Grammar Pruning: Enabling Low-Latency Zero-Shot Task-Oriented Language Models for Edge AI

Octavian Alexandru Trifan* **Jason Lee Weber** **Marc Titus Trifan**
Alexandru Nicolau **Alexander Veidenbaum**

University of California, Irvine

Abstract

Edge deployment of task-oriented semantic parsers demands high accuracy under tight latency and memory budgets. We present Grammar Pruning, a lightweight zero-shot framework that begins with a user-defined schema of API calls and couples a rule-based entity extractor with an iterative grammar-constrained decoder: extracted items dynamically prune the context-free grammar, limiting generation to only those intents, slots, and values that remain plausible at each step. This aggressive search-space reduction both reduces hallucinations and slashes decoding time. On the adapted FoodOrdering, APIMIXSNIPS, and APIMIXATIS benchmarks, Grammar Pruning with small language models achieves an average execution accuracy of over 90%—rivaling State-of-the-Art, cloud-based solutions—while sustaining at least 2x lower end-to-end latency than existing methods. By requiring nothing beyond the domain's full API schema values yet delivering precise, real-time natural-language understanding, Grammar Pruning positions itself as a practical building block for future edge-AI applications that cannot rely on large models or cloud offloading. [1]

1 Introduction

Recent growth in edge computing and "Internet-of-Things" has led to increased adoption of edge devices such as smartphones, wearables, and even embedded industrial systems. Consequently, there is an increasing demand for complex Natural Language Processing (NLP) on edge devices. One critical task for NLP on the edge is Task-Oriented Semantic Parsing (TOSP), which enables users to invoke device functionality via natural language, like telling a smart watch to set a timer or a phone to initiate a call. State-of-the-Art (SOTA) Large Language Models (LLMs) have been shown to solve this problem quite accurately when latency and computational costs are not a concern. However, in edge devices, high latency and extensive resource requirements are not feasible.

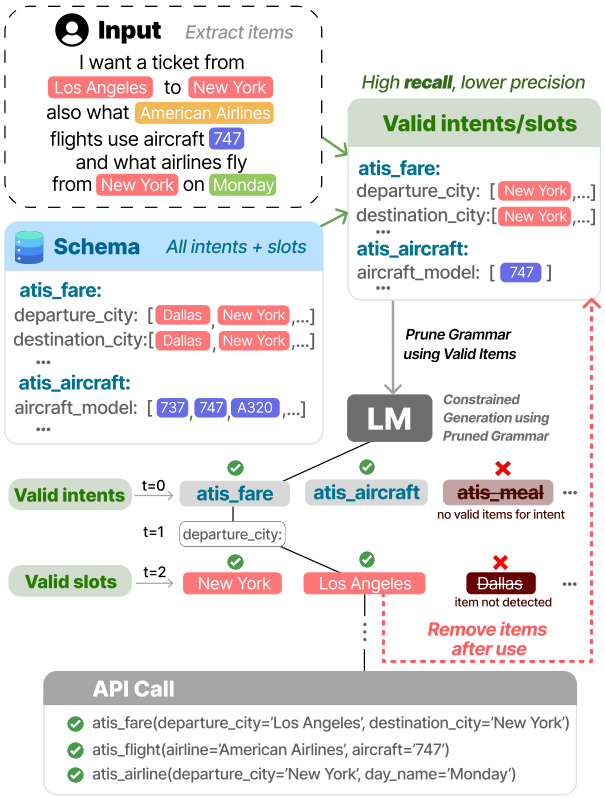

Figure 1: Grammar Pruning: Entities are extracted from the input and mapped to a set of valid items with the help of the schema. The existing grammar for generation is dynamically pruned so that the LM is restricted to only valid intents and slot values. Used items are removed, ensuring schema-compliant and hallucination-free generated API calls.

Existing approaches that leverage large reasoning-oriented models—such as DeepSeek's R1—prioritize accuracy at significant computational expense, with resource requirements far exceeding typical edge device constraints. Edge-based systems cannot tolerate high latency,

*Corresponding author: otrifan@uci.edu

[1] Code and adapted datasets are available at: github.com/octatrifan/grammar-pruning

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 16944–16957
November 4–9, 2025

nor can users accept the consequences of model hallucinations, which risk critical usability errors (e.g., incorrect alarms, erroneous financial interactions). Thus, a novel approach that carefully balances computational cost, latency, and accuracy is necessary.

To address this critical challenge, we introduce Grammar Pruning, a lightweight and schema-driven semantic parsing framework. By dynamically constraining the output space of small language models (SLMs), our approach ensures accurate, efficient, and schema-compliant parsing suitable for deployment in real-time edge environments.

2 Related Work

Task-Oriented Semantic Parsing

Task-oriented semantic parsing involves converting natural language user requests into structured, machine-readable representations to facilitate the execution of specific tasks. Numerous methods have been proposed to increase parsing performance, including hierarchical annotation schemes to improve accuracy and efficiency (Gupta et al., 2018), in-context learning approaches like kNN-ICL to improve compositional generalization (Zhao et al., 2024), and Stack-Transformers within shift-reduce parsing frameworks for diverse resource scenarios (Fernández-González, 2024). Furthermore, leveraging the intrinsic properties of ontology labels has also proven beneficial for semantic parsing in low-resource settings (Desai et al., 2021).

Recent attention has also been drawn towards zero-shot semantic parsing, where models classify new concepts without prior labeled examples by transferring knowledge from similar tasks or by leveraging semantic context. Many successful approaches use Chain-of-Thought (CoT) (Wei et al., 2022) prompting strategies, which decomposes tasks into smaller pieces and reasons about each part individually. Qin et al. (2025), for example, uses Divide-Solve-Combine as a 3-step process of splitting the input into multiple intents and solving them individually before aggregating the results. Mekala et al. (2022), decomposes semantic parsing tasks into abstractive and extractive question-answering problems. Similarly, schema-augmented methods, as employed by Rubino et al. (2022) on their FoodOrdering dataset, closely relate to our work. However, their approach does not explicitly address the computational efficiency and strict schema compliance, which is critical for edge-device applications, motivating our novel approach. MIXATIS and MIXSNIPS (Qin et al., 2020) have been widely used in academic research, lately including zero-shot techniques focused on prompting (Qin et al., 2025; Wu et al., 2021)

Schema-Augmented Methods

Modern systems move beyond flat slot labeling by leveraging explicit schema information to guide parsing and slot-filling (Zhang et al., 2023). Schema-augmented prompting allows models to interpret new APIs by conditioning on domain ontologies and slot descriptions, as shown in recent dialogue state tracking and semantic parsing tasks (Li et al., 2021). Task-oriented semantic parsing often employs a two-step pipeline, leveraging a predefined schema: first, Named Entity Recognition (NER) identifies entities in the input text; second, constrained generation techniques utilize predefined schemas to ensure the output aligns with the expected structure. This approach enhances accuracy and interpretability in applications such as knowledge base question answering and dialogue systems (Nam and Lee, 2023; Shrivastava et al., 2022; Gupta et al., 2022).

Constrained Decoding

Constrained decoding (CD) techniques play a fundamental role in enforcing structured outputs during generation, ensuring that models adhere to specified schemas or formatting rules. Such methods have been successfully applied across diverse applications including semantic role labeling (Deutsch et al., 2019), code validation (Scholak et al., 2021), secure code generation (Fu et al., 2024), and semantic parsing (Shin et al., 2021; Stengel-Eskin et al., 2023; Roy et al., 2023). Context-free grammars (CFGs) are widely used in constrained decoding to systematically restrict autoregressive language model outputs, thus reducing overall hallucinations and accuracy. Toolkits such as Guidance (Lundberg, 2023), LMQL (Beurer-Kellner, 2023), and Outlines (Willard and Louf, 2023) support flexible constraint integration for LLMs.

Input-Dependent Grammars

Recent methods have also explored dynamic grammar adjustments based on the specific input provided, referred to as input-dependent grammars. This concept was formally introduced by Geng et al. (2023), whose work primarily addressed tasks such

as entity disambiguation and constituency parsing. Other researchers have experimented with grammars generated dynamically by LLMs themselves as intermediate steps within CoT prompting strategies (Wang et al., 2023). Explicit prompting for reasoning steps (Nye et al., 2021; Wei et al., 2022; Yao et al., 2023) has proven beneficial, significantly improving output quality and inspiring SOTA models such as OpenAI's o1 (OpenAI, 2024) and Deepseek's R1 (Guo et al., 2025).

Despite these advances, dynamically-generated or reasoning-dependent grammar techniques often incur substantial computational costs, rendering them impractical for resource-constrained edge environments. The computational burden introduced by explicit reasoning or intermediate grammar generation highlights the need for alternative methods capable of maintaining accuracy without excessive resource usage.

Summary of Contribution

Existing literature illustrates significant progress in TOSP, NER, CD, and input-dependent grammar methods, yet often overlooks the stringent computational constraints inherent in edge-device scenarios. Prior work either demands extensive computational resources incompatible with edge hardware or compromises the accuracy due to overly simplistic constraints. To address this crucial gap, we introduce a novel approach based on *Grammar Pruning*, which dynamically contains the output generation space for SLMs. Our method uniquely combines schema-driven grammar constraints, efficient zero-shot parsing capabilities, and lightweight deployment, providing a practical solution specifically tailored for accurate, resource-efficient, and latency-sensitive edge applications.

3 Method

The overall architecture is illustrated in Figure 1, and each of the components will be discussed in-depth in the remainder of this section. The architecture remains constant across different datasets and downstream tasks, with minimal changes required. However, the following sections focus on our implementation and our experiments showing the efficiency of our method on the APIMIXATIS dataset, our own adaptation of the popular MIXATIS, with more details in Section 4.1. The dataset consists of multi-intent task-oriented parsing requests in the airline travel domain.

The modularity of our system allows for flexibility in deployment as well as incremental updates to each module, making it particularly suitable for resource-constrained devices. For edge devices, it is crucial that each module is both resource-efficient and low-latency capable. Below we describe in detail how the key modules work.

3.1 Item Extraction (NER) and Intent Validation

Crucially, for effective and lightweight item extraction, the system relies on the presence of a well-defined schema that details all permissible intents and their associated slots; without such a structured framework, accurately identifying and categorizing relevant information from user input becomes more challenging. For example, the APIMIXATIS schema defines permissible intents and slots (e.g., for flight booking, intents like `atis_fare`, `atis_aircraft`, and slots like `departure_city`, `destination_city`, `aircraft_model`).

We start with the set $C = \{c_1, c_2, \ldots, c_n\}$ that contains all the categories (corresponding to slot types, for example, `departure_city`, `aircraft_model`). For each category $c \in C$ we define $V_c = \{v_{c_1}, v_{c_2}, \ldots, v_{c_m}\}$ which corresponds to the possible values for each category (for example, "New York", "Los Angeles" for `departure_city`; "737", "747" for `aircraft_model`). All these values represent valid strings accepted by the backend engine or for slot filling. Therefore, the full schema S can be defined as encompassing all intent-slot structures and their permissible values.

For the item extraction system, each query to our system will receive the input x, which is a sequence $x = (w_1, w_2, \ldots, w_k)$ with words w_i. Our goal is to extract only the items (entities) that are needed for our downstream task, map them to their respective backend-valid values, and store them in a multiset $M_x \subseteq_{\text{mult}} S$, where S represents all possible item values defined in the schema:

$$M_x = \{\{(v, c) | (v, c) \in S, \rho(v, x_i) > \tau, x_i \subseteq x\}\}$$

where ρ is the function that controls the similarity between the extracted item from the input and a schema item, and τ is a threshold that must be achieved. We note that M_x is a multiset, as we allow duplicates to be included, because our function might extract the same item that is used multiple times in the same input (e.g., if "New York" appears multiple times and is relevant each time).

3.2 Recall vs Precision

Following this initial extraction, the new method emphasizes identifying "Valid intents/slots." This involves taking the extracted items M_x and evaluating them based on the schema S to determine I_x, the set of plausible intents and slot assignments for the given input.

The process for generating I_x from M_x and the full schema S follows 2 steps. First, we identify potential slot fillers: For each unique extracted entity v in M_x, we consult the schema S to find every possible slot c that v can populate. We then form a set of all valid (v, c) pairs. For instance, if "New York" is in M_x, this step would generate both ("New York", `departure_city`) and ("New York", `destination_city`) if the schema permits it. The second step is to validate intents: We iterate through all intents defined in the schema S. An intent is considered valid and added to I_x only if at least one of its required slots can be filled by an item in our set of potential slot fillers. For example, the intent atis_meal is discarded if no food-related items were extracted into M_x. The resulting set I_x is therefore a union of all valid intents and all potential (v, c) pairs.

This identification of I_x is performed with a strong emphasis on maximizing recall. Such prioritization is critical because any element or slot value erroneously omitted from I_x (a false negative) would mean the resulting pruned grammar G_x would be overly restrictive, preventing the LM from generating the correct output.

A consequence of this recall-centric approach is that the set I_x may include ambiguities (i.e., exhibit lower precision). For instance, an extracted entity like "New York" might be present in I_x as a valid candidate for both a `departure_city` slot and a `destination_city` slot if both are plausible according to the schema S and the extracted items M_x. The LM is later tasked with disambiguating these cases, operating with a grammar constrained by this potentially ambiguous I_x.

3.3 Grammar Pruning and Constrained Generation

We arrive at the grammar pruning step with the set of initially extracted items M_x, and more critically, the derived set of "Valid intents/slots" I_x (the nature and formation of which, including its recall-prioritized characteristic, are detailed in Section 3.1). Initially, our grammar is defined as $G = \text{Grammar}(S)$, where S is the full schema. This is the unpruned grammar, allowing the model to output from all possible intents and slots defined in the schema.

The function Grammar will now prune the initial grammar G based on the identified "Valid intents/slots" I_x. So, $G_x = \text{Grammar}(I_x)$. The output is structured, for example, as a series of API calls or structured data objects. By applying the function Grammar on I_x, we are forcing the LM to first select from valid intents, and for each chosen intent, to only allow values for its slots from the permissible items identified in I_x (which were derived from M_x). For example, if `atis_fare` is a valid intent, and `departure_city` is a slot for it, the grammar for `departure_city=` will only allow items $v \in \{v | (v, \text{departure\_city}) \in I_x \text{ for intent atis\_fare}\}$.

Moreover, the grammar pruning step is dynamic and iterative. As the LM generates output and fills slots (e.g., at step t, it selects `departure_city: Los Angeles`), that item can be considered "used." We can use the fact that M_x (and by extension, the available items in I_x) is a multiset to keep pruning the grammar whenever the LM uses an item. If (v_t, c_t) is used by the LM at step t, we can modify the available items and thus the grammar for subsequent steps:

$$M_x^{(t+1)} = M_x^{(t)} \setminus \{(v_t, c_t)\}$$

$$G_x^{(t+1)} = \text{Grammar}(I_x^{(t+1)})$$

where $I_x^{(t+1)}$ is derived from $M_x^{(t+1)}$. Therefore, the LM operates on an increasingly pruned grammar. Unconstrained Generation (vanilla Language Model inference) is defined by:

$$\hat{y} = \arg\max_y \prod_{t=1}^{T} p_\theta(y_t | x, y_{<t})$$

Let $L(G)$ be the language generated by G. Constrained decoding modifies the generation as follows:

$$\hat{y} = \arg \max_{y \in L(G_x^{(T)})} \prod_{t=1}^{T} p_\theta(y_t | x, y_{<t})$$

It follows that $(|I_x| \leq |S|) \rightarrow (|L(G_x)| \leq |L(G)|)$, therefore our model will have fewer items/structures to choose from, speeding up generation and reducing hallucinations. The LM, by operating with the constrained grammar G_x that permits the (potentially ambiguous but recall-rich) options

present in I_x, is responsible for disambiguating these possibilities during the generation process to produce the contextually appropriate structured output.

4 Experimental Setup

4.1 Datasets & Processing

Our experiments focus on zero-shot semantic parsing. We utilize three primary datasets which we release publicly, all adapted to output Python API call structures: FoodOrdering, APIMIXSNIPS, and APIMIXATIS. FoodOrdering is our adaptation of the full [2] dataset from (Rubino et al., 2022). APIMIXSNIPS and APIMIXATIS are evaluation-only sets derived from subsets of the original MixSNIPS and MixATIS benchmarks (Qin et al., 2020), respectively, which focused on only intent/slot extraction, not execution accuracy. We modified these subsets by converting all annotations to our Python API format and making minor alterations to some examples, such as rephrasing prompts for clarity and updating parameters for better real-world alignment. The rest of the input structure remained nearly identical. The fully adapted datasets are available in our code repository. An overview is presented in Table 1.

Dataset	Train	Eval	Intent	Slot	Val
FoodOrdering					
`Pizza`	10,000	348	2	10	166
`Burrito`	9,982	191	7	11	34
`Sub`	10,000	161	3	8	62
`Burger`	0	161	3	9	44
`Coffee`	0	104	1	9	43
APIMixSNIPS	0	103	7	44	259
APIMixATIS	0	103	16	59	215

Table 1: Overview of the adapted datasets. The Burger and Coffee subsets of FoodOrdering, as well as APIMIXSNIPS and APIMIXATIS, are evaluation-only. Intents, Slots, and Values (Val) are schema-dependent.

This transformation to Python API outputs for all datasets is a crucial pre-processing step, leveraging findings from Bogin et al. (2024). Their work demonstrates that parsing to rare Domain-Specific Languages (DSLs) from few examples is challenging. We improve semantic parsing effectiveness by: (1) using general-purpose programming languages like Python for the target representation (see Appendix A.1), and (2) augmenting prompts with a structured domain description (e.g., available Python classes and functions). This shift to Python-based outputs also considerably eases the grammar-building process.

4.2 Item Extraction (NER) Setup

For all three datasets used, the availability of a well-defined schema significantly simplifies the item extraction process.

Specifically, for datasets with very structured schemas like APIMIXSNIPS and APIMIXATIS, entity extraction can often be achieved with straightforward rule-based methods, potentially implemented using tools like spaCy (Honnibal et al., 2020). The clear mapping between schema items and expected user phrases in these datasets facilitates precise extraction.

While the schema in FoodOrdering also aids extraction, its greater linguistic diversity (e.g., variations like `"caramel"`, `"caramel syrup"`, `"caramel drizzle"`) may benefit from more robust techniques. In such cases, we can augment rule-based approaches with heuristics or lightweight fuzzy matching to accurately map varied expressions to schema items.

4.3 Grammar Operations

The Guidance[3] framework from Microsoft is employed to implement the grammar building and pruning process. Guidance provides a flexible programming paradigm that integrates control logic, such as conditionals and loops, with the generation process. We choose Guidance for its low computational overhead, making it particularly suitable for resource-constrained environments. An example of a grammar is provided in Appendix A.2.1. Guidance is optimized to batch non-generated text, meaning that when the grammar allows for just one possible next token, Guidance will simply append the token. This speedup is noticeable when there is a set structure, with a lot of boilerplate parts. We would like to note that our method is agnostic with respect to the framework used; however, the best results can be seen with a system that can handle the grammar pruning process very quickly.

4.4 Models

Given the latency-aware nature of our work, we focus on small, open-source models that are pre-trained for code generation. Specifically, we

[2] https://github.com/amazon-science/food-ordering-semantic-parsing-dataset

[3] https://github.com/guidance-ai/guidance

choose to experiment with both the 0.5B and 1.5B parameter models of the instruct-tuned family of Qwen2.5 Coder models (Hui et al., 2024) and the 0.6B, 1.7B, and 4B parameter models of the family of Qwen3 models (Yang et al., 2025). In addition, we test these models with full Floating-Point 16 (F16) weights and quantized 4-bit (Q4_K_M) model weights. As a comparison, Cross-TOP (Rubino et al., 2022) runs on a BART-Large (Lewis et al., 2019) model.

Using multiple models serves two key reasons. First, it demonstrates the robustness and generalizability of Grammar Pruning across different model architectures and scales. Second, it allows us to analyze how the performance of various methods, including our own, changes with model size, providing insight into the interplay between model capacity and explicit generation constraints.

The open-source nature of these models is crucial for two reasons: (1) we need to be able to deploy them directly on edge devices (in our case, using llama.cpp (Gerganov, 2023) and (2) we need logit access to properly run constrained decoding.

To test cross-schema knowledge transfer, the `PIZZA`, `BURRITO`, and `SUB` datasets are used for instruction fine-tuning (Wei et al., 2021), leaving the `BURGER` and `COFFEE` datasets for zero-shot experiments. For fine-tuning, we use the Unsloth (Daniel Han and team, 2023) framework, training for a full epoch. For the APIMIXSNIPS and APIMIXATIS datasets, out-of-the-box models are used. The fine-tuning is performed on a single GPU for less than 10 GPU hours total between each model and quantization.

4.5 Hardware

The FoodOrdering experiments were performed on an NVIDIA 4090 GPU, while the APIMIXSNIPS and APIMIXATIS experiments (including latency) were run on an NVIDIA 3090 GPU. Both cards are equipped with 24GB of VRAM.

4.6 Evaluation Metrics

Our evaluation framework employs Unordered Exact Matching (EM) with refined adjustments to ensure accuracy and flexibility. Since we utilize named parameters, their order does not impact equivalence, allowing for a more adaptable comparison. Likewise, within lists, element order is disregarded as long as their content remains consistent. While accuracy is important, it is also vital that our system performs efficiently on edge devices. Therefore, we also measure the average total latency across the datasets.

4.7 Baselines and Ablations

In order to demonstrate the effectiveness of Grammar Pruning, we are performing multiple ablations and comparing them against other methods.

In terms of ablations, we test 2 main changes: (1) we test using prompts both with and without appending the NER results, and (2) the grammar can be skipped entirely, or the full non-pruned grammar can be used. These are tested on the FoodOrdering dataset using the Qwen2.5 model.

For the cross-schema evaluation, the baseline is Cross-TOP (Rubino et al., 2022), which uses an augmented schema but no dynamic constrained decoding, comparing to their reported results.

We use the Qwen3 (Yang et al., 2025) model for the adapted APIMIXATIS and APIMIXSNIPS datasets, in the 2 CoT baselines, specifically Qwen3 in Thinking Mode and Divide-Solve-Combine Prompting (DSCP) (Qin et al., 2025). DSCP breaks down zero-shot multi-intent detection into three steps: dividing an utterance into single-intent parts, solving for each intent separately, and then combining the results. This method has shown impressive results for the original MIXATIS and MIXSNIPS datasets, however, while employing larger models, and focusing on intent-slot resolution, not execution accuracy.

We are also reporting results from GPT-4.1 mini, a non-reasoning proprietary model, while using their Function Calling setup, which allows us to pass our schema. Comparing to this model serves two purposes: it benchmarks our open-source models against a powerful, closed-source competitor, and it highlights the effectiveness of Grammar Pruning. By showing that our method with a small, local model can match or exceed GPT-4.1-mini's accuracy, we underscore the value of a targeted, algorithmic approach over relying solely on a larger, general-purpose model.

5 Results and Analysis

5.1 Item Extraction (NER) Accuracy

Our evaluation focuses on three key metrics: precision, recall, and the balance of these, F1 score. As shown in Table 2, the system achieves high precision and recall, with minor variations between datasets. Both food ordering datasets exhibit similar precision and recall of approximately 0.96

Dataset	Precision	Recall	F1
FoodOrdering			
Coffee	0.96	0.97	0.96
Burger	0.96	0.95	0.96
APIMixSNIPS	0.93	1.00	0.97
APIMixATIS	0.69	1.00	0.81

Table 2: Classification metrics of Rule-Based NER across FoodOrdering and API datasets.

within a very small margin of error (± 0.01).

However, the precision and recall of the APIMIXSNIPS and APIMIXATIS datasets present a different pattern. Both datasets exhibit perfect recall; that is, we extract all of the true positives from both datasets. As mentioned in Section 4.2, this is because there is a very clear mapping in these datasets between schema items and expected user phrases. While precision is still greater than 0.9 for APIMIXSNIPS, it is much lower in APIMIXATIS. Since we are heavily prioritizing recall in our item extraction, ambiguity can be introduced into our resulting extraction in instances where an extracted item may be usable with multiple different slots (see Section 3.2).

5.2 Accuracy Results

5.2.1 FoodOrdering

Table 3 shows that grammar pruning delivers substantial zero-shot performance gains over the CrossTOP baseline (Rubino et al., 2022). Accuracy rises from 73.3% to 96.2% on BURGER and from 54.8% to 91.1% on COFFEE, yielding improvements of 23 and 41 absolute points, respectively.

Augmenting the prompt with high-recall NER entities, but without any grammar, makes it hard for the small model to respect the schema, resulting in hallucinated slots/values, which sometimes are artifacts of fine-tuning. Adding full grammar–constrained decoding (CD) narrows the search space, yet the irrelevant menu items remain reachable, so performance barely exceeds the NER-only setting. By dynamically pruning the grammar to the utterance-specific subset, hallucinations are eliminated and near-perfect exact matches are obtained.

Grammar pruning benefits every tested backbone, from the 0.5B model up to 1.5B, and remains highly stable under 4-bit quantization: Q4 models trail their 16-bit counterparts by at most two percentage points.

To contextualize our findings, we also conducted preliminary experiments with larger models in the 8B-parameter class. As expected, we observed that baseline performance improved significantly, confirming the established trend that model scale enhances reasoning capabilities. Critically, however, this improved accuracy still fell considerably short of that achieved by our much smaller models enhanced with Grammar Pruning.

Since our investigation centers on models that are practical for edge deployment, this outcome is particularly salient. It highlights that for this class of structured generation tasks, our efficient pruning method can be a more direct and effective means of improving accuracy than relying on an increase in model parameters alone.

5.2.2 APIMixATIS and APIMixSNIPS

In Table 4, the grammar-pruning pipeline again dominates, attaining 96.1% EM on APIMIXSNIPS and 92.2% on APIMIXATIS with the 4 B backbone-matching GPT-4.1-mini + Function-Calling while using roughly one-tenth the parameters. Although the GPT model always produced syntactically well-formed calls, it occasionally supplied slot values that were outside the valid schema, leading to EM errors (a behavior we also observed in some few-shot experiments, see Appendix A.5).

DSCP decomposes an utterance into smaller subqueries, solves them, and recombines the partial outputs. In practice, the divide step often omits or invents entities, guaranteeing that the subsequent combine step assembles an invalid call. These malformed outputs are heavily penalized by the exact-match metric, limiting DSCP to 34.0% on APIMIXSNIPS and 45.6% on APIMIXATIS.

Qwen3 (CoT) often behaves like DSCP with extra self-verification steps, which explains the sizeable improvement over vanilla DSCP. Nevertheless, any spurious entity that appears in the reasoning phase leaks into the structured output, and it often fails to follow the correct syntax. As a result, Qwen3 (CoT) peaks at 51.5% EM on APIMIXSNIPS and 74.8% on APIMIXATIS.

Across all baselines, accuracy improves with the parameter count: both CoT variants gain over 20 percentage points between 0.6B and 4B parameters, consistent with prior findings that chain-of-thought prompting begins to pay off only for sufficiently large models. Grammar pruning, however, already yields high accuracy at 0.6B and retains it under 4-bit quantization, where the drop is never more

Approach	Burger				Coffee			
	Qwen2.5 0.5B		Qwen2.5 1.5B		Qwen2.5 0.5B		Qwen2.5 1.5B	
	Q4	F16	Q4	F16	Q4	F16	Q4	F16
Standard Prompting + No Grammar	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Standard Prompting + Full Grammar	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
NER-augmented Prompt + No Grammar	0.0	1.2	25.2	20.6	1.0	0.0	0.0	1.0
NER-augmented Prompt + Full Grammar	0.0	1.2	25.2	20.6	3.7	6.2	9.7	10.1
Cross-TOP (Rubino et al., 2022)	73.3				54.8			
Grammar Pruning	**87.0**	**90.4**	**94.6**	**96.2**	**91.1**	**91.1**	**91.1**	**91.1**

Table 3: Accuracy (%) on the COFFEE and BURGER datasets across different ablations. All models, including baselines, are fine-tuned on the other three FoodOrdering subsets (Pizza, Burrito, Sub) to test for cross-schema knowledge transfer.

Approach	APIMixSNIPS						APIMixATIS					
	0.6B		1.7B		4B		0.6B		1.7B		4B	
	Q4	F16	Q4	F16	Q4	F16	Q4	F16	Q4	F16	Q4	F16
Qwen3 Thinking	5.82	6.79	18.44	19.41	51.45	51.45	6.80	16.50	30.10	33.98	74.76	74.76
DSCP (Qin et al., 2025)	0.97	0.97	6.79	6.79	33.98	33.98	0.97	1.94	16.50	16.50	45.63	45.63
Constrained Decoding	2.91	2.91	3.88	3.88	36.89	36.89	4.85	6.80	11.65	13.59	34.95	36.89
GPT-4.1 mini + FC	91.26						92.23					
Grammar Pruning	**91.26**	**91.26**	**94.17**	**94.17**	**96.12**	**96.12**	**67.96**	**69.90**	**83.50**	**83.50**	**92.23**	**92.23**

Table 4: Accuracy (%) of various models and baselines. GPT-4.1 mini is presented directly alongside other model sizes for direct comparison. FC denotes "Function Calling". The model used is Qwen3 for all other baselines. None of the models are fine-tuned, as these datasets are designed for zero-shot evaluation."

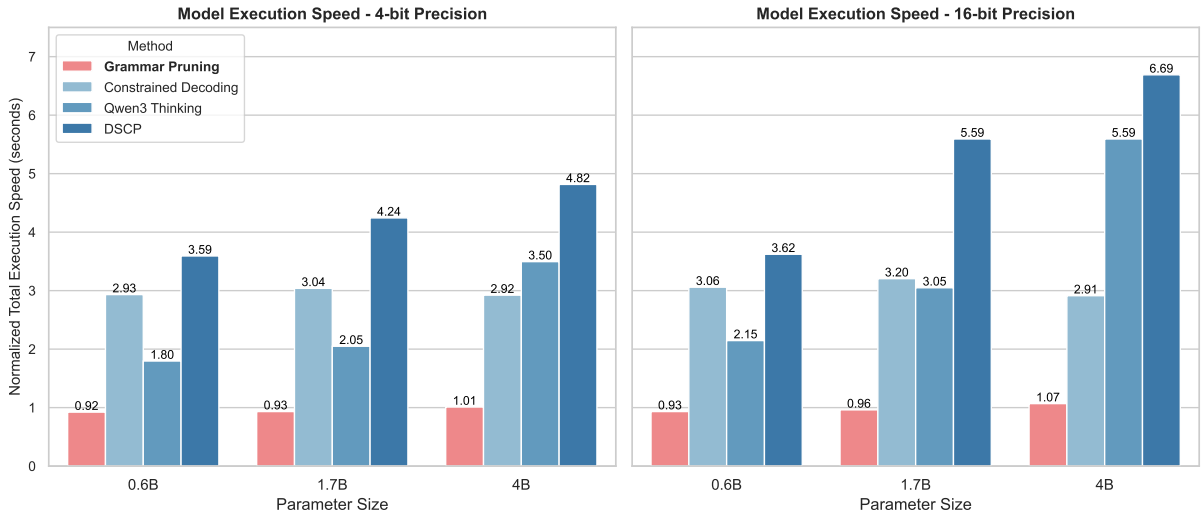

Figure 2: Average response time for a single prompt in seconds (single NVIDIA 3090 GPU). Grammar pruning outperforms all other techniques at each level of quantization and for each model size.

than two percentage points - an attractive property for edge deployment.

5.3 Latency Results

For task-oriented parsing on edge devices, a seamless user experience requires near real-time interaction. While the exact threshold varies by application, latencies over a few seconds can feel slow. The goal is often sub-second response times, which are challenging for generative models on resource-constrained hardware.

Figure 2 presents the mean end-to-end response times for the four methods tested in Table 4—Grammar Pruning, Constrained Decoding, Qwen3 Thinking, and DSCP. We test the 0.6B, 1.7B, and 4B model sizes with both 4-bit quantization and regular floating point 16 weights. Across every model size and quantization level, we see that Grammar

Pruning achieves sub-one-second latency while also performing over twice as fast as the next best method. Grammar Pruning also remains remarkably consistent, suggesting that the method is well-optimized for small models.

Constrained Decoding falls consistently between 2.9 and 3.2, roughly 3 times slower than Grammar Pruning and occasionally faster than Qwen3 in Thinking Mode. While Grammar Pruning has a runtime that will scale with the number of slot value entities extracted from the prompt since it will run out of generation options as entities are removed, there is no stopping mechanism in pure Constrained Decoding. In the case of small models with Constrained Decoding, they seem unable to stop themselves and will continue generating until they hit a manually-set or architecture-encoded context limit.

While Qwen3 in thinking mode performs better on the accuracy front than the other ablation methods, it does so at the cost of increased latency. In Figure 2, we see that as the precision and number of parameters increase, the latency of Qwen3 in Thinking Mode also increases. The downside to "thinking" before speaking is that more tokens are used in generation.

Lastly, DSCP underperforms all other methods over all precision levels and model sizes. In these ablations, we see that it is anywhere from 3 to almost 7 times worse than Grammar Pruning. However, the key difference is that Qwen3 has been explicitly trained for general reasoning across all parameter levels. Because of this, the model has been trained to stop roughly after a certain amount of thinking has been done. However, DSCP is purely a prompting technique that can be used with any model and has not been fine-tuned to stop after a certain amount of thinking. Because of this, Qwen3 in Thinking mode tends to stop sooner than DSCP, even though it has more "links" in its Chain-of-Thought.

6 Conclusion

In this work, we have presented **Grammar Pruning**, a lightweight, schema-driven framework for zero-shot Task-Oriented Semantic Parsing on edge devices. By pairing rule-based, schema-guided slot extraction with dynamic grammar constraints, our approach rigorously limits the language model's response space to only valid intents, slots, and slot value entities, minimizing hallucinations. Through evaluation of zero-shot benchmarks both with knowledge transfer—FOODORDERING—and without—APIMIXSNIPS and APIMIXATIS—Grammar Pruning significantly outperforms prior constrained-decoding baselines in terms of both accuracy and latency.

Limitations

Despite the empirical gains in accuracy and latency, we acknowledge several practical constraints that warrant further investigation and represent exciting avenues for future work.

First, the current NER module is essentially a schema-defined collection of rules. While this design minimizes computational overhead, it leaves the system vulnerable to novel lexical variants and domain drift. For example, a slang term like a cup of "Joe" for coffee would currently produce a false negative that propagates into the grammar, causing downstream failures. Future work must explore hybrid or self-refining extraction methods that maintain low latency while gracefully responding to out-of-schema inputs.

Second, Grammar Pruning presupposes a complete and static schema. When an application allows for user-defined extensions (e.g., custom menu items), the schema must be recreated. While our method transfers across schemas with zero fine-tuning, each new domain requires this explicit inventory. To substantiate the broader applicability of our approach, future work should evaluate Grammar Pruning on more diverse datasets and extend it to tasks beyond TOSP, such as structured web data extraction or specialized code generation, which could also benefit from dynamic, schema-driven constraints. Learning to automatically relax or expand the grammar for valid, out-of-schema user inputs remains an open challenge.

Finally, while latency was low on a consumer GPU, preliminary tests on a Raspberry Pi 5 (Appendix A.7) show that hardware plays a considerable role. Future work must involve device-specific optimizations to further reduce latency on highly constrained edge hardware.

In sum, Grammar Pruning is a promising bridge between reliability and efficiency on the edge, but its practicality outside of controlled testbeds hinges on adaptive extraction, schematic evolution, and additional optimization for edge devices.

References

Lukas et al. Beurer-Kellner. 2023. Lmql: Programming language for constraint-based language model querying. In *NeurIPS 2023*.

Ben Bogin, Shivanshu Gupta, Peter Clark, and Ashish Sabharwal. 2024. Leveraging code to improve in-context learning for semantic parsing. *Preprint*, arXiv:2311.09519.

Michael Han Daniel Han and Unsloth team. 2023. Unsloth.

Shrey Desai, Akshat Shrivastava, Alexander Zotov, and Ahmed Aly. 2021. Low-resource task-oriented semantic parsing via intrinsic modeling. *arXiv preprint arXiv:2104.07224*.

Daniel Deutsch, Shyam Upadhyay, and Dan Roth. 2019. A general-purpose algorithm for constrained sequential inference. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pages 482–492.

Daniel Fernández-González. 2024. Shift-reduce task-oriented semantic parsing with stack-transformers. *Cognitive Computation*, 16(6):2846–2862.

Yanjun Fu, Ethan Baker, Yu Ding, and Yizheng Chen. 2024. Constrained decoding for secure code generation. *arXiv preprint arXiv:2405.00218*.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured nlp tasks without finetuning. *arXiv preprint arXiv:2305.13971*.

G. Gerganov. 2023. ggerganov/llama.cpp: Llm inference in c/c++.

In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. *Preprint*, arXiv:2311.04934.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Sonal Gupta, Rushin Shah, Mrinal Mohit, Anuj Kumar, and Mike Lewis. 2018. Semantic parsing for task oriented dialog using hierarchical representations. *arXiv preprint arXiv:1810.07942*.

Vivek Gupta, Akshat Shrivastava, Adithya Sagar, and Denis Savenkov. 2022. Retronlu: Retrieval augmented task-oriented semantic parsing. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*.

Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength Natural Language Processing in Python.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2019. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *CoRR*, abs/1910.13461.

Shuyang Li, Jin Cao, Mukund Sridhar, Henghui Zhu, Shang-Wen Li, Wael Hamza, and Julian McAuley. 2021. Zero-shot generalization in dialog state tracking through generative question answering. *arXiv preprint arXiv:2101.08333*.

Scott Lundberg. 2023. Guidance: A guidance language for controlling large language models. Https://github.com/microsoft/guidance.

Dheeraj Mekala, Jason Wolfe, and Subhro Roy. 2022. Zerotop: Zero-shot task-oriented semantic parsing using large language models. *arXiv preprint arXiv:2212.10815*.

Daehwan Nam and Gary Lee. 2023. Semantic parsing with candidate expressions for knowledge base question answering.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*.

OpenAI. 2024. Openai o1 system card.

Libo Qin, Qiguang Chen, Jingxuan Zhou, Jin Wang, Hao Fei, Wanxiang Che, and Min Li. 2025. Divide-solve-combine: An interpretable and accurate prompting framework for zero-shot multi-intent detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25038–25046.

Libo Qin, Xiao Xu, Wanxiang Che, and Ting Liu. 2020. AGIF: An adaptive graph-interactive framework for joint multiple intent detection and slot filling. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1807–1816, Online. Association for Computational Linguistics.

Subhro Roy, Samuel Thomson, Tongfei Chen, Richard Shin, Adam Pauls, Jason Eisner, and Benjamin Van Durme. 2023. Benchclamp: A benchmark for evaluating language models on syntactic and semantic parsing. *Advances in Neural Information Processing Systems*, 36:49814–49829.

Melanie Rubino, Nicolas Guenon des Mesnards, Uday Shah, Nanjiang Jiang, Weiqi Sun, and Konstantine Arkoudas. 2022. Cross-TOP: Zero-shot cross-schema task-oriented parsing. In *Proceedings of the Third Workshop on Deep Learning for Low-Resource Natural Language Processing*, pages 48–60, Hybrid. Association for Computational Linguistics.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*.

Richard Shin, Christopher H Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. 2021. Constrained language models yield few-shot semantic parsers. *arXiv preprint arXiv:2104.08768*.

Akshat Shrivastava, Shrey Desai, Anchit Gupta, Ali Elkahky, Aleksandr Livshits, Alexander Zotov, and Ahmed Aly. 2022. Retrieve-and-fill for scenario-based task-oriented semantic parsing. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*.

Elias Stengel-Eskin, Kyle Rawlins, and Benjamin Van Durme. 2023. Zero and few-shot semantic parsing with ambiguous inputs. *arXiv preprint arXiv:2306.00824*.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. 2023. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, 36:65030–65055.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for llms. *arXiv preprint arXiv:2307.09702*.

Ting-Wei Wu, Ruolin Su, and Biing-Hwang Juang. 2021. A label-aware bert attention network for zero-shot multi-intent detection in spoken language understanding. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5090–5100, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. *arXiv preprint*. ArXiv:2505.09388 [cs].

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Xiaoying Zhang, Baolin Peng, Kun Li, Jingyan Zhou, and Helen Meng. 2023. SGP-TOD: Building task bots effortlessly via schema-guided LLM prompting. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13348–13369, Singapore. Association for Computational Linguistics.

Wenting Zhao, Ye Liu, Yao Wan, Yibo Wang, Qingyang Wu, Zhongfen Deng, Jiangshu Du, Shuaiqi Liu, Yunlong Xu, and Philip Yu. 2024. kNN-ICL: Compositional task-oriented parsing generalization with nearest neighbor in-context learning. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 326–337, Mexico City, Mexico. Association for Computational Linguistics.

A Appendix

A.1 Data Preprocessing

Below is an example of how all 5 datasets of the FoodOrdering dataset are changed to a Python structure instead of the original cluttered DSL. For example:

`"one regular latte light roast with an extra espresso shot and honey added and one large cappuccino with caramel syrup"`

is initially represented as:

```
(DRINK_ORDER (NUMBER 1) (SIZE regular) (
    DRINK_TYPE latte) (ROAST_TYPE
    light_roast) (TOPPING
    ESPRESSO_SHOT_1) (TOPPING honey)) (
    DRINK_ORDER (NUMBER 1) (SIZE large)
    (DRINK_TYPE cappuccino) (TOPPING
    caramel_syrup))
```

and transformed into:

```
[DrinkOrder(number=1, drink_type='latte'
    , roast_type='light_roast', size='
    regular', toppings=[Topping(name='
    ESPRESSO_SHOT_1'), Topping(name='
    honey')]), DrinkOrder(number=1,
    drink_type='cappuccino', size='large
    ', toppings=[Topping(name='
    caramel_syrup')])]
```

This ensures a compact yet flexible representation, suitable for downstream processing, which leverages the knowledge of Python from the pretraining phase.

A.2 Guidance Framework Code for Dynamic Constrained Decoding

This appendix provides the implementation of a grammar using the Guidance framework. The functions handle the generation of structured `DrinkOrder` outputs with dynamic constraints, represented by the list of possible values.

A.2.1 Drink Order Grammar

```
guidance(stateless=False)
def drinkOrderCoffee(lm):
    lm += "DrinkOrder("
    lm += select(["number="+regex("\d+")
        , ""], name='numberFlag')
    if drink_type_values:
      lm += select([", drink_type='" +
          select(drink_type_values, name
          ='drinkTypeName')+"'", ""],
          name='drinkTypeFlag')
      if lm['drinkTypeFlag'] != "":
        drink_type_values.remove(lm['
            drinkTypeName'])

    if roast_type_values:
      lm += select([", roast_type='"+
          select(roast_type_values, name
          ='roastTypeName')+"'", ""],
          name='drinkTypeFlag')
      if lm['drinkTypeFlag'] != "":
        roast_type_values.remove(lm['
            roastTypeName'])

    if size_values:
      lm += select([", size='"+select(
          size_values, name='sizeName')+
          "'", ""], name='sizeFlag')
      if lm['sizeFlag'] != "":
        size_values.remove(lm['sizeName'
            ])

    if style_values:
      lm += select([", style='"+select(
          style_values, name='styleName'
          )+"'", ""], name='styleFlag')
      if lm['styleFlag'] != "":
        style_values.remove(lm['
            styleName'])

    if topping_values:
      lm += select([", toppings=[", ""],
          name='toppingsFlag')
      if lm['toppingsFlag']:
        for i in topping_values[:]:
          lm += toppingCoffee()
          if not topping_values:
            lm += ']'
            break
          lm += select([", ", "]"], name
              ="finishedListToppings")
          if lm['finishedListToppings']
              == "]":
            break

    return lm + ")"
```

Listing 1: Function to generate a DrinkOrder object.

Description: This function generates a `DrinkOrder` grammar with attributes such as `number`, `drink_type`, `roast_type`, `size`, `style` and an optional list of `toppings`. Each attribute is constrained by dynamically updated grammar rules.

A.2.2 Topping Functions

```
@guidance(stateless=False)
def toppingCoffee(lm):
  lm += "Topping(name="
  if topping_values:
    lm += "'" + select(topping_values,
        name='toppingName') + "'"
    topping_values.remove(lm['
        toppingName'])

  if quantity_values:
    lm += select([", qualifier='" +
        select(quantity_values, name='
        qualifierName') + "'", ""], name
        ='qualifierFlag')
    if lm["qualifierFlag"] != "":
      quantity_values.remove(lm['
          qualifierName'])

  if not_values:
    lm += select([", negation=True", ""
        ], name='negationFlag')
    if lm['negationFlag'] != "":
      not_values.remove('not')

  lm += ")"
  return lm
```

Listing 2: Function to generate a Topping object.

Description: This grammar generates a `Topping` object with attributes such as `name`, `qualifier`, and `negation`. The values for each attribute are dynamically constrained by user input.

A.3 Valid Order Function

```
@guidance(stateless=False)
def validOrderCoffee(lm):
  lm += "["
  first = True
  for i in range(7):
```

```
    if drink_type_values:
      if not first:
        lm += select(", ", "")
      else:
        first = False

      lm += drinkOrderCoffee()
    else:
      break
  return lm +']'
```

Listing 3: Function to generate a valid order.

Description: This grammar generates a list of `DrinkOrder` objects, iterating through multiple orders.

A.4 Qualitative Error Analysis for FoodOrdering Baselines

The low accuracy scores for the ablation baselines in Table 3 can be attributed to two key factors: the strictness of our evaluation metric and a common failure mode in small language models. Our evaluation uses Unordered Exact Matching (EM), which penalizes any deviation from the ground-truth API call, assigning a score of 0 for outputs with even minor errors.

The primary failure mode observed was the *hallucination of parameters*. Even when guided by a full, unpruned grammar, the models often inferred and incorrectly assigned attributes that were not specified in the user's prompt. Because the full grammar makes all schema items from the training domains (Pizza, Burrito, Sub) technically reachable, the model would make contextually inappropriate assignments. For example, models would assign a `roast_style` to a `hot_chocolate` order or specify a coffee's `style` as `decaf` when no such information was present in the request.

For the baselines without NER augmentation, the small models struggled to perform both entity extraction and structured generation simultaneously. They often failed to detect all the correct items from the prompt before attempting to generate the output, leading to incomplete or malformed API calls.

These errors highlight the limitations of using small models with broad, unpruned constraints. While the full grammar prevents syntax errors, it does not prevent the model from navigating the large semantic space of all possible schema values incorrectly. In contrast, our **Grammar Pruning** method entirely eliminates these hallucinations by dynamically restricting the generation space to only those intents, slots, and values explicitly extracted from the user's prompt, ensuring near-perfect exact matches. While larger models (e.g., 8B+) show improved performance on these baselines, our approach enables smaller, edge-friendly models to achieve state-of-the-art accuracy.

A.5 Few-Shot Learning Sanity Check

While our primary focus is the strict zero-shot scenario, we performed a brief sanity check with GPT-4.1-mini using 3-shot prompting. We observed that the model still occasionally produced out-of-schema slot values, suggesting that a small number of examples may be insufficient to fully constrain the model against a large and complex schema. This confirmed our focus on a zero-shot approach where schema compliance is guaranteed by the generation method itself.

A.6 Prompt Caching

One of our optimization strategies is prompt caching (Gim et al., 2024), which significantly enhances efficiency in inference. Our prompts adhere to a structured instruction-based format, incorporating both input data and the extracted elements only at the end of each prompt. As a result, multiple queries exhibit a shared long prefix (as we augmented prompts with a structured domain description that includes the available classes and function), allowing us to cache this common segment rather than fully regenerating it for each request. Because this prefix accounts for $\approx 70\%$ of the total prompt length, this approach dramatically reduces cold-start overhead, optimizing response times and substantially accelerating Time-To-First-Token (TTFT), particularly for extensive prompts. By leveraging this caching mechanism, we achieve a considerable boost in computational efficiency and overall system responsiveness.

A.7 Raspberry Pi Latency Test Results

A.7.1 Time-to-first-token

Device	Qwen2.5 0.5B		Qwen2.5 1.5B	
	Q4	F16	Q4	F16
Raspberry Pi	6.46s	9.83s	17.69s	35.98s
Raspberry Pi + Cache	2.10s	2.70s	4.77s	10.07s
GPU	0.28s	0.30s	0.27s	0.27s
GPU + Cache	0.18s	0.18s	0.18s	0.19s

Table 5: Time-to-first-token (averaged) on the 2 local devices. "Cache" represents using Prompt Caching.

TTFT measures the delay from input submission to the first generated token, and is reported

in Table 5. On the Raspberry Pi, TTFT is greatly reduced by the quantization in both model sizes. However, it didn't have as big as an impact on the GPU, as times remained constant across quantization. Prompt caching proves to have a great impact on both the Pi and the GPU. With prompt caching, TTFT is sped up by up to 3.6x on the Pi and 1.6x on GPU.

A.7.2 Total Latency

For the Raspberry Pi, quantization plays a crucial role in reducing latency. The Qwen2.5 0.5B (Q4) model runs in 3.79s with CD, compared to 4.72s in F16, showing a clear performance advantage. The effect is even more pronounced for the 1.5B model, where Q4 achieves 7.29s with CD, while F16 takes 13.35s. On GPU, quantization has a minimal effect on total latency, as both Q4 and F16 models perform consistently, with differences of only 0.02-0.05s.

Using the grammar improves the speed for the larger 1.5B model on Raspberry Pi, as well as on the smaller model F16 quant. This suggests that constraining the model's output space reduces unnecessary computations, making inference more efficient. However, for the smaller 0.5B model, CD introduces a minor overhead, increasing latency slightly from 3.41s to 3.79s. Additionally, CD introduces a small latency overhead on GPU, increasing inference time by about 0.4s. While the difference is marginal, it indicates that grammar pruning slightly slows inference on fast executions, but with very little overhead.

Despite the higher latency compared to GPUs, the Raspberry Pi's performance (especially on the smaller quantized model) remains within a practical range for real-time applications.

Device	**Qwen2.5 0.5B**		**Qwen2.5 1.5B**	
	Q4	**F16**	**Q4**	**F16**
Raspberry Pi + CD	3.79s	4.72s	7.29s	13.35s
Raspberry Pi w/o CD	3.41s	5.55s	7.88s	15.73s
GPU + CD	0.73s	0.75s	0.77s	0.78s
GPU w/o CD	0.30s	0.32s	0.33s	0.38s

Table 6: Total average latency (time from input to output) on the 2 local devices. "CD" signifies Constrained Decoding (grammar pruning). "w/o CD" represents the unconstrained model (no grammar). Prompt Caching is used for all experiments.