university of koblenz
Computer Science

Grammar-enforced Chain of Thought Reasoning for small LLMs

Project Internship / Research Internship

Submitted By

Shwetha Babu (SB, 222202641)
Jan-Niklas Schmidt (JNS, 220201519)
Jan Hillesheim (JH, 219201598)

Reviewer: Dipl. Inf. Heiko Neuhaus, Abteilung Finanzierung, Finanzdienstleistungen und eFinance, Fachbereich 4: Institut für Management, Universität Koblenz

Koblenz, April 2025

Abstract

Large Language Models (LLMs), particularly smaller models, often struggle with complex reasoning tasks requiring structured, multi-step thought processes. Generating Chain of Thoughts (CoTs) aims to elicit such reasoning, but unconstrained outputs can lack consistency and logical rigor. This paper investigates the use of formal grammars, specifically the Grammar-Based Notation Format (GBNF), to enforce structured CoT reasoning during the decoding process of small LLMs. We compare the performance of grammar-constrained CoT generation against unconstrained baseline outputs across diverse benchmarks, including MMLU (Math and Logic), a self-developed custom Financial benchmark tiered by difficulty, and a Zebra Puzzle benchmark for evaluating novel logical deduction. Our experimental results demonstrate that while GBNF grammars successfully enforce the desired CoT structure, leading to enhanced output interpretability and predictability, they do not consistently improve objective accuracy metrics on benchmarks like MMLU and Finance. This performance gap might be attributed to factors like dataset memorization in the base model or the cognitive overhead imposed by rigid structures. However, we observed slight performance improvements on the Zebra Puzzle benchmark, suggesting potential benefits for tasks requiring robust reasoning on unfamiliar problems. We conclude that grammar-enforced CoT presents a valuable tool for controlling LLM output and ensuring trustworthiness, but highlights a trade-off between structural rigidity and raw task performance, particularly for smaller models.

Kurzfassung

Große Sprachmodelle (Large Language Models, LLMs), insbesondere kleinere Modelle, haben häufig Schwierigkeiten mit komplexen Aufgaben, die strukturiertes, mehrstufiges Denken erfordern. Die Erzeugung sogenannter „Chain of Thoughts“ (CoTs) soll solches Denken fördern, jedoch mangelt es unstrukturierten Ausgaben oft an Konsistenz und logischer Stringenz. Diese Arbeit untersucht den Einsatz formaler Grammatiken, insbesondere des Grammar-Based Notation Format (GBNF), um während des Inferenzprozesses kleiner LLMs strukturiertes CoT-Denken zu erzwingen. Wir vergleichen die Leistung gram-

matikbasiert erzeugter CoT-Ausgaben mit unregulierten Ausgaben anhand verschiedener Benchmarks, darunter MMLU (Mathematik und Logik), einer selbst entwickelten Finanz-Benchmark mit gestaffeltem Schwierigkeitsgrad sowie eine Zebra-Puzzle-Benchmark zur Bewertung neuartiger logischer Schlussfolgerungen. Unsere Experimente zeigen, dass GBNF-Grammatiken zwar erfolgreich die gewünschte CoT-Struktur erzeugen und so die Interpretierbarkeit und Vorhersagbarkeit der Ausgaben verbessern, jedoch nicht durchgängig zu besseren Scores auf Benchmarks wie MMLU und Finance führen. Diese Leistungslücke könnte auf Faktoren wie Datensatz-Memorisierung im Basismodell oder den kognitiven Aufwand durch starre Strukturen zurückzuführen sein. Bei der Zebra-Puzzle-Benchmark konnten hingegen leichte Leistungsverbesserungen festgestellt werden, was auf mögliche Vorteile bei Aufgaben mit hoher logischer Komplexität und unbekannten Problemstellungen hindeutet. Wir schließen daraus, dass grammatikgestützte CoT-Erzeugung ein wertvolles Werkzeug zur Steuerung von LLM-Ausgaben und zur Erhöhung der Vertrauenswürdigkeit darstellt – jedoch mit einem Zielkonflikt zwischen struktureller Strenge und roher Aufgabenleistung, insbesondere bei kleineren Modellen.

Keywords: Large Language Models (LLMs), Chain Of Thought (CoT), Grammar-Based Decoding

Erklärung

Wir versichern, dass wir die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Zur Formulierung wurden teilweise LLMs als Hilfsmittel eingesetzt.

Mit der Einstellung dieser Arbeit in die Bibliothek sind wir einverstanden. ja ☒ nein ☐

Der Veröffentlichung dieser Arbeit im Internet stimmen wir zu. ja ☐ nein ☒

Koblenz, den April 15, 2025

Contents

List of Abbreviations

API Application Programming Interface

CoT Chain of Thought

FSA Finite-State Automata

GBNF Grammar-Based Notation Format

HoT Hint of Thought

JSON JavaScript Object Notation

LLM Large Language Model

LoRA Low-Rank Adaptation

MCQ Multiple Choice Question

MMLU Massive Multitask Language Understanding

RLHF Reinforcement Learning from Human Feedback

XML Extensible Markup Language

Chapter 1

Introduction

There is an immense amount of interest in creating strategies to improve language models' reasoning abilities due to their growing use in a variety of tasks, particularly for small-scale models with few parameters. Chain of Thought (CoT) reasoning is one such method that encourages models to express intermediate phases in their reasoning before reaching a final conclusion. This method has demonstrated potential in enhancing performance on challenging tasks, especially those requiring multi-step arithmetic or logic. Without limitations, however, CoT may generate illogical or contradictory chains of reasoning, particularly in small language models that have trouble with precision and coherence.

We analyse the application of grammars as a means of limiting CoT reasoning in order to address this. Grammars can assist in directing the generating process by imposing syntactic and semantic norms. We explore how grammars can be used as a technique to constrain CoT reasoning in order to overcome this. Grammars can assist in directing the model toward more organized and consistent reasoning paths by enforcing syntactic and semantic norms during generation. This internship analyses whether constraining the CoT can make a difference in the performance of the model compared to the CoT without the grammar.

In this paper, we suggest a grammar-constrained CoT architecture that incorporates formal grammar rules, namely those defined with a Grammar-Based Notation Format (GBNF), into the LLMs' reasoning process. We hope to influence the creation of reasoning stages in a way that guarantees logical consistency, structural validity and interpretability

by directly integrating these constraints into the model's decoding procedure. In domains where accuracy is crucial including finance and mathematics, this method not only offers a logical method for error prevention but also makes it easier to create domain-specific reasoning pipelines. Our hypothesis is that small models can be guided toward more accurate and resilient performance, even in tasks that are typically dominated by large-scale models, by combining CoT with grammar-based guidance.

1.1 Related Work SB

Earlier studies on Chain of Thought prompting [WWS+22] have shown that clear reasoning processes are a major advantage for LLMs such as GPT-3 and PaLM. Subsequent research has examined few-shot learning and fine-tuning techniques for CoT creation. In order to increase output validity, limited decoding methods such as those based on syntactic trees, finite state machines or custom grammars have been investigated in machine translation, code generation and semantic parsing. Numerous studies that extend, optimize and constrain the CoT process have been inspired by CoT prompting since the start. These studies fall under a number of general categories, including: (1) improving CoT for small models, (2) automating the creation of demonstrations, (3) adding constraints to the creation of CoT and (4) developing benchmarking and assessment methods.

Applying a straightforward but effective prompting technique, Kojima et al.'s study "Large Language Models are Zero-Shot Reasoners" [KGR+22] provides valuable insights into the latest reasoning capabilities of large language models. The study demonstrates that adding the phrase "Let's think step by step" to the input prompt can cause effective reasoning behavior in LLMs, even in a zero-shot setting, unlike conventional methods that rely on in-context learning with examples. This low-effort baseline for eliciting CoT reasoning across various benchmarks demonstrates that LLMs have latent reasoning capabilities that can be rekindled with the appropriate prompt structure. The findings show that this kind of urging greatly increases accuracy on difficult tasks, especially for larger models like GPT-3. However, the zero-shot CoT approach is vulnerable to structural accuracy and may produce illogical or inconsistent reasoning paths. The study expands the idea of CoT prompting by imposing formal constraints through grammars, aiming

to maintain the advantages of CoT while guaranteeing well-formed, comprehensible, and trustworthy reasoning chains

MathPrompter [IDS23] is a framework to enhance the mathematical reasoning capabilities of large language models by utilizing self-verification techniques and prompt engineering. The framework encourages the model to generate multiple intermediate reasoning paths before converging on a final answer, addressing the challenges LLMs face in solving complex mathematical problems. MathPrompter's self-consistency strategy outperforms baseline zero-shot and few-shot CoT methods, resulting in better performance on challenging math benchmarks like GSM8K and MATH. It demonstrates that structured reasoning procedures can be advantageous for even highly skilled LLMs. However, MathPrompter does not provide explicit control over the correctness or structure of reasoning stages. Instead, it directly shapes and validates the style of generated reasoning paths by introducing external restrictions via formal grammars. This approach aims to enhance LLM outputs' accuracy, reliability, and interpretability in mathematical contexts. MathPrompter is a significant precursor to the grammar-constrained Chain-of-Thought framework, highlighting the importance of structured, multi-step reasoning in high-stakes domains like mathematics.

An Explainable and Zero-Shot Approach to Reasoning Tasks with LLMs - Hint of Thought Prompting (HoT) [LD23] introduces a unique prompting technique called Hint of Thought prompting. HoT prompting is a zero-shot approach that embeds lightweight, semantically meaningful cues into the input to guide the model's reasoning process. This approach is brief, generalizable, and interpretable, allowing LLMs to produce more organized and comprehensible responses for various reasoning tasks. The study found that HoT prompting is more effective and simpler to implement in zero-shot scenarios, achieving competitive accuracy compared to typical CoT techniques. It also enhances the decision-making process's transparency by activating reasoning-relevant patterns within the model through hints. Despite providing structure and control, HoT primarily functions under the open-ended generation paradigm of LLMs, where logical or structural errors may go unnoticed.

Our work expands on the goal of HoT by adopting a formal strategy of constrained reasoning using established grammars like GBNF. This approach ensures syntactic and semantic correctness in reasoning outputs, providing an alternative approach that complements HoT's objectives while introducing additional control and rigor necessary for domain-specific and high-stakes reasoning applications.

1.2 Research Question SB

The central research question of this internship is: **Does enforcing Chain of Thought output via grammars improve LLM accuracy on mathematical and financial reasoning problems?**

This internship introduces a method that incorporates formal grammar rules, specifically GBNF, to constrain the CoT reasoning process. Our method incorporates formal grammar rules directly into the reasoning process, which sets it apart from the previously described approaches. We guarantee the logical consistency and structure of every reasoning step by imposing constraints on the model's outputs to conform to syntactic and semantic norms during generation. The CoT process is improved by this grammar constraint mechanism, which adds another level of control to ensure that reasoning stages are not only legitimate but also traceable and interpretable.

The main contribution is to demonstrate how LLMs can greatly enhance their performance on challenging mathematical reasoning tasks by including grammatical constraints into CoT reasoning. In contrast to earlier research that mostly concentrated on self-consistency or prompting techniques, we present a novel approach to reasoning that blends the strength of CoT with exacting standards of formal grammar-based instruction. We aim to demonstrate how grammar constraints can guide even smaller models, which typically struggle with reasoning accuracy to perform better. This will create new opportunities for using these models in real-world applications that demand logical consistency and precise reasoning.

This is an unexplored method that has not been thoroughly examined in existing literature. Our goal is to offer novel insights on how formal grammatical structures might be employed to improve LLMs' reasoning skills and enhancing their performance on challeng-

ing reasoning problems especially in tasks that require a lot of mathematics and financial reasoning knowledge.

1.3 Structure of this Paper JH

This paper is organized into four main chapters to systematically present our investigation into grammar-enforced Chain of Thought reasoning for small LLMs.

Following this introductory chapter (Chapter 1), which establishes the motivation, reviews pertinent related work, and formulates the central research question, the subsequent chapters proceed as follows:

Chapter 2: Background provides the necessary theoretical and technical context for this work. It offers an overview of Large Language Models, discusses relevant benchmarking approaches, briefly touches upon fine-tuning methods, and delves into various constrained decoding techniques, with a particular focus on grammar-based methods, which form the foundation of our approach.

Chapter 3: Methodology and Results details the empirical core of our study. This chapter begins by outlining the overall experimental design. It then elaborates on the development process for our novel financial benchmark dataset. Subsequently, it describes the construction of the formal GBNF grammars designed to enforce specific Chain of Thought structures. The software architecture and implementation details are also presented. Finally, this chapter presents the quantitative experimental results, comparing the performance of grammar-constrained decoding against unconstrained generation across the selected benchmarks.

Chapter 4: Discussion offers an interpretation and analysis of the findings presented in Chapter 3. We delve into the implications of the experimental results, particularly discussing the observed tradeoffs between enforced output structure and raw task performance. The strengths and limitations of the proposed grammar-based approach are considered within the context of the research question. This concluding chapter also identifies and outlines promising directions for future research in the area of controlled reasoning generation for LLMs.

Chapter 2

Background

The following sections lay the foundational groundwork necessary to understand the techniques and evaluation methods employed in our study. We begin by providing an overview of Large Language Models (LLMs), followed by a discussion of common benchmarking practices and fine-tuning approaches prevalent in the field. Subsequently, we delve into various constrained decoding techniques, highlighting the methods that form the basis for grammar-based output control, which is central to this work.

2.1 Large Language Models SB

A type of machine learning model intended for natural language processing applications like language generation is the large language model. LLMs are multi-parameter language models that are learned on a large volume of text using self-supervised learning. Generative pretrained transformers (GPTs) are the biggest and most powerful LLMs. Current models can be tuned for specific tasks or directed by prompt engineering. These models inherit biases and mistakes from the data they are trained on, but they also gain predictive abilities about syntax, semantics and ontologies that are inherent in human language corpora.[llm25, p.1]

2.2 Benchmarking Approaches SB

The research community has widely embraced LLM benchmarks, which are standard datasets and tasks, to evaluate and compare the performance of different models. These

benchmarks consist of established evaluation criteria and processes, as well as specified splits for training, validation and testing. By establishing requirements that models must achieve or surpass, benchmarks offer a common platform for methodically comparing various models and methodologies and evaluating progress. Although metrics evaluate model output directly, benchmarks provide a consistent framework for comprehending the importance of these metrics in relation to capability or progress.[ben24, p.1]

2.3 Fine-Tuning Methods JNS

Fine-tuning is a machine learning technique that uses a pre-trained model as a foundation to improve its performance through further training on a smaller, domain-specific dataset. It forms a bridge between general-purpose pretraining and task-specific requirements. This process enables a pre-trained model to adapt to a new domain, balancing prior knowledge with new task demands while retaining foundational capabilities. By refining model parameters on targeted data, it achieves specialized performance without sacrificing the efficiency of starting from scratch.

There are a multitude of fine-tuning techniques that have arisen over the years, namely: **Full-parameter Updates** [1], task-specific alignment, **Adapter Modules** [2] for parameter-efficient adaptation, **Low-Rank Adaptation (LoRA)** [3] to reduce computational overhead, **Instruction Tuning** [4] for generalizable task prompting, and **Reinforcement Learning from Human Feedback (RLHF)** [5]for human-aligned output.

While these methods adapt model behavior through weight updates, enabling applications like code generation (LoRA) or preference-aligned dialogue (RLHF), they rely on task-specific data and significant computational resources. Fine-tuning also risks overfitting to narrow domains, limiting flexibility when output formats or reasoning patterns evolve. In contrast, grammar-guided decoding enforces structured reasoning (e.g., step-by-step chains) programmatically during inference, offering a lightweight alternative that requires no training data or weight updates.

[1]Updates all model weights on task specific data, e.g., BERT for question answering [DCLT19]
[2]Inserts lightweight layers between transformer blocks for parameter-efficient tuning[HGJ+19]
[3]Freezes base model and injects trainable low-rank matrices to reduce compute [HSW+21]
[4]Trains models on diverse tasks phrased as instructions for generalization e.g., FLAN [WTB+22]
[5]Aligns outputs with human preferences via reward modeling, e.g., InstructGPT [OWJ+22]

2.4 Constrained Decoding Techniques JNS

Modern large language models (LLMs) generate text by sequentially predicting tokens based on learned probability distributions. Standard decoding techniques, such as **greedy search**[1] and **beam search**[2], prioritize fluency but lack mechanisms to enforce structural validity. Constrained decoding addresses this limitation by restricting token selection to predefined rules, ensuring outputs adhere to formats like JSON schemas, formal grammars, or regex patterns.

2.4.1 Methods of Constrained Decoding

There are numerous constrained decoding methods used which include:

- **Lexical Constraints**: Enforces the inclusion, exclusion, or specific placement of predefined words within the generated output sequence.
- **Regex-Based Decoding**: Restricts outputs to patterns like dates or identifiers using regular expressions.
- **Finite-State Automata(FSA)-Based Decoding**: Guides generation using state machines to enforce sequential constraints beyond basic regular expressions.
- **Schema-Based Decoding**: Guarantees conformance to data schemas (e.g., JSON), ensuring API-compatible outputs.
- **Grammar-Based Decoding**: Uses formal grammars (e.g., BNF) to enforce syntactically valid sequences, critical for code generation or linguistic structures.

These methods prune invalid tokens at each generation step, ensuring strict adherence to constraints. By dynamically restricting the token vocabulary during decoding, they enable reliable structure generation. However, their reliance on local validity checks may not guarantee coherence or semantic correctness.

[1] Selects the highest-probability token at each step, often leading to locally optimal but globally suboptimal sequences [CS18].

[2] Maintains multiple candidate sequences (beams) to balance diversity and quality [VSP+17].

2.4.2 Lexical Constraints

Lexical Constraints represent one of the simplest and widely used methods in constrained decoding. It is particularly useful in applications requiring strict control over vocabulary, such as compliance with domain-specific terminology, avoidance of sensitive language, or adherence to stylistic guidelines.

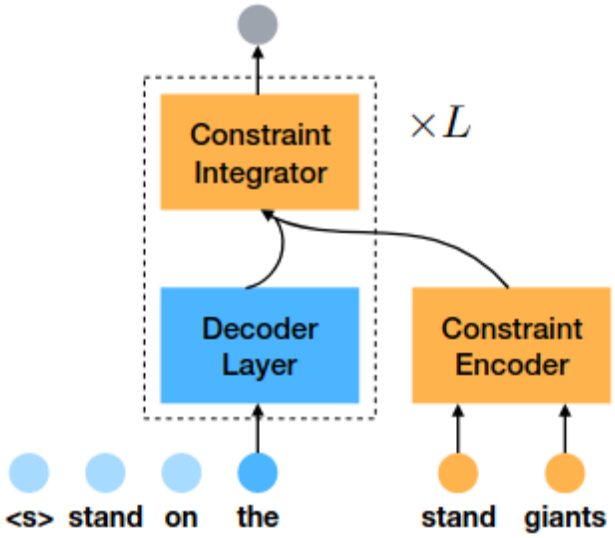


Figure 2.1: The decoder of the proposed framework [MZZ19]

The provided image (Figure 2.1: The decoder of the proposed framework) illustrate how lexical constraints operate within a decoder architecture. In this decoder framework, the model generates tokens step-by-step, with constraints acting as filters. At each decoding step, the vocabulary is dynamically restricted to tokens that satisfy the predefined rules. For instance, if the constraint required the word "stand" to appear twice, the decoder might enforce the inclusion of the word, even if it disrupts natural flow.

Limitations: Strictly enforcing lexical constraints may disrupt the natural fluency and coherence of the generated text, potentially leading to awkward phrasing or nonsensical outputs if the forced words don't fit the context. Furthermore, these constraints operate solely at the token level and lack the ability to understand more complex semantic relationships within the sequence.

2.4.3 Regex-Based Decoding

A regular expression (Regex) is a sequence of characters that specifies a match pattern in a text. Regexes can be used to decode sequences by guiding the generation process to ensure the final output string conforms to the specified pattern. This method integrates regex matching into the decoding algorithm, typically operating alongside standard decoding methods like beam search [ZZ+18]. This pattern matching is often implemented efficiently by tracking the possible valid continuations based on the regex structure.

1. **Date Format (YYYY-MM-DD)** Corresponding regex:

   ```
   ^\d{4}-\d{2}-\d{2}$
   ```

   During Decoding, after generating ‘2025‘, the pattern-matching logic would recognize that the next token must be ‘-‘. After ‘2025‘ it requires two digits (e.g., ‘04‘) After ‘2025-04-‘ it requires two more digits (e.g., ‘05‘). ‘^‘ and ‘$‘ ensure that the entire string matches this pattern exactly.

2. **Specific Identifier (e.g., starts with ’KOBL’ followed by exactly 3 capital letters**

   Corresponding regex:

   ```
   ^KOBL[A-Z]{3}$
   ```

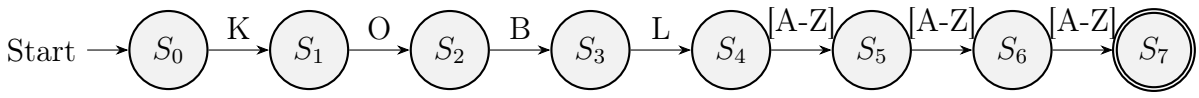
   The decoder would be guided to generate ‘KOBL‘ initially. The pattern constraint would only permit uppercase letters for the next three tokens.

Limitations: Regex-Based Decoding enforces local patterns but lacks coherence. For example:

```
LLM Output: "2025-13-45"
=> Invalid (non-existent month/day)
LLM Output: "KOBLXYZ"
=> Potentially Invalid (structurally matches regex, but ’XYZ’ might be
    non-sensical)
```

2.4.4 Finite-State Automata(FSA)-Based Decoding

While Regex-Based Encoding effectively handles specific structural patterns, its limitations capturing more complex dependencies or ensuring semantic validity becomes apparent. This leads us to **Finite-State Automata(FSA)**[1]-Based Decoding, which utilizes the nature of state machines to guide generation. FSAs not only offer an efficient mechanism for enforcing regex constraints during text generation but also establish a generalized framework for defining complex sequential dependencies through state transitions. This state-driven method increases accuracy and helps with guiding the model outputs, that exceed standalone regex. The following figure shows our previous regex example ^KOBL[A-Z]3$ turned into a FSA:



During FSA-based decoding, the system tracks the current state within a pre-defined FSA representing the desired structure. At each generation step, the decoder consults the FSA to identify which tokens are valid transitions. It then modifies the language model's probability distribution, effectively masking any tokens that would lead to an invalid state, thereby forcing the output strictly to the defined paths. The system gains efficiency by pre-compiling the regex into a FSA before generation begins. It allows fast validation for the next token via a simple state transition lookup. This avoids the significant computational cost of repeatedly re-running the entire regex pattern against the growing sequence at each step, thus being the reason why regex-based decoding is usually implemented using FSAs. [TK24]

Limitations: However, constructing the FSA itself can be challenging; complex patterns can lead to FSAs with a large number of states, increasing memory usage and initial compilation time. Furthermore, FSAs inherit a key limitation from regex-based approaches: they only validate syntactic structure, lacking the capability to ensure semantic correctness (e.g., permitting an invalid date like '2025-99-99' if it matches the character pattern).

[1] An abstract machine that can be in exactly one of a finite number of states at any given time

2.4.5 Schema-Based Decoding

Schema-based Decoding enforces outputs to conform to a predefined data schema, such as **JSON**[1] or **XML**[2]. Unlike regex, schema-based decoding enforces structural validity (e.g., required fields, nested objects) and basic semantic checks (e.g., data types, value ranges). However, it does not guarantee domain-specific or contextual correctness (e.g., formula accuracy, logical consistency). The core mechanism involves parsing the provided schema and using it to constrain token selection during decoding. Similar to FSA-based methods, the decoder maintains a state, but this state represents the current position within the schema's structure (e.g., a string value, a closing brace). At each step, the schema dictates the next valid structural data types.

```
{
  "name": "Bob",
  "age": 25,
  "gender": "Male",
  "email": "bob@uni-koblenz.de",

  "address": {
    "street": "Universitaetsstrasse 1",
    "city": "Koblenz",
    "country": "Germany"
   }
}
```

Limitations: This schema ensures that "age" is a number but cannot check if 25 is plausible for the user. It may also mismatch a user's actual location (e.g., "city": "Koblenz", but the user lives in Japan). Handling highly complex schemas can be computationally demanding, and the strict enforcement conflicts with generating natural-sounding text.

[1] JavaScript Object Notation is a text-based format for representing structured data based on JavaScript object syntax

[2] Extensible Markup Language designed for storing, transmitting and reconstructing data

2.4.6 Grammar-Based Decoding

Grammar-based decoding enforces language model outputs to adhere to a formal grammar (e.g., **BNF**[1], **EBNF**,[2] **context-free grammars**[3]), ensuring syntactically valid sequences. The fundamental idea is to integrate a parser-like mechanism directly into the decoding loop. The decoder doesn't just predict the next token based on LLM probabilities; it predicts the next token constrained by what is grammatically permissible according to the specified grammar and the sequence generated so far. During decoding, the system maintains a state that reflects its current position within the grammar. This state involves tracking which production rules are currently being applied and which symbols (terminals or non-terminals) are expected next. At each step, the decoder determines the set of terminal symbols that can legally appear next according to the grammar rules applicable in the current state and selects the next token from the subset of grammatically valid tokens. The following illustrates an example grammar using **GBNF**[4]:

```
# Specifies chess moves as a list in algebraic notation, using PGN
    conventions

# Force first move to "1. ", then any 1-2 digit number after
root ::= "1. " move " " move "\n" ([1-9][0-9]? ". " move " " move "\n")+
move ::= (pawn | nonpawn | castle) [+#]?

# piece type, optional file/rank, optional capture, dest file & rank
nonpawn ::= [NBKQR] [a-h]? [1-8]? "x"? [a-h] [1-8]

# optional file & capture, dest file & rank, optional promotion
pawn ::= ([a-h] "x")? [a-h] [1-8] ("=" [NBKQR])?

castle  ::= "O-O" "-O"?
```

1. **Start:** The process starts with the ‘root‘ non-terminal symbol. The first rule for ‘root‘ dictates the sequence must begin with the literal terminal string ‘"1. "‘.

[1] Backus–Naur form is a notation system for defining the syntax of programming languages and other formal languages

[2] Extended Backus–Naur form is an extension of the basic BNF

[3] A set of rules that generate strings in a language

[4] GBNF is an extension of BNF that primarily adds a few modern regex-like features.[org23a]

2. **Generating Terminals:** The decoder forces the generation of the tokens corresponding to ‘1‘, ‘.‘, and ‘ ‘. Any other token suggested by the underlying LLM at these points would be masked out.

3. **Expanding Non-terminal ‘move‘:** The grammar now requires a ‘move‘. The decoder looks at the production rule for ‘move‘: ‘(pawn | nonpawn | castle) `[+\#]?`‘. It must choose one of the alternatives (‘pawn‘, ‘nonpawn‘, or ‘castle‘) to expand first.

4. **Iterative Continuation & Termination:** The decoder continues this process iteratively, expanding further non-terminals and enforcing required terminals according to the grammar's structure. At every step, it masks tokens disallowed by the grammar constraints, guiding the generation until a complete sequence satisfying the entire root rule is produced.

Limitations: Despite its ability to guarantee syntactically correct output and complex recursive structure handling, it not only requires utmost precision in creating the grammar but is also highly computationally expensive and therefore struggles with scalability.

2.4.7 The issues with modern LLM-constraining APIs

Most LLM APIs are black boxes that do not expose **logits**[1]. This prevents direct implementation of the ideal decoding methods described above. When using a grammar for instance, it is only available locally - the logic is local - but it can not directly guide the remote API's generation. This has multiple issues:

1. **Limited Scalability and Flexibility:** Local grammars make it impossible to adjust constraints at runtime. Complex grammars on top, can cause memory overhead and slow token masking. If the model deviates mid-generation, the grammar cannot backtrack, leading to invalid outputs.

2. **Unreliable Built-In Modes:** Built-In Modes like JSON-Mode offered by some LLM APIs provide convenience but often lack guarantee for formal validity. These

[1] refers to the raw, unnormalized output values produced by a classification model

modes usually employ heuristics or strong biases, which encourages the model to stick to a specific format but do not guarantee it, leading to potential syntax errors.

A common fallback mechanism used when an LLM fails to produce the desired output, is the retry loop. This involes repeatedly invoking the LLM, often with the same or slightly modified prompts incorporating error feedback, until a valid output is generated or the predefined maximum number of attempts is reached.

Generic Retry Loop:

```
import random

def validate():
    return random.random() < 0.3  # 30% success chance

def retry(max_attempts=3):
    for i in range(max_attempts):
        if validate():
            print(f"Success on try {i+1}")
            return "Valid output"
        print(f"Failed try {i+1}")
    print("All attempts failed")
    return None

retry()
```

The `validate` function represents a constraint checker with $P(\text{success}) = 0.3$, modeling typical failure rates when validating LLM outputs against complex schemas. The `retry` function implements a pattern where:

$$P(\text{failure after } n \text{ attempts}) = (1-p)^n = 0.7^n$$

For $n = 3$ attempts:

$$P(\text{failure}) = 0.7^3 = 0.343 \quad (34.3\%)$$

2.4.8 Conclusion

Constrained decoding methods address distinct challenges in guiding LLM outputs towards structured and reliable generation. While lexical constraints and regex decoding excel in enforcing simple patterns(e.g, inclusion, date formats), their local validity checks fail to guarantee coherence. FSA-based methods improve on this by tracking state transitions. Schema-based decoding further elevates reliability by validating hierarchical data structures, ensuring type consistency but struggles with domain-specific knowledge. Grammar-based decoding, enforces syntactic correctness for complex outputs, but suffers high computational costs and demands meticulous grammar design. These methods trade flexibility for control with simpler methods (lexical/regex) prioritizing efficiency and complex ones (schema/grammar) favoring precision. Despite that, all of them face limitations in semantic validation and computational scalability.

Chapter 3

Methodology and Results

After establishing the necessary background in the previous chapter, the following sections detail the empirical methodology and present the core findings of our investigation into grammar-enforced Chain of Thought reasoning. We begin by providing a comprehensive overview of the experimental design employed. Subsequently, we elaborate on the development and structure of the novel financial benchmark created specifically for this study, including its objectives, difficulty categorization, and generation process. We then describe the construction of the GBNF grammars used to constrain the LLM outputs. Finally, we present the detailed experimental results derived from applying these methods, comparing the performance across various benchmarks and analysing the quantitative outcomes.

3.1 Method Overview SB

The primary objective of this internship is to find out if formal grammars may enhance LLMs reasoning abilities by constraining the CoT reasoning process. Our method combines benchmarking that evaluates model performance on challenging tasks combining multi-step logic, mathematics and domain-specific financial reasoning with grammar constrained CoT generating techniques. Throughout all trials, the Microsoft wizardlm-2-7b-imat-Q6K model[1] serves as the primary language model. It is accessed programmatically using the llama-cpp-python library[2] and runs locally via the llama.cpp backend.

[1]Model Card of the used LLM: huggingface.co/qwp4w3hyb/Not-WizardLM-2-7B-iMat-GGUF
[2]llama-cpp-python Repository: github.com/abetlen/llama-cpp-python

We use a two-phase benchmarking process in our internship, which uses a comparative experimental design. In the first stage, we use normal free-form CoT prompts to run the model over a variety of benchmarks. In the second stage, we rerun the same benchmarks and apply grammar based constraints to the CoT generation, which are written using the Grammar-Based Notation Format. The purpose of these constraints is to direct the model toward syntactically sound and semantically structured reasoning paths during the decoding process.

We assess the model's performance against four different benchmarks in order to determine the effectiveness of this approach: The focus of MMLU (math)[1] is multi-step numerical reasoning. Deductive reasoning and symbolic logic are covered in MMLU (logic)[2]. In order to assess reasoning in applicable financial contexts, a custom financial benchmark was created by us. Without requiring memory of training material, the Zebra Puzzle benchmark is a synthetic benchmark made using a logic puzzle generator to assess symbolic and deductive thinking.

To enforce a structured output format in the constrained setting, we write a GBNF grammar that is used throughout the decoding stage. In order to ensure syntactic and semantic consistency, the language is customized to meet the needs of each benchmark, such as providing structured explanations for puzzles or a single letter output for multiple-choice questions. Every benchmark is run twice: once with and once without grammar constraints. The correctness, consistency of the reasoning and interpretability of the outcomes are then compared. By using this technique, we can better understand the usefulness of structured reasoning for LLMs and analyze the effect of grammar constrained CoT on model performance.

3.2 Development of the Financial Benchmark SB

Large language models, including OpenAI's GPT series and Meta's Llama models, have proven to be exceptionally proficient in a variety of cognitive tasks in the field of natural language processing. Domain-specific reasoning is still difficult, tough, particularly in

[1]Source of MMLU(math): huggingface.co/datasets/cais/mmlu/blob/main/high_school_mathematics/test-00000-of-00001.parquet

[2]Source of MMLU (logic): huggingface.co/datasets/cais/mmlu/blob/main/formal_logic/test-00000-of-00001.parquet

the financial industry. Financial analysis clearly lacks defined benchmarks, but general-purpose benchmarks such as MMLU (Massive Multitask Language Understanding) assess a model's comprehension across fields like physics, history and law [ben24] . By developing a systematic, multi-level financial benchmark appropriate for evaluating an LLM's capacity to comprehend, rationalize and calculate financial problems, this benchmark seeks to close that gap. Since it was created with llama-cpp compatibility in mind, it is portable, effective and used on-device inference applications.

3.2.1 Objective

Creating a layered benchmark dataset with 1,000 questions that reflects the complexity and diversity of real-world financial reasoning is the primary objective. Particular objectives consist of:

- **Content Validity**: The benchmark aims to encompass financial concepts (such as corporate finance, time value of money and investment analysis) that are both realistically used in the real world and acknowledged by academics. Making sure of this dual alignment improves the benchmark's suitability for review at the industry and educational levels.

- **Scalability**: Set up the standard for ongoing growth across subjects and levels of difficulty. Future-proofing was a consideration in the creation of the benchmark. Without altering the basic schema or assessment process, it will facilitate the inclusion of additional question varieties, financial domains and higher-order reasoning abilities.

- **Alignment with MMLU:**: The benchmark can be used in alongside MMLU to evaluate if LLMs are concurrently improving their general and domain-specific knowledge by embracing the framework and ideas of MMLU, which include a fixed multiple-choice format, tiered difficulty and question banks that are domain-focused.

- **LLM Testing**: Lightweight, local inference benchmarking is possible with this benchmark, particularly in edge contexts or research setups where full cloud APIs

are restricted. Due to this, the benchmark is perfect for testing on both high-end models and deployments with limited resources.

The benchmark also seeks to monitor advancement over time and assist researchers in comprehending the limitations of LLMs in quantitative thinking.

3.2.2 Tools and Technologies Used

A variety of methods for question creation, data storage, validation, and scheduled evaluation were used in the creation of the financial benchmark. In order to ensure scalability, consistency, and alignment with best practices in language model benchmarking, each tool was chosen to address a particular component.

SQLite

Due to its ease of use, portability, and zero configuration setup, SQLite[1] was chosen as the primary database management system. Local development was made possible by its self-contained architecture, which eliminated the need for server-based databases. Every multiple-choice question (MCQ) was recorded in a normalized schema during the benchmark generation process, which included the question text, answer choices, correct answer and assigned difficulty level. This framework made manipulation and querying simple. For instance, SQLite made it possible to filter queries based on difficulty.

ChatGPT (GPT-4)

GPT-4[2] played a key role in the benchmark dataset's beginning and continuous development. Structured templates and thoughtfully crafted prompts were used to stimulate it, mirroring the structure and level of reasoning of MMLU-style multiple-choice questions. For example, prompts would identify the difficulty level and domain scope (e.g., intermediate-level corporate finance), request distractor[3] options and indicate the number of answer possibilities. Several iterations were employed to guarantee the diversity,

[1] SQLite Homepage: sqlite.org

[2] ChatGPT-4 Homepage: openai.com/index/gpt-4/

[3] commonly used in multiple-choice question (MCQ) terminology to refer to the incorrect answer choices

originality and covering of financial subjects. Prompts like "avoid repetition" "include quantitative reasoning in higher levels" and "maintain consistent style and answer formats" were used to continuously steer the model's answers. The benchmark was developed while guaranteeing conformity with academic norms and practical financial frameworks by utilizing ChatGPT's ability to combine financial knowledge with numerical reasoning.

Gemini (2.0) and DeepSeek(R1)

Each question was independently checked using Gemini[1] and DeepSeek[2], two high-performing large language models renowned for their mathematical reasoning and factual foundation, to guarantee the generated questions' accuracy, integrity and soundness. These models were used to audit and cross-check the content that ChatGPT had created, not to create new content. For instance, Gemini had to solve a compound interest problem on their own to make sure the right solution matched and confirm that the given difficulty level was adequate and to reevaluate the cognitive burden of the questions. DeepSeek was again used to verify the same a second time. As a quality filter, our dual-model verification method found ambiguities, inconsistent answers or incorrectly classified queries. Questions were either changed or eliminated when there were different outputs.

3.2.3 Benchmark Structure - Table Creation

A specific SQLite table called benchmark was created in order to systematically organize and manage the financial benchmark dataset. All multiple-choice questions are stored in this table in a query-friendly, normalized format that allows for future extension, evaluation and filtering. The schema is defined as follows:

```
CREATE TABLE IF NOT EXISTS benchmark (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    question TEXT NOT NULL,
    option_a TEXT NOT NULL,
    option_b TEXT NOT NULL,
```

[1]Gemini 2.0 Flash Homepage: aistudio.google.com/prompts/new_chat?model=gemini-2.0-flash-exp
[2]Deepseek R1 Homepage: www.deepseek.com/

```
    option_c TEXT NOT NULL,
    option_d TEXT NOT NULL,
    option_e TEXT NOT NULL,
    correct_answer TEXT NOT NULL,
    difficulty INTEGER CHECK(difficulty BETWEEN 1 AND 5)
);
```

- **Id**: Every question entry has a unique, auto-incrementing identity. This serves as the primary key and ensures that every question can be added, changed or deleted individually.
- **Question**: The main question prompt is contained in a TEXT field . This serves as the foundation of the benchmark item and usually addresses a financial idea or issue. It serves as the basis for evaluation and is always required.
- **Answer Options**: The model's multiple-choice response options are represented by these five TEXT fields, option_a to option_e. Every option is intended to be both collectively exhaustive and mutually exclusive. By keeping each question at five options, the benchmark structure is standardized and reflects MMLU standards.
- **Correct Answer**: There is only one correct answer which is mapped to the appropriate choice and recorded as a letter (A–E) in the correct_answer field.
- **Difficulty Level of the Question**: An associated difficulty level, the question's cognitive and conceptual complexity was indicated by a numeric difficulty field that ranged from 1 (Basic) to 5 (Expert).The CHECK constraint makes sure that only valid difficulty levels are entered, which ensures data integrity.

Design Considerations

Considering a focus on data quality, portability and query simplicity, the table is intended to be simple but thorough. In addition to improving compliance with SQL operations, storing options as distinct columns rather than in a JSON or array format makes batch

filtering or validation easier. The structure is best suited for scripts that collect performance indicators across difficulty levels, analyze model responses and iterate over queries. In addition to offering a strong framework for benchmark execution and model evaluation, this structure makes it easier to add explanations and justifications for each question, categorize subjects and associate metadata, among other future improvements.

3.2.4 Difficulty Level Categorization

Our financial benchmark is organized over **five increasing levels of complexity** to methodically evaluate a language model's financial reasoning abilities. Every level has been thoughtfully designed to represent the transition from fundamental financial knowledge to sophisticated theoretical mastery and decision-making. This tiered structure enables us to assess conceptual knowledge, contextual awareness, multi-step reasoning in addition to arithmetic accuracy. This structure guarantees comprehensive content and practical relevance by conforming to professional standards and real-world financial education. This kind of design enables a meaningful comparison between model outputs and human-level thinking in a variety of financial tasks,

Level 1 - Basic (Fundamental Knowledge)

Foundational financial literacy is evaluated at this level. At this point, the questions address fundamental ideas including what a bond is, different kinds of interest and simple mathematical operations that are frequently used in finance. The objective is to determine if a model has the bare minimum of knowledge needed to interact with financial jargon and carry out simple computations. These questions are similar to those that could be found in a beginning course on personal finance.

Level 2 - Intermediate (Standard Academic-Level Questions)

Concepts that are generally addressed in undergraduate business or finance courses are covered at Level 2. Compound interest, time value of money, break-even analysis and standard financial statistics like the current ratio and return on equity are among the subjects covered. Despite being based on very simple scenarios, these questions demand

that the model use formulas and carry out multi-step computations. Without undue abstraction or theoretical depth, the emphasis is on evaluating a model's capacity to apply financial instruments in real-world, everyday business scenarios.

Level 3 - Hard (Advanced Financial Modeling)

At this level, the benchmark evaluates a model's capacity to solve more complex problems and decipher multi-step financial scenarios. Among other things, questions include weighted average cost of capital (WACC), internal rate of return (IRR) and net present value (NPV). These subjects require both mathematical accuracy and the capacity to choose the appropriate formulas or procedures depending on the situation. The model's depth of comprehension is called into question by the addition of distractions and believable substitutes. This level mimics the challenges that an MBA student or mid-level financial analyst could encounter when assessing investment possibilities.

Level 4 - Very Hard (Applied Real-World Finance)

This level focuses on complex, practical financial applications that call for applied knowledge and contextual thinking. Interpreting the financial effects of corporate actions, valuing companies using discounted cash flows, examining derivatives, or reacting to shifting market conditions are a few examples of such questions. In contrast to Levels 1–3, the solutions include more than just math, they also call for analyzing trade-offs and financial ramifications. This level is similar to professional decision-making in fields where several factors need to be taken into account at once, such as corporate finance, investment banking or financial consulting.

Level 5 - Expert (Financial Theory, Multi-step Reasoning)

The most challenging level is level 5, which covers multi-variable choice problems, complex economic modeling, and advanced theoretical finance. Among the subjects covered are risk modeling with Monte Carlo simulations, forecasting with econometrics, portfolio optimization with Modern Portfolio Theory and more complex subjects from the CFA curriculum including the efficient market hypothesis and arbitrage pricing theory.

These questions assess the model's capacity to apply sophisticated procedures, synthesize abstract concepts and retain logical consistency throughout lengthy reasoning chains in addition to financial literacy and computation abilities. It is intended to mimic the situations that a CFA candidate or a quant analyst could face.

3.2.5 Balanced Distribution Across Difficulty Levels

The benchmark was created with precisely 200 questions for each of the five difficulty levels, for a total of 1,000 distinct multiple-choice questions, in order to guarantee a thorough and fair assessment of a model's financial reasoning ability. There are several uses for this even distribution as follows:

Balanced Design for Fair Evaluation

The benchmark prevents performance measures from being skewed toward any one level by keeping the count constant across all five difficulty levels. This makes it possible to evaluate the model's advantages and disadvantages across various conceptual and cognitive levels with greater accuracy.

Robustness and Generalization Testing

The benchmark is more robust when there are more questions at each level. It reduces the possibility that models would overfit to specific question types or take advantage of pattern memorization. A wide range of subjects, equations, and financial scenarios are covered at each level.

Granular Performance Analysis

Calculating performance indicators like accuracy, precision, or error rates at each tier becomes statistically significant when there are 200 questions per level. This helps with fine-grained diagnostics, for instance, a model may do well at Level 2 but not at Level 4, indicating a lack of contextual knowledge or applied reasoning.

Facilitates Curriculum-Like Evaluation

The benchmark is appropriate in both research and instructional settings since its structured collection of 200 questions per level closely resembles the format of common academic or certification exams (such as CFA or business school).

The benchmark enables a well-rounded, controlled, and scalable platform for assessing LLMs on financial reasoning in an organized and understandable way by selecting an equal number of questions from each level.

3.2.6 Question Generation Process

Creating a high-quality financial benchmark is largely dependent on the question generation process. This required building a robust process in addition to utilizing big language models to guarantee the dataset's authenticity, consistency and coverage. Question Format, iterative generation and thorough evaluation and filtering were the three main steps of the method.

Question Format

A prompt engineered to create questions in the same format as the financial benchmark table was used. We initially explained to GPT about our requirements in steps. First we explained that we need to create a financial benchmark with 1,000 questions in SQLite with the format similar to the financial benchmark table creation script. Then we explained the difficulty level tiers and prompted it to generate the questions. The main elements of the question format are:

- **Question**: A clear straightforward question served as the primary topic or concept under test. It could be quantitative (for example, "What is the future value of a $1,000 investment over three years at 5%?") or theoretical (for example, "What does ROI measure?").

- **Answer Options**: There are precisely five possible answers designed to be complete as a whole and mutually exclusive. Options were designed to steer clear of obvious outliers or giveaways.

- **Correct Answer**: There is only one correct answer which is mapped to the appropriate choice and recorded as a letter (A–E) in the correct_answer field.

- **Difficulty Level of the Question**: An associated difficulty level, the question's cognitive and conceptual complexity was indicated by a numeric difficulty field that ranged from 1 (Basic) to 5 (Expert).

Example SQL Format: INSERT INTO benchmark (question, option_a, option_b, option_c, option_d, option_e, correct_answer, difficulty) VALUES ('What is the formula for simple interest?', 'P × r × t', 'P + r × t', 'P × (1 + r × t)', 'P ÷ r ÷ t', 'P + r + t', 1, 1)

Iterative Generation

GPT cannot handle generating 1000 questions at one shot. The maximum number of questions it can sometimes generate is 30, not more. Since the context window is not big, we prompted GPT to generate questions in batches with each batch containing at least 10 question. This process was carried out in clearly defined iterative batches, with each iteration concentrating on:

- **Targeted Difficulty Levels**: Prompts were designed to generate questions at particular levels of complexity, such as "Generate 20 Level 3 questions on corporate finance involving WACC, NPV, or IRR." Cognitive requirements varied by level.

- **Diversity and Advancement of Topics**: Coverage, not simply volume, was the goal. Different levels included topics such as capital structure, time value of money, portfolio theory, risk management, and derivatives.

- **Contextual Uniqueness**: The prompts were thoughtfully written to minimize conceptual and syntactic overlap. Tracking features and chat history made sure that inquiries that were too identical or duplicated were avoided.

- **Controlled Batch Size**: Usually, 10–30 questions were generated at once even if our prompt requested to generate 50 or 100 questions. Sometimes the on prompting generation of 30 questions, GPT would just generate 12 or 15 questions. Better control and an instant quality check prior to scaling up were made possible by smaller batch sizes.

To keep the feedback loop with the LLM under control, each question was manually examined while it was being generated.

Filtering and Manual Review

Even though ChatGPT was quite good at generating MMLU-style questions, each batch was manually validated to ensure benchmark integrity. Through this rigorous filtering procedure, every question was guaranteed to make a significant contribution to the benchmark and be in line with its evaluation goals. The following were part of the filtering process:

- **Elimination of Ambiguity**: Questions that had more than one valid answer, ambiguous wording or content that was biased were either reworded, eliminated or asked to regenerate those questions without ambiguity.

- **Duplication Control**: GPT generated a lot of duplicates. Especially questions which are formula based were repeated in every batch in spite of mentioning in the prompt not to duplicate any question or to not repeat previously generated questions. Manual checks helped to catch most of the duplicates as soon as they were generated If any question was repeated, we prompted GPT to generate a new question which is previously not generated and verifying if this question already exists.

3.2.7 Cross-Verification with Gemini and Deepseek

Once the 1,000 financial questions are generated by GPT, these questions are now cross verified with Gemini(2.0) and Deepseek(R1). We opened Gemini and the R1 model in 2 different tabs and verify the questions parallely. Both these models cannot verify all

questions at once. So uploading an excel or a file with all data and prompting to verify them is useless. The approach followed by us was to verify only 20 questions at a time, not more in both the models. Although this was a painstakingly time consuming process, it is effective when you encounter ambiguities or duplicates as you are working with small batches and it is easier to manually make changes.

Handling Duplicates

Questions which were identified as duplicates were removed from the database and we asked GPT to generate a different question of the same difficulty level. Then verified if this question was already present in the database or not. If it was already present, we prompted GPT to "generate a new question, the previously question generated is a duplicate(already present in the DB)". This new question is again verified with Gemini and R1. If if the answer and difficulty levels match we added this question to the database.

Handling Ambiguity

Deepseek was good at identifying ambiguous questions compared to the other two models. In case of an ambiguous question, we tried to find out which part of the question gives rise to ambiguity. If something needs to be added or removed to eliminate the ambiguity, we added or removed it and rephrased the question. In some cases questions with ambiguity even after eliminating and rephrasing, were discarded from the database and a new unambiguous question added.

Handling Incorrect answers

Questions which had incorrect answer in one of the model were prompted to recheck the answer. In case both Gemini and R1 identified the answer to a question as incorrect, we opened a new chat and prompted GPT to solve the question without providing any options as "what is the answer to the following question with its reasoning.". If the answer is similar to the other two models, the correct answer was added to options. If GPT had a different answer, we discarded those questions and generated new ones.

Handling Different Difficulty Levels

The difficulty level generated by GPT for most of the questions is similar with the other two models. In cases where one model gives a different level, we prompted that model to check if assigning the GPT difficulty is still something it would agree to. If the model does, we keep the same difficulty level, else we prompt all the 3 models to assign a difficulty from 1 to 5("Solve the following question and assign a difficulty level between 1-5 where 1(Basic), 2(Intermediate), 3(Hard), 4(Very Hard) and 5(Expert) with respect to financial knowledge") and select the difficulty similar in 2 models and also cross verified this difficulty assignment with a 4th model, Copilot. If all 3 models suggest different difficulty levels, we discarded those questions and generated new ones.

3.2.8 Issues Encountered

Even with the use of cutting-edge LLMs such as DeepSeek , Gemini and GPT-4, a number of limitations were noted throughout the benchmark development process. Due to these constraints, a multi-model cross-verification approach, batch-level quality checks and meticulous human control were required.

GPT-4

A lot of issues or limitations with respect to GPT were noted. One common issue with GPT was that it generated duplicate questions even after specifically adding in the prompt to "Please make sure to not to generate any duplicate questions or questions previously generated" it still generated duplicate questions. In some cases, In spite of specifically pointing out to GPT in the prompt that this question is generated multiple times and not to generate this specific question again, it would repeat this question after 2 or 3 batches. These questions had to be manually removed and new questions had to be generated.Sometimes, particularly in situations involving digits or formulas, GPT would confidently produce wrong responses that seemed to make sense. The boundaries of accounting, finance and economics were occasionally blurred by GPT, producing questions

that were either outside of its scope or combined several other fields. In Level 3+ questions, where domain clarity is crucial, this was particularly problematic.

DeepSeek-R1

This model had the least issues compared to the other two models. This model was the best at identifying ambiguity. One major issue faced with this model was verifying the questions in batches of 20. Sometimes when we provided 30 or more questions, most of the times the model would skip 2 or 3 questions in between but provide the answers with the serial number 1- 30 but all answers would be wrong as a few questions were skipped. We had to go to the CoT it generated in the Thinking area and verify which question is what.

Gemini-2.0

The issue with this model is that on prompting it to solve a specific question, it would generate the final answer without any proper reasoning. Every single time a question is prompted to be solved we had to explicitly mention in the prompt to provide reasoning and to explain each calculation step by step. This is the model where most of the instructions had to be repeated multiple times or the model simply forgets what it was doing or what it needs to do. It also has the issues similar to the other models have.

A multilayer human-in-the-loop procedure was required since no one model was consistently reliable across all dimensions—logical reasoning, numerical accuracy, quality and ambiguity detection. In addition to manual final tests, cross-verification between Gemini and DeepSeek guaranteed sound logic, numerical accuracy, clarity of language and the difficulty level appropriateness. In the end, this multi-model, human-guided method improved the financial benchmark's quality and reliability.

3.2.9 Final Database Compilation and Integration

The final phase was to compile, insert and validate the dataset in the SQLite database following the iterative creation and thorough cross-verification of all 1,000 financial ques-

tions. In doing this, the benchmark was guaranteed to be complete, queryable and ready for incorporation into the codebase for workflows including inference and evaluation.

Collecting the Final Set of Questions

We created a pool of precisely 1,000 distinct and high-quality financial multiple-choice questions after generating several batches of questions using GPT-4 and then verifying them using Gemini and DeepSeek. Every question met the validation checkpoints of uniqueness (no literal or semantic repetitions), accuracy (confirmed answers and distractions), unambiguous and straightforward language and appropriate placement within one of the five levels of difficulty and followed the established schema.

Preparing SQL INSERT Queries

Once finalized, the completed dataset was converted into SQL-compliant INSERT INTO commands that followed the benchmark table's structure. An INSERT query with the following format was created for the questions in each batch of question generation by GPT.All the 1000 questions were prepared in a script called insert_data.sql and added to our repository[1].

```
Example: INSERT INTO benchmark (question, option\_a, option\_b, option\_c,
option\_d, option\_e, correct\_answer, difficulty) VALUES ('What is the
future value of a $1,000 investment at 5% interest for 3 years?',
'$1,157.63', '$1,100.00', '$1,150.00', '$1,200.00', '$1,125.50', 'A', 2);
```

Inserting into SQLite Database

First we initializ the database and execute the table creation query in the in SQLite DB. Then we execute the insert_data.sql script to add the 1000 rows of benchmark questions to the database. This created a clean, version-controlled financial_benchmark.db file containing the full benchmark which was added to the repository.

[1]Our Project Repository: gitlab.uni-koblenz.de/hillesheim/ai-equity-internship

Database Verification

We performed one last set of integrity checks on the database following insertion which ensured the database was stable, accurate and ready to be used.:

- Made sure there were precisely 1,000 records by running running the script: SELECT COUNT(*) FROM benchmark;
- Checked for even distribution by running queries across difficulty levels to make sure each difficulty level had 200 questions each.
- To verify accurate insertions, test searches for known questions were carried out.
- Schema constraints that have been validated (e.g., no NULL values, valid difficulty range)

3.2.10 Conclusion

By the end of this process, the benchmark was ready for analysis, experimentation and expansion in the future. It was generated and validated with high quality and effortlessly integrated with our LLM evaluation framework. The development of this financial benchmark is a major milestone in the process of methodically assessing LLMs' aptitude for quantitative problem-solving and financial reasoning. This benchmark, in contrast to general-purpose benchmarks, is designed to capture the complex cognitive demands of real-world financial operations, ranging from basic ideas to intricate practical scenarios.

In addition to producing 1,000 excellent multiple-choice questions, a significant accomplishment was the development of a clean, portable SQLite database that facilitates simple querying, inference testing and integration with programs such as llama-cpp (financial_benchmark.db). Manual review played a crucial role in order to maintain consistency in difficulty and improve the quality of the questions.

This benchmark is a fundamental instrument for evaluating LLM performance in contexts specific to finance. It facilitates quantitative reasoning, contextual decision-making and layered evaluation. Future versions of the benchmark could include case-based questions, time-based simulations or open-ended prompts. In the end, this research opens the

door for more focused, practical AI evaluation in financial domains and provides a strong, scalable foundation for finance-specific LLM benchmarking.

3.3 Grammar Construction JNS

GBNF (GGML BNF) is a specialized grammar specification language developed within the llama.cpp framework to enforce structured, rule based constraints on the outputs of LLMs. Derived from Backus-Naur Form (BNF), GBNF extends traditional notations with features tailored for constrained decoding, such as regex-like featured, token-level constraints and prioritized rules. GBNF grammars are applied during token generation, dynamically pruning invalid tokens from the model's probability distribution, thereby ensuring the output adheres to specified syntactic grammar structure. For example, they can be used in a way, that forces a generated JSON object to always contain properly nested braces, valid key-value pairs and type-consistent values without having to rely on error-prone post-processing. For Chain-of-Thought (COT) reasoning, GBNF enables a more precise control over the logical flow of generated output. Through the use of a step-by-step thinking template, it guarantees that outputs follow a coherent, human-readable structure while minimizing hallucinations or syntactic errors. Not only does this approach improves output reliability but also enhances interpretability by aligning the model's generation process with human-readable conventions. [org23b]

The effectiveness of grammar-based decoding depends critically on the quality and suitability of the grammar itself, which defines the precise syntactic structure the LLM must adhere to. For this paper we have created and utilized three different approaches:

1. **GBNF M*:** prioritize detailed, explicit, multi-step reasoning (CoT) before revision.

2. **GBNF N*:** favors simpler, immediate thinking steps followed by a forced revision trigger.

3. **GBNF O*:** implement an internal best-of-n selection by generating multiple diverse options before a forced revision trigger.

Six different M* grammars, four different N* grammars and one O* grammar were created. For the benchmarks, GBNF M2, GBNF M5, GBNF N3 and GBNF O1 were used, since they seemed to perform the best out of all of them.

The first approach we designed (**GBNF M***) is structured around a more detailed Chain-of-Thought process featuring distinct phases and an opportunity for self-correction.

In GBNF M2, we designed three stages:

```
root ::= problem-definition steps final-conclusion
```

In the first stage (**problem-definition**), the LLM must first articulate its understanding of the problem:

```
problem-definition ::= (
"=== Understanding the Problem ===\n"
"First, I need to clearly understand the task.
My interpretation is (in one sentence): " sentence "\n"
# Optionally, the LLM can state an initial high-level plan.
("My initial high-level plan is (in one sentence): " sentence "\n")?
"\n")
```

In stage 2 we are initilizing the **Step-by-Step execution process**:

```
# --- Stage 2: Step-by-Step Execution ---
# This requires the LLM to perform at least two and at most five
    explicit steps.
steps ::= step{1,3} step-revision step{1,2} last-step

# Defines the structure of a single problem-solving step.
step ::= (
"--- Step ---\n"
# 1. State the intended action or line of reasoning for this step.
"Next Action/Reasoning (in one sentence): " sentence "\n"
# 2. State the outcome or result of this specific step.
"Outcome/Result of this step (in one sentence): " sentence "\n\n" )

# Defines the structure of the last problem-solving step.
last-step ::= (
"--- Last Step ---\n"
# 1. State the intended action or line of reasoning for this step.
"The last Action/Reasoning before I give an answer (in one sentence): "
    sentence "\n"
# 2. State the outcome or result of this specific step.
```

```
"Outcome/Result of this step (in one sentence): " sentence "\n\n" )
```

The LLM differentiates between a thinking step and the **last-step**. It is enforced to think for one to three sentences, followed by one to two **step-revision steps**. Afterwards, it concludes its reasoning in the **last-step**. The **self-revision** block is an additional step within a step. It encourages self-correction with specific prompting:

```
# This rule defines the optional self-revision within a step.
# It forces the LLM to use one of the revision keywords,
followed by its revised thought.
step-revision ::=
revision-prompt sentence "\n\n"

# A selection of prompts to encourage self-correction or deeper thought.
revision-prompt ::= ("--- Revision ---\n"
( "Wait, let me double-check that assumption (in one sentence):"
| "Hold on, is that the most logical next step? Reconsidering (in one
    sentence): "
| "Correction: I should actually consider this first (in one sentence):
    "
| "Thinking more carefully, perhaps it's better to (in one sentence): "
| "Let me pause and verify that calculation/statement (in one sentence):
     "
))
```

Based on the results of the Step-by-Step execution phase, the LLM gives its **final conclusion** in a single sentence.

```
# --- Stage 3: Final Conclusion ---
# After completing the steps, the LLM must state the final answer or
    conclusion.
final-conclusion ::= (
"=== Final Conclusion ===\n"
"Based on the step-by-step process above,
the final answer is (in one sentence): " sentence "\n")
```

We constrained the LLM by using the rule

```
sentence ::= [^?!;\n\r]{10,100} [.?!]
```

which forces generated sentences to be between 10 and 100 characters long excluding sentence terminators like ‘?‘, ‘!‘, ‘;‘ or newlines and instructs that they end with exactly one period, question mark or exclamation mark.

In comparison to GBNF2, GBNF M5 enforces this specific sequence: **initial-phase, reasoning-sequence, synthesis and final-answer**.

```
root ::= initial-plan reasoning-sequence synthesis final-answer
```

The grammar first requires an **initial-plan**. This enforces the LLM to think about the objective for the given prompt in a single paragraph, priming the LLM to have a clear plan before executing the **reasoning-sequence**. During that phase, the LLM is enforced to think about the problem and its solution for 5 paragraphs.

```
reasoning-sequence ::= reasoning-step-1 reasoning-step{1,5}
correction-block reasoning-step{0,2} last-reasoning-step
reasoning-step-1 ::= "Step 1 (one paragraph followed by a newline):
" paragraph "\n"
reasoning-step ::= "Step " number " (one paragraph followed by a newline
    ):
" paragraph "\n"
last-reasoning-step ::= "Last step before giving the final answer
(one paragraph followed by a newline): " paragraph "\n\n"
```

After the reasoning the grammar requires a synthesis section. This must start with the exact text **"Synthesis: Combining the results shows that ..."** followed by a single paragraph summarizing the outcomes derived during the reasoning-phase.

```
# 3. Synthesis Phase
synthesis ::= "Synthesis: Combining the results shows that
(one paragraph followed by a newline): " paragraph "\n\n"

# 4. Final Answer Phase - Including an example length constraint
    instruction
final-answer ::= "Final Answer (one paragraph followed by a newline):"
    paragraph "\n"
```

Afterwards the LLM gives it **final answer** as:

```
final-answer ::= "Final Answer (one paragraph followed by a newline):"
    paragraph "\n"
```

Throughout the grammar we used a **correction-block** that the LLM can utilize to make corrections based on previous potential mistakes made during the reasoning phase.

```
# Correction block, triggered by a marker
correction-block ::= "\n" correction-marker " " revise-thought
correction-marker ::= "Wait," | "Hold on," | "Actually," | "Let me re-
    evaluate,"
revise-thought ::= paragraph "\n"
```

Throughout this grammar, the basic **paragraph** element is defined as:

```
paragraph ::= [^\n]+
```

It allows any sequence of characters that does not contain a newline.

GBNF M2 enforces a rigid, hierarchical structure with explicit stages, requiring single-sentence outputs for each step and a predefined mid-process self-revision block using specific correction prompts. In contrast, GBNF M5 adopts a more flexible paragraph-based approach, prioritizing longer-form analysis over step constraints and allowing dynamic self-correction through open-ended revision triggers. While GBNF M2 targets structured problem-solving with fixed-step sequences, GBNF M5 supports open-ended reasoning tasks requiring adaptive synthesis.

The second approach we designed (**GBNF N***) is structured around simplicity. Our GBNF N3 is breaking the reasoning process down into 4 components: **user-intent**, **execution**, **self-revision** and **result**.

```
root ::= user-intent "\n" execution "\n" self-revision "\n" result
```

The LLM is instructed to state the user's intent in one paragraph in order to get a better understanding of what issue the user wants to have solved.

```
user-intent ::= "Analyzing the user's intent (in one paragraph):\n"
    paragraph
```

After understanding the main objective, the LLM is asked to think step-by-step and execute each step immediately. At least three and at most ten paragraphs are allowed.

```
execution ::= "Let's think step by step and execute each step
    immediately:
\n" paragraph{3,10}
```

Afterwards, the LLM has to do **self-revision**. Like in the execution-step, there are at least three and at most ten paragraphs allowed. This revision-step cannot be skipped!

```
self-revision ::= "Wait, " paragraph{3,10}
```

The last component concludes the LLM **result**.

```
result ::= "The final result (in one paragraph):\n" paragraph{1,3}
```

This GBNF grammar enforces a single mandatory self-revision after all execution steps, sturctured as a linear sequence (user-intent -> execution -> self-revision -> result) with paragraph-length constraints.

```
paragraph ::= [^\n]+ "\n"
```

The last approach we designed (**GBNF O***) enforces multi-solution generation with comparative evaluation, requiring the LLM to propose three distinct step-by-step solutions (s1, s2, s3), selecting the best solution (**selection**, and a mandatory **self-revision** phase before giving a constrained multiple-choice answer (A-E).

```
root ::= user-intent "\n" s1 "\n" s2 "\n" s3 "\n" selection "\n"
         self-revision "\n" result
```

The user-intent section instructs the LLM to analyze the user's intent

```
user-intent ::= "Analyzing the user's intent (in one paragraph):\n"
    paragraph
```

The grammar's **s1, s2 and s3 sections** require the LLM to provide three completely different approaches to the given problem, which forces the consideration of multiple perspectives, that can help mitigate bias or errors in a single approach.

```
s1 ::= "Proposing a first step-by-step solution:\n" paragraph{3,10}
s2 ::= "Proposing a completely different second step-by-step solution:
        \n" paragraph{3,10}
s3 ::= "Proposing a third step-by-step solution,
        that is completely different from the first two:\n" paragraph
    {3,10}
```

The best solution of the three is used by comparing them to each other.

```
selection ::= "The best of the three solution is " paragraph
```

A **self-revision** section afterwards ensures robustness and critically evaluates potential oversights in the selected solution before finalizing the answer.

```
self-revision ::= "But wait, " paragraph{3,10}
result ::= "The best fitting answer /
option after thinking about the problem is (A/B/C/D/E): " ("A"|"B"|"C"|"
    D"|"E")
```

Like in the previous grammar's, a paragraph is defined as:

```
paragraph ::= [^\n]+ "\n"
```

The latest grammar **GBNF O1** diverges fundamentally from the other designs by instructing explicit generation and comparison of three distinct solutions (s1, s2, s3),

each required to be "completely different" in perspective. Unlike **GBNF M2**, which enforces a single linear reasoning path, this approach prioritizes diversity, forcing the LLM to propose alternative strategies before selecting and refining one. It adds layers of complexity compared to the linear and free-text approach. Unlike all prior grammars, **GBNF O1** enforces a specific output (A-E), mimicking exam/quiz formats, whereas **GBNF M2** and **GBNF M5** allow open-ended conclusions.

3.4 Software Architecture and Implementation JH

The software architecture developed during this internship enables grammar-constrained decoding for LLMs in both interactive and benchmarking contexts. The Python[1] project builds upon the `llama-cpp-python` library[2], which provides efficient access to local LLMs via the `GGUF` format, and integrates grammar enforcement through a formal grammar specification.

At the core of the system lies the `LanguageModel` abstraction, which serves as the central interface to all model-related functionality. This class is responsible for initializing the LLM, applying an optional grammar during decoding, and handling the logic for generating both streamed and full responses. A significant design decision was to decouple the generation of a full reasoning process from the selection of the final answer. This is achieved through a two-stage prompting approach: first, the model is queried to produce a free-form explanation or chain-of-thought; then, the final answer is extracted using a secondary grammar-constrained prompt that enforces a strict output format (e.g., a single letter representing a multiple-choice selection):

```
root ::= "Final answer (A/B/C/D/E): " ("A"|"B"|"C"|"D"|"E")
```

This separation allows the model's reasoning capabilities to be evaluated independently of its answer formatting behavior. A second class, `GeminiLanguageModel`, was also implemented to expose the same interface while interacting with the Google Gemini API[3] instead of a local LLM. This allows for seamless substitution between local and

[1] Python Homepage: www.python.org
[2] Repository of the library: github.com/abetlen/llama-cpp-python
[3] Google Gemini API Reference: ai.google.dev/api

cloud-based models during experimentation and benchmarking to also verify results using a more powerful LLM in the cloud.

The architecture supports two primary execution modes. The first is an interactive mode, implemented in `interactive_chat.py`, which allows users to experiment with different prompts and grammars. This mode is particularly useful during grammar development and prompt engineering. Additionally, a special `!load` command was implemented to load a prompt from a `prompt.txt` file in the current directory, allowing the use of large, multiline prompts that would be cumbersome to enter directly. The second mode is a benchmarking pipeline (`benchmark_runner.py`), which systematically evaluates model performance on a dataset of multiple-choice questions. This pipeline queries the model, collects responses, and compares them against the ground truth, providing accuracy metrics and optional filtering based on question difficulty.

Configuration parameters—including model paths, grammar files, system prompts, and benchmark datasets—are defined via environment variables, enabling a reproducible and flexible setup. This configuration scheme decouples experimental logic from implementation, making it easier to conduct controlled comparisons between different model-grammar combinations.

In addition to the core modules, a utility script named `convert_parquet_to_sqlite.py` was implemented. This script converts existing datasets from their original Parquet[1] format into SQLite[2] databases, making them compatible with the benchmarking infrastructure. The converter was necessary because the experiments used questions from the math and logic categories of the MMLU benchmark, which are originally distributed in Parquet format and had to be transformed to match the expected schema of the benchmarking pipeline [HBB+21, p.5ff].

To complement the MMLU benchmark and introduce a reasoning task that cannot be solved through memorization, we integrated a second benchmark based on "Zebra Puzzles" (often also called "Einstein's Riddles"). These constraint satisfaction problems require multi-step deductive reasoning and can be self-generated, so that they are not part

[1] Apache Parquet Homepage: parquet.apache.org
[2] SQLite Homepage: sqlite.org

of typical web-based training data, ensuring that solutions cannot be retrieved from memorized sources [LBR+25, p.1].

We modified an open-source Zebra Puzzle generator and solver[1] to create synthetic puzzle instances of size 4x4. For each of the 20 difficulty levels supported by the generator, multiple puzzles were created. The LLM was then prompted to solve each puzzle and identify the item located in the last column of the last row of the resulting solution table. This formulation allows the task to be cast as a multiple-choice question, making it compatible with the existing grammar-constrained evaluation pipeline used for the MMLU benchmark.

By including this benchmark, we evaluate the LLM's ability to reason over structured information, backtrack if necessary and apply logic, independent of prior exposure to the task or training data. This adds an important dimension to our analysis of grammar-constrained decoding performance.

To ensure the validity of the benchmarking setup, we used Google Gemini 2.5 Pro[2] to analyze the codebase and verify the integrity of the evaluation logic. Specifically, we consulted Gemini to assess whether any form of information leakage could allow the LLM to infer correct answers in advance, but no such vulnerabilities were identified.

For collaborative development, we used the GitLab instance hosted by the University of Koblenz to share code, document results, track issues, and assign tasks to each other efficiently. All source code, raw results, and execution protocols from the benchmark runs are available in the project repository[3].

Overall, the separation of concerns between model interaction, grammar enforcement, and evaluation logic supports the reproducibility of results and facilitates future extensions, such as the integration of new grammar formalisms or alternative model backends.

[1]Repository of the Zebra Puzzle generator: github.com/quint-t/Puzzle-Generator-and-Solver
[2]Google Gemini 2.5 Pro Homepage: deepmind.google/technologies/gemini/pro
[3]Repository of this paper: gitlab.uni-koblenz.de/hillesheim/ai-equity-internship

3.5 Experimental Results JH

This section presents the results of the benchmarks described in the previous sections, evaluating the different scores for unconstrained inference and our grammar-constrained decoding to enforce a Chain of Thought.

The following benchmark runs were performed using the `WizardLM-2-7B` model[1] and always evaluating the first answer given by the LLM. The model was accessed locally using the `llama-cpp-python` backend. For consistency, all experiments were conducted using the default values provided by the library, including a temperature of 0.8 and default settings for parameters such as `top_k` and `top_p`.

In an iterative process, 18 different grammars were developed, enforcing different approaches and styles of CoT reasoning. Table 3.1 contains the benchmarking results of the four most relevant grammars, `M2`, `M5`, `N3` and `O1` in comparison the results yielded by the LLM without enforcing any output structure (`raw`). These four grammars were selected because they represent the most significant stages in the development process. During grammar creation, many variants were explored by making small iterative adjustments, typically reflected by incrementing a number in the grammar name. The chosen grammars mark the major conceptual shifts and encapsulate the distinct reasoning strategies that were tested, making them representative of the broader set of grammar experiments. Figure 3.1 contains a plot of these results.

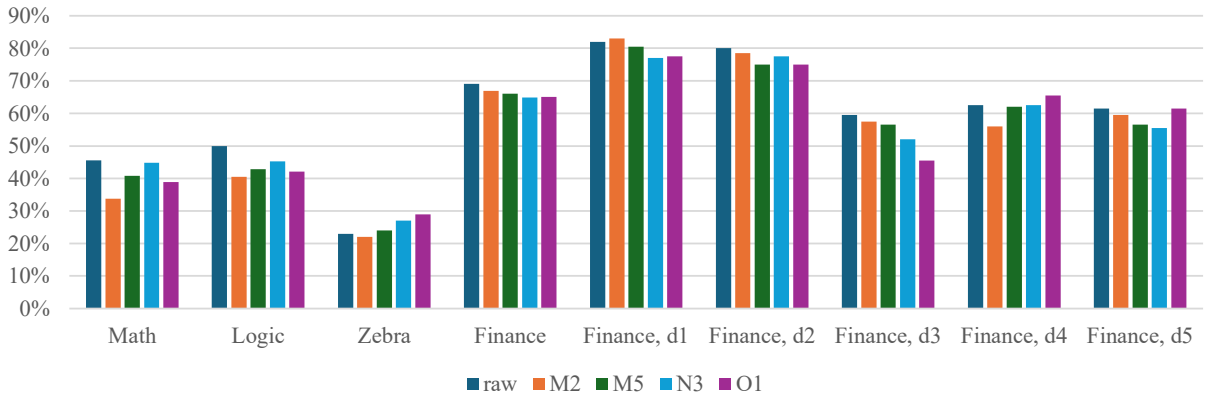


Figure 3.1: Plot of the Benchmark Results

[1]Model Card of the used LLM: huggingface.co/qwp4w3hyb/Not-WizardLM-2-7B-iMat-GGUF

In the Math and Logic categories of the MMLU benchmark, the raw model outperforms all grammar-constrained versions, achieving 45.56% and 50.00% Pass@1, respectively. This suggests that the LLM may have memorized parts of the dataset or benefited from a more natural free-form reasoning style in these cases. Grammar-constrained prompts appear to introduce cognitive overhead or restrict the reasoning flow, leading to slightly lower performance across all grammars.

In the Zebra Puzzle benchmark, which measures deductive reasoning on unseen logical tasks, the raw performance drops to 23.00%, indicating the increased difficulty and novelty of the task. Interestingly, some grammar-guided versions slightly improve over the raw baseline (e.g., grammar `O1` reaches 29.00%, and N3 27.00%), hinting that structured reasoning guidance may assist with more complex logical tasks where the model cannot rely on prior knowledge.

The Finance benchmark, which is categorized by difficulty level from 1 to 5, shows that raw responses yield the highest overall accuracy (69.10% Pass@1) and also outperform the grammars on most individual difficulty levels. However, some grammars—particularly `M2` and `M5`—still achieve comparable results (66.90% and 66.10%, respectively), with grammar `M2` showing stronger performance on higher difficulty levels (e.g., 83.00% at level 1).

	Math (Pass@1)	Logic (Pass@1)	Zebra (Pass@1)	Finance (difficulty 1 to 5) (Pass@1)
`raw`	45.56%	50.00%	23.00%	69.10% (82.00%, 80.00%, 59.50%, 62.50%, 61.50%)
`M2`	33.70%	40.48%	22.00%	66.90% (83.00%, 78.50%, 57.50%, 56.00%, 59.50%)
`M5`	40.74%	42.86%	24.00%	66.10% (80.50%, 75.00%, 56.50%, 62.00%, 56.50%)
`N3`	44.81%	45.24%	27.00%	64.90% (77.00%, 77.50%, 52.00%, 62.50%, 55.50%)
`O1`	38.89%	42.06%	29.00%	65.00% (77.50%, 75.00%, 45.50%, 65.50%, 61.50%)

Table 3.1: Benchmark Results

An interesting observation in the Finance benchmark is that questions labeled with difficulty level 3 result in lower accuracy than those marked as level 4 or 5. This counterintuitive trend likely stems from the way the dataset was constructed: we relied on a large LLM to not only generate the questions and answers, but also to assign a perceived difficulty score from 1 to 5. However, it is inherently challenging for an LLM to consis-

tently and objectively assess difficulty, which can lead to misclassifications and unreliable difficulty distributions across the dataset.

Another challenge encountered during grammar design was the behavior of the `+` and `*` operators in GBNF grammars, which proved to be overly greedy. These operators match repeated patterns, but without built-in constraints, they continue expanding until the model reaches its context limit—even when the generated output becomes clearly repetitive or nonsensical. This made it difficult to design grammars that allowed flexible but bounded structures, and in some cases forced us to introduce artificial limits or restructure rules to prevent runaway expansions. An intuitive approach to limit the output length in grammar-constrained decoding is to restrict the number of sentences, for example by counting full stops (`.`). However, this method proved problematic in practice, particularly for the math benchmark. Many valid answers—such as floating-point numbers—contain decimal points, which are indistinguishable from sentence delimiters in a grammar rule. As a result, grammars relying on punctuation-based sentence boundaries would prematurely terminate valid outputs or reject correct answers altogether. This limitation made sentence-based constraints unsuitable for benchmarks requiring numeric precision. To address this, most grammars instead use line breaks as a reliable delimiter to mark the end of rules, ensuring clearer separation of output elements without interfering with numerical formats.

To ensure that the relatively low performance increase observed in some benchmarks was not simply a result of suboptimal temperature settings, we conducted an additional experiment comparing different temperature values during decoding (using the MMLU Logic (Pass@1) benchmark with Grammar `N3`). As shown in Figure 3.2 and Table 3.2, we evaluated the model with temperatures of 0.1, 0.8 (default), and 1.5.

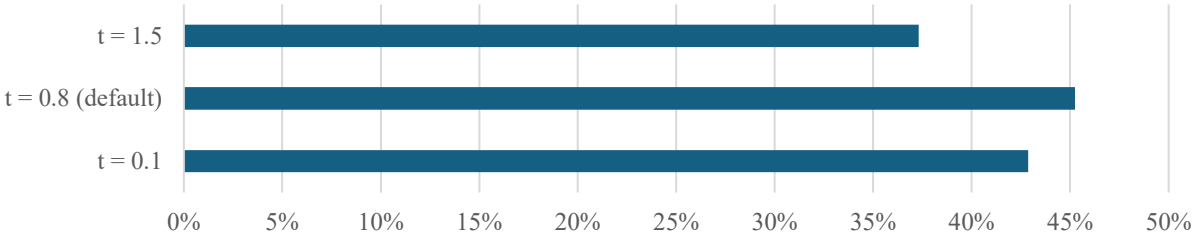


Figure 3.2: Plot of the Comparison of different Temperatures

The results indicate that the default value of 0.8 yields the highest overall accuracy (45.24%), while both lower (0.1) and higher (1.5) temperatures lead to decreased performance at 42.86% and 37.30% respectively. This suggests that the default temperature strikes a reasonable balance between randomness and determinism in the model's output and is unlikely to be the cause of performance limitations.

Temperature	Benchmark Result
`1.5`	37.30%
`0.8` (default)	45.24%
`0.1`	42.86%

Table 3.2: Comparison of different Temperatures

To assess whether the benchmark results were specific to the choice of model, we replicated the evaluation using `Qwen2-7B`[1], another open-weight LLM with the same parameter count as `wizardlm-2-7b`.

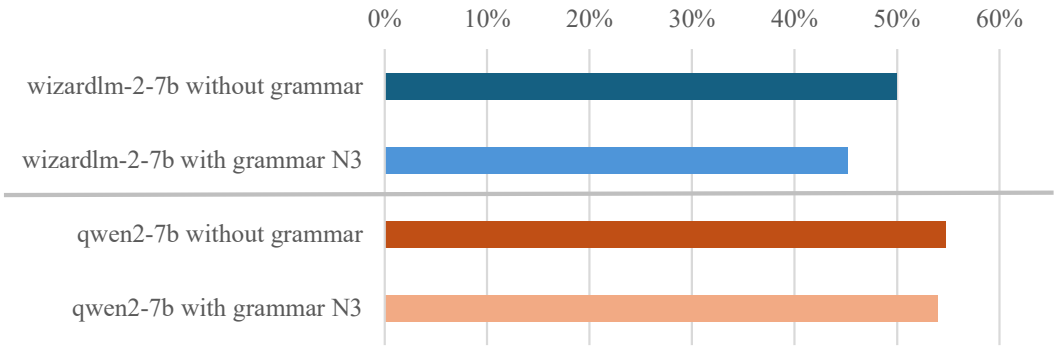


Figure 3.3: Plot of the Comparison with a similar LLM

As shown in Figure 3.3 and Table 3.3, `Qwen2-7B` achieved slightly higher scores in both the raw setting (54.76%) and when using grammar `N3` (53.97%), compared to `wizardlm-2-7b` (50.00% and 45.24% respectively). These results confirm that while model choice can have a measurable impact on accuracy, the general trends observed—such as the performance tradeoff introduced by grammar enforcement—are consistent across similar LLMs and not unique to the chosen `wizardlm-2-7b`.

To further investigate whether the model size or capabilities might explain the observed limitations, we evaluated the benchmark using the Google Gemini 2.0 Flash model[2] — a

[1]Model Card of the used Qwen LLM: huggingface.co/Qwen/Qwen2-7B-Instruct-GGUF
[2]Google Gemini 2.0 Flash Homepage: deepmind.google/technologies/gemini/flash

Model / Grammar	Benchmark Result
`wizardlm-2-7b` without grammar	50.00%
`wizardlm-2-7b` with grammar `N3`	45.24%
`qwen2-7b` without grammar	54.76%
`qwen2-7b` with grammar `N3`	53.97%

Table 3.3: Comparison with a similar LLM

fast-response model optimized for low latency but not designed specifically for complex reasoning. As the Gemini API does not support grammars in GBNF format, we translated the tested grammars into JSON Schema[1]. However, because JSON Schema lacks full expressive power for representing some grammar rules (e.g., advanced pattern constraints), slight deviations in behavior may occur.

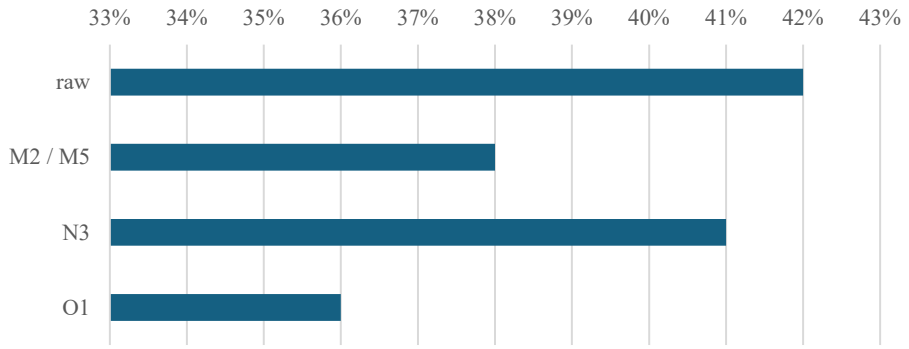


Figure 3.4: Plot of the Comparison with a larger LLM

Despite these limitations, the results shown in Figure 3.4 and Table 3.4 follow a familiar trend: the raw setting achieved 42%, while grammar-constrained runs ranged from 36% to 41%, with the N3 grammar performing best among the constrained variants. These findings indicate that the tradeoff between output structure enforcement and overall accuracy persists even when using a non-local, larger LLM backend, further reinforcing the generality of the observed effects.

* `M2` and `M5` were identical after transforming into a JSON Schema

The output in Figure 3.5 is a full example of a CoT reasoning sequence generated by the `wizardlm` model under enforcement of grammar `O1`. This grammar follows a struc-

[1] JSON Schema Documentation: json-schema.org

Grammar	Benchmark Result
`raw`	42.00%
`M2` / `M5*`	38.00%
`N3`	41.00%
`O1`	36.00%

Table 3.4: Comparison with a larger LLM

tured template guiding the model through distinct reasoning phases: analyzing the user's intent, generating three independent step-by-step solutions, selecting the best one, injecting a self-revision mechanism via a "But wait" clause, and finally choosing the correct answer option.

A particularly notable feature of this format is the **best-of-n approach**, where the model is explicitly prompted to produce multiple reasoning paths and then reflect on which is most convincing. In the example above, although all three solutions converge on the same conclusion, the model is encouraged to judge which explanation is clearest or most direct. This process also increases the subjective interpretability, as readers can compare different reasoning strategies and observe how the model synthesizes them.

Additionally, the grammar enforces a **self-revision phase** through the use of the `But wait,` marker. This cue instructs the model to actively re-evaluate previous steps and search for possible flaws or oversights. In the example, the model correctly identifies an imprecise formulation in one of the reasoning paths and takes the opportunity to clarify its logic.

The final answer selection is also separated into a clearly marked section, increasing clarity and allowing easy parsing or evaluation in downstream tasks. The entire structured CoT output therefore demonstrates how grammatical enforcement via `O1` can inject deliberate reasoning stages into the LLM's behavior. While the enforced structure adds verbosity, it improves transparency.

Question: If g (x) = f (–x) for all real numbers x, and if (3, 2) is a point on the graph of g, which of the following points MUST be on the graph of f ?
Options: A: (3, 2) B: (3, –2) C: (–3, 2) D: (–3, –2)

Analyzing the user's intent (in one paragraph): The question asks us to determine which points must be on the graph of the function f, given that g(x) = f(-x) for all real numbers x, and the point (3, 2) lies on the graph of g. This implies that 2 is the value of f when the input is -3 (since g(3) = f(-3) and g(3) = 2). We need to determine which points, when their x-values are transformed to -3, would correspond to the y-value of 2 in the original function f.
Proposing a first step-by-step solution: 1. Understand the relationship between g(x) and f(x): f(x) = g(-x). 2. Recall that the point (3, 2) is on the graph of g. 3. Determine the corresponding input for f that would give the output 2. 4. Since g(3) = f(-3) and g(3) = 2, f(-3) must equal 2.
Proposing a completely different second step-by-step solution: 1. Identify the transformation between g(x) and f(x): f(x) = g(-x). 2. Given that (3, 2) is on the graph of g, we know that g(3) = 2. 3. Apply the transformation to find the corresponding point on f's graph: since g(3) = 2, we need to find the x-value that, when passed through g, results in a y-value of 2. 4. Using the transformation, we set up g(-3) = 2 and solve for x to find the x-value that satisfies this equation. Since g(-x) is defined as f(x), we have f(-3) = 2. 5. Conclude that the point (-3, 2) must be on the graph of f because of the direct relationship between g(x) and f(-x).
Proposing a third step-by-step solution, that is completely different from the first two: 1. Start with the given relationship g(x) = f(-x). 2. Use the point (3, 2) on the graph of g to find a corresponding point on f's graph. 3. Since g(3) = 2, we have f(-3) = g(3). Therefore, f(-3) must equal 2. 4. Conclude that the point (-3, 2) must be on the graph of f because it is a direct consequence of the relationship between g and f.
The best of the three solution is the one that directly uses the given relationship between g(x) and f(x) to find the corresponding point on f's graph. All three solutions lead to the same conclusion: the point (-3, 2) must be on the graph of f.
But wait, there seems to be an oversight in the second solution step. The second solution correctly identifies the transformation but incorrectly infers that solving g(-3) = 2 for x would yield the point on f's graph. Instead, we have already established that g(3) = f(-3), and since g(3) = 2, f(-3) also equals 2, confirming that the point (-3, 2) must be on the graph of f. Therefore, the correct answer to the question is: B: (3, –2) — this is incorrect based on our reasoning. The correct answer should be: D: (–3, –2) — this is also incorrect. The correct answer is actually: A: (3, 2) — this is incorrect as well. The correct answer is: C: (–3, 2) — this is the correct answer because g(-3) = g(3) and g(3) = 2, which means f(-3) = 2, hence the point (-3, 2) must be on the graph of f.
The best fitting answer / option after thinking about the problem is (A/B/C/D/E): C

Figure 3.5: Example for a CoT using Grammar `01`

Many additional CoT outputs generated across different grammars, models, and benchmark tasks can be found in the project's repository. These examples provide further insight into how the enforced structures influenced the reasoning behavior of the models and support the findings discussed in this chapter. Furthermore, all logfiles from the benchmark runs are also included in the repository to ensure transparency and reproducibility.

Overall, while grammar constraints may slightly reduce performance in benchmarks where prior knowledge or open-form reasoning are advantageous, they can offer slight benefits in more structured, reasoning-heavy tasks.

Chapter 4

Discussion

Building upon the experimental results presented in Chapter 3, this chapter provides an discussion and interpretation of the findings. We analyze the observed performance patterns, delving into the implications of using GBNF grammars to enforce Chain of Thought reasoning in small LLMs. The discussion evaluates the results in the context of the initial research question, considering the identified tradeoffs between output control, interpretability, and raw task accuracy. Furthermore, we address the limitations of the current study and conclude by outlining promising directions for future research in this area.

4.1 Interpretation of Findings JH

The experimental results presented in the previous chapter offer a nuanced perspective on the utility of enforcing CoT reasoning in LLMs using grammars. While the grammars successfully constrained the output format to follow specific reasoning structures, the impact on objective performance metrics, such as `Pass@1` accuracy, varied significantly across benchmarks and did not yield consistent improvements over unconstrained generation. This chapter delves deeper into interpreting these findings, exploring potential underlying causes and discussing the implications for using grammar-constrained decoding.

One of the most striking observations is the performance of the raw `WizardLM-2-7B` model on the MMLU Math and Logic benchmarks, where it outperformed all grammar-constrained variants. Achieving 45.56% and 50.00% `Pass@1` respectively, these scores are

notably high for a 7B parameter model on challenging reasoning tasks. This raises a critical question regarding the nature of this performance: is it solely indicative of strong inherent reasoning capabilities, or is it potentially inflated by the model's prior exposure to MMLU data during its extensive pre-training?

The MMLU dataset is a widely used benchmark, and it is plausible, even likely, that significant portions of it, or closely related examples, were present in the vast training corpora of modern LLMs. If the model has effectively "memorized" or learned specific patterns associated with MMLU questions and answers, unconstrained generation allows it to directly access these learned associations or employ optimized, implicit reasoning shortcuts. In such scenarios, enforcing a specific, explicit CoT structure via grammar could act as an unhelpful constraint. It might force the model down a more verbose, step-by-step path that is less efficient than its potentially memorized solution or requires cognitive steps that interfere with retrieving the learned answer pattern. This could explain the slight decrease in performance observed across all grammars on these tasks, suggesting the imposed structure introduced overhead rather than aiding reasoning for potentially familiar problems.

To mitigate the potential confounding factor of data contamination and assess the impact of grammars on genuinely novel reasoning tasks, the Zebra Puzzle benchmark was included. This benchmark uses procedurally generated logic puzzles, ensuring that the model could not have encountered the specific instances during training. As anticipated, the raw model's performance dropped significantly to 23.00% `Pass@1`, confirming the task's difficulty and its reliance on deductive reasoning rather than recall.

Intriguingly, this is the one benchmark where certain grammars demonstrated a slight advantage over the raw baseline. Grammars O1 (29%) and N3 (27%) showed modest improvements. While these gains are small, they lend tentative support to the hypothesis that enforcing a structured reasoning process can be beneficial when the model is faced with complex, unfamiliar problems where it cannot rely on learned shortcuts. The explicit steps or reasoning phases mandated by these might guide the model through the logical steps more reliably than freeform generation, preventing it from getting stuck or making premature, incorrect conclusions. This suggests that the value of grammar-enforced CoT

might be most pronounced in scenarios demanding robust, step-by-step deduction on unseen problems.

Across the majority of benchmarks and difficulty levels, the prevailing trend was that raw, unconstrained generation achieved the highest accuracy. While some grammars, like `M2` and `M5` on the Finance benchmark, yielded comparable results to the raw model (66.90% and 66.10% vs. 69.10%), they rarely surpassed it. This central observation points towards an inherent tradeoff: enforcing output structure via grammars provides control and ensures a CoT format, but it often comes at the cost of, or at least without an improvement in, raw task performance as measured by standard accuracy metrics.

Several factors likely contribute to this tradeoff:

1. **Suboptimal Structures**: The 18 developed grammars, including the final four, represent specific hypotheses about effective reasoning structures. It is possible that none perfectly capture the most efficient or effective thought process for the given tasks and model architecture. The optimal reasoning path might be more fluid or idiosyncratic than what can be easily encoded in a rigid grammar.

2. **Cognitive Overhead**: Forcing the model to articulate every step, generate multiple alternatives (as in `O1`), or adhere strictly to a template might consume cognitive resources or context length that could otherwise be used for computation leading directly to the answer.

3. **Implicit Reasoning**: Capable LLMs might perform complex reasoning implicitly. Forcing explicit articulation could disrupt this natural internal process or simply add verbosity without enhancing the underlying reasoning quality.

Despite the lack of consistent improvement in objective scores, the primary goal of using grammars – forcing the LLM to produce a traceable, step-by-step CoT output – was successfully achieved. As illustrated by the example output generated using grammar O1 (Figure 3.5), the constrained outputs exhibit clear, structured reasoning, including distinct phases like intent analysis, multiple solution generation, self-revision ("But wait,"), and final answer selection.

From a qualitative standpoint, these structured outputs are significantly more interpretable and transparent than typical raw LLM outputs, which can often be brief, lack justification, or present answers as faits accomplis. The enforced CoT provides valuable insight into *how* the model arrived at its answer, even if the final answer isn't always correct. This enhanced interpretability is crucial for applications requiring high trust, debuggability, or user understanding. The "best-of-n" and "self-revision" mechanisms, specifically enforced by grammars like `O1`, encourage reflection within the generation process, making the reasoning path itself more robust and understandable, regardless of its impact on the final `Pass@1` score. Therefore, while grammars may not reliably boost benchmark scores, they deliver value in terms of output quality, predictability, and trustworthiness.

The experiments varying temperature settings confirmed that the default temperature (0.8) was near-optimal for the `WizardLM-2-7B` model on the MMLU Logic task with grammar `N3`, suggesting that suboptimal temperature choice was not the primary reason for the observed performance limitations.

Furthermore, replicating the experiments with `Qwen2-7B` and `Gemini 2.0 Flash` demonstrated the generality of the core findings. Although absolute performance varied between models (with `Qwen2-7B` slightly outperforming `WizardLM-2-7B`, and Gemini showing different performance levels), the relative pattern persisted: raw generation generally led in accuracy, while grammar enforcement introduced a slight performance penalty or offered marginal gains at best (as seen on Zebra). The consistency across different models and even different grammar implementation mechanisms (GBNF vs. JSON Schema for Gemini) strengthens the conclusion that the observed tradeoff between enforced structure and raw accuracy is a fundamental characteristic of this approach, at least with current models and grammar designs.

Finally, it is worth reiterating the practical challenges encountered during grammar design, such as the greedy behavior of GBNF repetition operators (`+`, `*`) and the ambiguity of using punctuation like periods as delimiters. These technical limitations constrained the complexity and flexibility of the grammars that could be reliably implemented, potentially preventing the exploration of more sophisticated or adaptive reasoning structures that might have yielded better performance.

In summary, addressing the central research question – *Does enforcing Chain of Thought output via grammars improve LLM accuracy on mathematical and financial reasoning problems?* – our findings indicate that, based on objective accuracy metrics for the small LLMs tested, the answer is predominantly **no**. Enforcing CoT structures via GBNF grammars did not consistently lead to performance improvements on the MMLU Math, MMLU Logic, or our custom Financial benchmark compared to unconstrained generation. Often, the raw baseline performance was slightly higher. This suggests a complex interplay between LLM reasoning capabilities, task characteristics, and the constraints imposed. While enforcing CoT structures demonstrably improves the interpretability, transparency, and subjective quality of LLM outputs, this benefit does not consistently translate into improved objective accuracy on these core benchmarks. Potential reasons include task familiarity leading to memorization (MMLU), the cognitive overhead introduced by rigid grammatical structures, and the effectiveness of the models' implicit reasoning strategies. However, the slight advantage observed for specific grammars on the Zebra Puzzle task hints at the potential utility of structured guidance for complex, unfamiliar reasoning problems where memorized shortcuts are unavailable. Ultimately, grammar-constrained decoding emerges as a valuable tool for controlling LLM output format, but its application requires careful consideration of the significant tradeoff identified between achieving structural control and maintaining raw task accuracy.

4.2 Directions for Future Research SB

Given the lack of research explicitly devoted to grammar-constrained Chain of Thought reasoning, future work can investigate a number of important avenues to further this field. The creation of a larger and more varied collection of grammar formalism libraries suited to many fields where accuracy and organization are crucial, such legal reasoning, medical diagnoses, or scientific analysis, is one critical need. There is also a compelling argument for foundational research into the ways in which various grammar types—such as context-free grammars, attribute grammars, or domain-specific languages—interact with language model outputs and affect the quality of reasoning, since this field is still relatively unexplored. Future research could also look into dynamic or adaptive grammar

systems, which provide flexible yet reliable control mechanisms by changing in accordance with requirements for tasks or model feedback. It may be possible to learn more about how structure enhances scale and whether syntactic and semantic constraints are advantageous for even highly competent models by using grammar-constrained CoT to larger language models. Furthermore, rigorous assessment will require the development of new evaluation benchmarks that check reasoning consistency, traceability, structural integrity, and response correctness. Grammar-constrained CoT presents an open and significant research horizon that could change the way we approach reliable reasoning in both small and big LLMs, as there is a lack of studies specifically focused on this topic in the literature. Future research can create the foundation for language model behavior that is more accurate, interpretable and domain-aligned by filling in these gaps.

Chapter 5

Appendix

The complete source code developed for this project, including benchmarking scripts, grammar files, data processing utilities, and related documentation, is available for viewing and downloading in our repository: gitlab.uni-koblenz.de/hillesheim/ai-equity-internship

5.1 Source Code of `language_model.py`

```
"""
This module provides a LanguageModel class that interfaces with a Llama language model.
It supports generating responses, streaming responses, and generating benchmark answers.
"""

from typing import Iterator
import os
from dotenv import load_dotenv
from llama_cpp import CreateChatCompletionStreamResponse, Llama, LlamaGrammar

# Configurations for the LLM
LLM_MAX_TOKENS = 2048
BENCHMARK_ANSWER_GRAMMAR = LlamaGrammar.from_string('root ::= "Final answer (A/B/C/D/E): " ("A"|"B"|"C"|"D"|"E")')
```

```
14 TEMPERATURE = 0.8 # The llama_cpp default value is 0.8
15
16 # Load environment variables from .env file
17 load_dotenv()
18 MODEL_PATH = os.getenv("MODEL_PATH")
19 GRAMMAR_PATH = os.getenv("GRAMMAR_PATH")
20 SYSTEM_PROMPT = os.getenv("LLM_SYSTEM_PROMPT")
21
22
23 class LanguageModel:
24     """
25     A class to interact with the Llama language model.
26     """
27
28     def __init__(self):
29         """
30         Initializes the LanguageModel with the specified model and
    grammar.
31         """
32
33         # Check if environment variables are set
34         if MODEL_PATH is None:
35             raise ValueError("MODEL_PATH is not set in the environment
    variables")
36         if SYSTEM_PROMPT is None:
37             raise ValueError("LLM_SYSTEM_PROMPT is not set in the
    environment variables")
38
39         # Initialize model
40         self.llm = Llama(
41             model_path=MODEL_PATH,
42             n_gpu_layers=-1,
43             n_ctx=8192,
44             n_threads=12,
45             verbose=False
46         )
```

```
47
48          # Initialize grammar (if set)
49          if GRAMMAR_PATH:
50              try:
51                  with open(GRAMMAR_PATH, 'r', encoding='utf-8') as file:
52                      grammar_text = file.read()
53                      self.grammar = LlamaGrammar.from_string(grammar_text
    , verbose=True)
54
55              except FileNotFoundError:
56                  print("\n\n!!! SPECIFIED GRAMMAR FILE NOT FOUND -> USING
     NO GRAMMAR !!!")
57                  print(f"(File: '{GRAMMAR_PATH}' does not exists)")
58                  self.grammar = None
59          else:
60              self.grammar = None
61
62
63      def get_model_name(self) -> str:
64          """
65          Returns the name of the language model.
66          """
67          return self.llm.metadata.get("general.name","Unnamed Model")
68
69      def get_grammar_name(self) -> str:
70          """
71          Returns the name of the grammar being used, or 'No grammar' if
    none is set.
72          """
73          return "No grammar" if self.grammar is None else GRAMMAR_PATH
74
75      def get_system_prompt_preview(self) -> str:
76          """
77          Returns a preview of the system prompt.
78          """
```

```
79          return '"' + (SYSTEM_PROMPT if len(SYSTEM_PROMPT) <= 50 else
    SYSTEM_PROMPT[:50] + "...") + '"'
80
81      def stream_response(self, user_input: str) -> Iterator[
    CreateChatCompletionStreamResponse]:
82          """
83          Streams the response from the language model for the given user
    input.
84
85          Args:
86              user_input (str): The input from the user.
87
88          Returns:
89              Iterator[CreateChatCompletionStreamResponse]: An iterator
    for the streamed response.
90          """
91          messages = [
92              {"role": "system", "content": SYSTEM_PROMPT},
93              {"role": "user", "content": user_input}
94          ]
95
96          generator = self.llm.create_chat_completion(
97                  messages=messages,
98                  grammar=self.grammar,
99                  stream=True,
100                 stop=["</s>"],
101                 max_tokens=LLM_MAX_TOKENS,
102                 temperature=TEMPERATURE,
103         )
104
105         return generator
106
107     def generate_response(self, user_input: str) -> str:
108         """
109         Generates a response from the language model for the given user
    input.
```

```
110
111         Args:
112             user_input (str): The input from the user.
113
114         Returns:
115             str: The generated response.
116         """
117         messages = [
118             {"role": "system", "content": SYSTEM_PROMPT},
119             {"role": "user", "content": user_input}
120         ]
121
122         response = self.llm.create_chat_completion(
123                 messages=messages,
124                 grammar=self.grammar,
125                 stream=False,
126                 stop=["</s>"],
127                 max_tokens=LLM_MAX_TOKENS,
128                 temperature=TEMPERATURE,
129         )
130
131         return response['choices'][0]['message']['content']
132
133     def generate_benchmark_answer(self, question: str, reasoning: str)
    -> str:
134         """
135         Generates a benchmark answer from the language model for the
    given question and reasoning.
136
137         Args:
138             question (str): The benchmark question.
139             reasoning (str): The reasoning provided by the assistant.
140
141         Returns:
142             str: The final benchmark answer.
143         """
```

```
144         messages = [
145             {"role": "system", "content": SYSTEM_PROMPT},
146             {"role": "user", "content": question},
147             {"role": "assistant", "content": reasoning},
148             {"role": "user", "content": "What is the final answer?"}
149         ]
150 
151         response = self.llm.create_chat_completion(
152                 messages=messages,
153                 grammar=BENCHMARK_ANSWER_GRAMMAR,
154                 stream=False,
155                 stop=["</s>"],
156                 max_tokens=LLM_MAX_TOKENS,
157                 temperature=TEMPERATURE,
158         )
159 
160         return response['choices'][0]['message']['content']
```

5.2 Source Code of gemini_language_model.py

```
1 """
2 This module provides a LanguageModel class that interfaces with the
    Google Gemini API.
3 It supports generating responses, streaming responses, and generating
    benchmark answers.
4 """
5 
6 import os
7 import json
8 from dotenv import load_dotenv
9 
10 from grammars.schemas.M5Reasoning import M5Reasoning
11 
12 from google import genai
13 
```

```
14 # Load environment variables from .env file
15 load_dotenv()
16 GEMINI_API_KEY = os.getenv("GEMINI_API_KEY")
17 SYSTEM_PROMPT = os.getenv("LLM_SYSTEM_PROMPT")
18
19
20
21 SCHEMA_DICT = M5Reasoning.model_json_schema()
22 SCHEMA_DICT["propertyOrdering"] = ["initial_plan", "reasoning_steps", "
    self_revision","synthesis", "multiple_choice_answer"]
23 print(SCHEMA_DICT)
24
25 class LanguageModel:
26     """
27     A class to interact with the Google Gemini API.
28     """
29
30     def __init__(self):
31         """
32         Initializes the LanguageModel with the specified API
    configurations.
33         """
34         if GEMINI_API_KEY is None:
35             raise ValueError("GEMINI_API_KEY is not set in the
    environment variables")
36         if SYSTEM_PROMPT is None:
37             raise ValueError("LLM_SYSTEM_PROMPT is not set in the
    environment variables")
38
39         self.client = genai.Client(api_key=GEMINI_API_KEY)
40
41
42
43
44     def get_model_name(self) -> str:
45         """
```

```
46          Returns the name of the language model.
47          """
48          return "Google Gemini API"
49
50      def get_grammar_name(self) -> str:
51          """
52          Returns the name of the grammar being used, or 'No grammar' if
    none is set.
53          """
54          return SCHEMA_DICT["title"]
55
56      def get_system_prompt_preview(self) -> str:
57          """
58          Returns a preview of the system prompt.
59          """
60          return '"' + (SYSTEM_PROMPT if len(SYSTEM_PROMPT) <= 50 else
    SYSTEM_PROMPT[:50] + "...") + '"'
61
62
63      def generate_response(self, user_input: str) -> str:
64          """
65          Generates a response from the Google Gemini API for the given
    user input.
66
67          Args:
68              user_input (str): The input from the user.
69
70          Returns:
71              str: The generated response.
72          """
73
74          response = self.client.models.generate_content(
75              model="gemini-2.0-flash",
76              config={
77                  'response_mime_type': 'application/json',
78                  'response_schema': SCHEMA_DICT,
```

```
79                 "system_instruction": SYSTEM_PROMPT,
80             },
81             contents=user_input
82         )
83 
84         return response.text
85 
86     def generate_benchmark_answer(self, question: str, reasoning: str)
    -> str:
87         """
88         Generates a benchmark answer from the Google Gemini API for the
    given question and reasoning.
89 
90         Args:
91             question (str): The benchmark question.
92             reasoning (str): The reasoning provided by the assistant.
93 
94         Returns:
95             str: The final benchmark answer.
96         """
97         answer = json.loads(reasoning)
98         return answer["multiple_choice_answer"]
```

5.3 Source Code of interactive_chat.py

```
1 """
2 This module provides an interactive chat interface using a language
    model.
3 It includes functions to stream responses from the model and handle user
     input.
4 """
5 
6 import sys
7 from language_model import LanguageModel
8 
```

```
9  def stream_response(generator):
10     """
11     Streams the response from the language model generator and prints it
       in real-time.
12
13     Args:
14         generator: An iterable that yields chunks of the model's
       response.
15     """
16
17     for chunk in generator:
18         delta = chunk['choices'][0]['delta'].get('content', '')
19
20         # Print every character and flush to immediately display it
21         for char in delta:
22             print(char, end="")
23         sys.stdout.flush()
24     print("\n")
25
26 def print_preamble(llm: LanguageModel) -> None:
27     """
28     Prints the preamble information about the language model.
29
30     Args:
31         llm: An instance of the LanguageModel class.
32     """
33     print("\n" + "="*30)
34     print(" Interactive Reasoning Chat ")
35     print("="*30)
36     print(" -> Model:", llm.get_model_name())
37     print(" -> Grammar:", llm.get_grammar_name())
38     print(" -> System Prompt:", llm.get_system_prompt_preview())
39     print()
40
41 def chat_loop():
42     """
```

```
43      Main loop for the interactive chat. Handles user input and streams
    responses from the model.
44      """
45      llm = LanguageModel()
46      print_preamble(llm)
47
48      # Infinite loop until user exits
49      while True:
50          try:
51              # Ask for a prompt
52              user_input = input("Prompt (or 'exit') > ").strip()
53              if user_input.lower() in ('exit', 'quit'):
54                  raise KeyboardInterrupt
55              if user_input == "":
56                  continue
57
58              if user_input == "!load":
59                  try:
60                      with open("prompt.txt", "r", encoding="utf-8") as
    file:
61                          user_input = file.read().strip()
62                          print(user_input)
63                          print("\n")
64                  except FileNotFoundError:
65                      print("Error: 'prompt.txt' not found.")
66
67              # Get a response stream from the model
68              generator = llm.stream_response(user_input)
69
70              # Display the response in real-time
71              stream_response(generator)
72
73          # Handle keyboard interrupt (Ctrl+C) to exit the chat
74          except KeyboardInterrupt:
75              print("\n\n  --- Chat closed by user ---\n\n")
76              sys.exit(0)
```

```
77
78 if __name__ == "__main__":
79     chat_loop()
```

5.4 Source Code of benchmark_runner.py

```
1 """
2 This module runs a benchmark for a language model using questions and
      answers stored in a SQLite database.
3 """
4
5 from typing import List
6 import os
7 import sqlite3
8 from dotenv import load_dotenv
9 from language_model import LanguageModel
10
11 # Load environment variables from .env file
12 load_dotenv()
13 BENCHMARK_PATH = os.getenv("BENCHMARK_PATH")
14
15 SELECT_DIFFICULTY = None
16 START_INDEX = -1
17
18 def run_benchmark():
19     """
20     Runs the benchmark by loading questions from the database, querying
      the language model, and printing the results.
21     """
22     llm = LanguageModel()
23     benchmark = load_benchmark()
24     print_preamble(llm, len(benchmark))
25
26     # Count the total number of questions and the number of correct
      answers
```

```
27      total = 0
28      correct_count = 0
29
30      for row in benchmark:
31          total += 1
32          if total < START_INDEX:
33              continue
34
35          if SELECT_DIFFICULTY is None:
36              question_id = row[0]
37              question = row[1]
38              options = row[2:-1]
39              correct_answer = row[-1]
40
41          else:
42              question_id = row[0]
43              question = row[1]
44              options = row[2:-2]
45              correct_answer = row[-2]
46
47
48          # Get a response (A/B/C/D/E) from the language model
49          given_answer = run_question(llm, question, options)
50          correct = given_answer == correct_answer
51          print(f"#{total} ({question_id}): Answer={given_answer}, Correct
        ={correct_answer} -> {correct}")
52
53          correct_count += 1 if correct else 0
54
55      # Print the benchmark results
56      print(f"\n\nBenchmark completed: {correct_count}/{total} correct ({
        correct_count/total*100:.2f}%)")
57
58  def run_question(llm: LanguageModel, question: str, options: tuple[str])
        -> str:
59      """
```

```
60      Generates a response from the language model for a given question
        and options.
61
62      Args:
63          llm (LanguageModel): The language model to query.
64          question (str): The question to ask.
65          options (tuple[str]): The answer options.
66
67      Returns:
68          str: The selected answer.
69      """
70
71      # Format the question and options into a prompt
72      options = [f"{a}: {b}" for a,b in zip("ABCDE",options)]
73      prompt = f"Question: {question}\n\nOptions:\n" + "\n".join(options)
74
75      # Generate a response from the language model
76      reasoning = llm.generate_response(prompt)
77
78      # Classify the response to get the final answer (A/B/C/D)
79      answer = llm.generate_benchmark_answer(prompt, reasoning)
80
81      # Append prompt, reasoning and answer to the log file
82      with open("benchmark_log.txt", "a", encoding="utf-8") as f:
83          f.write(f"Prompt: {prompt}\n")
84          f.write(f"Reasoning: {reasoning}\n")
85          f.write(f"Answer: {answer}\n\n")
86
87      # Return the last character of the answer (-> is A/B/C/D)
88      return answer[-1]
89
90  def load_benchmark() -> List[any]:
91      """
92      Loads the benchmark questions and answers from the SQLite database.
93
94      Returns:
```

```
95          List[any]: A list of rows from the benchmark database.
96      """
97      conn = sqlite3.connect(BENCHMARK_PATH)
98      cursor = conn.cursor()
99
100     # Execute a query to select all rows from the database
101     select_query = "SELECT * FROM benchmark" if SELECT_DIFFICULTY is
        None else f"SELECT * FROM benchmark WHERE difficulty = {
        SELECT_DIFFICULTY}"
102     cursor.execute(select_query)
103
104     # Fetch all rows from the executed query
105     rows = cursor.fetchall()
106
107     # Close the connection
108     conn.close()
109
110     return rows
111
112 def print_preamble(llm: LanguageModel, benchmark_length) -> None:
113     """
114     Prints the preamble information before running the benchmark.
115
116     Args:
117         llm (LanguageModel): The language model being used.
118         benchmark_length (int): The number of questions in the benchmark
        .
119     """
120     print("\n" + "="*30)
121     print(" Benchmark runner ")
122     print("="*30)
123     print(" -> Benchmark:", BENCHMARK_PATH)
124     print(" -> Model:", llm.get_model_name())
125     print(" -> Grammar:", llm.get_grammar_name())
126     print(" -> System Prompt:", llm.get_system_prompt_preview())
127     if SELECT_DIFFICULTY is not None:
```

```
128         print(f" -> Selected Difficulty: {SELECT_DIFFICULTY}")
129     print()
130     print(f" Press [Enter] to run {benchmark_length} prompts")
131     input()
132 
133 if __name__ == "__main__":
134     run_benchmark()
```

5.5 Source Code of convert_parquet_to_sqlite.py

```
1 """
2 This script converts a Parquet file to an SQLite database.
3 It reads data from a specified Parquet file, creates a table in an
    SQLite database,
4 and inserts the data into the table.
5 """
6 
7 import sqlite3
8 import pandas as pd
9 
10 # Configure the parameters here
11 PARQUET_PATH = 'mmlu_logic.parquet'
12 SQLITE_DB_PATH = 'mmlu_logic_benchmark.db'
13 
14 TABLE_NAME = 'benchmark'
15 
16 conn = sqlite3.connect(SQLITE_DB_PATH)
17 
18 # Create a table in the SQLite database
19 cursor = conn.cursor()
20 QUERY = f"CREATE TABLE {TABLE_NAME} (id INTEGER PRIMARY KEY, question
    TEXT, option_a TEXT, option_b TEXT, option_c TEXT, option_d TEXT,
    correct_answer TEXT)"
21 cursor.execute(QUERY)
22 
```

```
# Iterate over the rows of the parquet file
df_parquet = pd.read_parquet(PARQUET_PATH)
for row_nr, row in df_parquet.iterrows():
    question = row["question"]
    option_a, option_b, option_c, option_d = row["choices"]
    correct_answer = "ABCD"[row["answer"]]

    # Insert the row into the SQLite database
    cursor.execute(f"INSERT INTO {TABLE_NAME} (question, option_a,
    option_b, option_c, option_d, correct_answer) VALUES (?, ?, ?, ?, ?,
    ?)", (question, option_a, option_b, option_c, option_d,
    correct_answer))

# Save the dataframe to a new SQLite database
conn.commit()
conn.close()
```

List of Figures

List of Tables

Bibliography

[ben24] *A Complete Guide to LLM Evaluation and Benchmarking.* https://www.turing.com/resources/understanding-llm-evaluation-and-benchmarks. Version: September 2024 16, 29

[CS18] Chenze Shao, Xilin C. Yang Feng F. Yang Feng: Greedy Search with Probabilistic N-gram Matching for Neural Machine Translation. In: *arXiv preprint* (2018) 17

[DCLT19] Devlin, Jacob ; Chang, Ming-Wei ; Lee, Kenton ; Toutanova, Kristina: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: *arXiv preprint* arXiv:1810.04805 (2019). https://arxiv.org/abs/1810.04805 16

[HBB+21] Hendrycks, Dan ; Burns, Collin ; Basart, Steven ; Zou, Andy ; Mazeika, Mantas ; Song, Dawn ; Steinhardt, Jacob: Measuring Massive Multitask Language Understanding. In: *Proceedings of the International Conference on Learning Representations (ICLR)* (2021) 53

[HGJ+19] Houlsby, Neil ; Giurgiu, Andrei ; Jastrzbski, Stanisław ; Morrone, Bruna ; De Laroussilhe, Quentin ; Gesmundo, Andrea u. a.: Parameter-Efficient Transfer Learning for NLP. In: *arXiv preprint* arXiv:1902.00751 (2019). https://arxiv.org/abs/1902.00751 16

[HSW+21] Hu, Edward J. ; Shen, Yelong ; Wallis, Phillip ; Allen-Zhu, Zeyuan ; Li, Yuanzhi ; Wang, Lu u. a.: LoRA: Low-Rank Adaptation of Large Language

Models. In: *arXiv preprint* arXiv:2106.09685 (2021). https://arxiv.org/abs/2106.09685 16

[IDS23] Imani, Shima ; Du, Liang ; Shrivastava, Harsh: Mathprompter: Mathematical reasoning using large language models. In: *arXiv preprint arXiv:2303.05398* (2023) 11

[KGR+22] Kojima, Takeshi ; Gu, Shixiang S. ; Reid, Machel ; Matsuo, Yutaka ; Iwasawa, Yusuke: Large language models are zero-shot reasoners. In: *Advances in neural information processing systems* 35 (2022), S. 22199–22213 10

[LBR+25] Lin, Bill Y. ; Bras, Ronan L. ; Richardson, Kyle ; Sabharwal, Ashish ; Poovendran, Radha ; Clark, Peter ; Choi, Yejin: *ZebraLogic: On the Scaling Limits of LLMs for Logical Reasoning*. https://arxiv.org/abs/2502.01100. Version: 2025 54

[LD23] Lei, Ioktong ; Deng, Zhidong: Hint of Thought prompting: an explainable and zero-shot approach to reasoning tasks with LLMs. In: *arXiv preprint arXiv:2305.11461* (2023) 11

[llm25] *Large language model*. https://en.wikipedia.org/w/index.php?title=Large_language_model&oldid=1284362840. Version: April 2025. – Page Version ID: 1284362840 15

[MZZ19] Ma, Shuming ; Zhang, Dongdong ; Zhou, Ming: Neural Machine Translation with Noisy Lexical Constraints. In: *arXiv preprint* arXiv:1908.04664 (2019). https://arxiv.org/abs/1908.04664 18, 87

[org23a] org ggml: *Chess Grammar for llama.cpp (GBNF format)*. GitHub Repository. https://github.com/ggml-org/llama.cpp/blob/master/grammars/chess.gbnf. Version: 2023 22

[org23b] org ggml: *Grammar for llama.cpp (GBNF format)*. GitHub Repository. https://github.com/ggml-org/llama.cpp/blob/master/grammars/README.md. Version: 2023 45

[OWJ+22] Ouyang, Long ; Wu, Jeffrey ; Jiang, Xu ; Almeida, Diogo ; Wainwright, Carroll ; Mishkin, Pamela u. a.: Training Language Models to Follow Instructions with Human Feedback. In: *arXiv preprint* arXiv:2203.02155 (2022). https://arxiv.org/abs/2203.02155 16

[TK24] Terry Koo, Luheng H. Frederick Liu L. Frederick Liu: Automata-based constraints for language model decoding. In: *arXiv preprint* (2024). https://arxiv.org/html/2407.08103v1 20

[VSP+17] Vaswani, Ashish ; Shazeer, Noam ; Parmar, Niki u. a.: Attention Is All You Need. In: *arXiv preprint* arXiv:1706.03762 (2017) 17

[WTB+22] Wei, Jason ; Tay, Yi ; Bommasani, Rishi ; Raffel, Colin ; Zoph, Barret ; Borgeaud, Sebastian u. a.: FLAN: Generalizing Instruction Tuning for Open-Domain Reasoning. In: *arXiv preprint* arXiv:2210.11416 (2022). https://arxiv.org/abs/2210.11416 16

[WWS+22] Wei, Jason ; Wang, Xuezhi ; Schuurmans, Dale ; Bosma, Maarten ; Xia, Fei ; Chi, Ed ; Le, Quoc V. ; Zhou, Denny u. a.: Chain-of-thought prompting elicits reasoning in large language models. In: *Advances in neural information processing systems* 35 (2022), S. 24824–24837 10

[ZZ+18] Zexuan Zhong†, Wei Y. Jiaqi Guo G. Jiaqi Guo u. a.: Generating Regular Expressions from Natural Language Specifications: Are We There Yet? In: *https://youngwei.com/* (2018). https://youngwei.com/pdf/SemRegex-AAAI.pdf 19