes a byte-level pushdown automaton to interpret the context-free grammar. This byte-level design allows each character edge to include one or more bytes, handling irregular token boundaries and supporting tokens containing sub-UTF8 characters. The automaton's structure is optimized to accelerate matching, as described in §3.4. In the preprocessing phase, we generate an adaptive token mask cache, as detailed in §3.1, which accelerates runtime mask generation by precomputing context-independent tokens. The effectiveness of this cache is further enhanced by context extension in §3.2. At runtime, the token mask cache quickly generates most of the mask, while the persistent execution stack in §3.3 efficiently processes the rest context-dependent tokens. Additionally, mask generation and LLM inference are overlapped in §3.5 to minimize the overhead of constrained decoding. Once the LLM generates a new token under the mask constraint, this token is then used to update the stack state of the pushdown automaton for the next mask generation.

3.1 Adaptive Token Mask Cache

To accelerate the generation of the token mask cache, the adaptive token cache categorizes tokens into two types (Figure 4): context-independent tokens, which constitute the vast majority and can be pre-computed, and context-dependent tokens, which require slower, on-the-fly processing but are relatively few. This token classification relates to how tokens are validated by the pushdown automaton. We found that, considering the transition of the stack state, the process of matching tokens to the automaton can be divided into three categories:

1. The matching process expands into a child rule, pushing new elements onto the stack.
2. The matching process advances within the current rule, updating the stack top node to a new position.
3. The matching process reaches the end of the current rule and returns to a parent rule, popping elements from the stack.

Validating tokens in the former two cases only relies on the stack top node, which represents the position within the current rule, so we define these tokens as *context-independent tokens*. The tokens in the third type, however, requires inspecting the entire running stack in validation, and are defined as *context-dependent tokens*. For every node of the pushdown automaton, there is a set of context-independent tokens with this node being at the top of the stack at runtime, and their validity can be determined ahead of time. Therefore, we precompute the validity of these tokens and store them in a cache with the stack top node as the key, which we refer to as the adaptive token mask cache. It also adaptively selects the most efficient storage format based on the cache's contents, as explained in the next paragraph.

At runtime, we retrieve the validity of context-independent tokens directly based on the top of the stack to generate the token mask. The remaining few context-dependent tokens are validated by executing the pushdown automaton with the full stack. If parallel stacks exist due to the ambiguity of the grammar, the token masks for every stack is merged into a final token mask by finding the union of the accepted tokens in each mask. The computation for the token mask is significantly reduced because our method do not need to check context-independent tokens at runtime. Experiments show that context-dependent tokens account for only a minor proportion, amounting to less than 1% (1134 out of 128k) for the Llama-3.1 model using JSON grammar.

Adaptive storage. The token mask cache adopts an adaptive storage format to reduce memory usage, as illustrated in Figure 5. For each automaton node, the token mask cache divides the vocabulary into three parts: the accepted context-independent tokens, the rejected context-independent tokens, and the context-dependent tokens. Since these three parts together cover all tokens, it is sufficient to store only the two smaller subsets. We observe that, for a set of context-independent tokens, they tend to be either almost entirely accepted, namely *accept-heavy* cases, or almost entirely

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.

rejected, namely *reject-heavy* cases. This arises because, if wildcards can be matched from the current node, such as the wildcard `[^"\]*` in the rule of string, nearly all tokens are valid; whereas if the node only accepts a few specific characters, nearly all tokens are invalid. Based on this observation, we designed the following adaptive storage format:

1. For accept-heavy cases, we store the rejected context-independent tokens and context-dependent tokens in two arrays.
2. For reject-heavy cases, we store the accepted context-independent tokens and context-dependent tokens in two arrays.
3. For rare cases where the accepted and rejected tokens are roughly equal, we store the accepted and rejected context-independent tokens and compress them into a bitset matching the vocabulary size.

Thus, in both accept-heavy and reject-heavy cases, the adaptive storage format only requires storing a small subset of tokens, significantly reducing memory usage. In practice, we will enumerate the three storage types, calculate their respective costs, and choose the storage type with the smallest size. For Llama-3.1 model and JSON grammar, this adaptive storage method can effectively reduce the total memory usage to 0.2% (from 160 MB to 0.46 MB).

Additionally, when multiple parallel stacks exists, we need to merge the token masks. The merging algorithm of token masks is optimized based on storage type, as shown in Algorithm 1. For an accept-heavy mask (many accepted tokens, storing only rejected tokens), it intersects the rejected tokens with $PartialRej$. For a reject-heavy mask (many rejected tokens, storing only accepted tokens), it combines accepted tokens with $PartialAcc$. In the final mask, the rejected tokens are the set difference $PartialRej \setminus PartialAcc$. This algorithm limits set operations to small token subsets, thus enhancing efficiency.

Algorithm 1 Efficiently Merge Token Masks

Input: Token masks for k parallel stacks $\{M_i = (Acc_i, Rej_i)\}_{i=1}^{k}$, vocabulary $\mathcal{V}$.
Output: The final token mask $M = (Acc, Rej)$.
Initialize $PartialAcc \leftarrow \emptyset$, $PartialRej \leftarrow \mathcal{V}$
for $i = 1$ **to** k **do**
  if M_i is accept-heavy **then**
    M_i only stores rejected token list Rej_i
    $PartialRej \leftarrow PartialRej \cap Rej_i$
  else
    M_i only stores accepted token list Acc_i
    $PartialAcc \leftarrow PartialAcc \cup Acc_i$
  end if
end for
$M \leftarrow (\mathcal{V} \setminus (PartialRej \setminus PartialAcc),$
$\quad PartialRej \setminus PartialAcc)$

3.2 Context Expansion

Although the adaptive token mask cache effectively reduces the number of tokens checked at runtime, checking all context-dependent tokens remains an efficiency bottleneck at runtime. To further reduce the number of context-dependent tokens, XGrammar introduces context expansion, which leverages the grammar's context information to reject more context-dependent tokens during preprocessing, as shown in Figure 6.

As described in the last section, a token is context-dependent when we reach the end of the current rule during matching, but there is still a remaining part of the token that requires further checking by returning to the parent rules. However, through analysis of the grammar, we can observe that in a large portion of the cases, the remaining part of a token is invalid. This is because, for many rules, when they reach their end, there are only a limited number of positions they can return to within their parent rules, and the set of strings that can be further matched from those positions is also limited.

Based on this observation, context expansion precomputes, for each rule, the set of strings that can be accepted after returning to the parent rules, called the *expanded suffix*. The remaining unmatched part of the context-dependent tokens should either be a prefix of the expanded suffix or start with the expanded suffix. Otherwise, they are rejected.

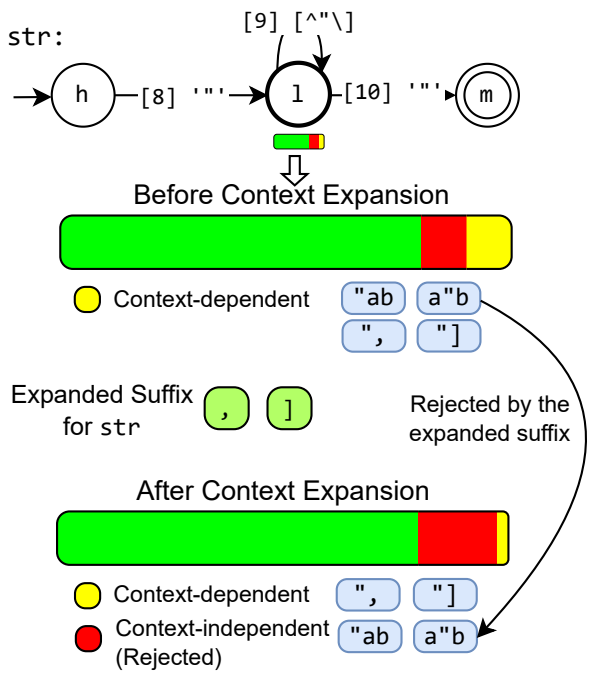

Figure 6. The context expansion. Each rule obtains a set of expanded suffices, representing the set of strings that must be matched after completing this rule. The remaining unmatched part of the context-dependent tokens should either be a prefix of the expanded suffix or start with the expanded suffix. Otherwise, they are rejected.

This filtering process effectively reduces the number of context-dependent tokens by eliminating those that would fail in higher-level rule contexts. Applied to the Llama-3.1 model and JSON grammar, this technique reduces context-dependent tokens by 90% (from 1,134 to 120).

Algorithm 2 describes the context expansion process that finds the expanded suffix of each rule. For a rule R, we utilize a finite state automaton (FSA) $\mathcal{A}_R^{\text{ctx}}$ (ctx is the abbreviation for context) to represent the expanded suffix, and that is extracted from the pushdown automata. We first find all edges $e = (s, t)$ in the pushdown automata that references R and belongs to rule R'. R' is not necessarily different from R. Then we find a subgraph of the automaton of rule R' starting from t to represent the possible strings that can follow R via depth-first search (DFS). However, we will not consider edges in the subgraph that reference other rules to avoid recursive references between rules, so the edges in the extracted subgraph will only have character labels.

Algorithm 2 Extract the Expanded Suffix Automaton

Input: Pushdown automaton $\mathcal{P}$, rule R
Output: Expanded context FSA $\mathcal{A}_R^{\text{ctx}}$ for R
Initialize $\mathcal{A}_R^{\text{ctx}}$ as an empty FSA
for edge $s \xrightarrow{R} t$ in $\mathcal{P}$ referencing R **do**
  { $\mathcal{A}_\delta$ is an FSA for the partial result }
  Initialize $\mathcal{A}_\delta$ as an empty FSA, $visited \leftarrow \{\}$
  Add node t to $\mathcal{A}_\delta$
  EXTRACTONE(t, $\mathcal{A}_\delta$, $visited$)
  { Merge the partial result into the final result }
  $\mathcal{A}_R^{\text{ctx}} \leftarrow$ FSAUNION($\mathcal{A}_R^{\text{ctx}}$, $\mathcal{A}_\delta$)
end for

function EXTRACTONE($start$, $\mathcal{A}_\delta$, $visited$)
  if $start$ in $visited$ **then**
    return
  end if
  Add $start$ to $visited$
  { Stop search for nodes with rule-referencing edges }
  if $start$ is a final node in $\mathcal{P}$ **or**
      has an edge referencing another rule **then**
    Mark $start$ as final in $\mathcal{A}_\delta$
    return
  end if
  {Now all outward edges of $start$ are character edges}
  for edge $start \xrightarrow{c} end$ from $start$ **do**
    Add end and $start \xrightarrow{c} end$ to $\mathcal{A}_\delta$
    EXTRACTONE(end, $\mathcal{A}_\delta$, $visited$)
  end for
end function

If a node has both character edges and edges referencing other rules, we will stop the search at this node. The extracted subgraph is then merged into $\mathcal{A}_R^{\text{ctx}}$. This process is repeated for all rules, and the extracted $\mathcal{A}_R^{\text{ctx}}$ is used to reject context-dependent tokens cannot match any string in it after finishing matching rule R.

Although we do not consider rule-referencing edges when extracting the expanded context automata, this algorithm can still extract many useful context information. That is because the inlining optimization introduced in §3.4 inlines fragment rules into their parent rules, reducing the need to check into child rules to reject context-dependent tokens.

3.3 Persistent Execution Stack

As the grammar engine still needs to handle context-dependent tokens, we need to efficiently execute the pushdown automata for these tokens. Additionally, we also need to execute the pushdown automata for preprocessing the context-independent token sets for all positions in the pushdown automata. In both cases, we need to maintain multiple

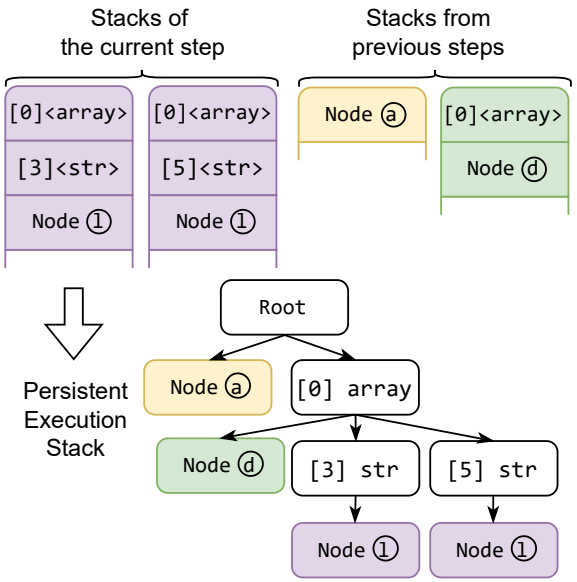

Figure 7. The persistent stack organizes multiple matching stacks from the current step, as well as stacks from previous steps, into a single tree. It reduces memory consumption and supports rolling the state back to previous steps.

parallel stacks and branch out as we match the characters in each token. To support efficient state branching, we introduce the persistent execution stack (Driscoll et al., 1989) to manage the multiple stacks and efficiently execute the pushdown automata. It can also manage the stacks from previous time points and enable the state rollback operation, effectively speeding up the execution of the pushdown automata on a set of tokens.

As shown in Figure 7, the persistent execution stack manages a set of stacks, which are either the parallel stacks from the current time point or the stacks from previous time points, into a single tree, and every stack is represented by a path from the root node on the tree. The stack top node is stored as a pointer to the node in the tree. Since the stacks from adjacent time points often share most of the deeper elements and only a few nodes are pushed or popped, this merging avoids memory redundancy for storing multiple stacks. When matching a new character from a token, we may need to split the stack into multiple stacks due to the ambiguity of the grammar, each corresponding to a different expansion of grammar rules. In this case, we only need to split the branch for that stack instead of copying the whole stack, which reduces the overhead of state branching.

Additionally, the persistent execution stack enables fast state rollback by maintaining the stack from previous time points. At runtime, a sliding window of history is maintained. To roll back to a previous state, we only need to change the current stack pointers, which requires constant time. This rollback operation is particularly useful for checking a large set of tokens, as many tokens share a common prefix with other tokens, such as `read`, `ready`, and `reader` all sharing the prefix `read`. All the checked tokens are sorted in lexicographical order to find the maximum length of the common prefixes. Then the tokens are checked one by one, and before checking each token, the state rolls back to just after the common prefix with the previous token. Therefore, we can avoid the redundant checks of these common prefixes, reducing the number of characters that need to be checked. For Llama-3.1 model and JSON grammar, this approach reduces the number of characters that need to be checked across the entire vocabulary to 30%, significantly speeding up the preprocessing stage.

The rollback operation enables more applications with efficient structured generation. There are many LLM applications that involve rolling back the output to a previous token. For instance, the jump-forward decoding (Yin et al., 2024) requires retokenization, which involves rolling back some tokens in the context and then inserting new tokens. To ensure structured generation can continue after rolling back tokens, we can roll back the automaton state simultaneously with the output token rollback. There are also many LLM applications that requires LLMs generate in a tree structure, such as in Tree-of-thought (Yao et al., 2024), SGLang (Zheng et al., 2024), and the speculative model in the speculative decoding algorithm SpecInfer (Miao et al., 2024). We can maintain the automata state for every branch of the output tree, and when the output branches, we can quickly split the automaton state, maintaining separate matching states for each output branch. This branching is fast because we only need to maintain the stack top pointer on the tree for every branch. Therefore, the persistent execution stack enables us to ensure efficient structured generation for all these applications.

3.4 Pushdown Automata Structure Optimizations

We will perform additional optimizations to improve the structure of pushdown automata to speed up the efficiency of final execution. These optimizations draw from traditional compiler optimization concepts, but we find them particularly useful for efficient constrained decoding.

Rule inlining. There could be many fragment rules, i.e. rules with only a few elements, in the specified context-free grammar, which are then converted into small FSA in the pushdown automaton. On the one hand, this increases the ambiguity of the grammar since we need to inspect into these fragment rules and check during the execution of the pushdown automata. On the other hands, during context expansion, references to fragment rules are not considered,

so the extracted context automata will be smaller. We will miss the opportunity to reject context-dependent tokens based on the structure of these fragment rules.

To address this issue, we introduce an automatic inlining strategy (Scheifler, 1977) for fragment rules. We iteratively pick rules that do not reference other rules and inline them into the parent rules. To avoid the explosion of the automaton size, we limit the size of the inlined rule and the size of inlined result to constants. This inlining process almost eliminated fragment rules, thereby improving the efficiency of token checking and enhancing the effectiveness of the context expansion.

Pushdown automata node merging. For pushdown automata, in many cases, the ambiguity comes from multiple outward edges of a node with the same label. When matching tokens, if we arrive at this node, and the next character just matches the label, the matching stack will be split into multiple stacks, one for each outward edge. The increase in the number of stacks increases the computation as we need to check the context-dependent tokens for each stack and merge the token masks. To reduce this kind of ambiguity, the node merging algorithm merges the subsequent nodes that satisfy: a) they are pointed to by edges with the same label originating from the same point b) they are not pointed to by other edges.

Additionally, the epsilon edge also increases the ambiguity of the matching process. An epsilon edge $s \xrightarrow{\epsilon} t$ in the automata means that the matching process can directly move from s to t without consuming any characters. If the matching process arrives at s, the execution stack will split into two stacks, one with s at the top and the other with t, both of which can continue matching. To reduce this kind of ambiguity, the node merging algorithm also merges the nodes s and t into a single node, as long as s has no other outward edge or t has no zero inward edge.

These two optimizations preserves the equivalence of the automaton, but reduces the number of nodes and edges. At runtime, the number of stacks and the computation required for token checking are reduced, speeding up the mask generation process.

3.5 Overlapping Mask Generation and LLM Inference

With the optimizations mentioned above, the token mask generation process is significantly accelerated, but it still requires CPU computation. To further eliminate the overhead of constrained decoding, we overlap the computation for mask generation with the LLM inference process, as shown in Figure 8. We observed that the mask generation process and LLM inference process can be overlapped. That is because the mask generation only requires CPU, and only depends on the previously generated tokens. The LLM inference process except the sampling stage only requires GPU, and also only depends on the previously generated tokens. Therefore, we can parallelize the mask generation process on the CPU with the LLM inference process on the GPU. We will synchronize before sampling, and the GPU will obtain the mask from the CPU and perform masked sampling to generate the new token. Additionally, the preprocessing stage can also be overlapped with the LLM prefilling stage, where the LLM processes the prompt. This orchestration between CPU and GPU ensures that the token restrictions are applied seamlessly, with almost zero overhead for LLM inference. In practice, the time for mask generation is less than the time for LLM inference, so the mask generation process will not become the bottleneck of the generation process.

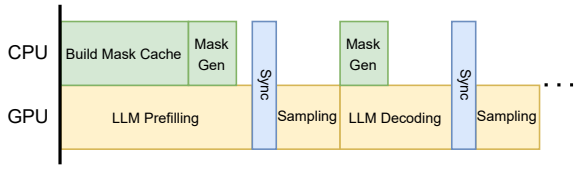

Figure 8. Overlapping building the mask cache with LLM prefilling, and mask generation with LLM decoding to minimize the overhead.

4 Evaluation

We implement XGrammar in 12,000 lines of core C++ code, and we provide Python bindings to facilitate seamless integration with LLM inference frameworks. In this section, we evaluate XGrammar to answer the following questions:

- Can XGrammar efficiently support each step of constrained decoding? (§4.1)
- Does XGrammar achieve minimal overhead for end-to-end structured generation in LLM serving? (§4.2)
- How effective is each optimization technique introduced in XGrammar? (§4.3)
- How does XGrammar effect downstream structured generation tasks? (§4.4)

4.1 Mask generation efficiency

This section evaluates the efficiency of mask generation to measure the overhead introduced by constraint decoding. We first assess regex-based methods using JSON schemas, which can be converted into regex. To test more complex cases beyond regex capabilities, we evaluate context-free grammars, including unconstrained JSON (from ECMA-404 (Ecma International, 2013)), XML (based on the XML

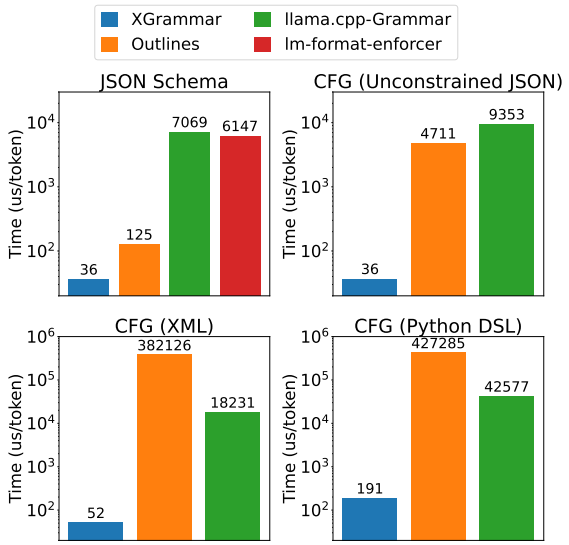

Figure 9. Per token mask generation latency. XGrammar consistently outperforms existing constrained decoding libraries.

1.0 standard (Bray et al., 2008)), and a Python DSL (adapted from the Python Grammar Specification (Python Software Foundation, 2024)). Unconstrained JSON cannot be handled by regex-based methods due to its support for arbitrarily nested lists and objects. The Python DSL covers basic control flow (if, for, while) and data types (str, int, float, bool) but ignores indentation. For JSON schema and unconstrained JSON, we use the JSON-mode-eval dataset (Nous-Research, 2024), and for XML and Python, we use a synthetic dataset. For baselines, we choose three popular constrained generation libraries: Outlines (Willard & Louf, 2023) (v1.0), the grammar engine in llama.cpp (Gerganov, 2023) (b3998), and lm-format-enforcer (Gat et al., 2025) (v0.10.9, a regex-based method that does not support CFG). All methods are evaluated on Llama-3.1-8B-Instruct using an AMD Ryzen 9 7950X CPU and an NVIDIA RTX 4090 GPU.

The results are shown in Figure 9. XGrammar consistently achieves the lowest latency across all tasks, with under 40 µs per token for JSON Schema and CFG (JSON), and under 200 µs for XML and Python DSL. It delivers up to 3x speedup on JSON Schema and over 100x on CFG, compared to the best baseline in each case.

4.2 End-to-End LLM Engine Evaluation

This section evaluates XGrammar in LLM serving scenarios. We co-design XGrammar with the serving engines using the overlapping technique introduced in §3.5. We integrate it into widely used end-to-end LLM serving engines, including the C++-based MLC-LLM (MLC team, 2023a) and the Python-based SGLang (Zheng et al., 2024), showcasing

Table 1. End-to-end structured generation efficiency across different models, measured in time per output token (ms) on the JSON Schema task and the Llama-3.1 8B model. XGrammar demonstrates superior performance across different models.

Model	SGLang + Outlines	SGLang + XGrammar
Llama-3.1 8B	44.2	6.8
DeepSeek-V2-Lite 16B MOE	15.8	4.8

XGrammar's adaptability and efficiency across different deployment environments.

We first compare the efficiency of several LLM engines that support structured generation, including vLLM (Kwon et al., 2023b) (v0.6.3) with Outlines and llama.cpp with its built-in grammar engine. Efficiency is measured by the average time per output token (TPOT), which reflects the overhead of applying constraints during token generation. The evaluations are conducted using Llama-3.1-8B-Instruct under both JSON schema and CFG (unconstrained JSON). All tests are run in an online serving setting with fixed batch sizes, on hardware with an AMD EPYC 7R13 CPU and an NVIDIA H100 GPU. To ensure a fair comparison, we set a fixed maximum output length. On average, the input has 139 tokens, and the generated output has 53 tokens.

The experiment results are shown in Figure 10. XGrammar achieves the best TPOT among all baselines for both JSON Schema and CFG. The computation of vLLM and llama.cpp is hindered by their grammar engines' longer preprocessing and per-token processing time. The decrease in TPOT speed in vLLM becomes particularly noticeable with larger batch sizes. This is because a larger batch size leads to higher throughput, putting greater pressure on grammar processing on the CPU side. Overall the XGrammar-based structured generation solutions can bring up to 80x output token rate compared to existing solutions. This proves the effectiveness of XGrammar's system optimizations and its co-design with the serving engine.

We also study the effectiveness of XGrammar across different models in end-to-end serving scenarios. As the result shown in Table 1, for various models, SGLang integrated with XGrammar consistently outperforms its integration with Outlines, demonstrating the robustness of XGrammar's effectiveness across model architectures. This experiment also provides direct evidence that XGrammar outperforms other constrained decoding library when running on the same serving engine.

Additionally, we examine the overhead of XGrammar's constrained decoding in end-to-end scenarios by measuring performance on the MLC-LLM engine with and without

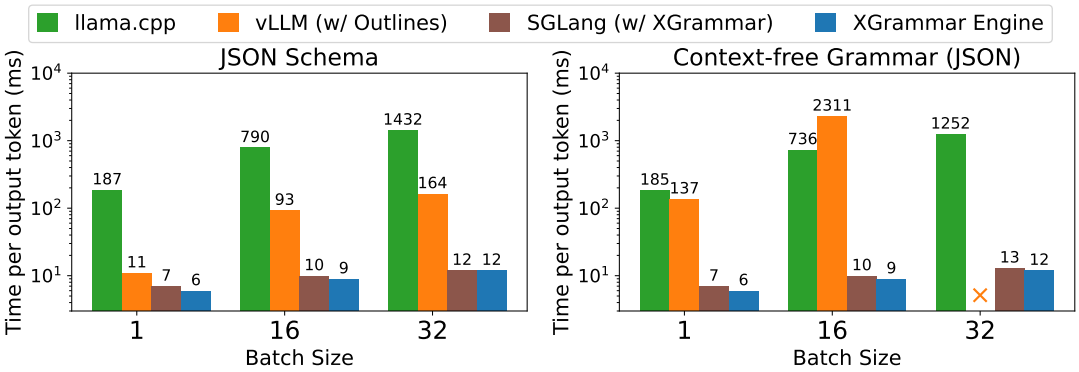

Figure 10. End-to-end evaluation on Llama 3.1 inference with structured constraints. Some results with a batch size of 32 are not reported because their API call time exceeded the API timeout limit of 600 seconds.

Table 2. The impact on performance of enabling and disabling XGrammar tested on the MLC-LLM Engine and the Llama-3.1 8B model. TPOT (ms) is reported. XGrammar introduces minimal overhead to the serving engine with better generation quality.

Task	Batch Size	TPOT w/o XGrammar	TPOT w/ XGrammar
JSON Schema	1	6.2	6.3
	16	9.0	9.2
CFG (JSON)	1	6.3	6.3
	16	9.0	9.1

Table 3. Ablation study of optimization techniques in XGrammar.

Optimization	Per-token latency (ms)
PDA Baseline	65.776
+ Node merging	38.280 (1.7×↓)
+ Adaptive token mask cache	0.154 (248.6×↓)
+ Rule inlining	0.035 (4.4×↓)
+ Context Expansion	0.018 (1.9×↓)

XGrammar. As shown in Table 2, enabling XGrammar enhances output quality with nearly zero overhead in TPOT. This is attributed to efficient mask generation and the overlapping of grammar processing with GPU execution.

4.3 Ablation Study of Optimization Techniques

In this section, we investigate the impact of various optimizations introduced in xgrammar on mask generation performance, to better illustrate our design decisions. We begin by implementing a baseline using a pushdown automaton parser without any optimizations, where each token mask is generated by checking the entire vocabulary to determine whether parsing can proceed. Building on this baseline, we progressively add the optimizations described in this paper, namely node merging, the adaptive token mask cache, rule inlining, and context expansion. For each configuration, we measure the average mask generation time on the CFG (unconstrained JSON) task with the Llama-3.1 8B model and the json-mode-eval dataset. As shown in Table 3, the results show that the adaptive token mask cache has the greatest impact in terms of speedup, while other techniques, including node merging, rule inlining, and context expansion, also yield noticeable improvements.

Table 4. Impact of XGrammar on structured generation tasks. XGrammar ensures 100% syntactic correctness of the generated outputs.

Task	Accuracy w/o XGrammar	Accuracy w/ XGrammar
Function calling	62%	100%
XML code generation	80%	100%

4.4 Impact of XGrammar on Structured Generation Tasks

XGrammar can improve the generation quality of LLMs by ensuring that the output strictly adheres to the given format. We evaluate the impact of XGrammar on two strcutured generation tasks: function calling (i.e., JSON generation guided by a JSON schema) and XML code generation. For function calling, we use the json-mode-eval dataset, while for XML code generation, we rely on a synthetic dataset. We measure the syntactic correctness of the generated function calling outputs and XML code using LLaMA-3.1 8B. As shown in Table 4, XGrammar significantly improves generation accuracy. We observe that without XGrammar, the model often includes additional explanations alongside the intended code output, or the generated JSON contains an unexpected type. This makes the output unsuitable for direct use by downstream applications. XGrammar avoids this issue by enforcing grammar constraints.

5 Related Work

Several works looked at algorithm improvements for structured generation. (Koo et al., 2024) proposes an algorithm to convert character-level pushdown automata to token-level pushdown automata. (Wang et al., 2023) specifies LLM output structure through prompting. (Rozière et al., 2024; Chaudhary, 2023; Li et al., 2023) explore finetuning LLMs for higher quality structured generation. XGrammar's approach is orthogonal to these methods and can be combined with these approaches.

There has also been some previous work focusing on constrained decoding. Some methods use regex to represent syntax, such as lm-format-enforcer (Gat, 2024), but they cannot handle more complex CFGs. Synchromesh (Poesia et al., 2022) and llama.cpp (Gerganov, 2023) utilizes LR parser and PDA respectively to handle CFG and generate the mask, but requires checking the entire vocabulary at runtime, which incurs large overhead. Outlines (Willard & Louf, 2023) handles the grammar through a lexer and parser, and uses caching to accelerate mask generation, but it only considers the most recent lexer token, which may lead to incorrect judgments when LLM tokens span across multiple lexer tokens (Appendix A in (Koo et al., 2024)). Syncode (Ugare et al., 2024) uses a lexer and parser with a cache spanning multiple lexer tokens, but requires all tokens to be processed offline, leading to substantial preprocessing overhead. In this work, XGrammar leverages a PDA to support CFGs and introduces an adaptive token mask cache to adaptively handle tokens during preprocessing and runtime, combining other system optimizations to achieve minimal overhead in both stages.

Guidance (Guidance-ai, 2024), LMQL (Beurer-Kellner, 2023), SGLang (Zheng et al., 2024) provide flexible ways to declare the structures. XGrammar is complementary to these improvements and can be used as the backend engine to speedup their execution.

LLM serving engines (MLC team, 2023a; Zheng et al., 2024; Kwon et al., 2023a; hiworldwzj et al., 2024) employ various techniques to support efficient LLM generation for multiple concurrent users, including engine-level techniques such as continuous batching (Yu et al., 2022) for dyanmic request scheduling, and low-level KV cache technique PagedKVCache (Kwon et al., 2023a) for efficient memory management. Also, AICI (Moskal et al., 2024) proposes a CPU-GPU parallel computation paradigm to accelerate structured generation. These LLM serving engines can leverage XGrammar for efficient, structured generation on top of their existing LLM inference techniques.

6 Conclusion

We proposed XGrammar, a flexible and efficient structured generation engine for LLMs. XGrammar separates the vocabulary into context-independent tokens and context-dependent ones. It prechecks the context-dependent tokens and stores the result in an adaptive token mask cache. We further introduce a persistent stack to speed up the execution of context-dependent checks. Finally, we co-design the grammar engine with LLM inference to overlap grammar execution with GPU computation. Our system greatly speeds up the token mask generation process in token mask and enables zero overhead structure generation in end-to-end LLM inference flows. We hope our system can enable a broader range of structure generation across platforms.

Acknowledgements

This work was supported in part by NSF award CNS-2211882, and gifts from OctoAI, Qualcomm, and CMU opensource software fellowships. We are grateful to Sasa Misailovic for his guidance as the shepherd of this paper. We also appreciate the valuable feedback and discussions from the DeepSeek, SGLang, TensorRT-LLM, vLLM, and WebLLM teams (listed alphabetically). Additionally, we thank Weihua Du, Haoran Peng, Xinyu Yang, Zihao Ye, Jieyu Zhang, Zhihao Zhang, and Ligeng Zhu for their insightful input and thoughtful conversations.

References

Beurer-Kellner, L. GitHub - eth-sri/lmql: A language for constraint-guided and efficient LLM programming. — github.com. https://github.com/eth-sri/lmql, 2023. [Accessed 31-10-2024].

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. Extensible Markup Language (XML) 1.0 (Fifth Edition). https://www.w3.org/TR/xml, November 2008. W3C Recommendation, 26 November 2008.

Bridle, J. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In Touretzky, D. (ed.), *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989. URL https://proceedings.neurips.cc/paper_files/paper/1989/file/0336dcbab05b9d5ad24f4333c7658a0e-Paper.pdf.

Chaudhary, S. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca, 2023.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Chomsky, N. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956. doi: 10.1109/TIT.1956.1056813.

Deutsch, D., Upadhyay, S., and Roth, D. A general-purpose algorithm for constrained sequential inference. In Bansal, M. and Villavicencio, A. (eds.), *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pp. 482–492, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/K19-1045. URL https://aclanthology.org/K19-1045.

Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(89)90034-2. URL https://www.sciencedirect.com/science/article/pii/0022000089900342.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Tan, X. E., Xie, X., Jia, X., Wang, X., Goldschlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang, Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Grattafiori, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Vaughan, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Franco, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Wyatt, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Ozgenel, F., Caggioni, F., Guzmán, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Thattai, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Damlaj, I., Molybog, I., Tufanov, I., Veliche, I.-E., Gat,

I., Weissman, J., Geboski, J., Kohli, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Prasad, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Huang, K., Chawla, K., Lakhotia, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Tsimpoukelli, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Laptev, N. P., Dong, N., Zhang, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O., Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P., Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Li, R., Hogan, R., Battey, R., Wang, R., Maheswari, R., Howes, R., Rinott, R., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Kohler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Albiero, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wang, X., Wu, X., Wang, X., Xia, X., Wu, X., Gao, X., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Hao, Y., Qian, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., and Zhao, Z. The llama 3 herd of models, 2024a. URL https://arxiv.org/abs/2407.21783.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024b.

Ecma International. ECMA-404 The JSON Data Interchange Standard. Online, 2013. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.

Evey, R. *The Theory and Applications of Pushdown Store Machines*. Mathematical linguistic and automatic translation: Report to National Science Foundation. Harvard University, 1963. URL https://books.google.com/books?id=mg4yAAAAIAAJ.

Gat, N. GitHub - noamgat/lm-format-enforcer: Enforce the output format (JSON Schema, Regex etc) of a language model — github.com. https://github.com/noamgat/lm-format-enforcer, 2024. [Accessed 31-10-2024].

Gat, N. et al. lm-format-enforcer. https://github.com/noamgat/lm-format-enforcer, 2025. Accessed: 2025-03-27.

Gerganov, G. GitHub - ggerganov/llama.cpp: LLM inference in C/C++ — github.com. https://github.com/ggerganov/llama.cpp, 2023. [Accessed 31-10-2024].

Guidance-ai. GitHub - guidance-ai/guidance: A guidance language for controlling large language models. — github.com. https://github.com/guidance-ai/guidance, 2024. [Accessed 31-10-2024].

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 185–200, 2017.

hiworldwzj, shihaobai, sufubao, WANDY666, FlyingFlame, llehtahw, LiangLiu, wxd000000, fuheaven, XHPlus, Chielo, Yong, Y., and_gate, sangchengmeng, wangzhihong, singularity, Yang, S., SiYu, W., Tracin, Granger, E., Husain, H., R, S. A. G. A., SunXiaoye, Peng, T., Uranus, Bai, Y., Fan, Y., bingo, liuhuakai, and XFPlus. *ModelTC/lightllm*. 10 2024. URL https://github.com/ModelTC/lightllm.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

Koo, T., Liu, F., and He, L. Automata-based constraints for language model decoding, 2024. URL https://arxiv.org/abs/2407.08103.

Kuchnik, M., Smith, V., and Amvrosiadis, G. Validating large language models with relm. *Proceedings of Machine Learning and Systems*, 5:457–476, 2023.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023a.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention, 2023b. URL https://arxiv.org/abs/2309.06180.

LangChain. Tool Calling with LangChain — blog.langchain.dev. https://blog.langchain.dev/tool-calling-with-langchain/, 2024. [Accessed 26-10-2024].

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder: may the source be with you!, 2023. URL https://arxiv.org/abs/2305.06161.

Liu, B., Jiang, Y., Zhang, X., Liu, Q., Zhang, S., Biswas, J., and Stone, P. Llm+p: Empowering large language models with optimal planning proficiency, 2023. URL https://arxiv.org/abs/2304.11477.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Zhang, Z., Wong, R. Y. Y., Zhu, A., Yang, L., Shi, X., Shi, C., Chen, Z., Arfeen, D., Abhyankar, R., and Jia, Z. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pp. 932–949. ACM, April 2024. doi: 10.1145/3620666.3651335. URL http://dx.doi.org/10.1145/3620666.3651335.

MLC team. MLC-LLM, 2023a. URL https://github.com/mlc-ai/mlc-llm.

MLC team. WebLLM, 2023b. URL https://github.com/mlc-ai/web-llm.

Moskal, M., Musuvathi, M., and Kıcıman, E. AI Controller Interface. https://github.com/microsoft/aici/, 2024.

Mozannar, H., Bansal, G., Fourney, A., and Horvitz, E. Reading between the lines: Modeling user behavior and costs in ai-assisted programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 1–16, 2024.

NousResearch. NousResearch/json-mode-eval · Datasets at Hugging Face — huggingface.co. https://huggingface.co/datasets/NousResearch/json-mode-eval, 2024. [Accessed 31-10-2024].

OpenAI. Function Calling - OpenAI API. https://platform.openai.com/docs/guides/function-calling, 2024. [Accessed 26-10-2024].

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., Bello, I., Berdine, J., Bernadett-Shapiro, G., Berner, C., Bogdonoff, L., Boiko, O., Boyd, M., Brakman, A.-L., Brockman, G., Brooks, T., Brundage, M., Button, K., Cai, T., Campbell, R., Cann, A., Carey, B., Carlson, C., Carmichael, R., Chan, B., Chang, C., Chantzis, F., Chen, D., Chen, S., Chen, R., Chen, J., Chen, M., Chess, B., Cho, C., Chu, C., Chung, H. W., Cummings, D., Currier, J., Dai, Y., Decareaux, C., Degry, T., Deutsch, N., Deville, D., Dhar, A., Dohan, D., Dowling, S., Dunning, S., Ecoffet, A., Eleti, A., Eloundou, T., Farhi, D., Fedus, L., Felix, N., Fishman, S. P., Forte, J., Fulford, I., Gao, L., Georges, E., Gibson, C., Goel, V., Gogineni, T., Goh, G., Gontijo-Lopes, R., Gordon, J., Grafstein, M., Gray, S., Greene, R., Gross, J., Gu, S. S., Guo, Y., Hallacy, C., Han, J., Harris, J., He, Y., Heaton, M., Heidecke, J., Hesse, C., Hickey, A., Hickey, W., Hoeschele, P., Houghton, B., Hsu, K., Hu, S., Hu, X., Huizinga, J., Jain, S., Jain, S., Jang, J., Jiang, A., Jiang, R., Jin, H., Jin, D., Jomoto, S., Jonn, B., Jun, H., Kaftan, T., Łukasz Kaiser, Kamali, A., Kanitscheider, I., Keskar, N. S., Khan, T., Kilpatrick, L., Kim, J. W., Kim, C., Kim, Y., Kirchner, J. H., Kiros, J., Knight, M., Kokotajlo, D., Łukasz Kondraciuk, Kondrich, A., Konstantinidis, A., Kosic, K., Krueger, G., Kuo, V., Lampe, M., Lan, I., Lee, T., Leike, J., Leung, J., Levy, D., Li, C. M., Lim, R., Lin, M., Lin, S., Litwin, M., Lopez, T., Lowe, R., Lue, P., Makanju, A., Malfacini, K., Manning, S., Markov, T., Markovski, Y., Martin, B., Mayer, K., Mayne, A., McGrew, B., McKinney, S. M., McLeavey, C., McMillan, P., McNeil, J., Medina, D., Mehta, A., Menick, J., Metz, L., Mishchenko, A., Mishkin, P., Monaco, V., Morikawa, E., Mossing, D., Mu, T., Murati, M., Murk, O.,

Mély, D., Nair, A., Nakano, R., Nayak, R., Neelakantan, A., Ngo, R., Noh, H., Ouyang, L., O'Keefe, C., Pachocki, J., Paino, A., Palermo, J., Pantuliano, A., Parascandolo, G., Parish, J., Parparita, E., Passos, A., Pavlov, M., Peng, A., Perelman, A., de Avila Belbute Peres, F., Petrov, M., de Oliveira Pinto, H. P., Michael, Pokorny, Pokrass, M., Pong, V. H., Powell, T., Power, A., Power, B., Proehl, E., Puri, R., Radford, A., Rae, J., Ramesh, A., Raymond, C., Real, F., Rimbach, K., Ross, C., Rotsted, B., Roussez, H., Ryder, N., Saltarelli, M., Sanders, T., Santurkar, S., Sastry, G., Schmidt, H., Schnurr, D., Schulman, J., Selsam, D., Sheppard, K., Sherbakov, T., Shieh, J., Shoker, S., Shyam, P., Sidor, S., Sigler, E., Simens, M., Sitkin, J., Slama, K., Sohl, I., Sokolowsky, B., Song, Y., Staudacher, N., Such, F. P., Summers, N., Sutskever, I., Tang, J., Tezak, N., Thompson, M. B., Tillet, P., Tootoonchian, A., Tseng, E., Tuggle, P., Turley, N., Tworek, J., Uribe, J. F. C., Vallone, A., Vijayvergiya, A., Voss, C., Wainwright, C., Wang, J. J., Wang, A., Wang, B., Ward, J., Wei, J., Weinmann, C., Welihinda, A., Welinder, P., Weng, J., Weng, L., Wiethoff, M., Willner, D., Winter, C., Wolrich, S., Wong, H., Workman, L., Wu, S., Wu, J., Wu, M., Xiao, K., Xu, T., Yoo, S., Yu, K., Yuan, Q., Zaremba, W., Zellers, R., Zhang, C., Zhang, M., Zhao, S., Zheng, T., Zhuang, J., Zhuk, W., and Zoph, B. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Pearce, H., Tan, B., Ahmad, B., Karri, R., and Dolan-Gavitt, B. Examining zero-shot vulnerability repair with large language models, 2022. URL https://arxiv.org/abs/2112.02125.

Poesia, G., Polozov, O., Le, V., Tiwari, A., Soares, G., Meek, C., and Gulwani, S. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*, 2022.

Python Software Foundation. Python full grammar specification. https://docs.python.org/3/reference/grammar.html, 2024. Accessed: 2025-03-05.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Scheifler, R. W. An analysis of inline substitution for a structured programming language. *Commun. ACM*, 20 (9):647–654, September 1977. ISSN 0001-0782. doi: 10.1145/359810.359830. URL https://doi.org/10.1145/359810.359830.

Scholak, T., Schucher, N., and Bahdanau, D. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In Moens, M.-F., Huang, X., Specia, L., and Yih, S. W.-t. (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9895–9901, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.779. URL https://aclanthology.org/2021.emnlp-main.779.

Schützenberger, M. On context-free languages and push-down automata. *Information and Control*, 6(3):246–264, 1963. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(63)90306-1. URL https://www.sciencedirect.com/science/article/pii/S0019995863903061.

Ugare, S., Suresh, T., Kang, H., Misailovic, S., and Singh, G. Syncode: Llm generation with grammar augmentation, 2024. URL https://arxiv.org/abs/2403.01632.

Wang, B., Wang, Z., Wang, X., Cao, Y., Saurous, R. A., and Kim, Y. Grammar prompting for domain-specific language generation with large language models, 2023. URL https://arxiv.org/abs/2305.19234.

Wang, C., Cho, K., and Gu, J. Neural machine translation with byte-level subwords, 2019. URL https://arxiv.org/abs/1909.03341.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. H. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021. URL https://arxiv.org/abs/2109.00859.

Willard, B. T. and Louf, R. Efficient guided generation for llms. *arXiv preprint arXiv:2307.09702*, 2023.

Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: deliberate problem solving with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2024. Curran Associates Inc.

Yin, L., Sheng, Y., and Zheng, L. Fast json decoding for local llms with compressed finite state machine. https://lmsys.org/blog/2024-02-05-compressed-fsm/, February 2024. Accessed: 2025-03-26.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Zakai, A. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 301–312, 2011.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

A Formal Definition of the PDA Variant

In our paper, we define a variant of the pushdown automaton (PDA) that is equivalent to the original definition, but is designed to facilitate the description of the parsing algorithm and the construction of the token mask cache, since the keys of the token mask cache are precisely the states in this PDA. It is defined as the tuple

$$P = \big(R, \Sigma, \{A_r\}_{r\in R}, q_{\text{main}}, \delta\big),$$

where:

- R is a finite set of grammar rules.
- Σ is a finite input alphabet.
- For each rule $r \in R$, the corresponding finite state automaton is given by

  $$A_r = \big(Q_r, \Sigma \cup R, q_r^{\text{start}}, F_r, \delta_r\big),$$

  where Q_r is a finite set of states, $q_r^{\text{start}} \in Q_r$ is the start state, $F_r \subseteq Q_r$ is the set of accepting states, and δ_r is the transition function defined over Q_r. The transition labels in A_r are drawn from the alphabet $\Sigma \cup R$, which includes both input characters and rule references.
- q_{main} is the start state corresponding to the main rule.
- δ is the global transition function that governs the operation of the PDA by handling two kinds of transitions:
  - *Character transitions*: When in a state $q \in Q_r$, reading an input symbol $a \in \Sigma$ may lead to a transition within the same automaton, i.e., $q \xrightarrow{a} q'$.
  - *Rule reference transitions*: When in a state $q \in Q_r$, a transition labeled by a rule reference to $s \in R$ allows the PDA to push the current return information onto the stack and jump to the start state q_s^{start} of A_s.

The parsing state is represented by a set of pairs $\{(s_i, q_i)\}$, where s_i denotes the content of the stack (encoding return information), q_i is the current state (with $q_i \in Q_r$ for some $r \in R$). There can be multiple such pairs because the pushdown automaton can contain non-deterministic transitions, which means there could be multiple possible parsing stacks and states. In the main body of the paper, for simplicity, we place the current state q_i at the top of the stack. Thus, the parsing state is represented as a set of parsing stacks.

We now describe a formal transformation that converts the above variant PDA definition into a standard PDA. To obtain a standard PDA, we construct a new pushdown automaton

$$P' = \big(Q, \Sigma, \Gamma, \delta', q_0, F\big)$$

as follows. The state set Q is defined as the union of all states from the FSAs:

$$Q = \bigcup_{r\in R} Q_r,$$

and we set the initial state to be $q_0 = q_{\text{main}}$. The stack alphabet Γ is chosen to record return information; we define

$$\Gamma = \{\, (r, q) \mid r \in R,\ q \in Q_r \,\},$$

so that each symbol (r, q) encodes the context of a rule call, namely the originating rule and the return state.

The transition function δ' of the standard PDA is then defined to simulate the behavior of the variant PDA. For each character transition in some δ_r, if

$$q \xrightarrow{a} q' \quad \text{with } a \in \Sigma \text{ and } q, q' \in Q_r,$$

we include in δ' the transition

$$\delta'(q, a, \gamma) \ni (q', \gamma) \quad \text{for all } \gamma \in \Gamma.$$

For each rule reference transition in δ_r, if

$$q \xrightarrow{s} q' \quad \text{with } s \in R,$$

we simulate the recursive call by defining an ϵ-transition that pushes the return information and transfers control to the called rule. Formally, we set

$$\delta'(q, \epsilon, \gamma) \ni \Big(q_s^{\text{start}},\ (r, q') \cdot \gamma\Big) \quad \text{for all } \gamma \in \Gamma,$$

where q belongs to the automaton A_r, and $(r, q') \cdot \gamma$ denotes the stack obtained by pushing (r, q') onto γ. Finally, when

an automaton A_r reaches an accepting state $q \in F_r$, the standard PDA simulates the return from a recursive call by popping the top of the stack. That is, if the current stack has the form $(r', q') \cdot \gamma$, we define

$$\delta'(q, \epsilon, (r', q')) \ni (q', \gamma).$$

The set of accepting states F is defined as those states in $\bigcup_{r \in R} F_r$ that are reached with an empty stack.

This construction shows that character transitions in the variant PDA directly correspond to state transitions without stack operations in P', while rule reference transitions correspond to stack push and jump operations. Similarly, completing the match of an FSA and returning to the parent rule is achieved by a stack pop and a transition to the stored return state. In this way, the standard PDA P' exactly simulates the recursive behavior of our variant PDA.

B Synergy between XGrammar and Jump-forward Decoding

Jump-forward decoding is a technique designed to accelerate structured generation. When, based on the current input, the following output can be deterministically inferred from the grammar, it bypasses LLM decoding and sampling by directly tokenizing and appending the output to the context. This improves the end-to-end efficiency of LLM generation. This technique is orthogonal to the constrained decoding adopted by XGrammar. XGrammar further supports jump-forward decoding and demonstrates that the two can be effectively combined to further improve efficiency.

We evaluate performance using the Llama-3.1-8B-Instruct model served through the SGLang engine, running on a machine with an AMD Ryzen 9 7950X CPU and an NVIDIA RTX 4090 GPU. All experiments are conducted with a batch size of 1. We compare our method, XGrammar, against the existing decoding backend Outlines, testing both with and without jump-forward decoding enabled. We measure time per output token as the evaluation metric.

As shown in Figure 11, XGrammar consistently outperforms Outlines, and achieves the best efficiency overall when combined with jump-forward decoding—demonstrating its ability to better leverage structural constraints for faster generation.

C Cross-platform Deployment of XGrammar

We study bringing XGrammar to a wide variety of platforms. We leverage Emscripten (Zakai, 2011) to compile XGrammar into WebAssembly (Haas et al., 2017) and build a JavaScript binding. This approach enables XGrammar to run in client-side browsers on portable devices like laptops and mobile phones. We further integrate the web-binding with the in-browser LLM inference framework WebLLM (MLC team, 2023b) to enable structured generation.

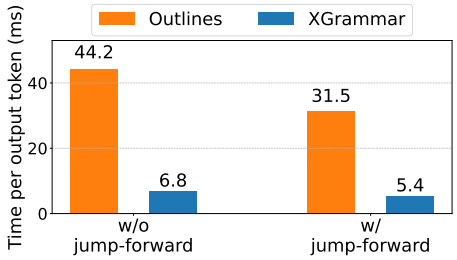

Figure 11. The performance comparison with and without jump-forward decoding on the JSON schema task and the SGLang engine. XGrammar combined with jump-forward can achieve optimal TPOT.

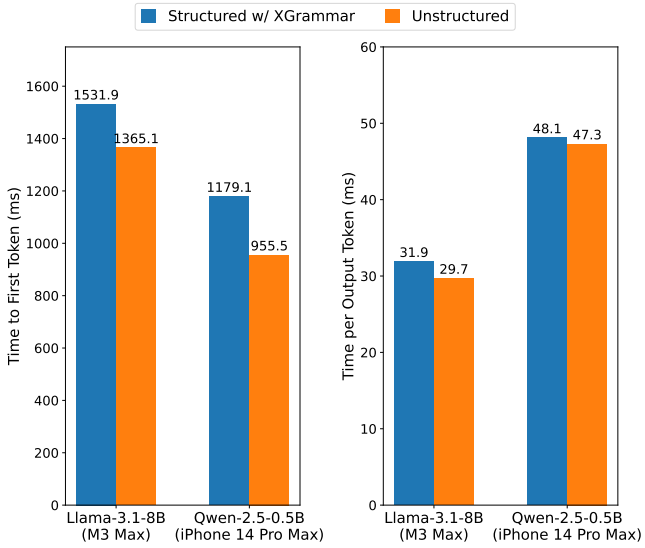

Figure 12. End-to-end performance comparison between structured generation with XGrammar and unstructured generation in browser JavaScript environment.

We evaluate the end-to-end performance with the JSON-mode-eval dataset, using 4-bit quantized models Llama-3.1-8B-Instruct (Dubey et al., 2024b) on a MacBook Pro M3 Max (MacOS 14.5) with Google Chrome, and Qwen-2.5-0.5B-Instruct (Yang et al., 2024) on an iPhone 14 Pro Max (iOS 18) with Safari.

The results are shown in Figure 12. We compare the time to first token (TTFT) and time per output token (TPOT) between structured generation with XGrammar and non-structured generation while ensuring the number of generated tokens is the same. The results show that XGrammar brings close to zero overhead in both settings, enabling a great potential to support future on-device agents with high

performance.