PROMTEC: Fast LLM Inference Decoding using Prompt Multi-Lookup with Template Database and Common Sequences

Alan Chi-Man Lee[*1], Wing-Sun Cheng[2], Calvin Chun-Kit Chan[1]
[1]Department of Information Engineering, The Chinese University of Hong Kong, [2]RISKSIS
{lcm123, ckchan}@ie.cuhk.edu.hk marco.cheng@risksis.com

Abstract

We propose PROMTEC, a novel multi-faceted approach to accelerate the inference of large language models (LLMs) by leveraging three key techniques: Prompt Multi-Lookup, Template Datastore, and Common Sequences methods. Prompt Multi-Lookup enhances the autoregressive decoding efficiency by generating multiple candidate sequences from context. Template Datastore exploits structured patterns, particularly in mathematical and code generation tasks, to enable fast and accurate candidate generation. Common Sequences optimize inference by precomputing frequent short sequences in specialized domains. For mathematical generation, PROMTEC achieves a 3.91× speedup on the miniF2F benchmark. For code generation, it achieves up to a 4.23× speedup on the HumanEval benchmark. This work highlights the potential of integrated candidate generation to accelerate LLM inference while maintaining high-quality outputs.

1 Introduction

Large Language Models (LLMs) are crucial in numerous applications, including question answering, program synthesis, and task automation (Devlin et al., 2019; Brown et al., 2020; Zhang et al., 2022; Touvron et al., 2023). However, the significant inference costs and time-consuming autoregressive decoding processes pose substantial challenges.

The need for efficient token generation is heightened by inference-time scaling, which requires generating longer outputs for complex tasks (Zhong et al., 2024). Multi-agent and pipelined LLM systems improve accuracy and reliability but suffer from long response times due to sequential processing stages (Wang et al., 2024; Santhanam et al., 2024).

Speculative decoding has emerged as a promising technique to accelerate LLM output speed (Leviathan et al., 2023; Chen et al., 2023; Miao et al., 2024; Cai et al., 2024; Zhang et al., 2024; Lin et al., 2024). This method allows LLMs to verify multiple tokens in a single forward pass using small draft models or additional decoding heads, reducing latency. However, it faces limitations such as the need for high-quality draft models, increased GPU memory usage, and orchestration complexity (Chen et al., 2024; Li et al., 2024).

Recent advancements have integrated additional decoding heads directly into the LLM, utilizing its final hidden representations (Cai et al., 2024). While addressing some challenges of separate draft models, this approach still requires fine-tuning and additional GPU memory, which can be substantial for large models (Grattafiori et al., 2024). Complementing speculative decoding, tree attention modifies traditional attention mechanisms to verify multiple sequences more efficiently (Cai et al., 2024; Miao et al., 2024). By structuring tokens hierarchically, tree attention allows parallel verification of multiple speculative paths, enhancing decoding efficiency.

Despite these advancements, there remains substantial room for improvement in the speed and efficiency of LLM inference. We propose PROMTEC, which integrates several novel methods to further accelerate the autoregressive decoding process. Our contributions include:

1. **Prompt Multi-Lookup**: This method enhances string matching techniques to generate multiple candidate sequences from a given input prompt. By leveraging high n-gram overlaps between the input and output sequences, it effectively reduces decoding time by reusing previously seen subsequences.

2. **Template Datastore**: Targeting frequent sequences in domains such as mathematics and code generation, this method constructs a trie structure of normalized patterns. By efficiently matching these patterns with input sequences, it gener-

[*]Corresponding author

ates high-quality speculative candidates that significantly speed up the decoding process.

3. **Common Sequences**: This method focuses on generating candidate sequences based on commonly occurring tokens and single-variable formulae, particularly in specialized domains like mathematics, physics, and computer science.

We evaluate PROMTEC on the miniF2F benchmark for mathematical generation. Our experiments demonstrate substantial improvements in generation speed, achieving up to a 3.91× speedup over standard autoregressive decoding. We extend our evaluation to code generation tasks using the HumanEval benchmark, where PROMTEC achieves a significant speedup of up to 4.23×. Additionally, we provide a detailed analysis of the relationship between the number of candidate sequences and the mean accepted tokens, highlighting the efficiency gains and potential limitations of each component.

2 Related Works

2.1 Speculative Decoding

Speculative Decoding, a Draft-then-Verify strategy, accelerates inference by generating multiple potential tokens at each step and verifying them against the target Language Model (LLM) (Xia et al., 2023). This approach builds on Blockwise Decoding (Stern et al., 2018), which used additional FFNN heads in the Transformer decoder for multi-token generation, validated by the LLM.

Advancements include using smaller LLMs for drafting (Chen et al., 2023; Leviathan et al., 2023) and integrating FFNN heads directly into the target LLM for parallel token generation (Cai et al., 2024). Other methods involve subprocesses or adaptive layer skipping for efficiency (Yang et al., 2024; Zhang et al., 2024), and learnable tokens for better parallel decoding (Monea et al., 2023).

PLD / LLMA (Saxena, 2023; Yang et al., 2023) uses high n-gram overlap between input and output to speed up decoding by matching n-grams in the input. REST (He et al., 2024) retrieves potential tokens from a pre-built datastore. These advancements highlight speculative decoding's versatility and potential to significantly accelerate the inference process while maintaining high-quality outputs.

2.2 Tree Attention

Tree attention (Spector and Re, 2023; Cai et al., 2024; Miao et al., 2024) enhances traditional causal attention by compressing multiple sequences into a single merged sequence with a hierarchical structure. This prevents sibling token interference and allows parallel verification of multiple speculative paths, significantly boosting the efficiency and quality of speculative decoding. Tree-based speculation, as refined by SpecInfer (Miao et al., 2024), further improves this by enabling simultaneous verification of various candidate sequences, streamlining the inference process and ensuring coherent, accurate outputs.

3 Method

3.1 Prompt Multi-Lookup

The PLD / LLMA method (Saxena, 2023; Yang et al., 2023) can only generate at most one candidate sequence in each retrieval step. We improve by generating multiple candidate sequences in each step from the prompt. Incorporating the idea from SpecInfer (Miao et al., 2024), these candidate sequences can then be organized in a token tree structure, where each node represents a sequence of speculated tokens. This tree-based approach allows for the parallel verification of multiple draft sequences against the LLM, significantly increasing the number of generated tokens in a single decoding step and improving the success rate of verification.

To generate multiple candidate sequences from a given input prompt, we uses a reversed Z-function algorithm to efficiently identify matching suffixes within the prompt. The Z-function (Gusfield, 1997) computes an array that, for each position in the reversed prompt, gives the length of the longest substring starting from that position which matches a prefix of the reversed prompt. The Z-function algorithm is provided in Appendix A. By focusing on non-zero Z-values, we can efficiently locate positions within the prompt that have repeating patterns or suffixes.

The process begins by reversing the input prompt and then computing the Z-function for the reversed sequence. This reversed Z-array enables the identification of suffixes that match previous subsequences within the prompt. We filter out the positions corresponding to non-zero Z-values since they indicate matches, and select the top positions based on their Z-values to ensure we retrieve the most significant matches. We set the last element in the Z-array to zero to avoid self-matching.

From these identified positions, we generate the candidate sequences by extracting subsequences of a specified length from the prompt, starting just after each identified match. This approach ensures that multiple candidate sequences, each representing a plausible continuation of the prompt, are generated. The number of candidates and the length of each candidate sequence can be controlled by parameters, allowing for flexible and efficient generation. Setting the number of candidates to be 1 is equivalent to the PLD method. The algorithm is summarized in Algorithm 2 in Appendix B.

3.2 Template Datastore

To further optimize the speculative inference, we utilize a template-based method targeting patterns of frequent sequences in math formulae and code generation.

Templates construction To build the templates for mathematical generation, we start with a collection of LaTeX formulae that serve as the raw patterns. These patterns are preprocessed to replace variables with placeholders to normalize and generalize the sequences. Specifically, non-command segments in the LaTeX strings are identified and their variables are replaced with unique negative tokens representing variable placeholders. This normalization helps in identifying and matching patterns with slight variations in variable names.

Once the patterns are normalized, they are tokenized using a tokenizer. The tokenized patterns are then used to create a trie structure datastore, which is efficient for prefix searching and pattern matching. The trie has two types of edges, exact and wildcard edges. An exact edge connects a node to a positive-valued node, which corresponds to a non-variable token. A wildcard edge connects a node to a negative-valued node, which corresponds to a variable placeholder. See Figure 1.

Templates retrieval The process begins at the root node of the trie. The prefix is examined token by token. For each token in the prefix, the method attempts to follow the corresponding edge in the trie. If an exact match is found for the current token, the method proceeds to the next level of the trie. Simultaneously, the method checks for wildcard edges in each node, allowing it to match variable placeholders that can represent any token. This

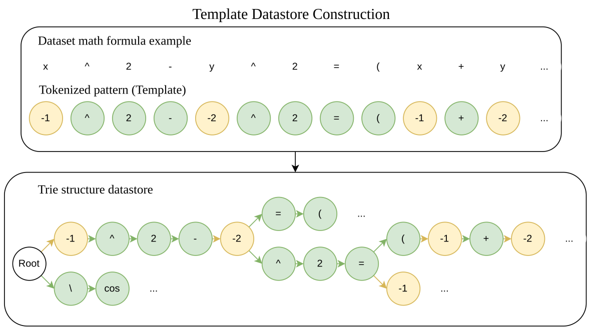
Template Datastore Construction
Dataset math formula example
x ^ 2 - y ^ 2 = (x + y ...
Tokenized pattern (Template)
-1 ^ 2 - -2 ^ 2 = (-1 + -2 ...
Trie structure datastore
Root
-1 ^ 2 - -2
= (...
^ 2 = (-1 + -2 ...
-1 ...
\ cos ...

Figure 1: Template and trie structure datastore construction example. Yellow edges correspond to wildcard edges and green edges correspond to exact edges.

dual approach ensures a comprehensive search that accounts for both exact and flexible matches.

As the traversal progresses, the values stored in each visited node are collected. If the end of the prefix is reached, the traversal stops and the collected values are returned. The collected values form a list of relevant patterns that are considered for candidate sequence generation. See Step 1 in Figure 2.

Pattern pruning Due to the constraint of increased decoding time for a larger number of candidate sequences described in section 4.2.1, we must reduce the number of retrieved patterns to retain the high-quality sequences only. From the matched list of relevant patterns, we construct another trie. We call this a candidate trie. Candidate trie aggregates the frequency of the matching patterns to prioritize more commonly occurring continuations. Each node in this trie reflects the frequency of sequences it holds, influenced by both exact and wildcard matches found in the initial trie, as illustrated in Step 2 in Figure 2. This frequency-based prioritization ensures that the speculative candidates generated are statistically significant.

From the candidate trie, we generate potential continuations of the input prompt. The trie is traversed using a combination of a priority queue and a depth-first search up to a specified depth or length, using frequency to guide the traversal (Step 3 in Figure 2). This assembles the most frequently occurring sequences into speculative candidates.

Subsequently, the speculative candidates are refined and verified. Placeholder tokens within these candidates are substituted back with the actual variable names that appear in the current context. Placeholders with different values are forced to substitute with different variable names, illustrated in

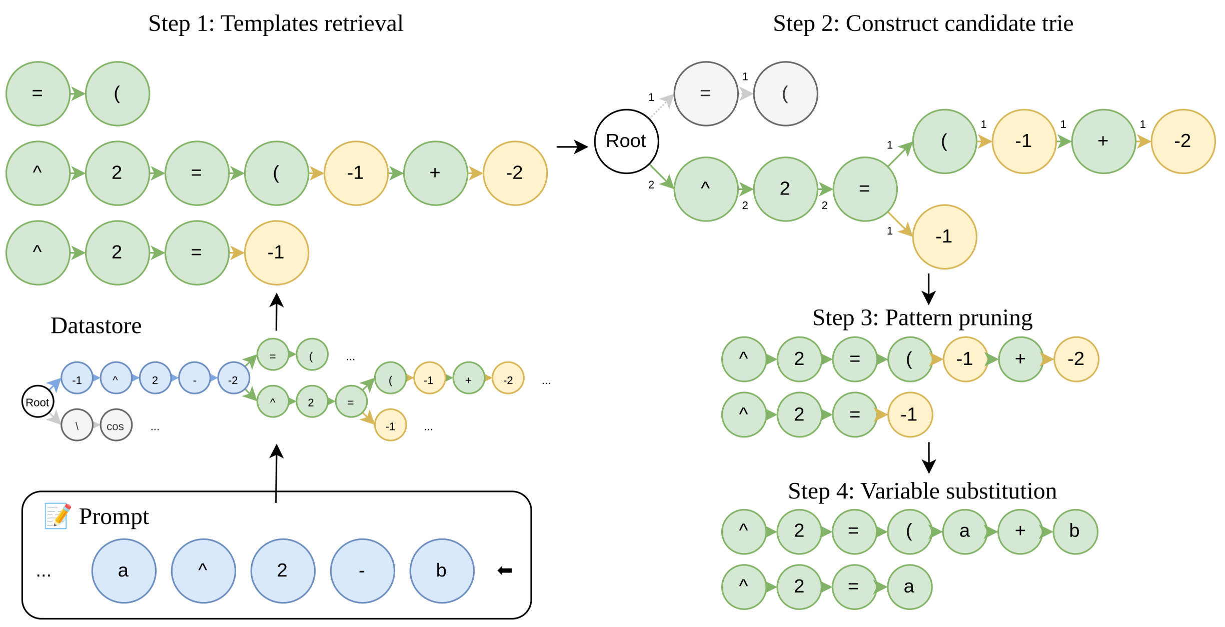
Step 1: Templates retrieval
Step 2: Construct candidate trie
Root
Datastore
Root
Step 3: Pattern pruning
Step 4: Variable substitution
Prompt

Figure 2: Overview of the Template Datastore method.

Step 4 in Figure 2. Any placeholder token and its subsequent tokens are removed if the substitution cannot be made. This context-aware substitution guarantees that the generated candidates are not only syntactically correct but also meaningful in the given context.

Lastly, we implement a fallback mechanism using a datastore introduced in He et al., 2024. If the trie structure does not yield sufficient speculative candidates, the system queries this datastore for other potential sequences. This scenario occurs when LLM is generating uncommon math expressions or regular narrative text. This fallback mechanism enhances the system's ability to handle diverse and unforeseen input scenarios.

Code generation extension Rather than using LaTeX formulae, we use code samples as raw patterns. Each code variable may correspond to more than one tokens, unlike variables in mathematical formulae. The tokens for each code variables are replaced with one negative-valued placeholder. During the trie traversal in templates retrieval, the search logic is modified to allow wildcard edges to match with multiple placeholders.

3.3 Common Sequences

We also generate candidate sequences based on common sequences of tokens. This method primarily targets short sequences involving single-variable formulae and commonly occurring tokens that appear in specialized domains, especially in mathematics, physics, and computer science.

Single variable formulae In a mathematical context, expressions involving single variables are particularly common. Variables often appear encapsulated within delimiters like dollar signs, forming structures such as `"$x$"`.

The context is decoded to extract variables using a utility function that identifies formulae and their associated variables. Single-variable formulae are generated by converting the variable strings into token IDs and surrounding them with token IDs representing the dollar signs. This ensures the generated formulae are consistent with the expected tokenized format. These formulae are immediately added to the list of candidate sequences.

Common tokens In addition to variable-centric expressions, common tokens such as transitional phrases like "Therefore," are prevalent in the discourse of these specialized domains. If the number of single-variable formulae is insufficient to meet the required number of retrievals, the method supplements the remaining slots with common token sequences collected from datasets. Each common token sequence is truncated to the desired length before being added to the candidate list.

Code generation extension Instead of extracting math variables from the context, the utility function is modified to extract code variables. The necessity of adding dollar sign delimiters and common tokens are removed for code generation.

3.4 Draft verification and acceptance

The aforementioned methods will retrieve multiple candidate sequences in parallel. Many of these sequences share common prefixes. We build a token tree from the retrieved candidate sequences and construct a pseudo sequence from the token tree using breadth-first search. Hence, each candidate is a subsequence of the pseudo sequence, thereby consolidating common prefixes to appear only once. We then follow the attention strategy presented in Cai et al., 2024; Miao et al., 2024; Spector and Re, 2023 to construct the tree attention mask. An example is given in Figure 3.

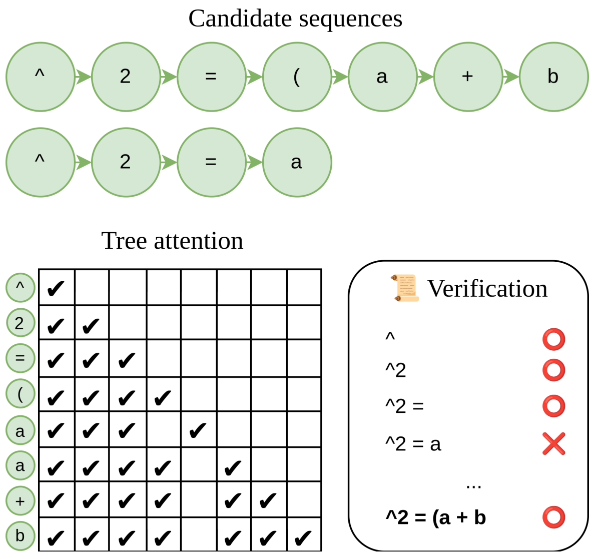

Figure 3: Candidate verification example.

We follow He et al., 2024 to obtain the conditional probability distribution at each token position. We then generate new tokens by sampling from this distribution. Next, we verify if these sampled tokens match the corresponding tokens in the draft. If they do, we sequentially accept them until we encounter a discrepancy. From the first error onwards, all subsequent tokens in the draft are disregarded. This ensures that the final sequences generated through our method align precisely with those produced by traditional autoregressive generation methods.

4 Experiment

4.1 Experimental Setup

Dataset and model We conduct experiments on the miniF2F (Zheng et al., 2022) dataset from OpenAI and HumanEval (Chen et al., 2021). The miniF2F dataset consists of 244 test statements and 244 validation statements, encompassing formalized versions of Olympiad-type problems. These problems span various subdomains of mathematics and difficulty levels, offering a benchmark for formal mathematical reasoning across different formal systems. HumanEval is a collection of 164 programming challenges written by humans, aimed at evaluating models' ability to generate Python code. Each problem comes with a docstring that serves as a prompt for creating the solution. We compare the generation speed of standard autoregressive generation, speculative decoding (Leviathan et al., 2023; Chen et al., 2023) that leverages a small draft LM to generate a single candidate sequence and use LLM to verify the candidate tokens, REST datastore (He et al., 2024), and our method. We employ Llama 2 (Touvron et al., 2023) for mathematical generation and Code Llama (Rozière et al., 2024) for code generation. We use their 7B configurations, with a maximum generation limit of 512 tokens. All experiments are conducted on a single NVIDIA RTX4090 GPU and 32 CPU cores. All results are averaged across 10 different runs.

Sampling strategy We implement greedy sampling for the LLM. At each decoding step, greedy sampling selects the token with the highest probability. Our method ensures that only draft tokens that align with those sampled from the language model are accepted. Consequently, the sequences produced through our approach are indistinguishable from those generated by conventional autoregressive methods.

Baselines We implement speculative decoding (Leviathan et al., 2023; Chen et al., 2023), PLD (Saxena, 2023) and REST (He et al., 2024) as the baselines for comparison. For the small draft LM used in speculative decoding, we adopted TinyLlama 2 1B trained by Ayoola, 2023. Following REST, we construct the REST datastore using data derived from UltraChat (Ding et al., 2023), which consists of around 774K conversations from ChatGPT.

Templates and Common tokens For mathematical generation, we use the MathBridge (Jung et al., 2024) dataset to construct templates and common tokens. MathBridge contains approximately 23 million LaTeX formulae paired with the corresponding mathematical spoken sentences, and the context before and after the formulae. For code generation, we use the Python pretraining code from The

Stack (Kocetkov et al., 2022) dataset, which contains 2.7M Python code samples, to construct code templates.

Metrics The first metric is *Throughput*, which is the average number of tokens generated for the LLM in one second. The second metric that we use is *Mean Accepted Tokens* (MAT), which is computed as the ratio of the length of the generated tokens to the number of forward steps taken by the LLM. If L denotes the length of the generated tokens and F represents the number of forward steps taken, the MAT is:

$$\text{MAT} = \frac{L}{F}$$

4.2 Hyperparameters

4.2.1 Token Tree Size and Forward Time Trade-off

In our analysis of the speculative inference process, we discovered a significant relationship between the latency of language model forward passes and the complexity of the token tree used for generating candidate sequences. Specifically, we observe that the logarithm of the LLM's forward time is linearly proportional to the number of nodes in the token tree, as shown in Figure 4. Mathematically, this relationship can be expressed as:

$$\log(\text{Forward Time}) = m \times (\text{no. of Nodes}) + c$$

where m is the proportional constant, and c is a constant offset.

This result underscores an inherent trade-off between the number of candidate sequences and the inference latency. While generating a larger number of candidate sequences can potentially improve the accuracy and robustness of the model's predictions, it also increases the complexity of the token tree, thereby elevating the computational cost and time required for the forward pass. Consequently, excessively expanding the number of candidate sequences may lead to diminishing returns in performance due to the increased latency.

To optimize the balance between inference efficiency and the quality of speculative decoding, it is crucial to limit the number of candidate sequences. This ensures that, while the model benefits from the diversity of candidates, it remains computationally feasible and responsive.

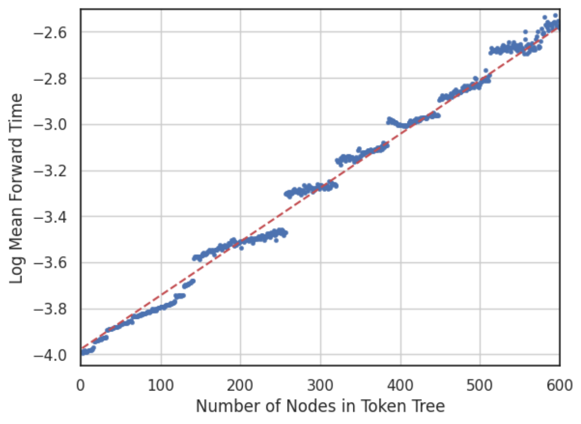

Figure 4: Log mean forward time against different token tree sizes with regression line.

Method	Max seq. len	Max no. of seq.
Speculative	5	1
REST	8	29
PLD	12	1
Prompt	12	5
Template	8	29
Common	3	5

Table 1: Maximum candidate length and number used in our experiments.

4.2.2 Candidate number and length tuning

To determine the optimal hyperparameters for our candidate generation methods, we utilize Optuna (Akiba et al., 2019), an automatic hyperparameter optimization framework. Optuna facilitates efficient and scalable optimization of hyperparameters through an exploration-exploitation balance. The optimization process involved selecting the maximum sequence length and the maximum number of sequences for each of the candidate generation methods, including Prompt Multi-Lookup, Template Datastore, and Common Sequences. The optimization objective is to minimize the runtime taken on validation set of the miniF2F dataset. The optimized parameters are shown in Table 1. To compare the baseline REST datastore with our Template datastore, we have set the same parameters on them. Also, to compare the baseline PLD with our Prompt Multi-Lookup method, the same maximum candidate length is enforced. The maximum number of candidates used in speculative decoding and PLD is 1 as they can only generate and verify a single candidate sequence in each forward pass.

4.3 Main Results

4.3.1 Evaluation on Mathematical Generation

We conducted extensive evaluations of our candidate generation methods on the miniF2F benchmark, both on the validation and test sets. Table 2 compares the generation speed of various configurations of our methods with baseline methods, specifically standard autoregressive decoding, speculative decoding, REST and PLD.

PROMTEC exhibits a notable improvement in the generation speed compared to all baseline methods, achieving a performance increase of 3.87× to 3.91× for Llama2 on the miniF2F benchmark. These empirical findings underscore the effectiveness of our approach in accelerating the LLM generation process.

Speculative decoding, while achieving a high MAT due to the high-quality candidate sequences generated by the small speculative model, demonstrates lower throughput compared to REST and PLD. This reduced throughput is primarily because the time taken for the small speculative model to produce a candidate is longer than the time required by retrieval-based methods. Consequently, although speculative decoding benefits from accurate candidate sequences, it does not match the efficiency of REST and PLD in terms of generation speed.

Impact of Prompt Multi-Lookup PLD, which generates only one candidate sequence at a time, is outperformed by our Prompt Multi-Lookup method. While PLD efficiently leverages string matching to produce a single candidate sequence, our Prompt Multi-Lookup method enhances this approach by generating multiple candidate sequences in each retrieval step. This capability allows for parallel verification of these sequences, significantly increasing the number of tokens generated in a single decoding step and improving overall throughput. As a result, the Prompt Multi-Lookup method not only maintains the efficiency of PLD but also offers superior performance by producing more candidate sequences simultaneously, achieving up to a 2.80× speedup on the miniF2F benchmark.

Impact of Template Datastore Our Prompt Multi-Lookup method is significantly enhanced by incorporating the Template Datastore. The Template Datastore improves the results by providing structured patterns, particularly in math formulae generation, allowing for a more efficient and accurate generation of candidate sequences. When comparing Prompt + Template (which achieves a speedup of 3.79×) with Prompt + REST (which achieves a speedup of 2.92×), it is evident that the Template Datastore performs better, especially in mathematical contexts. The Template Datastore's focus on frequent sequences and its ability to handle variable placeholders make it more effective in generating high-quality speculative candidates for math-related content than the REST method. Further, when we combine Prompt + Template with REST (which has a speedup of 3.72×), the addition of REST shows minimal improvement, indicating that the Template Datastore is capable of handling the candidate generation efficiently on its own. This suggests that REST is largely replaceable by the Template Datastore, particularly in domains where structured patterns and frequent sequences are prevalent.

Impact of Common Sequences The inclusion of Common Sequences in our Prompt + Template method can further enhance the speedup by a small amount. By targeting short, frequently occurring sequences and single-variable formulae, the Common Sequences method complements the Template Datastore. This additional layer of candidate generation slightly improves the overall generation speed, making the combined approach of Prompt + Template + Common even more efficient, with a 3.87× speedup over the autoregressive decoding baseline.

4.3.2 Evaluation on Code Generation

We conducted a thorough evaluation of our code generation methods on the HumanEval benchmark with the same parameters derived in Section 4.2.2, focusing on the effectiveness and speed of our approaches. Table 3 presents the results, highlighting the throughput, Mean Average Time (MAT), and speedup of various configurations of our methods against the baseline autoregressive decoding.

By incorporating Prompt Multi-Lookup method alone, we observe a speedup of 3.88×. The addition of Template Datastore further boosts performance, achieving a speedup of 4.22×. The combination of Prompt Multi-Lookup, Template Datastore, and Common Sequences results in the highest performance, reaching a speedup of 4.23×.

It is important to note that our Prompt Multi-Lookup, Template Datastore, and Common Sequences methods can be executed in parallel, and

	miniF2F val			miniF2F test		
Method	**Throughput**	**MAT**	**Speedup**	**Throughput**	**MAT**	**Speedup**
Autoregressive Decoding	55.56	1.00	1.00×	55.56	1.00	1.00×
Speculative Decoding	91.84	3.46	1.65×	90.32	3.51	1.63×
REST	98.71	1.56	1.78×	96.53	1.54	1.74×
PLD	138.50	2.03	2.49×	138.37	2.07	2.49×
Prompt	155.19	2.35	2.79×	155.32	2.39	2.80×
Prompt + REST	162.09	2.54	2.92×	162.00	2.59	2.92×
Prompt + Template	212.63	3.52	3.83×	210.53	3.55	3.79×
Prompt + Template + REST	208.79	3.57	3.76×	206.56	3.60	3.72×
Prompt+Template+Common	**217.32**	**3.66**	**3.91**×	**215.07**	**3.70**	**3.87**×

Table 2: Generation speed of Llama2 7B on miniF2F benchmark with baselines and different settings of our methods.

	HumanEval		
Method	**Throughput**	**MAT**	**Speedup**
Autoregressive Decoding	55.57	1.00	1.00×
Prompt	215.42	4.37	3.88×
Prompt + Template	234.41	5.69	4.22×
Prompt + Template + Common	**235.26**	**5.74**	**4.23**×

Table 3: Generation speed of Code Llama 7B on HumanEval benchmark with PROMTEC extended for code generation.

the average time required for retrieval is less than 1ms. This extremely small retrieval time is practically negligible, further underscoring the efficiency of our methods. By performing these operations concurrently, we can maintain high throughput and rapid candidate sequence generation without any significant impact on overall performance.

4.4 Analysis of MAT vs Number of Candidates

To better highlight the strengths and limitations of each component, we analyze the impact of the number of candidate sequences on the Mean Accepted Tokens for our Prompt Multi-Lookup, Template Datastore, and Common Sequences generation methods. Figure 5 shows the results for a range of candidate sequences from 1 to 50.

For the Prompt Multi-Lookup method, MAT increases rapidly from 2.03 to 2.35 as the number of candidate sequences increases from 1 to 5. Beyond this point, the MAT continues to improve gradually, reaching a stable value of 2.40 at around 15 candidate sequences. After which the gains become negligible. This behavior can be attributed to the inherent limitation of the Prompt Multi-Lookup method: it cannot suggest many candidates due to the typically short length of prompts, which restricts the number of possible suffix matches.

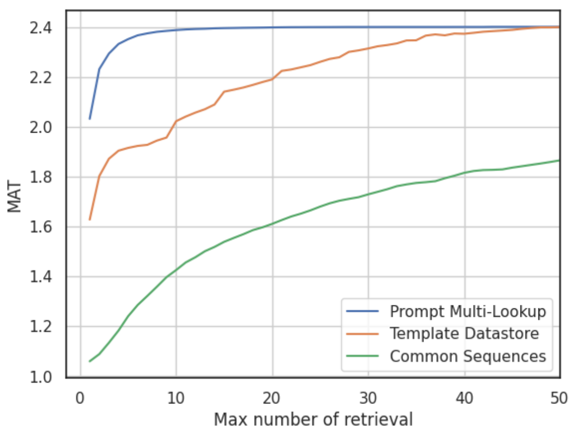

Figure 5: Mean accepted tokens for Prompt Multi-Lookup, Template Datastore and Common Sequences for various maximum number of retrieved candidate sequences.

For the Template Datastore method, its MAT starts at 1.63 for a single candidate and increases steadily as more candidates are added. It reaches a stable value of 2.38 at 40 candidate sequences. This indicates that the Template method continues to benefit from additional candidate sequences up to a higher threshold compared to the Prompt Multi-Lookup method.

The MAT for the Common Sequences method remains lower compared to the Prompt Multi-Lookup and Template Datastore methods. This is because the Common Sequences method produces lower quality candidates, as it takes less information about the context into consideration. While it targets short, frequently occurring sequences and single-variable formulae, it lacks the depth of contextual understanding provided by the Prompt Multi-Lookup and Template Datastore methods.

These findings emphasize the importance of optimizing the number of candidate sequences for each method to achieve a balance between MAT improvement and computational efficiency. The negligible retrieval time for our components further underscores the efficiency and practicality of these approaches in enhancing language model performance.

5 Conclusion

This study introduced a novel approach to enhancing the efficiency of Large Language Models (LLMs) by integrating Prompt Multi-Lookup, Template Datastore, and Common Sequences methods. These techniques collectively improve the speed of autoregressive decoding by generating multiple candidate sequences efficiently. Our evaluations on the miniF2F benchmark demonstrate significant speedups, achieving up to a 3.91$\times$ improvement over standard methods. Additionally, on the HumanEval benchmark for code generation, our methods achieve up to a 4.23$\times$ speedup. This approach outperforms existing speculative decoding techniques, offering a scalable solution for faster LLM inference while maintaining high-quality outputs. Moreover, we performed an ablation study to understand the individual contributions of each component to the overall performance improvement. Future work can explore further optimizations and applications to larger models and other domains.

Limitations

The limitations of our work are as follows:

- The effectiveness of the Template Datastore heavily relies on the availability of high-quality domain-specific templates. While this approach showed significant improvements in mathematical and code generation tasks, its performance may degrade in domains where structured patterns are less prevalent or harder to define.
- The construction of code templates is code-language dependent, which may limit the generalizability of our approach to other programming languages.
- The common tokens in the Common Sequences method are language dependent, which may reduce its effectiveness across different natural languages.

References

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631.

Odunusi Abraham Ayoola. 2023. Tinyllama-2-1b-miniguanaco.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *Preprint*, arXiv:2401.10774.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. *Preprint*, arXiv:2302.01318.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen

Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. 2024. Sequoia: Scalable, robust, and hardware-aware speculative decoding. *Preprint*, arXiv:2402.12374.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. *Preprint*, arXiv:1810.04805.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing chat language models by scaling high-quality instructional conversations. *Preprint*, arXiv:2305.14233.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Dan Gusfield. 1997. *Simple Uniform Preprocessing for Linear-time Pattern Matching*. Cambridge University Press.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason D. Lee, and Di He. 2024. Rest: Retrieval-based speculative decoding. *Preprint*, arXiv:2311.08252.

Kyudan Jung, Sieun Hyeon, Jeong Youn Kwon, Nam-Joon Kim, Hyun Gon Ryu, Hyuk-Jae Lee, and Jaeyoung Do. 2024. Mathbridge: A large corpus dataset for translating spoken mathematical expressions into $latex$ formulas for improved readability. *Preprint*, arXiv:2408.07081.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 tb of permissively licensed source code. *Preprint*, arXiv:2211.15533.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. *Preprint*, arXiv:2211.17192.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. Eagle-2: Faster inference of language models with dynamic draft trees. *Preprint*, arXiv:2406.16858.

Feng Lin, Hanling Yi, Hongbin Li, Yifan Yang, Xiaotian Yu, Guangming Lu, and Rong Xiao. 2024. Bita: Bi-directional tuning for lossless acceleration in large language models. *Preprint*, arXiv:2401.12522.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. 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, page 932–949. ACM.

Giovanni Monea, Armand Joulin, and Edouard Grave. 2023. Pass: Parallel speculative sampling. *Preprint*, arXiv:2311.13581.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna

Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Keshav Santhanam, Deepti Raghavan, Muhammad Shahir Rahman, Thejas Venkatesh, Neha Kunjal, Pratiksha Thaker, Philip Levis, and Matei Zaharia. 2024. Alto: An efficient network orchestrator for compound ai systems. In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 117–125, New York, NY, USA. Association for Computing Machinery.

Apoorv Saxena. 2023. Prompt lookup decoding.

Benjamin Spector and Chris Re. 2023. Accelerating llm inference with staged speculative decoding. *Preprint*, arXiv:2308.04623.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models. *Preprint*, arXiv:1811.03115.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *Preprint*, arXiv:2307.09288.

Zilong Wang, Zifeng Wang, Long Le, Huaixiu Steven Zheng, Swaroop Mishra, Vincent Perot, Yuwei Zhang, Anush Mattapalli, Ankur Taly, Jingbo Shang, Chen-Yu Lee, and Tomas Pfister. 2024. Speculative rag: Enhancing retrieval augmented generation through drafting. *Preprint*, arXiv:2407.08223.

Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. 2023. Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation. *Preprint*, arXiv:2203.16487.

Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Inference with reference: Lossless acceleration of large language models. *Preprint*, arXiv:2304.04487.

Seongjun Yang, Gibbeum Lee, Jaewoong Cho, Dimitris Papailiopoulos, and Kangwook Lee. 2024. Predictive pipelined decoding: A compute-latency trade-off for exact llm decoding. *Preprint*, arXiv:2307.05908.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2024. Draft & verify: Lossless large language model acceleration via self-speculative decoding. *Preprint*, arXiv:2309.08168.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. Opt: Open pre-trained transformer language models. *Preprint*, arXiv:2205.01068.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *Preprint*, arXiv:2109.00110.

Tianyang Zhong, Zhengliang Liu, Yi Pan, Yutong Zhang, Yifan Zhou, Shizhe Liang, Zihao Wu, Yanjun Lyu, Peng Shu, Xiaowei Yu, Chao Cao, Hanqi Jiang, Hanxu Chen, Yiwei Li, Junhao Chen, Huawen Hu, Yihen Liu, Huaqin Zhao, Shaochen Xu, Haixing Dai, Lin Zhao, Ruidong Zhang, Wei Zhao, Zhenyuan Yang, Jingyuan Chen, Peilong Wang, Wei Ruan, Hui Wang, Huan Zhao, Jing Zhang, Yiming Ren, Shihuan Qin, Tong Chen, Jiaxi Li, Arif Hassan Zidan, Afrar Jahin, Minheng Chen, Sichen Xia, Jason Holmes, Yan Zhuang, Jiaqi Wang, Bochen Xu, Weiran Xia, Jichao Yu, Kaibo Tang, Yaxuan Yang, Bolun Sun, Tao Yang, Guoyu Lu, Xianqiao Wang, Lilong Chai, He Li, Jin Lu, Lichao Sun, Xin Zhang, Bao Ge, Xintao Hu, Lian Zhang, Hua Zhou, Lu Zhang, Shu Zhang, Ninghao Liu, Bei Jiang, Linglong Kong, Zhen Xiang, Yudan Ren, Jun Liu, Xi Jiang, Yu Bao, Wei Zhang, Xiang Li, Gang Li, Wei Liu, Dinggang Shen, Andrea Sikora, Xiaoming Zhai, Dajiang Zhu, and Tianming Liu. 2024. Evaluation of openai o1: Opportunities and challenges of agi. *Preprint*, arXiv:2409.18486.

A Z function

The Z-function is a powerful algorithm used in string processing to compute an array of values that represent the longest substring starting from each position in the string, which matches a prefix of the string. The algorithm is given in Algorithm 1. This function is particularly useful in pattern matching. The Z-function is efficient and runs in linear time, making it suitable for various applications in string matching and text processing. This function is instrumental in the Prompt Multi-Lookup method described in Section 3.1, where it helps identify matching suffixes within the prompt to generate multiple candidate sequences.

Algorithm 1 Z-function

```
Input: $s$: The input array of token IDs.
Output: Z-array.
 1: Initialize $z$ array with the length of $s$.
 2: Set $(l, r) \leftarrow (0, 0)$
 3: for $i$ from $1$ to $n - 1$ do
 4:     if $i \leq r$
 5:         $z[i] = \min(r - i + 1, z[i - l])$
 6:     else
 7:         $z[i] = 0$
 8:     end if
 9:     while $i+z[i] < n$ and $s[z[i]] = s[i+z[i]]$ do
10:         $z[i] = z[i] + 1$
11:     end while
12:     if $i + z[i] - 1 > r$
13:         $l = i$
14:         $r = i + z[i] - 1$
15:     end if
16: end for
17: return $z$
```

B Prompt Multi-Lookup Algorithm

Algorithm 2 Prompt Multi-Lookup

```
Input: $prompt$: The input prompt
       $retrieve\_len$: Length of each retrieved sequence
       $retrieve\_num$: Number of sequences to retrieve
Output: A list of candidate sequences.
 1: prompt_rev = reverse $prompt$
 2: z_arr = z_function(prompt_rev)
 3: z_arr_rev = reverse z_arr
 4: Set the last element of z_arr_rev to 0.
 5: Identify non-zero elements in z_arr_rev.
 6: k = min ($retrieve\_num$, number of non-zero elements).
 7: Initialize retrieved_seqs $\leftarrow$ []
 8: Get top-k indices based on z_arr_rev values.

 9: for each position $p$ in top-k indices do
10:     Add $prompt[p+1 : p+1+retrieve\_len]$ to retrieved_seqs.
11: end for
12: return retrieved_seqs
```