UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Towards Augmenting and Evaluating Large Language Models

Permalink

https://escholarship.org/uc/item/5gt674rh

Author

Liu, Tianyang

Publication Date

2024

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Towards Augmenting and Evaluating Large Language Models

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Computer Science

by

Tianyang Liu

Committee in charge:

Professor Julian McAuley, Chair
Professor Taylor Berg-Kirkpatrick
Professor Zhiting Hu

2024

Copyright

Tianyang Liu, 2024

All rights reserved.

The Thesis of Tianyang Liu is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

DEDICATION

This work is dedicated with deep affection and immense gratitude to my beloved grandmother. Her recent departure has created an indelible void in my heart. Regrettably, the vast distance between us—me in the United States and her in China—prevented me from being by her side in her final moments. This missed opportunity has imbued me with a profound sense of loss and sorrow, a sentiment that mere words fail to fully capture.

Grandma, your strength was a beacon that guided me, yet I wish you hadn't had to be so strong, for so long, all on your own, without telling us anything. In the future, I will carry your legacy forward and keep your light shining bright within me.

The autumn chill that wakes me up
You loved the amber skies so much
Long limbs and frozen swims
You'd always go past where our feet could touch
And I complained the whole way there
The car ride back and up the stairs
I should've asked you questions
I should've asked you how to be
Asked you to write it down for me
Should've kept every grocery store receipt
'Cause every scrap of you would be taken from me
Watched as you signed your name Marjorie
All your closets of backlogged dreams
And how you left them all to me

What died didn't stay dead
What died didn't stay dead
You're alive, you're alive in my head
What died didn't stay dead
What died didn't stay dead
You're alive, so alive
And if I didn't know better
I'd think you were singing to me now
If I didn't know better
I'd think you were still around
I know better
But I still feel you all around
I know better
But you're still around

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am profoundly grateful to the many individuals who have supported and guided me throughout this remarkable journey.

I extend my heartfelt thanks to UC San Diego for fostering an academic environment that has greatly contributed to my personal and academic development. The opportunities and challenges I have encountered here have played a crucial role in shaping my journey in research.

I am deeply appreciative of my collaborators, Shibo Hao, Canwen Xu, Fei Wang, and Zhen Wang, for your invaluable insights and expertise. Collaborating with you has not only enriched my research but has also made the experience incredibly fulfilling.

My gratitude extends to my professors, Julian McAuley, Zhiting Hu, and Muhao Chen, for your guidance and support. Your advice has been essential in helping me navigate the complexities of my research endeavors.

A special thank you to Professor Julian McAuley for your confidence and the opportunity you provided during a particularly competitive application season. Your support has been incredibly motivating and has significantly influenced my academic journey.

I owe an immense debt of gratitude to my parents, whose unwavering support and love have been the cornerstone of my achievements.

And most importantly, to my grandmother. I hope you were still around. I know you are still around.

Chapter 3 and 6, in part, is a reprint of the material as it appears in "ToolkenGPT: Augmenting Frozen Language Models with Massive Tools via Tool Embeddings." by Shibo Hao, Tianyang Liu, Zhen Wang and Zhiting Hu, which was published at *Conference on Neural Information Processing Systems (NeurIPS)*, 2023. This thesis author was the co-primary investigator and author of this paper.

Chapter 4 and 6, in part, is a reprint of the material as it appears in "RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems." by Tianyang Liu, Canwen Xu and Julian McAuley, which was published at *The International Conference on Learning*

Representations (ICLR), 2024. This thesis author was the primary investigator and author of this paper.

Chapter 5 and 6, in part,is a reprint of the material as it appears in "Rethinking Tabular Data Understanding with Large Language Models." by Tianyang Liu, Fei Wang and Muhao Chen, which was published at *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2024. This thesis author was the primary investigator and author of this paper.

VITA

2018–2022	Bachelor of Engineering, Wuhan University
2022–2024	Master of Science, University of California San Diego
2024	Intern, Nvidia

PUBLICATIONS

"StarCoder 2 and The Stack v2: The Next Generation." arXiv Preprint, 2024.
"RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems." The International Conference on Learning Representations (ICLR), 2024.
"Rethinking Tabular Data Understanding with Large Language Models." arXiv Preprint, 2023.
"ToolkenGPT: Augmenting Frozen Language Models with Massive Tools via Tool Embeddings." Advances in Neural Information Processing Systems (NeurIPS), 2023.
"RoseMatcher: Identifying the impact of user reviews on app updates." Information and Software Technology, 2023
"Architecture Decisions in AI-based Systems Development: An Empirical Study." IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2023.
"The Role of User Reviews in App Updates: A Preliminary Investigation on App Release Notes." Asia-Pacific Software Engineering Conference (APSEC), 2021.

ABSTRACT OF THE THESIS

Towards Augmenting and Evaluating Large Language Models

by

Tianyang Liu

Master of Science in Computer Science

University of California San Diego, 2024

Professor Julian McAuley, Chair

In the rapidly evolving field of Natural Language Processing (NLP), the advent of Large Language Models (LLMs) marks a significant milestone, setting new standards in language understanding and generation. This thesis focuses on augmenting and evaluating LLMs, introducing ToolkenGPT, a novel method to integrate external tools via tool embeddings to enrich model functionality and adaptability and RepoBench, a benchmark for assessing the proficiency of LLMs in handling repository-level code auto-completion. Additionally, this thesis rethinks approaches towards tabular data reasoning, exploring how LLMs can be better tailored to understand and interpret structured data formats effectively.

Chapter 1

Introduction

With the advent of large language models (LLMs) [12, 20, 153, 91, 92, 94, 124, 125], the field of natural language processing (NLP) has witnessed a transformative shift, leading to an unprecedented expansion in the range of potential applications. From generating human-like text to facilitating complex conversational systems, LLMs have become a cornerstone in the development of AI-driven solutions across various sectors including healthcare, education, customer service, and content creation. Their ability to understand, generate, and interact using natural language has opened new horizons, making technology more accessible and intuitive for human users.

However, despite their impressive capabilities, LLMs are not without their limitations, which pose significant challenges to their practical application and reliability. One of the most notable issues is their propensity for generating hallucinated content [137] – fabrications or false information that may appear convincing but lacks factual accuracy. This tendency not only questions the trustworthiness of the output but also limits their use in scenarios where precision and truthfulness are paramount. Moreover, LLMs often struggle with processing long-context prompts, where the retention and coherent integration of information across a lengthy text become cumbersome, leading to a decline in performance [78]. Their handling of calculations and structured data, like tables or knowledge graphs also reveals a gap in their capabilities, as their primary architecture design is tuned towards language processing rather

than numerical analysis or structured data interpretation. Furthermore, the static nature of their training datasets means that LLMs cannot incorporate real-time information and it can be impractical to consistently update parameters by continuing training [134], making it a herculean task to maintain their knowledge base as up-to-date.

In response to these limitations, there is a growing trend towards augmenting LLMs [86] with external information access. By integrating external tools and functionalities, such as the ability to call functions or utilize specialized software within their processing, LLMs can overcome some of their inherent constraints. Techniques like Retrieval-Augmented Generation (RAG) [68] present solutions by combining the generative prowess of LLMs with the ability to query external databases or documents in real-time, thereby enriching the model's output with up-to-date and factual information. This synergy between LLMs and external resources opens up new avenues for applications that require a higher degree of accuracy and real-time data, providing a more robust framework for LLMs to operate within.

On the other hand, evaluating LLMs, in light of these advancements and challenges, is also important. As the application domains of LLMs expand, so does the necessity for comprehensive evaluation metrics and methodologies that can accurately assess their performance, reliability, and ethical implications. Evaluation goes beyond mere benchmarking on standard datasets; it encompasses a holistic analysis of the model's ability to produce coherent, contextually appropriate, and factually accurate content, its resilience against generating hallucinations, and its performance in long-context scenarios. Furthermore, evaluation must also consider the model's ethical implications, such as biases in generated content and the potential for misuse, ensuring that advancements in LLMs contribute positively to society. The nuanced nature of these models, combined with their wide-ranging applications, makes the task of evaluation complex yet critically important. It serves not only as a measure of progress but also as a guide for future research directions, ensuring that the development of LLMs remains aligned with ethical standards and practical utility. In this context, the thorough evaluation of LLMs is an indispensable component of their development, acting as the cornerstone upon which the future

of human-like artificial intelligence is built.

This thesis is structured as follows:

Chapter 1 introduces the motivation behind this research, delving into the challenges and opportunities in the current landscape of large language models (LLMs) and their applications.

Chapter 2 lays the foundation by discussing the basics, including what is a large language model, its underlying principles.

Chapter 3 introduces ToolkenGPT, a novel method that learns and integrates external tools via tool embeddings, significantly enriching the model's functionality and adaptability.

Chapter 4 presents RepoBench, a benchmark specifically designed to assess the proficiency of Code LLMs in handling repository-level code auto-completion tasks. This benchmark aims to evaluate the models' ability to understand and generate code in a long and contextually relevant manner.

Chapter 5 focuses on tabular data reasoning, exploring and comparing strategies for tailoring LLMs to more effectively understand, interpret, and interact with tabular data.

Chapter 6 presents a thorough review of related work.

Chapter 7 concludes the thesis while discussing the novel possibilities and future outlook of LLMs.

Chapter 2

Background

The field of NLP has witnessed remarkable advancements with the advent of LLMs, enabling unprecedented capabilities in text generation, understanding, and translation. These powerful models leverage neural network architectures, specifically the Transformer architecture [128], to process sequential data with remarkable efficiency and accuracy. The Transformer architecture introduced a novel self-attention mechanism, which allows the model to contextualize and relate different parts of the input sequence. This mechanism computes the attention weights by measuring the similarity between a query vector Q and key vectors K from the input, and then weighing the corresponding value vectors V. Mathematically, the attention function is defined as:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1}$$

where, d_k represents the dimensionality of the keys, and the softmax function ensures that the attention weights sum to 1. This self-attention mechanism enables the model to capture long-range dependencies within the input sequence, a critical aspect for understanding and generating coherent natural language.

LLMs are typically trained on massive text corpora, leveraging self-supervised learning techniques. During training, the model learns to predict the next token in a sequence based on

the preceding tokens, optimizing a loss function such as the cross-entropy loss:

$$L(\theta) = -\sum_{i=1}^{N} \log p_\theta(x_i|x_{i-1},...,x_1) \tag{2.2}$$

where $L(\theta)$ is the loss with respect to the model parameters θ, and $p_\theta(x_i|x_{i-1},...,x_1)$ represents the conditional probability of predicting the next token x_i given its predecessors, as modeled by the LLM. The self-attention mechanism and the extensive training on large corpora enable LLMs to capture intricate patterns and nuances in natural language, allowing them to generate human-like text, understand and analyze textual data, and perform various NLP tasks with remarkable accuracy.

Furthermore, LLMs can be fine-tuned on specific tasks or domains, enhancing their performance and adapting them to specialized applications. This fine-tuning process involves further training the pre-trained LLM on a smaller, task-specific dataset, adjusting the model parameters to better capture the nuances and intricacies of the target task or domain.

Despite their impressive language abilities, LLMs face inherent limitations in their capacity to interact with the real world, perform computations, and incorporate dynamic, real-time information. These models are primarily trained on static text corpora, lacking the ability to directly interact with external systems, execute calculations, or update their knowledge based on evolving events. As a result, recent research has explored augmenting LLMs with external tools and knowledge sources, enabling them to use tools, leverage structured data, perform reasoning, integrate information from diverse sources, etc. This enhances the capabilities of LLMs by combining their language understanding and generation abilities with external tools and knowledge bases, thereby addressing the inherent limitations of these models in interacting with the world and performing non-linguistic tasks.

Another key challenge faced by LLMs is the limited context window size, which restricts their ability to process and understand long-range dependencies in textual data. Conventional LLMs are typically trained on fixed-length sequences, limiting their capacity to capture and

leverage information from extended contexts effectively. Efforts have been made to address this limitation through techniques such as sparse attention mechanisms, which aim to selectively attend to relevant parts of the input sequence, reducing computational complexity and enabling the processing of longer sequences.

The success of LLMs has sparked a paradigm shift in NLP, paving the way for more advanced and versatile language models. These models are now being employed in a wide range of applications, including machine translation, text summarization, question answering, content generation, and various other domains. However, LLMs are not without their challenges. Their enormous size and computational requirements pose significant challenges in terms of training and deployment. Additionally, there are concerns regarding the potential for biases and ethical issues stemming from the training data and model outputs. Ongoing research efforts aim to address these challenges, further enhancing the capabilities and trustworthiness of LLMs, while exploring new frontiers in language understanding and generation.

Chapter 3

ToolkenGPT: Augmenting Frozen Language Models with Massive Tools via Tool Embeddings

Augmenting large language models (LLMs) with external tools has emerged as a promising approach to solving complex problems. In this chapter, we propose an alternative approach, **ToolkenGPT**,where each tool is represented as a token ("*toolken*") with its own embedding, facilitating dynamic tool integration within LLMs. Our method significantly enhances LLMs' performance across multiple domains, surpassing existing baselines, by enabling the flexible use of an extensive tool set.[1]

3.1 Introduction

Large Language Models (LLMs) [12, 20, 124, 94] have established themselves as powerful tools for diverse real-world applications, ranging from writing assistance to automated customer support [10, 13, 30]. As these models continue to evolve, there is a growing interest in their potential to interact with the real world and enhance their functionality through integration with other tools, such as the calculator, databases, etc [97, 123, 111, 102]. The capability of these models to master and control a wide array of tools not only serves as an indicator of their intelligence, but also signals a promising path to overcome some of their fundamental

[1] Code is available at https://github.com/Ber666/ToolkenGPT

Table 3.1. Comparison of different tool learning paradigms. Plug-&-Play means the LLMs can be equipped and unequipped with a tool flexibly. Note that it doesn't indicate zero-shot tool learning.

Tool Learning Paradigms	Frozen LMs	Massive Tools	Plug-&-Play	Ability to Use Extensive Data
Fine-tuning [111, 97]	✗	✗	✗	✓
In-context learning [144, 102, 15]	✓	✗	✓	✗
ToolkenGPT (**Ours**)	✓	✓	✓	✓

weaknesses. These include updating the latest world knowledge [87], reducing their hallucinations [109, 115], and executing symbolic operations [28, 36, 95], etc. However, the rapid emergence of new tools, such as advanced software libraries, novel APIs, or domain-specific utilities [75, 70, 58], introduces additional richness and complexity to the task of tool learning for LLMs. This continuous evolution accentuates the importance of empowering LLMs with the ability to adapt and master massive new tools swiftly.

Recent advancements in LLMs have witnessed two primary lines of research approaches for tool integration with LLMs [86, 143, 102] (Table 3.1). The first paradigm involves fine-tuning LLMs to learn specific tools [97]. For example, there are enormous efforts to integrate the retrieval tool into LLMs [42, 67, 115, 11] and the recent Toolformer [111] fine-tuned GPT-J to learn five tools. While this method could yield promising results, it is computationally expensive and lacks the adaptability to new tools. The second approach relies on in-context learning [144, 96, 102], where LLMs learn how to use the tool through in-context demonstrations provided in the prompt. This method allows LLMs to handle newly introduced tools and drives successful applications like LangChain [15] and ChatGPT Plugin [2]. However, in-context learning comes with its own unique limitations. Specifically, it struggles with the inherent limitation of context length, making it impossible to demonstrate massive tools in the context. Also, mastering new tools simply via few-shot examples could be challenging. For example, even the latest models like GPT-4 face difficulties when handling unusual tools [13].

[2] https://openai.com/blog/chatgpt-plugins

In this chapter, we introduce ToolkenGPT, an alternative solution that enables LLMs to master massive tools without the need for any LLM fine-tuning, while still allowing for quick adaptation to new tools. The key idea of ToolkenGPT is to represent each tool as a new token ("*toolken*") to augment the vocabulary. Specifically, each tool is associated with an embedding inserted into the LLM head like a regular word token embedding. During generation, once a toolken is predicted, the LLM temporarily switches into a special mode (through prompting) to produce input arguments for the tool to execute, and inject the outputs back into the generation (see Figure 3.1). This approach offers an efficient way for LLMs to master tools by only learning the lightweight toolken embeddings. Consequently, ToolkenGPT combines the strengths of both fine-tuning and in-context learning paradigms while avoiding their limitations (Table 3.1): Compared to in-context learning that can only accommodate a small number of tools and few-shot demonstrations, ToolkenGPT allows massive tools (by simply inserting respective toolkens in the vocabulary) and can use extensive demonstration data for learning toolken embeddings; In contrast to LLM fine-tuning, the tool embeddings not only requires minimal training cost, but also provide a convenient means for plugging in arbitrary new tools on the fly by expanding the toolken vocabulary.

We demonstrate the flexibility and effectiveness of our ToolkenGPT in leveraging numerous external tools for solving a diverse set of problems, spanning from numerical reasoning to knowledge-based question answering and embodied plan generation. In complex numerical reasoning problems that involve a number of mathematical tools (numerical operations such as finding *greatest common divisor*), we show that ToolkenGPT can effectively utilize these tools during the reasoning process, which outperforms some of latest popular approaches, such as Chain-of-Thought [135] and ReAct [144]. For knowledge-based question answering, ToolkenGPT accommodates a substantial number of relation APIs (over 200) from the knowledge base, thereby facilitating factual predictions. Furthermore, we apply our framework to task planning for embodied agents, where an agent interacts with an environment using tools, namely the actions and objects. The findings illustrate that our method offers better grounding

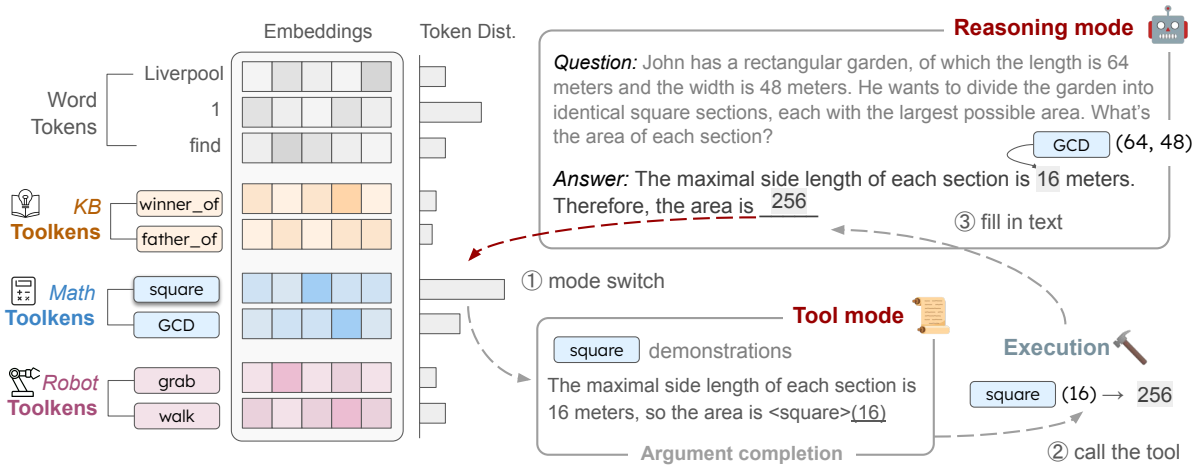

Figure 3.1. Overview of ToolkenGPT framework. Toolken embeddings are appended to the language model head like regular word tokens. In the "reasoning mode" for solving the problem, the LLM generates text as usual, except that any plugged-in toolkens are also considered for the next token generation. Once a toolken is predicted, (1) the LLM switch to the "tool mode", which provides a few demonstrations of the same tool to complete the arguments. Then, (2) the tool call is executed, and (3) the result is sent back to the text to continue the reasoning mode until the final answer is generated.

by learning toolken embeddings for 58 grounded actions and objects than previous in-context learning and specialized decoding methods.

3.2 ToolkenGPT for Mastering Massive Tools

In this section, we present ToolkenGPT, which enables LLMs to learn and use massive tools for complex problem-solving without the need for heavily fine-tuning the LLM. We begin by introducing the background and notations of language modeling for tool use. Typically, LLMs model the probability of a sequence of word tokens $s = (t_1, t_2, ..., t_n)$ as $P(s) = \sum_i^n P(t_i \mid t_{<i})$, where each word token comes from the vocabulary of the LLM, i.e. $t_i \in \mathcal{V}$ and $t_{<i}$ denotes the partial word token sequence before i-th step. In practice, the user often sets the prefix of a sequence (referred to as the prompt) to steer LLMs to generate desired contents, e.g., answering a question. Taking a step deeper, the distribution of the next token is predicted as $P(t_i|t_{<i}) = \text{softmax}(W_v \cdot h_{i-1})$, where $h_{i-1} \in \mathbb{R}^d$ is the last hidden state of the current context and

$W_{\mathcal{V}} \in \mathbb{R}^{|\mathcal{V}| \times d}$ is the embedding matrix for word tokens (also known as language model head).

Given a set of useful tools $\mathcal{T} = \{\tau_1, \tau_2, ...\}$, our goal is to enable LLMs to call a subset of these tools for solving the complex problem. Our flexible formulation allows tools to play a role by either returning some results that can help LLMs with text generation (e.g. calculation) or affecting the real-world environment (e.g. robot action). To call a tool during generation, the LLM first needs to select a tool and then input the arguments. In the running examples shown in Figure 3.1, during the answer generation process ("reasoning mode"), a math operator `square` is selected as the tool, and an operand `16` is generated as the argument in the "tool mode". Once the external tool receives the call, it executes the tool and returns the result `256`, back to the "reasoning mode".

3.2.1 Framework Overview

The core idea of ToolkenGPT is explicitly formulating tools as tokens (called "*toolkens*"). Each toolken is parameterized as a *toolken embedding* vector, and we denote a set of toolken embeddings as a matrix, i.e. $W_\tau \in \mathbb{R}^{|\mathcal{T}| \times d}$. Assuming we have trained toolken embeddings (to be described in Section 3.2.2), we first give an overview of our framework by introducing how it works in inference. As shown in Figure 3.1, the LLM is in the reasoning mode by default, generating the next token. Our framework allows the LLM to consider word tokens and toolkens uniformly. Specifically, the tool embedding matrix is concatenated with $W_{\mathcal{V}}$. Therefore, the LLM predicts the next token with the probability as follows:

$$P(t_i | t_{<i}) = \text{softmax}([W_{\mathcal{V}}; W_\tau] \cdot h_{i-1}) \tag{3.1}$$

where the next token can be either a word token or a toolken, i.e. $t_i \in \mathcal{V} \cup \mathcal{T}$, and $[;]$ is the concatenation operation. As we can see, our formulation of tools as toolken embeddings naturally allows for the fast adaption of new tools by expanding the toolken embedding matrix easily.

To execute a tool, the LLM switches into the "tool mode" once its toolken is predicted as the next token (as shown in the "mode switch" in Figure 3.1), which aims to generate the arguments for the tool. Specifically, the LLM pauses the generation and appends the current generated context to another prompt. The prompt in tool mode consists of in-context demonstrations for the predicted tool, showing how to generate the tool arguments by quoting the tool calls in a special syntax of `[tool](arguments)`. Then the LLM can follow the pattern in demonstrations to complete the arguments of the current tool call. Contrasting previous methods [144, 102] that fully rely on in-context learning for tool learning, our framework only leaves the easy work of completing arguments to in-context learning. Besides, there would be abundant context space for extensive demonstrations of a single specified tool. This design shares similarities with the classic divide-and-conquer methods [65, 61, 29]. Finally, the arguments are sent to the specified tool for execution, and the returned value is sent back to the text in the reasoning mode.

3.2.2 Learning Toolken Embeddings

Our framework keeps the original LLM parameters frozen and introduces a minimal additional training overhead with the toolken embeddings, W_τ. This embedding matrix contains the only parameters to optimize, but unlike other efficient LLM tuning methods, e.g., prompt tuning [66, 133] or prefix tuning [73], it does not require the gradients flowing through the major body of LLM parameters, leading to much stable and efficient training. Therefore, the tuning of toolken embeddings maintains nearly the same GPU memory as LLM inference. Whenever a new tool is added, the toolken embedding can be conveniently expanded and then, subsequent training on tool demonstration data involving the new tool gradually refines its embedding. Moreover, unlike in-context learning methods that only digest a few examples as training signals, ToolkenGPT is capable of tuning toolken embeddings from massive demonstrations.

Drawing parallels to how infants learn a new tool through demonstrations from adults [31], in this chapter, we primarily focus on learning toolken embeddings with tool demonstrations,

which can be either in-domain training data or synthetic data generated by LLMs (see Section 3.3.1 and Section 3.3.2). We first describe the format of training data and the training objective and we use the same example from Figure 3.1 to showcase how it can be used for training. Specifically, *"the area is 256 square feet ..."* can be tokenized into a word token sequence $s =$ ("the", "area", "is", "2", "5", "6", "square", "feet", ...). To indicate when to predict the toolkens, we need a parallel sequence mixed with word tokens and toolkens, i.e. $s' =$ ("the", "area", "is", "`[square]`", "`[N/A]`", "`[N/A]`", "square", "feet", ...). The subsequence of ("2", "5", "6") in s is where the returned tool results should fill in, and we choose the corresponding first token in s' as the toolken for the tool call with the following tokens are filled with `[N/A]`, indicating neglect in loss calculation. Thus, given a dataset composed of paired sequences $\mathscr{D} = \{(s, s')\}$, the training objective of ToolkenGPT is:

$$\mathscr{L}(W_\tau) = \sum_{(s,s')\in\mathscr{D}} \sum_{i=1}^{N} -\log P(t'_i|t_{<i}) \mathbb{1}_{t'_i \neq \texttt{[N/A]}} \tag{3.2}$$

where $P(t'_i|t_{<i})$ is defined in Eq.(3.1), and $\mathbb{1}_{t'_i \neq \texttt{[N/A]}}$ is the indicator function signaling we ignore the `[N/A]` tokens during the training. Thus, our training process is largely consistent with the inference in the reasoning mode. That is, to call a tool, the only job for the LLM is to predict a toolken at the beginning, and then the returned value will be filled back to the text. Here, `[N/A]` is introduced to skip the generation of the returned value of a tool call.

There are two primary ways to get the paired data. First, some datasets provide ground truth tool calls along with natural language sequences, e.g. the facts in KB supporting the answer to a question (Secion 3.3.2), or the calculation trace for solving a math problem (Section 3.3.1). To use the data for supervised learning, we preprocess them to get the paired data required for training as described in the above paragraph. Second, we explore synthesizing tool demonstrations with LLMs, sharing a similar idea to self-instruct [131]. An intuitive interpretation of this process is to distill the knowledge inside LLM to the new toolken embeddings. Specifically, we can prompt LLMs with the tool document and a few demonstrations with a special syntax

indicating tool calling, e.g., *The capital of U.S. is <capital>("U.S.")="Washington D.C."* Conditioned on that, the LLMs can generate some new use cases that utilizes the given tool and quote the tool call with the same syntax. We can then easily locate the tool calls and process the data into the paired data for training.

3.3 Experiments

In this section, we apply ToolkenGPT to three distinct applications characterized by meaningful tool-use scenarios: arithmetic tools for numerical reasoning, database APIs for knowledge-based question answering, and robot actions for embodied plan generation. We focus on how methods can accurately call the tools and how successfully they can solve the tasks. Our experiments show that ToolkenGPT can efficiently master massive tools while leveraging them to solve complex problems with improved performance, consistently better than advanced prompting techniques.

3.3.1 Numerical Reasoning

LLMs often struggle with mathematical tasks since the models are inherently designed for probabilistic estimation rather than symbolic operations. In this section, we aim to assess the tool-learning capabilities of ToolkenGPT, compared with in-context tool learning (e.g., ReAct [144]). We first demonstrate that ToolkenGPT consistently matches or outperforms the performance of in-context learning with the availability of four basic arithmetic functions ($+$, $-$, $\times$, $\div$). Moreover, to benchmark the tool-handling capability in more complex math problems, we include more available tools, i.e., an expanded (13) set of functions, and create a set of synthetic data. The results show that ToolkenGPT significantly outperforms baselines by training only on the synthetic data. Note that our focus is not to reach a state-of-the-art accuracy; Rather, the experiment is designed to evaluate the tool learning ability in the setting where certain tools are available.

Datasets. To evaluate the tool-learning proficiency in numerical reasoning comprehensively, we curate two new test datasets: (1) **GSM8K-XL**, an enhanced version of the existing GSM8K [23] dataset. GSM8K is a dataset of linguistically diverse grade school math word problems, involving performing a sequence of calculations using 4 basic arithmetic operations ($+$, $-$, $\times$, $\div$) to reach the final answer. In the original GSM8K dataset, the numbers for calculations are typically small, which might be less challenging for the recent powerful LLMs [23, 13]. So in the test set, we magnify the numbers to increase the computational difficulty for LLMs, which results in the GSM8K-XL dataset, featuring 568 test cases with much larger numbers. (2) **FuncQA** is a synthetic dataset we created to increase the complexity of math problems involving more arithmetic tools, which serves as a much more challenging benchmark to test the model's tool-learning capabilities. Specifically, This dataset requires at least 13 operators (e.g., `power`, `sqrt`, `lcm`) to solve, and it is challenging for both humans and LLMs to solve without an external calculator. Furthermore, FuncQA is categorized into two subsets: 68 one-hop questions ($\text{FuncQA}_{\text{one}}$) solvable with just one operation, and 60 multi-hop questions ($\text{FuncQA}_{\text{multi}}$) requiring a few reasoning steps.

To train the toolken embeddings used in GSM8K-XL, we preprocess the original training set of GSM8K which has the calculation annotation as described in Section 3.2.2. We get 6,054 examples, of which 1,000 were allocated for validation, and 5,054 for the training data. For the FuncQA dataset, we prompt ChatGPT to generate some one-hop QA patterns for each operator, and then randomly assign values to the patterns. This process yields 47 training data points and 3 validation data points for each operator, resulting in a total of 611 samples for training and 39 samples for validation.

Comparison Methods. We train toolken embeddings for each available math operator as described in Section 3.2.2. During inference, we prompt the LLM with 4-shot Chain-of-Thought [135] examples to enhance the reasoning ability of LLMs. The following baselines are evaluated for comparison: (1) *0-shot CharGPT* is the straightforward method asking LLMs to answer a question. No examples will be provided in the context and tools are not available.

Table 3.2. Results on the GSM8K-XL and FuncQA datasets. The numbers in parentheses indicate how many available tools are available. For GSM8K-XL and $\text{FuncQA}_{\text{one}}$ dataset, accuracy is evaluated based on an exact match (float numbers rounded to two decimals). In $\text{FuncQA}_{\text{multi}}$, we allow a margin of error of 0.1% to account for potential errors at each step of multi-hop reasoning.

Method	GSM8K-XL (4)	FuncQA (13)	
		One-Hop	Multi-Hops
0-shot ChatGPT	0.17	0.55	0.09
CoT [135]	0.18	0.20	0.03
ReAct [144]	0.32	0.57	0.06
ToolkenGPT (Ours)	**0.33**	**0.73**	**0.15**

We use ChatGPT as the base LLM in our experiment. This baseline measures the ability of the LLM to answer complex numerical reasoning problems with its own reasoning and calculation ability. (2) *Chain-of-thougts (CoT)* [135] is a more advanced prompting techniques. In this approach, a series of interconnected prompts are carefully crafted to guide the LLMs through a step-by-step reasoning process. The example reasoning chains are the same as the ones we used for ToolkenGPT, but no functions are available. (3) *ReAct* [144] combines reasoning and tools by prompting the LLMs to generate verbal reasoning traces and tool calls in an interleaved manner. Concretely, instead of just providing reasoning chains such as "... *The cost is 50*3.2=160*", ReAct incorporates special syntax to call operators, e.g."... *The cost is 50*3.2=<multiply>(50,3.2)=160*". Once the syntax is detected during inference, the tool would be called to calculate the result. We use the same reasoning chain examples as in both CoT and ToolkenGPT, with only slight differences in the tool calling syntax. LLaMA-33B [124] is used as the LLM for all settings other than zero-shot prompting.

Result Analysis. Table 3.2 shows the performance of all the methods on the GSM8K-XL and FuncQA datasets. On the GSM8K-XL dataset, 0-shot ChatGPT and few-shot learning with CoT struggle to calculate large numbers without the help of tools, while ReAct and ToolkenGPT manage to increase accuracy consistently by a large margin. Generally, both methods can call the correct tools when necessary, as the toolset is comprised of only the four basic operators.

However, for both $\text{FuncQA}_{\text{one}}$ and $\text{FuncQA}_{\text{multi}}$ datasets, learning to call applicable tools becomes challenging to ReAct as the number of tools increases. In ReAct, though all the tools are listed at the beginning of the prompt, it is infeasible to include demonstrations of every tool in the limited context (In our experiment, we provide 4 examples including 5 tool demonstrations). As a result, ReAct is susceptible to missing tool calls, making wrong tool calls, and predicting wrong arguments, especially for the tools not demonstrated in context. ToolkenGPT outperforms all the baselines across both one-hop and multi-hop scenarios, showing superior tool learning ability when there are numerous tools. It is important to note that even though toolken embeddings are trained solely using *one-hop synthetic data*, and *without any CoT examples*, they still manage to enhance performance in multi-hop problem contexts and can be integrated effectively with CoT prompting. This implies a degree of generalization of toolken embeddings, which is a very desired property that lowers the requirements of in-domain training data.

3.3.2 Knowledge-based Question Answering

LLMs are known to often make factual errors and hallucinate [54, 151, 150, 6] because of their limited knowledge [44]. Equipping them with access to knowledge bases (KBs) has been a promising research direction to reduce their hallucinations [115]. We formulate the access to the KB as APIs querying the database [122, 34]. Thus, each relational query can be treated as a tool to which the input argument is a subject entity, and the output is the corresponding tail entity. An example tool call is "`P1346`(2005-06 FA CUP) $\rightarrow$ LIVERPOOL F.C." "P1346" is a relation identifier in Wikidata, representing the winner of a competition or similar event (referred to `winner_of` below for ease of reading). In this section, we show that ToolkenGPT can accurately query a large knowledge base of *up to 234 tools (relations)*. We further show that even *only with synthetic data* (as described in Section 3.2.2 and explained below), we can train strong toolken embeddings that outperform popular tool-learning methods.

Dataset. KAMEL [59] is a question-answering dataset built with the facts in Wikidata. In line with ToolFormer [111], which uses its earlier version [100] as a benchmark to evaluate the

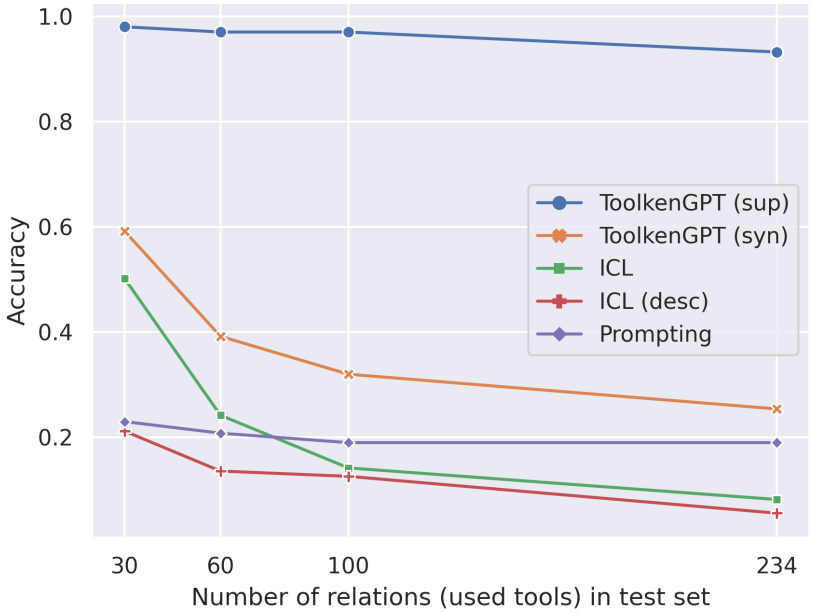

Figure 3.2. Performance of ToolkenGPT and baselines on 4 testsets involving different numbers of tools (relations) from KAMEL. ICL is short for In-context Learning [102]. Due to the context length limit of 2048 tokens, we list the descriptions and demonstrations of up to 30 relations for ICL and up to 60 relation descriptions for ICL (desc).

tool use, we adopt KAMEL to evaluate the use of KB query tools. KAMEL contains knowledge about 243 relations from Wikidata, each of which is associated with a question template (e.g. `winner_of`: "*Who is the winner of [S]?*") to turn a fact in Wikidata into a question. We have 234 tools in total for this dataset. In order to analyze the performance provided with different numbers of tools, we create four subsets by sampling from the original test set. Each subset consists of questions related to different numbers of relations, corresponding to 30, 60, 100, and 234, respectively. The size of each subset is 500.

Comparison Methods. We set up two different variants of the ToolkenGPT framework. (1) *ToolkenGPT (sup)*: We sample 200 examples per relation from the training set of KAMEL and train the toolken embeddings via supervised learning. This setting represents real-world scenarios where sufficient in-domain training data is available. (2) *ToolkenGPT (syn)*: In a more challenging setting where we assume in-domain training data is not available, we use the text description of each relation to synthesize training data with ChatGPT, e.g. "*The Nobel Peace Prize in 2020 was*

awarded to the United Nations World Food Programme for its efforts...", where the underlying tool call is `winner_of`(NOBEL PEACE PRIZE IN 2020)→UNITED NATIONS WORLD FOOD PROGRAMME. On average, 40 examples are used to train each toolken embedding.

We introduce the following baselines for comparisons: (1) *Prompting* [59] is a straightforward method that answers the questions with the LLM's internal knowledge. We frame each question within the prompt "*Question: [QUESTION]*\n*The answer is*" and ask the LLM to continue the sentence. (2) *In-context Learning (ICL)* [102] is a standard method to augment LLMs with tools. Before asking the question, we list the tool demonstrations and descriptions of all available tools. The demonstrations are shown in a specific syntax so that the LLM can generate in a similar style to be parsed. An example demonstration for `winner_of` is *"Question: Who is the winner of 2005-06 FA Cup?\nAnswer: The answer is <winner_of>(2005-06 FA Cup)=Liverpool F.C."* In a recent survey [102], this setting is referred to as "few-shot". (3) *In-context Learning (desc)* [102] is another common practice to augment LLMs with tools. The descriptions of all available tools will be provided in context, but their demonstrations are not directly shown. Instead, we show 8 demonstrations of the tools not included in the test subset to inform LLMs about the tool call format. This setting is referred to as "zero-shot" in Qin et al. [102]. The base model for all methods is LLaMA-13B [124].

Result Analysis. We show the experiment results on 4 testsets involving different numbers of relations in Figure 3.2. Note that the number of involved relations is the number of tools we can use. For all testsets, the accuracy of Prompting is about 20%, which indicates LLMs still struggle to store accurate facts in their parameters and it's necessary to augment them with a knowledge base. ToolkenGPT (sup) achieves the highest results with a large margin, showing that learning toolken embeddings is an effective method when there is massive in-domain training data. On the contrary, even though In-context learning also sees in-domain training data in the context, it still gets confused about which tools to call. Furthermore, the context length limit leads to drastic performance drops when there are more than 30 tools to use. The failure in the many-tools scene reveals the fundamental limitation of the in-context learning

paradigm. ToolkenGPT (syn) also outperforms all other baselines in all subsets, without seeing any in-domain training data. The synthetic training data, often in very different expression styles from the dataset, still helps the LLM understand these relations.

This success reflects the flexibility of our framework which can be applied even if there is no in-domain training data available. In-context learning (desc) generally fails in this task, because the LLM has difficulties memorizing text descriptions shown in contexts and mapping them to relation identifiers. The results provide more evidence to the previous discovery that LLMs have trouble using unfamiliar tools [13]. Based on this observation, it is reasonable to speculate that LLMs mostly *recall* the tools from their identifier instead of really *learning* to use tools from their descriptions.

3.3.3 Embodied Plan Generation

Recently, there have been many research attempts to utilize LLMs as the controller of embodied agents [51, 117, 1, 53, 139]. Despite the preliminary success of prompting LLMs, teaching LLMs about an environment and enabling them to make grounded predictions remain challenging. As discussed in Mialon et al. [86], tools that gather additional information (e.g. math or KB tools) and tools that have an effect on the physical world (e.g. actions taken by embodied agents) can be called in similar styles by the LLM. In this section, we demonstrate how our framework can also be applied to plan generation for embodied agents. Compared to previous methods that prompt LLMs, our ToolkenGPT can understand the environment better by learning toolken embeddings for agent action and object.

Dataset. VirtualHome [101] is a simulation platform for typical household activities, and ActivityPrograms knowledge base [101] consists of many tasks with plans executable in VirtualHome. We derive a subset of 297 tasks from ActivityPrograms.

Specifically, for each task, the model is given a high-level **goal** (e.g. *"Read book"*), a detailed **instruction** (e.g. *"I would go lie down in my bed and open the book and start reading."*, and a description of the **environment**, which includes the initial state of the agent, and the object

Table 3.3. Results on VirtualHome. Grounding means the proportion of scripts in which all the actions and objects can be grounded to the environment. Executable means the proportion of scripts that can be executed in VirtualHome without violating any rules. Success means the proportion of scripts that leads to the correct final state. Success (R) is a relaxed variant meaning the proportion of scripts that have reached the correct final state, but not necessarily ending with it.

Method	Grounding	Executable	Success	Success (R)
In-context Learning	0.74	0.42	0.20	0.30
+ Translation [51]	**1.00**	0.52	0.24	0.32
+ Grounded Decoding [53]	**1.00**	0.66	0.38	0.42
ToolkenGPT (Ours)	**1.00**	**0.82**	**0.68**	**0.70**

list of the environment (e.g. *"I am in ['home_office']. The objects I can manipulate are ['mail', 'freezer', 'television', ..., 'novel']"*. The model is expected to output an executable plan, which is an ordered list of verb-object instructions (e.g. *"[FIND] <novel>"*). Each task comes with an initial and final state graph, enabling the verification of the generated plans with the simulator and the comparison of the resulting final state with ground truth. We split the dataset into a training set of 247 tasks and a test set of 50 tasks, with a total of 25 verbs and 32 objects used in the dataset.

Comparison Methods. We consider all the actions and objects in VirtualHome as tools. With an additional `[END]` function indicating the end of a plan, we have 58 toolkens in total. For this dataset, we do not need the argument generation process described in Figure 3.1 because the tools do not take arguments. During inference, ToolkenGPT alternatively generates action toolkens and object toolkens, and ends with the `[END]` toolken. The toolken embeddings are trained on the training set.

We compare our method to the following baselines: (1) *In-context Learning* prompts the LLM and parses its outputs as the plan. The LLM is shown with the action list, 3 demonstration plans, and a new task with its goal, detailed description, and environment description. This method is the base of most recent methods [51, 1, 53] that apply LLMs to embodied AI. (2) *Translation* [51]: To avoid plans that include unavailable actions or objects, Huang et al. [51]

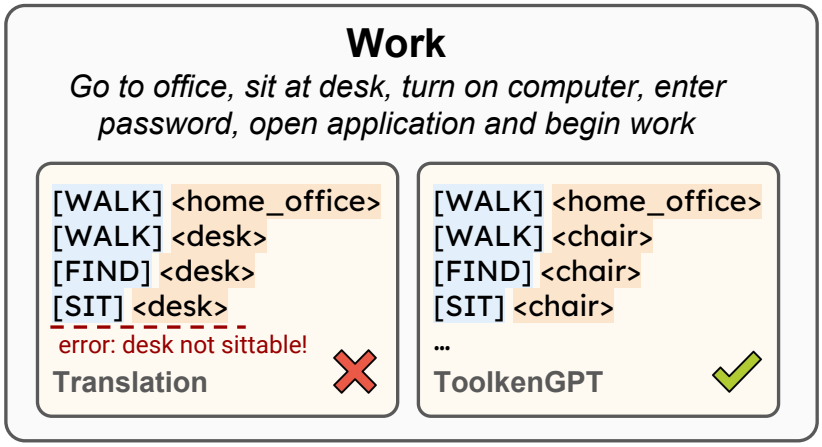

Figure 3.3. Case study on VirtualHome. ToolkenGPT predicts a successful script while other baselines fail to produce an executable one due to their misunderstanding of the `SIT` action.

proposes to use a translation model to translate the LLM's generation to admissible instructions. Following Huang et al. [51], we use SentenceRoBERTa-large [107] and translate the actions or objects to available ones with the highest cosine similarities. (3) *Grounded Decoding* [53] is a recent decoding-stage grouding method. The next token is predicted considering both LLM logits and "grounded functions". Specifically, we apply the affordance grounding function [53], encouraging LLMs to generate valid actions and objects. We do not consider other previous methods that heavily fine-tune the whole language model [72]. The base model of all methods is LLaMA-13B [124].

Result Analysis. We list results in Table 3.3. Though all valid actions and objects are explicitly listed in the context for the LLM using In-context Learning, it sometimes fails to ground its prediction to admissible instructions. Even though the actions and objects are valid, they often violate the physical rule in VirtualHome, resulting in a low success rate. We notice that while most of the plans generated with In-context Learning appear reasonable to humans, they are not grounded to the specific environment of VirtualHome. Translation [51] helps solve some shallow grounding problems, e.g. [tv] $\rightarrow$ `[television]`, while Grounded Decoding [53] further improves executable and success rate by considering grounding earlier in the decoding stage. Although these methods ensure all plans are grounded, neither significantly

Table 3.4. Comparison between ToolkenGPT and fine-tuning (LoRA) in terms of training cost and performance on FuncQA dataset. Both methods are based on Llama-7B.

Method	One-hop	Multi-hop	Computing Resource	Training Time
ReAct	0.40	0.03	-	-
Prompting	0.10	0.00	-	-
Fine-tune w/ LoRA [50]	0.62	0.07	8 $\times$ A100 (80G)	40 min
ToolkenGPT	0.55	0.06	1 $\times$ RTX3090 (24G)	2 min

improves the LLM's understanding of actions and objects, leading to unsatisfactory executable and success rates. ToolkenGPT not only predict valid actions and objects naturally by its design, but also achieves the highest success rate by learning toolken embeddings from more training tasks. A concrete example is shown in Figure 3.3 to illustrate the difference: All the baselines predict `[SIT]` <`desk`>, presumably guided by the description "sit at desk", but in VirtualHome `[SIT]` refers to "sit on", and a desk is regarded as not sittable. ToolkenGPT is the only one to successfully learn this rule from demonstrations and instead predict `[SIT]` <`chair`>.

3.3.4 Analysis

Computational Cost. We conduct experiments to compare ToolkenGPT with fine-tuning, specifically using LoRA [50], in terms of computation efficiency and performance. Due to the cost of fine-tuning LLMs, we implement both methods on LLaMA-7B. The results are listed in Table 3.4.

Fine-tuning LLMs results in slightly better performance than ToolkenGPT on FuncQA. Even though we apply LoRA, which is known for efficiency, the time consumption for fine-tuning exceeds significantly when compared to training toolken embeddings. It is also worth noting that ToolkenGPT enjoys additional benefits other than efficiency (Table 3.1), especially the plug-and-play of massive tools, thanks to the decoupled parameters for different tools.

Ablation Study. The design of ToolkenGPT benefits both tool selection and argument completion (tool mode in Figure 3.1). To understand their respective contributions to the performance, we further implement a baseline combining ReAct-style prompting and the sub-

Table 3.5. Ablation study on FuncQA dataset with LLaMA-30B.

Method	One-hop	Multi-hop
ReAct	0.57	0.06
+ Tool mode	0.60	0.07
ToolkenGPT	0.73	0.15

Table 3.6. ToolkenGPT with different configurations of training data on KAMEL.

# Examples	Synthetic	Supervised
10	0.36	0.56
20	0.46	0.90
40	0.52	0.95

routine of argument completion (tool mode). In the tool mode, the LLM is prompted with demonstrations using only the selected tool, which will provide more relevant knowledge than ReAct prompt for argument completion. As shown in Table 3.5, adding a tool mode could indeed improve the vanilla ReAct prompting method by enhancing the accuracy of argument completion. However, ToolkenGPT still outperforms this improved baseline by a large margin, indicating that toolken embeddings effectively help LLMs to decide when and which tool to call.

Training Data. In this section, we explore the effects of training data on learning the toolken embeddings. We choose to extend our experiments on KAMEL (Section 3.3.2), because there are two different sources of training data and it is easy to process or synthesize more data. Specifically, we sample 10/20/40 training examples of each tool for both ToolkenGPT (sup) and ToolkenGPT (syn), and report the accuracy on the test set involving 30 tools.

The results are summarized in Table 3.6. Under the same budget of data size, training with supervised data leads to better performance. Though we do not observe obvious mistakes in most of synthetic data instances, the distribution gap between synthetic data and test set may prevent toolken embedding from performing well. A larger training set benefits the performance for both data sources.

Chapter 4

RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems

Large Language Models (LLMs) have greatly advanced code auto-completion systems, yet lack benchmarks for multi-file programming scenarios. In this chapter, we introduce RepoBench, a benchmark for repository-level code auto-completion, covering Python and Java, with tasks for code retrieval, completion, and complex pipelines. Each task respectively measures the system's ability to retrieve the most relevant code snippets from other files as cross-file context, predict the next line of code with cross-file and in-file context, and handle complex tasks that require a combination of both retrieval and next-line prediction. RepoBench aims to facilitate a more complete comparison of performance and encouraging continuous improvement in auto-completion systems. RepoBench is actively maintained with the latest code, serving as a live benchmark publicly available at https://github.com/Leolty/repobench.

4.1 Introduction

Large language models (LLMs; [12, 20, 124, 94]) have been instrumental in paving new avenues for innovative applications across diverse domains, with programming being a notably attractive and promising domain [16, 127, 5, 132]. In particular, the rise and application of code auto-completion systems like GitHub's Copilot [1], driven by OpenAI's Codex [16], have the

[1] https://github.com/features/copilot

potential to substantially changed the manner in which we interact with code. These changes facilitate coding for beginners and improve efficiency of the coding process for experienced developers.

A variety of code auto-completion models [16, 39, 33, 90, 71, 3] have emerged in recent years, each boasting unique capabilities and performance characteristics. This emergence of models emphasizes the increasing importance of AI in the realm of programming, leading to a more diversified and competitive landscape. However, current evaluation datasets and benchmarks [84, 105, 4] predominantly focus on completion tasks within the scope of a single file. This focus fails to reflect the complexity and intricacies of real-world programming scenarios, where developers frequently work on multi-file projects, often navigating through and understanding code spanning several repositories.

Recognizing the need for a more comprehensive evaluation, we introduce RepoBench, a new benchmark for evaluating the effectiveness of repository-level code auto-completion systems. Specifically, RepoBench offers three distinct evaluation sub-tasks, each emphasizing a unique aspect of a fully functioning code auto-completion system: (1) **The Retrieval Task (RepoBench-R)**, which tests the system's ability to retrieve the most relevant code snippets, thereby providing the necessary context for the prediction of the next line of code. (2) **The Code Completion Task (RepoBench-C)**, where the task is to predict the next line of code given a pre-defined context. The context can involve content from different files (cross-file context) and within the file (in-file context) with a moderate length setting that can fit most models. (3) **The End-to-End Pipeline Task (RepoBench-P)**, which is designed to simulate the complete process of a code auto-completion system like GitHub Copilot - first retrieving relevant snippets and then completing the code by predicting the next line. In this scenario, the system may encounter a large set of potential snippets for retrieval, resulting in longer and broader contexts, which leads to the need for the system to optimize the efficient selection of numerous candidates to facilitate code completion while ensuring that the extensive context remains within the system's processing capabilities.

We conduct a series of experiments on RepoBench, analyzing the efficacy of various retrieval methods and code completion models of different magnitudes, and the assessment of their combined performance in a full pipeline, providing some insights for future research and development. Our results underscore the significance of code models that can manage extended contexts and maintain generalizability in real-world coding environments.

4.2 The RepoBench Dataset

RepoBench is a live benchmark for auto-code completion, with a commitment to continuously incorporate the latest data for model evaluation. This section introduces the construction and findings of RepoBench's inaugural iteration (v1.0).

4.2.1 Data Sources

Github-Code Dataset: The first source of RepoBench is the `github-code` dataset[2], which consists of a vast collection of code files sourced from GitHub repositories under open-source licenses with a data cutoff date of *March 16, 2022*. Specifically, we aggregate files based on their repository name as the github-code dataset is originally stored at the file-level. Given that the code in this dataset has been widely utilized for training various models [71, 90], we primarily use this dataset for constructing our training data. The use of this data for training specifically addresses the adoption of patterns that concatenate cross-file context and in-file context for next-line prediction. Fine-tuning on this dataset is optional, as sufficiently robust models may already exhibit this generalizability.

Newly Crawled GitHub Data: To mitigate impacts regarding data leakage and memorization, we augment the dataset by incoporating the most recent, non-forked GitHub repositories that are permitted under their respective licenses. Specifically, we use GitHub's official API to crawl Python and Java repositories created after *February 9, 2023*, which aligns with the newest knowledge cutoff date of The Stack [63], and before *August 3, 2023*. This newly-crawled data

[2]https://huggingface.co/datasets/codeparrot/github-code

serves exclusively as our test set for evaluation.

Continuous Updates: In response to the rapid advancement of Code LLMs and their training datasets, RepoBench is committed to a regimen of continuous updates, to ensure that RepoBench keeps pace with the latest developments and avoids potential data leakage, which could compromise the integrity of model evaluations. As of this writing, RepoBench v1.1 is already available. Detailed discussions on RepoBench v1.1 can be found in Section 4.4.

4.2.2 Data Processing

The data processing procedure for this study involves multiple steps. For the training data sourced from `github-code`, repositories with a number of Python or Java files between 32 and 128 are selected. This range is chosen to ensure an adequate cross-file dependency while avoiding excessive complexity and keeping the data volume within a reasonable range. While for the newly crawled test data, we do not set file number constraints to ensure a thorough evaluation. To identify cross-file dependencies and their usage, we use tree-sitter[3] to parse each file. This parsing is primarily directed at import statements, enabling us to identify all cross-file modules and the lines utilizing these modules (termed cross-file lines). Further, we track the corresponding code snippets that define these imported modules.

After processing the data, our dataset comprises 10,345 Python and 14,956 Java historical repositories, serving as training data and are available for optional fine-tuning. Additionally, we have 1,075 Python and 594 Java new repositories from GitHub designated as test data for evaluation.

4.2.3 Task Construction

Task Settings To effectively evaluate next-line prediction in auto-completion systems, we define three settings:

- **Cross-File-First (XF-F):** This is the most challenging setting, where we mask the first

[3]https://tree-sitter.github.io/tree-sitter/

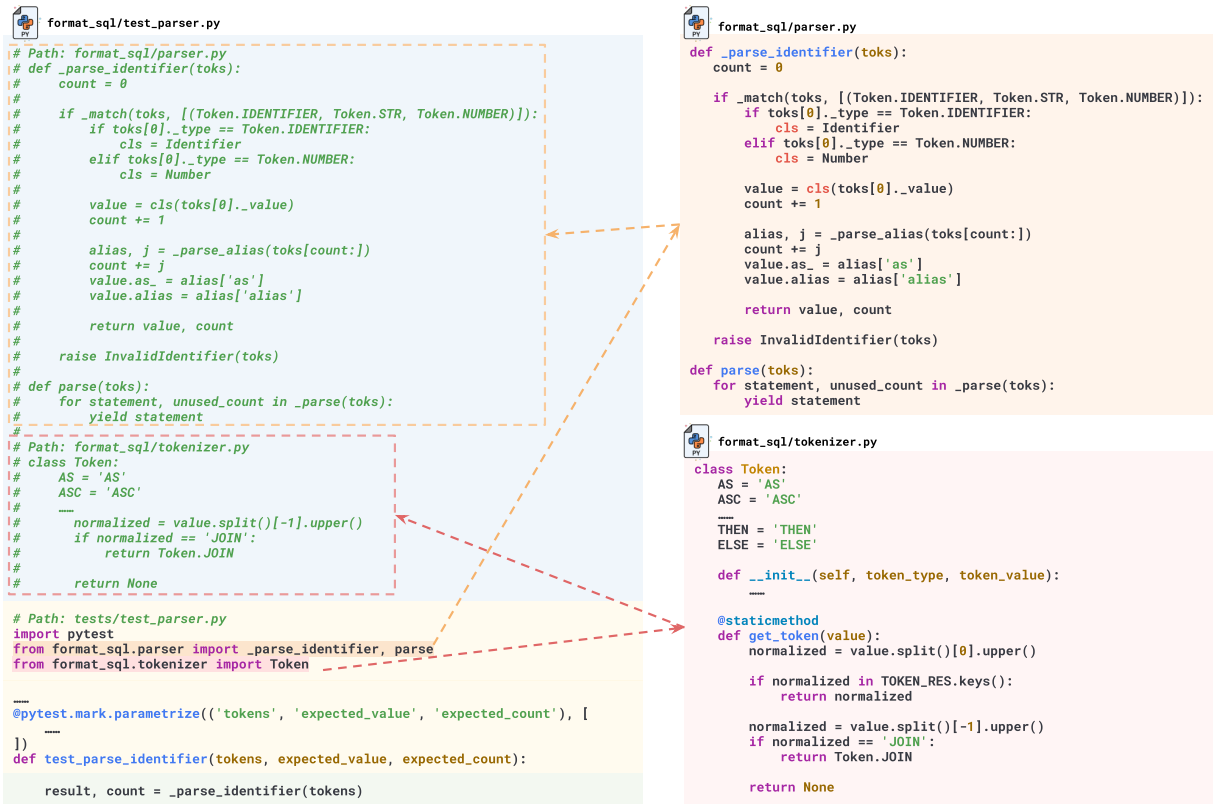

Figure 4.1. Construction of a prompt for repository-level cross-file code completion. The commented cross-file context (path + snippet), parsed from import statements using `tree-sitter`, is concated with the in-file context (path + import statements + preceding lines), which cropped to a maximum of 30 lines in RepoBench to form the input prompt, with the objective is to predict the next line. Note that for clarity, certain lines of code are omitted in this figure, which is an abbreviated and simplified version derived from a real example.

appearance of a cross-file line within a file. In this setting, there is no prior usage of the module in the in-file context to aid the prediction, thereby requiring the system to handle long-range cross-file context for better accuracy.

- **Cross-File-Random (XF-R):** In this setting, we mask a random and non-first occurrence of a cross-file line. Unlike the XF-F setting, the prior in-file usage of the module may serve as a hint for the prediction.

- **In-File (IF):** In this setting, we mask an in-file line that does not involve any cross-file modules. This setting serves as a robustness test to ensure that the incorporation of

cross-file context does not greatly affect the accuracy of predictions.

Note that RepoBench-R (Retrieval) is designed with only XF-F and XF-R settings, as IF does not involve retrieval and thus cannot be evaluated in this task, while both RepoBench-C (Code Completion) and RepoBench-P (Pipeline) involve all three settings: XF-F, XF-R, and IF.

RepoBench-R RepoBench-R targets the retrieval component of a repository-level auto-completion system, focusing on extracting the most relevant code snippet from a project repository for next-line code prediction.

In RepoBench-R, every snippet parsed from import statements is treated as a potential candidate for next-line prediction, where only one 'gold snippet' is the optimal context for prediction. This task considers scenarios with 5 or more candidate snippets, and specifically, we categorize them into two subsets: those with 5-9 candidates as the *easy* subset, and those with 10 or more candidates as the *hard* subset. As demonstrated in Table 4.1 (top), both the *easy* and *hard* subsets contain 12,000 samples for the XF-F setting, whereas for the XF-R setting, each subset consists of 6,000 samples. We also provide training data for optional usage, further details can be also located in Table 4.1 (bottom). For evaluative purposes, the Accuracy@k (acc@k) metric is employed to assess retrieval performance. The *easy* subset is evaluated using acc@1 and acc@3, while the *hard* subset is examined through acc@1, acc@3, and acc@5 metrics.

RepoBench-C RepoBench-C simply focuses on the prediction of the next line of code, given a set of in-file context (including several preceding lines and import statements), and cross-file context.

In RepoBench-C, as shown in Figure 4.1 the prompt is created by combining all the parsed snippets as cross-file contexts and an in-file context. The in-file context includes import statements and several preceding lines of code with a maximum limit of 30 lines. To address the varying context length in existing models, RepoBench-C is divided into two subsets: RepoBench-C-2k and RepoBench-C-8k. RepoBench-C-2k, designed for models with a token limit of 2,048, holds prompts that do not exceed 1,925 tokens. Concurrently, RepoBench-C-8k is architected

Table 4.1. *(Top)* Test data overview for RepoBench across Python and Java for 3 different tasks; *(Bottom)* Training data for RepoBench.

Lang.	Task	Subset	XF-F	XF-R	IF	Mean Candidates	Mean Tokens
Python	RepoBench-R	Easy	12,000	6,000	-	6.7	-
		Hard	12,000	6,000	-	17.8	-
	RepoBench-C	2k	12,000	5,000	7,000	-	1,035
		8k	18,000	7,500	10,500	-	3,967
	RepoBench-P		10,867	4,652	6,399	24	44,028
Java	RepoBench-R	Easy	12,000	6,000	-	6.8	-
		Hard	12,000	6,000	-	25.5	-
	RepoBench-C	2k	12,000	5,000	7,000	-	1,093
		8k	18,000	7,500	10,500	-	4,179
	RepoBench-P		10,599	4,459	6,196	26	139,406

Language	Task	XF-F	XF-R	IF
Python	Code Retrieval	175,199	86,180	-
	Code Completion	349,023	179,137	214,825
java	Code Retrieval	340,121	216,642	-
	Code Completion	683,890	447,464	709,218

with a higher threshold, encompassing up to 7,685 tokens, apt for models with an 8,192 token limit (e.g., StarCoder [71]) or 8,000 token limit (e.g., Codex [16]).

RepoBench-C is designed primarily for 0-shot learning, in order to examine the model's ability to handle long-range contexts. Despite this, we also provide a large amount of training data to allow fine-tuning, thereby enhancing the transfer capabilities of relatively smaller models, and for the test set, we allocate more data under XF-F settings compared with XF-R and IF settings. Details of this data are provided in Table 4.1. For evaluation metrics, we adopt Exact Match (EM) and Edit Similarity (Edit Sim) [121] following the previous work [84] and extend our evaluation with CodeBLEU [108] to evaluate the accuracy of the predicted code line.

RepoBench-P RepoBench-P evaluates the model's performance by combining the scenes of RepoBench-R and RepoBench-C: retrieval of relevant snippets and next-line code prediction,

presenting a challenging pipeline task. This task mirrors complex real-world scenarios that a practical auto-completion system would face, assessing the model's comprehensive performance and flexibility.

In RepoBench-P, each setting (XF-F, XF-R, and IF) requires the model to first identify the most pertinent snippets and then employ these snippets as cross-file context in conjunction with the in-file context to predict the subsequent line. Contrary to specifying a maximum token limit, we define a minimum token threshold: 12,000 for Python and 24,000 for Java, and the gold snippet retrieval process requires a minimum of 10 candidates. Due to the substantial amount of data resulting from these constraints, we opt to down-sample to ensure parity between Java and Python datasets. Details of this data are provided in Table 4.1. For evaluating the predicted next line, we also use the Exact Match, Edit Similarity amd CodeBLEU metrics, in line with the RepoBench-C setting.

4.3 Experiments

In this section, we present the detailed introduction, baselines, results and analysis of our three tasks defined above in RepoBench.

4.3.1 RepoBench-R

The primary objective of the retrieval task in RepoBench-R is to identify the most relevant code snippets to predict the next line given an in-file context. The process generally involves cropping certain lines of the in-file code before the predicted line, followed by the calculation of the degree of relevance (we use the term 'similarity' uniformly) between the cropped code and each candidate snippet. Formally, the general method for retrieval can be mathematically formulated as follows:

$$\underset{i\in\{1,\ldots,n\}}{\arg\overset{k}{\max}} f(C[-m:], S_i)$$

where C denotes the in-file code, S_i refers to the i-th candidate snippet, n is the total number of candidate snippets, m is the number of lines from the in-file code retained, k represents the top k candidates to be retrieved, and f is the function computing the similarity (or other scores) between the cropped in-file code and the candidate snippets.

Baseline. In our baseline approach, three strategies are employed for the retrieval task: (1) **Random Retrieval** involves retrieving code snippets in a random manner, serving as a lower-bound benchmark against which we can compare the effectiveness of the other retrieval methods. To ensure the stability and reliability of our results, this random process is repeated 100 times and the outcomes are averaged. (2) **Lexical Retrieval** uses Jaccard Similarity and Edit Similarity to assess the relevance between the cropped code from the in-file context and the candidate code snippets. (3) **Semantic Retrieval** applies encoder models, including CodeBERT [32] based on BERT [24], UnixCoder [39] based on UniLM [27] (we use the `Encoder-only` mode), and InstructOR [119] to obtain the code embeddings. We also include some other encoder-decoder or decoder models, include CodeGPT [84], CodeT5+ [132] and CodeGen [90]. Cosine Similarity is employed to measure the semantic similarity between the cropped code and the candidate snippets. In baseline, we crop $m = 3$ lines from the in-file code as specified in the general method ($C[-m:]$), indicating the last three lines before the line to predict. All computations for determining the similarity score are executed at the token level.

Results and Analysis. Table 4.2 presents a detailed comparison of different retrieval strategies in RepoBench-R. We have the following abservations: (1) **InstructOR Outperforms Retrieval Models, Followed by UniXcoder:** InstructOR consistently outperforms other retrieval models across tasks, with UniXcoder achieving comparable results despite having only 1/10 of InstructOR's parameters. This performance of UniXcoder can be partially attributed to its unique approach that includes multi-modal data representation learning and the utilization of both multi-modal contrastive learning (MCL) [37] and cross-modal generation tasks (CMG). (2) **Jaccard Similarity Offers a Competitive Lexical Retrieval Alternative:** Within the lexical retrieval category, Jaccard similarity has shown to be competitive, offering a viable and light-

Table 4.2. Baseline weighted average results of RepoBench-R on Python and Java retrieval tasks for *Easy* and *Hard* subset. The models used are `codebert-base` for CodeBERT, `unixcoder-base` for UniXcoder, `CodeGPT-small-py` and `CodeGPT-small-small` for CodeGPT in Python and Java respectively, and `codegen-350M-mono` and `codegen-350M-multi` for CodeGen in Python and Java respectively, `codet5p-220m` for CodeT5+ and `instructor-xl` for InstructOR.

Language	Retrieval	Model	Params.	Easy		Hard		
				acc@1	acc@3	acc@1	acc@3	acc@5
Python	Random	-	-	15.66	46.96	6.43	19.31	32.12
	Lexical	Jaccard	-	21.97	53.75	10.47	25.93	40.01
		Edit	-	18.69	50.98	7.83	21.77	36.49
	Semantic	CodeGPT	124M	16.18	47.05	8.27	22.79	36.35
		UniXcoder	125M	27.09	60.42	18.48	39.69	**54.00**
		CodeBERT	125M	16.94	48.27	6.72	19.89	33.05
		CodeT5+	220M	18.32	50.95	8.58	23.03	36.24
		CodeGen	350M	21.03	54.27	13.20	31.64	46.22
		InstructOR	1.5B	**28.22**	**62.77**	**19.10**	**39.91**	53.54
Java	Random	-	-	15.36	46.03	5.61	16.89	28.16
	Lexical	Jaccard	-	16.58	48.49	7.84	20.83	32.80
		Edit	-	15.19	45.92	6.09	17.63	28.69
	Semantic	CodeGPT	124M	16.46	48.46	7.87	22.97	37.53
		CodeBERT	125M	15.68	46.51	6.02	17.52	28.80
		UniXcoder	125M	19.61	52.96	12.23	28.74	41.88
		CodeT5+	220M	16.12	47.67	6.46	18.50	30.50
		CodeGen	350M	**20.09**	52.60	11.09	29.32	44.22
		InstructOR	1.5B	19.94	**53.91**	**13.07**	**31.28**	**44.52**

weighted alternative to semantic methods. (3) **Python Code Retrieval Tasks Yield Higher Accuracy than Java:** Tasks involving Python code retrieval tend to yield higher accuracy than those with Java, which is partially hypothesized to be due to the common Python practice of defining function arguments in close proximity to their calls, thereby providing valuable contextual cues for retrieval. In contrast, Java's extensive use of class structures may introduce additional complexities into the retrieval process.

4.3.2 RepoBench-C

The code completion task, RepoBench-C, aims to predict the next line of code based on a given in-file context (C_{in}), consisting of import statements and preceding lines before the target line, as well as a cross-file context (C_x), comprising snippets from other files parsed by import statements. This task commonly uses autoregressive language models trained on code for prediction. The formal expression of this task can be illustrated as follows:

$$P(Y) = \prod_{i=1}^{n} P(y_i | y_{<i}, C_x, C_{in}) \tag{4.1}$$

where $P(Y)$ is the joint probability of all tokens in the predicted sequence Y. The variable y_i denotes the i^{th} token in sequence Y, while $y_{<i}$ symbolizes the sequence of all preceding tokens. C_x and C_{in} represent the cross-file context and the in-file context, respectively. This product notation represents the autoregressive assumption that each token y_i is conditionally dependent on all preceding tokens $y_{<i}$ and the given contexts C_x and C_{in}.

Baseline. To establish a performance baseline, our benchmark compares 4 series of models: (1) **Codex**[4] [16] (i.e. `code-davinci-002`), developed by OpenAI, recognized for its code generation capabilities and serving as the base model for Copilot.(2) **CodeGen** [90], a family of autoregressive language models for program synthesis, available in multiple size variants (350M, 2B, 6B, 16B). We use `CodeGen-Mono` for Python and `CodeGen-Multi` for Java in our evaluations. (3) **StarCoder** [71], a 15.5B parameter model trained across over 80 programming languages. We adopt `StarCoder` for Python and `StarCoderBase` for Java. (4) **CodeLlama** [110], a family of large language models for code based on Llama 2 [124], include with 7B, 13B and 34B parameters. Similarly, we use `CodeLlama-Python` for Python and `CodeLlama` for Java.

Results and Analysis. Table 4.3 presents the updated results of RepoBench-C. Our findings on the two RepoBench-C subsets provide several insights: (1) **CodeLlama-34B Excels**

[4]Codex has been decommissioned as of January 4th, 2024 and is no longer accessible.

Table 4.3. Performance comparison of models on RepoBench-C across Python and Java, using weighted average Exact Match (EM), Edit Similarity (Edit Sim), and CodeBLEU scores from three settings, for 2k (top) amd 8k (bottom) subset.

Model	Params.	Python			Java		
		EM	Edit Sim	CodeBLEU	EM	Edit Sim	CodeBLEU
CodeGen	350M	20.71	64.21	32.60	21.21	63.62	37.18
CodeGen	2.7B	27.35	68.28	38.34	28.31	69.15	43.84
CodeGen	6.1B	31.67	70.67	42.15	29.59	70.27	44.87
CodeLlama	7B	34.10	71.24	43.46	35.80	76.75	49.98
CodeLlama	13B	36.18	72.25	45.60	36.26	75.72	49.44
StarCoder	15.5B	31.67	71.27	41.46	37.35	77.00	51.81
CodeGen	16.1B	33.41	71.20	43.58	30.45	70.29	45.91
CodeLlma	34B	**37.40**	**72.98**	**47.04**	39.41	78.52	53.56
Codex	-	31.31	72.21	41.45	**42.47**	**80.01**	**55.62**

Model	Params.	Python			Java		
		EM	Edit Sim	CodeBLEU	EM	Edit Sim	CodeBLEU
CodeLlama	7B	33.24	70.44	43.14	33.45	74.33	47.64
CodeLlama	13B	35.56	71.57	45.10	36.26	75.72	49.44
StarCoder	15.5B	29.93	68.84	40.39	32.49	74.29	46.92
CodeLlama	34B	**36.26**	**72.19**	**45.71**	36.84	76.06	50.77
Codex	-	32.13	71.89	42.27	**40.52**	**77.97**	**53.63**

in Python for Both 2k and 8k: CodeLlama-34B achieves the highest performance across all metrics for Python code generation tasks in both the 2k and 8k subsets, outperforming Codex and other models. (2) **Codex Maintains Dominance in Java for both 2k and 8k:** Despite the strong performance of CodeLlama and other models, Codex consistently emerges as the top-performing model for Java code generation tasks across both the 2k and 8k subsets. This demonstrates Codex's specialized capabilities for multilingual code generation.

4.3.3 RepoBench-P

RepoBench-P combines the retrieval and code completion tasks to form a pipeline, where the goal is to first retrieve the most relevant code snippets given an in-file context and then predict the optimal next line of code based on the retrieved snippets and the in-file context.

This pipeline approach aims to leverage the strengths of both tasks to enhance code assistance systems' capabilities. The formal expression of RepoBench-P can be represented as follows:

$$P(Y) = \prod_{i=1}^{n} P(y_i | y_{<i}, S_1, \ldots, S_k, C_{in}) \tag{4.2}$$

where $P(Y)$ denotes the joint probability of all tokens in the predicted sequence Y. y_i represents the i-th token in sequence Y, while $y_{<i}$ symbolizes the sequence of all preceding tokens. $S_1, \ldots, S_k$ refer to the retrieved code snippets, and C_{in} represents the in-file context. This product notation signifies the autoregressive assumption that each token y_i is conditionally dependent on all preceding tokens $y< i$, the given in-file context C_{in}, and the retrieved snippets $S_1, \ldots, S_k$.

Baseline. To establish a performance baseline for the end-to-end task RepoBench-P, we test Codex (code-davinci-002) as the base model. We reserve 1,600 tokens for the in-file context, with a cropping limit of 60 preceding lines. Any unused tokens from this allocation are then filled by the cross-file context, up to a total prompt size of 6,400 tokens.

For the retrieval component, we delve into several strategies for retrieving relevant snippets from the cross-file context: (1) **Gold-Only**: In cross-file completions, the cross-file context includes just the 'gold snippet'. For in-file completions, the context is left empty. (2) **Gold-Filled**: The cross-file context integrates the 'gold snippet' alongside with other randomly fetched snippets until the 6,400-token capacity is filled. Inspired by the work of [78], we employ two variant strategies for the placement of the 'gold snippet': *Gold-Filled-Head*, where the 'gold snippet' is positioned at the beginning; and *Gold-Filled-Tail*, where it is positioned at the tail-end. (3) **UniXcoder**: Using UniXcoder as cross-file context retriever, snippets are obtained based on their relevance to the cropped preceding three in-file lines while adhering to a 6,400-token limit for the input length. The includes the *UniXcoder-H2L (High-to-Low)* variant, ranking snippets from the most to least relevant, and the *UniXcoder-L2H (Low-to-High)* approach, ranking in the reverse order. (5) **Random**: Cross-file context snippets are totally randomly selected without considering their relevance until the token limit is reached. Due to the constraints imposed by

Table 4.4. Comparison of various retrieval strategies on the RepoBench-P for Python and Java using Codex [16] (code-davinci-002). Each strategy is evaluated in terms of Exact Match (EM) and Edit Similarity (ES) metrics for XF-F, XF-R, and IF settings. 'All' represents the average performance over the mixture of all test data, weighted by the size of each test setting. Strategies (*Gold-Only* and *Gold-Filled*), marked with an asterisk (*), include gold snippets for benchmarking purposes and serve only as references; they do not embody oracle capabilities.

Retrieval Method	Python			Java		
	EM	**Edit Sim**	**CodeBLEU**	**EM**	**Edit Sim**	**CodeBLEU**
Baseline	33.15	72.19	44.07	40.40	74.28	56.64
Random	34.94	73.10	45.89	40.77	74.51	56.88
Jaccard	36.46	73.66	46.76	41.48	**74.93**	**57.39**
UniXcoder-H2L	<u>36.61</u>	<u>74.02</u>	**47.28**	<u>41.70</u>	<u>74.82</u>	<u>57.29</u>
UniXcoder-L2H	**37.11**	**74.31**	<u>47.15</u>	**41.83**	**74.93**	57.12
Gold-Only*	35.79	73.58	46.55	41.62	74.95	57.09
Gold-Filled-Head*	36.07	73.67	46.90	41.59	74.88	56.99
Gold-Filled-Tail*	36.18	73.76	46.75	41.49	74.91	57.09

the codex rate limit, we are unable to perform multiple runs for the random retrieval, which is necessary to mitigate the inherent randomness; consequently, the results presented should be considered indicative and not conclusive. (6) **Baseline**: In this strategy, a token limit of 6,400 is solely allocated for the in-file context, abstaining from using any cross-file context during completion. It serves as a fundamental point of comparison, highlighting the model's performance when exclusively dependent on the in-file context.

Results and Analysis. Table 4.4 presents a comparison of various retrieval strategies using Codex in RepoBench-P. From this comparison, we present the following insights: (1) **Inclusion of Cross-file Contexts Improves Performance:** Integrating more cross-file contexts enhances performance, irrespective of retrieval quality. Even randomly selected contexts significantly boost performance, potentially by fostering contextual understanding, enabling the model to draw from a broader code repository. (2) **Effective Retrieval Enhances Performance:** Retrievers deploying specific models or methods like UniXcoder model, outperform random retrieval systems. Notably, this improvement is not confined to cross-file line prediction (XF-F

and XF-R) but is also observed in in-file next-line prediction (IF), highlighting the value of retrieving code related to current code in the same repository as cross-file contexts, even if the succeeding line does not encompass cross-file modules. (3) **Placement Order of Retrieved Code Snippets Matters:** The positioning of related code snippets influences code completion effectiveness. Positioning higher similar code snippets adjacent to or in close proximity to the line requiring completion tends to improve code completion performance.

4.4 Additional Introduction of RepoBench V1.1

In this section, we provide a description of the newest version of RepoBench v1.1[5,6] as of the writing. This version of RepoBench, also including datasets for Python and Java, was constructed from GitHub data spanning from `October 6, 2023`, to `December 31, 2023`. To further address potential concerns of data leakage and memorization, which could bias model evaluations, we also performed a deduplication process on the Stack v2 [82].

In RepoBench v1.1, to accommodate models capable of handling longer inputs and to streamline the evaluation process, we have reduced the volume of the test dataset and categorized the next-line prediction tasks into distinct levels based on their prompt lengths utilizing OpenAI's GPT-4 tokenizer for tokenization. The levels are determined by specific token ranges, including `Level 2k` (640 $\sim$ 1,600 tokens), `Level 4k` (1,600 $\sim$ 3,600 tokens), `Level 8k` (3,600 $\sim$ 7,200 tokens), `Level 12k` (7,200 $\sim$ 10,800 tokens), `Level 16k` (10,800 $\sim$ 14,400 tokens), `Level 24k` (14,400 $\sim$ 21,600 tokens), `Level 32k` (21,600 $\sim$ 28,800 tokens), `Level 64k` (28,800 $\sim$ 57,600 tokens), and `Level 128k` (57,600 $\sim$ 100,000 tokens).

RepoBench v1.1 is featured in the StarCoder 2 [82] technical report as a benchmark for evaluating the capabilities of base models for repository-level code completion. It includes evaluations at 5 levels, specifically 2k, 4k, 8k, 12k, and 16k, to accommodate the 16k sequence length for most of the current open-source models. Table 4.5 presents the performance of

[5]https://huggingface.co/datasets/tianyang/repobench_python_v1.1
[6]https://huggingface.co/datasets/tianyang/repobench_java_v1.1

Table 4.5. Average exact match (EM), edit similarity (Edit Sim), and CodeBLEU scores for open-access base models on RepoBench v1.1. (Table copy from StarCoder 2 techincal report [82])

Model	Python			Java		
	EM	**Edit Sim**	**CodeBLEU**	**EM**	**Edit Sim**	**CodeBLEU**
StarCoderBase-3B	29.99	69.37	36.77	36.01	74.18	45.30
DeepSeekCoder-1.3B	31.02	70.07	37.88	37.75	75.66	46.69
StableCoder-3B	**34.48**	**71.79**	**40.43**	**40.13**	**76.56**	**49.00**
StarCoder2-3B	32.47	71.19	39.25	38.46	76.53	47.96
StarCoderBase-7B	32.70	71.08	39.48	37.97	75.66	47.47
CodeLlama-7B	33.85	71.79	40.47	39.61	76.71	48.92
DeepSeekCoder-6.7B	**36.79**	**73.85**	**42.65**	**42.87**	**78.93**	**51.69**
StarCoder2-7B	33.72	72.07	40.34	39.84	77.23	48.96
StarCoderBase-15B	33.51	71.64	40.39	39.34	76.24	48.36
CodeLlama-13B	35.50	72.98	42.02	41.27	77.57	50.26
StarCoder2-15B	**36.99**	**74.08**	**43.25**	**42.57**	**79.05**	**51.45**
CodeLlama-34B	37.22	73.77	43.38	42.35	78.22	50.99
DeepSeekCoder-33B	**39.25**	**75.20**	**45.21**	**44.59**	**79.92**	**52.70**

open-access base models on RepoBench v1.1.

In comparison, it is noteworthy that StarCoder2 exhibits a significant improvement over StarCoder1, which can be attributed to two main factors. Firstly, the increase in context length has likely contributed to better model performance. More importantly, however, is the pre-training of StarCoder2 at the repository-level, underscoring the essential role of repository-level training in enhancing model capabilities.

Chapter 5

Rethinking Tabular Data Understanding with Large Language Models

Large Language Models (LLMs) have shown to be capable of various tasks, yet their capability in interpreting and reasoning over tabular data remains an underexplored area. In this context, this chapter investigates from three core perspectives: the robustness of LLMs to structural perturbations in tables, the comparative analysis of textual and symbolic reasoning on tables, and the potential of boosting model performance through the aggregation of multiple reasoning pathways.

5.1 Introduction

Large Language Models (LLMs; Brown et al. 12, Chowdhery et al. 20, Zhang et al. 153, OpenAI 91, 92, 94, Touvron et al. 124, 125) have revolutionized the field of NLP, demonstrating an extraordinary ability to understand and reason over rich textual data [136, 130, 159, 64, 74]. On top of LLMs' existing capabilities for NLP, further bolstering their potential for decision-making by drawing from external knowledge sources remains an exciting research frontier [88, 86, 43, 57]. Amongst such knowledge sources, tabular data serve as a ubiquitous kind due to their expressiveness for relations, properties and statistics, and their being easy to construct by human curators.

Like humans, LLMs can also benefit from reading tabular data accompanying text.

Marek Plawgo

Year	1999	2000	...	2008	2012
Competition	European Junior Championships	World Junior Championships	...	Olympic Games	European Championships
Venue	Riga, Latvia	Santiago, Chile	...	Beijing, China	Helsinki, Finland
Position	4th	1st	...	7th	18th (sf)
Event	400 m hurdles	400 m hurdles	...	4x400 m relay	400 m hurdles
Notes	52.17	49.23	...	3:00.32	50.77

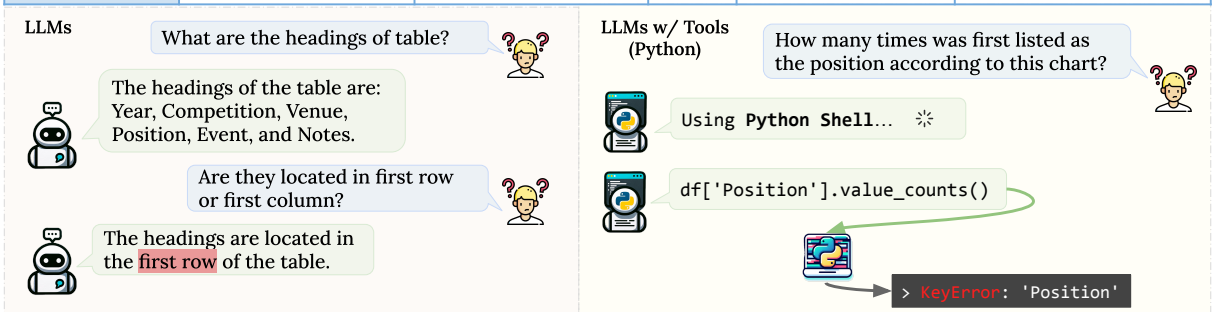

Figure 5.1. A demonstration of the challenges faced by LLMs in comprehending and interpreting table structures. In the first example, despite the LLM correctly identifying table headings, it falters in accurately determining the headings' positions within the table structure. In the second example, the model using Python Shell as an external tool erroneously defaults to interpreting headings (located in first column) as column headers, leading to subsequent mistakes in the generated code. Some logos in this and subsequent figures are generated by OpenAI's DALL-E3 [93].

However, as indicated in Figure 5.1, the structural nature of tables presents unique challenges to these models. Inherently designed to parse and process vast expanses of unstructured textual content, LLMs confront a paradigm shift when facing tabular data. Linearizing tables to suit the LLM paradigm can obscure the inherent structural and relational information, making tasks such as precise localization and complex statistical analyses. Additionally, the design variations in tables, whether 'column tables' with headers in the first row or 'row tables' with headers in the first column, further complicate the interpretation process. Beyond structural concerns, numerical reasoning and aggregation over tabular data present another layer of complexity. While LLMs excel at textual understanding, they occasionally stumble when confronted with tasks necessitating precise numerical computation within tables. Moreover, tables often present a dense amalgamation of textual or numerical data. The sheer volume and intricacy of this information can risk overshadowing crucial details, potentially impeding the LLM's decision-

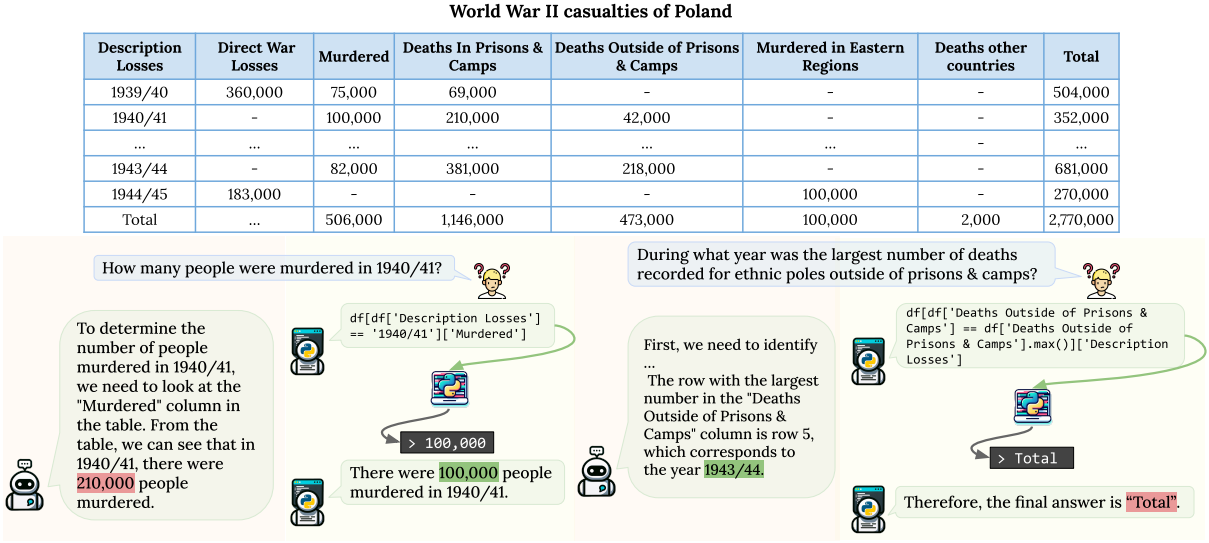

Figure 5.2. Illustrative examples sampled from the WIKITABLEQUESTIONS dataset, wherein a comparison is exhibited between textual reasoning via direct prompting and symbolic reasoning through Python Shell interactions. ***Top***: The table and its title. ***Bottom Left***: The first question example where textual reasoning erroneously interprets due to the limitation of precision localization, while symbolic reasoning accurately locates the answer with Python code. ***Bottom Right***: The second question example where textual reasoning successfully identifies the answer, but symbolic reasoning mistakenly treats the special row `total` row as the final answer.

making abilities [113].

With the emergence of instruction fine-tuning techniques [134, 22] and the application of Reinforcement Learning from Human Feedback (RLHF) [118, 35, 21], LLMs have witnessed significant enhancements in their alignment capabilities, paving the way for transitioning from few-shot to zero-shot learning settings [64]. In light of these advancements, this chapter delves deep into the the challenges and intricacies of tabular understanding and reasoning by LLMs, exemplified in Figure 5.2. We organize our exploration around three **pivotal research questions**: (1) How well do LLMs perceive table structures and how can we ensure robustness against structural variations? (2) Comparing textual and symbolic reasoning for table data in LLMs, which prevails in effectiveness, and what advantages and challenges manifest in each strategy? (3) Will the aggregation of multiple reasoning pathways enhance the accuracy and reliability of

tabular data interpretation by LLMs?

In pursuit of answering the aforementioned research questions, we conduct experiments on SOTA LLMs such as GPT-3.5 [92]. Our findings in Section 5.3 underscore that while LLMs are adept at semantically interpreting tables, their capability to resist structural variance (Section 5.3.1) and understand table structures (Section 5.3.2) is suboptimal. Motivated by these findings, we propose a table structure normalization method to enhance LLMs' resilience against structural table variations in Section 5.3.3. Intriguingly, Section 5.4.1 reveals that textual reasoning surpasses symbolic reasoning in contexts with limited table content, defying conventional conceptions of symbolic reasoning's dominance in other domains [86]. Both textual and symbolic reasoning strategies encompass different advantages and challenges, which is detailed in Section 5.4.2. To harness the unique strengths of each, we implement mix self-consistency mechanism (Section 5.5) that remarkably attains SOTA performance on Table QA, exemplifying the synergistic potential when both reasoning strategies are aggregated.

5.2 Preliminaries

This section succinctly introduces the foundational aspects of our study over structurally perturbed tabular data. Section 5.2.1 formally defines the problem, delineating the critical notations and conceptual frameworks, and Section 5.2.2 explicates our experimental setup details, elucidating dataset choice, model utilization, and evaluation strategy.

5.2.1 Problem Definition

Question answering (QA) over tabular data, commonly known as the TableQA task, is an important challenge in NLP. In this study, we targets TableQA to explore and enhance the proficiency of LLMs, in reasoning over tabular data. Additionally, we probe the robustness and adaptability of these models by introducing structural perturbations to tables.

Let $\mathscr{T}$ represent a table consisting of $\mathscr{R}$ rows and $\mathscr{C}$ columns, and τ represent its title/caption. Each cell in $\mathscr{T}$ is denoted by $\mathscr{T}_{i,j}$, where $i \in [0, \mathscr{R}-1]$ and $j \in [0, \mathscr{C}-1]$. $\mathscr{T}_{0,j}$ are

headers. Given a question $\mathscr{Q}$ pertaining to the table, our task is to identify an answer $\mathscr{A}$. This answer is generally a collection of values, denoted as $\{a_1, a_2, \ldots, a_k\}$, where $k \in \mathbb{N}^+$.

Furthermore, to delve deeper into the structural comprehension of LLMs, we introduce structural perturbations, which include:[1]

1. **Transposed Table ($\mathscr{T}^\top$):** A table obtained by converting rows to columns and vice-versa, maintaining the row and column order:

$$\mathscr{T}^\top_{i,j} = \mathscr{T}_{j,i} \quad \forall i \in [0, \mathscr{R}-1], j \in [0, \mathscr{C}-1].$$

2. **Row Shuffled Table ($\mathscr{T}_\Pi$):** A table obtained by randomly shuffling the rows (excluding the headers) with a random permutation function π, while keeping the order of columns unchanged:

$$\mathscr{T}_{\Pi_{i,j}} = \mathscr{T}_{\pi(i),j} \quad \forall i \in [1, \mathscr{R}-1], j \in [0, \mathscr{C}-1]$$

3. **Row Shuffled and Transposed Table ($\mathscr{T}^\top_\Pi$):** A table obtained by first randomly shuffling the rows (excluding headers) and then applying transposition:

$$\mathscr{T}^\top_{\Pi_{i,j}} = \mathscr{T}_{j,\pi(i)} \quad \forall i \in [1, \mathscr{R}-1], j \in [0, \mathscr{C}-1]$$

Defining our research problem more formally: our primary objective is to investigate the function, f, that can appropriately answer the posed question using the provided table. Specifically, this function will take three arguments: the table variant $\mathscr{T}' \in \{\mathscr{T}, \mathscr{T}^\top, \mathscr{T}_\Pi, \mathscr{T}^\top_\Pi\}$, its title τ, and the question $\mathscr{Q}$. It will output an answer $\mathscr{A}$. The entire problem can be formally framed as:

$$f(\mathscr{T}', \tau, \mathscr{Q}) \rightarrow \mathscr{A}, \quad \forall \mathscr{T}' \in \{\mathscr{T}, \mathscr{T}^\top, \mathscr{T}_\Pi, \mathscr{T}^\top_\Pi\}$$

[1]Column shuffling was not employed as the typical number of columns is limited and this shuffling had minimal impact on accuracy [157].

5.2.2 Experimental Setup

This section details the experimental setup adopted in our study, including the datasets employed, model selection, evaluation metrics, reasoning methods, and other details.

Dataset. We used the WIKITABLEQUESTIONS (WTQ; [98]) dataset for our experiments. The test set comprises 421 tables. Each table provides up to two question-answer pairs; if a table has fewer than two, only one was chosen, totaling 837 unique data points. With our four table configurations (original and three perturbations), the overall evaluation data points amount to $837 \times 4 = 3,348$.

Models. We employ the GPT-3.5 [92] series for our research. Given that tables usually have extensive data, depending on the prompt length, we dynamically use `gpt-3.5-turbo-0613` and `gpt-3.5-turbo-16k-0613`, with a primary aim to optimize cost when querying the API.

Evaluation Metrics. Following prior works [56, 89, 19, 145], we employ *Exact Match Accuracy* as the evaluation metric to validate predictions against ground truths, embedding instructions in prompts for consistent and parseable outputs.

Reasoning Methods. Our evaluation hinges on two distinct zero-shot reasoning approaches:

- **Direct Prompting (DP)** is a textual reasoning method that prompts LLMs to answer questions in a zero-shot manner. Rather than directly providing the answer, LLMs are instructed to reason step-by-step before concluding.

- **Python Shell Agent (PyAgent)** is a symbolic reasoning approach where the model dynamically interacts with a Python shell. Specifically, LLMs use the Python Shell as an external tool to execute commands, process data, and scrutinize results, particularly within a pandas dataframe, limited to a maximum of five iterative steps.

Other Details. Depending on the scenario, we adjust the temperature setting. In cases not employing self-consistency, we set it to 0. For scenarios involving self-consistency, the

Table 5.1. Accuracy of GPT-3.5 under different table perturbations using Direct Prompting (DP) and Python Shell Agent (PyAgent).

Perturbation	DP	PyAgent
Original ($\mathscr{T}$)	59.50	55.91
+Shuffle ($\mathscr{T}_{\Pi}$)	52.21 -12.25%	47.91 -14.31%
+Transpose ($\mathscr{T}^{\top}$)	51.14 -14.05%	12.45 -77.73%
+Transpose&Shuffle ($\mathscr{T}_{\Pi}^{\top}$)	37.51 -36.96%	8.96 -83.97%

temperature is set to 0.8. Importantly, it should be noted that all prompts are deployed in a zero-shot manner, without any demonstrations or examples.

5.3 LLM Robustness to Structural Perturbations

This section explores how LLMs interpret varied table structures in response to our first research question (Section 5.1). We probe the impact of three table perturbations on LLM performance (Section 5.3.1), uncover LLMs' challenges and limitations for direct table transposition and recoganize tranposed tables (Section 5.3.2), and introduce a structure normalization strategy (NORM) to mitigate these issues (Section 5.3.3).

5.3.1 Impacts of Table Perturbations on LLMs

In Section 5.2.1, we present three types of structural table perturbations: transposition ($\mathscr{T}^{\top}$), row-shuffling ($\mathscr{T}_{\Pi}$), and their combination ($\mathscr{T}_{\Pi}^{\top}$). As demonstrated in Table 5.1, both reasoning methods, DP and PyAgent, exhibit significant performance declines, with more pronounced when transposition is applied. DP consistently outperforms PyAgent largely across perturbations, indicating that textual reasoning tends to be more resilient to these structural changes. This resilience can be attributed to LLMs' ability to grasp semantic connections and meanings irrespective of structural shifts. In contrast, symbolic reasoning, exemplified by PyAgent, is heavily reliant on table structure, making it more vulnerable, especially to

Table 5.2. Evaluation results of GPT-3.5 on the 421 distinct tables of WTQ dataset, including three tasks: *Transposer* involving switching between original ($\mathscr{T}$) and transposed tables ($\mathscr{T}^\top$), *Detector* for identifying need for table transposition (0 for no transposition, 1 for transposition required), and *Determinator* to choose probable table headings either from the first row ($\mathscr{T}_{0,*}$) or the first column ($\mathscr{T}_{*,0}$).

LLMs As	Task Description	Accuracy
Transposer	$f(\mathscr{T}) \rightarrow \mathscr{T}^\top$	53.68
	$f(\mathscr{T}^\top) \rightarrow \mathscr{T}$	51.07
Detector	$f(\mathscr{T}) \rightarrow 0$	93.35
	$f(\mathscr{T}^\top) \rightarrow 1$	32.54
Determinator	$f(\mathscr{T}, \mathscr{T}_{0,*}, \mathscr{T}_{*,0}) \rightarrow \mathscr{T}_{0,*}$	97.39
	$f(\mathscr{T}^\top, \mathscr{T}_{0,*}, \mathscr{T}_{*,0}) \rightarrow \mathscr{T}_{*,0}$	94.77

transposition.

5.3.2 Limitations of Table Transposition with LLMs

To better understand LLMs' capabilities with regards to table structures, we investigate their ability on detecting tables in need of transposition and performing table transposition.

LLMs as Transposition Detectors. Given a table $\mathscr{T}$, the goal is to detect whether a table should be transposed for better comprehension by LLMs. This is formulated as a binary classification task:

$$f(\mathscr{T}) \rightarrow 0, \quad f(\mathscr{T}^\top) \rightarrow 1,$$

Where 0 denotes 'no need of transposition' and 1 indicates 'transposition needed'. Table 5.2 shows the results. GPT-3.5 correctly classified 93.35% of original tables $\mathscr{T}$ as not requiring transposition. However, its accuracy dramatically decreased to 32.54% on transposed tables $\mathscr{T}^\top$. Our observations highlight that LLMs suffer from structural bias in the interpretation of table orientations, predominantly leading to recommendations against transposition.

LLMs as Table Transposers. The objective is to switch between original and transposed table formats. Specifically, the goal is to directly yield $\mathscr{T}^\top$ given $\mathscr{T}$, and vice versa. Formally,

the task is:

$$f(\mathscr{T}) \rightarrow \mathscr{T}^{\top}, \quad f(\mathscr{T}^{\top}) \rightarrow \mathscr{T}$$

We observed that GPT-3.5's proficiency in this task is limited, with an accuracy of 53.68% transposing row tables and 51.07% for the inverse operation, suggesting that LLMs can not transpose tables precisely.

5.3.3 Table Structure Normalization

In addressing structural variations in tables, our goal is to ensure consistent interpretation and utility across diverse table structures. To normalize various table structures into well-ordered row-tables prior to downstream tasks, we introduce NORM, which is a two-stage normalization strategy: the first stage detects column-tables and transposing them into row-tables, while the second stage sorts the row-tables for enhanced comprehensibility. Through this approach, NORM accommodates for structural perturbations without compromising the understanding of the standardized row-tables.

Content-Aware Transposition Determination In the straightforward methods mentioned in Section 5.3.2, LLMs are affected by the loss of structure information of the table. Our approach aims to reduce this structural dependence by introducing a content-aware determination process, which leverages the semantic reasoning capabilities of LLMs, instead of perceiving the table's structure. Specifically, we analyze the inherent content within the first row ($\mathscr{T}_{0,*}$) and the first column ($\mathscr{T}_{*,0}$) of a given table ($\mathscr{T}$) to decide which is more semantically fitting to serve as the table's heading. This content-aware approach can be mathematically modeled as:

$$\begin{cases} f(\mathscr{T}, \mathscr{T}_{0,*}, \mathscr{T}_{*,0}) \rightarrow \mathscr{T}_{0,*} \\ f(\mathscr{T}^{\top}, \mathscr{T}_{0,*}, \mathscr{T}_{*,0}) \rightarrow \mathscr{T}_{*,0} \end{cases}$$

Here, a selection of the first row suggests that the current table structure is preferred, whereas opting for the first column signifies a need for transposition. Results in Table 5.2 highlight

Table 5.3. Accuracy of GPT-3.5 under different table perturbations for Direct Prompting (DP) and Python Shell Agent (PyAgent) with NORM applied.

Method	$\mathscr{T}$	$\mathscr{T}_{\Pi}$	$\mathscr{T}^{\top}$	$\mathscr{T}_{\Pi}^{\top}$
DP	59.50	52.21	51.14	37.51
+NORM	58.66 -1.41%	58.66 +12.35%	58.30 +14.00%	57.71 +53.85%
PyAgent	55.91	47.91	12.43	8.96
+NORM	56.87 +1.72%	57.11 +19.20%	55.44 +346.02%	55.08 +514.73%

capability of GPT-3.5 in discerning table headings semantically, with accuracies of 97.39% and 94.77% respectively for original table and tranposed table.

Row Reordering. Upon transposition, our next objective is to ensure the logical coherence of the table data through reordering the rows. We instruct LLMs to suggest improved reordering strategies.

Due to the subjective nature involved in identifying the most suitable order of a tabular data, and given that there are no widely recognized standards for this process, the effectiveness of the proposed sorting strategy will be evaluated based its downstream impact on the results of table QA task. We notice that when the entire well-ordered table is exposed, GPT-3.5 occasionally suggests alternative sorting strategies, leading to unnecessary complexity. To counteract this tendency and ensure a better sorting proposal, we strategically present the model with only the first three and the last three rows of the table. This selective exposure typically allows the model to discern logical ordering patterns without being influenced by existing table configurations.

Table 5.3 underscores the efficacy of NORM when applied prior to the two reasoning methods – DP and PyAgent. Demonstrably, NORM robustly mitigates structural perturbations, optimizing table comprehensibility for LLMs. The results illustrate that applying NORM does not detrimentally affect the original results ($\mathscr{T}$), and it effectively refines perturbated data, aligning the outcomes closely with the original results, and in some instances, even showing slight improvement. This suggests that NORM as a preprocessing step for preparing tabular data

can enhance robust analysis by LLMs.

In addressing our initial research question, the analysis indicates that **LLMs' performance is sensitive to table structural variations**, with significant struggles observed in accurately interpreting the same tabular content under transposition and shuffling. While **textual reasoning demonstrates some resilience** to structural variations, **symbolic reasoning is significantly impacted**, particularly with transposed tables. **The NORM strategy effectively navigates these challenges** by eliminating dependency on table structures, providing consistent interpretation across diverse table structures without compromising the integrity or meaning of the original content.

5.4 Comparing Textual and Symbolic Reasoning

In this section, we delve into the comparison of textual and symbolic reasoning methods in LLMs for tabular data understanding (Section 5.4.1), further conducting a detailed error analysis (Section 5.4.2) to address the second research question (Section 5.1). We evaluate the performance of each reasoning strategy using GPT-3.5, shedding light on their strengths and challenges. In Section 5.3.3, we explored NORM to mitigate structural perturbations, enhancing generalized LLM performance and successfully restoring perturbed tables to accuracy levels similar to their original states. Therefore, subsequent analyses will exclusively consider the original tables ($\mathscr{T}$).

5.4.1 Results

Table 5.4 showcases the performance of GPT-3.5 when employed for both direct textual reasoning using DP and interactive symbolic reasoning using PyAgent. By instructing the model with the CoT [136] reasoning strategy to *think step by step, and then give the final answer*, we can achieve an accuracy of 58.66%. This surpasses the StructGPT's Iterative Reading-

[2]For SC, results are derived by conducting an average over 100 shuffles to accommodate instances of ties during majority voting. In the Mix-SC method, DP-derived answers are prioritized over PyAgent due to DP's observed superior performance. All experiments regarding SC follow this.

Table 5.4. Performance on the sampled WTQ dataset ($\mathscr{T}$). ★ denotes methods based on Codex, while ♠ represents those based on GPT-3.5. The term SC refers to self-consistency.[2] NORM w/o RESORT means that reordering stage for NORM is not performed.

Method	Accuracy (%)
Few-shot Prompting Methods	
BINDER★ [19]	63.61
BINDER♠ [19]	55.07
DATER w/o SC★ [145]	61.75
DATER w/ SC★ [145]	68.99
Zero-shot Prompting Methods	
STRUCTGPT♠ [55]	51.77
NORM+DP♠	58.66
NORM+PYAGENT♠	56.87
NORM+PYAGENT-OMITTED♠	52.45
NORM+DP&PYAGENT w/ EVAL♠	64.22
DP w/ SC♠	66.39
+NORM♠	64.10
+NORM w/o RESORT♠	66.99
PYAGENT w/ SC♠	61.39
+NORM♠	63.77
+NORM w/o RESORT♠	62.84
DP&PYAGENT w/ MIX-SC♠	73.06
+NORM♠	72.40
+NORM w/o RESORT♠	73.65

then-Reasoning method, which concentrates reasoning tasks by continually collecting relevant evidence. For tables with limited tokens, symbolic reasoning via PyAgent offers an accuracy of 56.87%, which is slightly behind the accuracy by DP in a single attempt. A distinct advantage of symbolic reasoning is its ability to only present parts of the table in the prompt. As our experiments revealed, after excluding the central rows and showcasing only the initial and final three rows, we manage to maintain an accuracy of 52.45% with a 4.42% drop compared to the full-table PyAgent results. This makes it possible to deal with larger tables with numerous rows using LLMs with limited context window. In the following sections, we will present a comprehensive analysis of the discrepancies and errors observed across these methods.

Table 5.5. Error types of DP and PyAgent methods. †This does not imply that PyAgent does not make *table interpretation* errors; these are included under *coding errors* to avoid overlapping. Note that the percentages for each reasoning method might not sum up to 100%; the remaining percentage points are attributable to *other errors*, such as problems with dataset labeling, which are not categorized here.

Error Types	DP	PyAgent	Description
Table Misinterpretation	42%	-†	LLMs incorrectly interpret the content in tables.
Coding Errors	-	38%	LLMs produce inaccurate code, typically due to issues with minor details.
Misalignment Issue	24%	28%	Outputs are conceptually correct but the answers do not align with the instructions.
Logical Inconsistency	20%	10%	LLMs exhibit failures in reasoning, leading to contradictions or inconsistencies.
Execution Issue	-	12%	Issues emerge related to the execution of Python code.
Resorting Issue	10%	8%	The resorting stage in NORM changes the answers of some sequence-dependent questions.

5.4.2 Error Analysis

To elucidate the challenges and limitations of DP and PyAgent, this section presents an in-depth error analysis by sampling 50 erroneous outputs for each. Table 5.5 summarizes the predominant error types for DP and PyAgent methods. *Table interpretation errors* significantly afflict the DP method, comprising 42% of its total errors, highlighting substantial challenges for LLMs in accurately interpreting table data. PyAgent primarily struggles with *coding errors*, constituting 38% of its total errors. These errors either originate from misunderstandings of table content, often overlooking subtle details, or manifest as inherent deficiencies in coding capabilities. These prevalent errors underscore the intrinsic challenges and limitations each method faces in the reasoning process.

In response to the second research question, the analysis indicates **DP marginally surpasses PyAgent within single attempts**. Despite this, PyAgent can handle larger tables by processing partial table views. Notably, **DP encounters difficulties in accurate table**

interpretation, while **PyAgent reveals instability in coding capabilities**.

5.5 Reasoning Aggregation

This section examines how combining multiple reasoning pathways can boost LLMs' accuracy in interpreting tabular data, which is in response to the third research question (Section 5.1).

5.5.1 Methods

Self-Consistency. Previous work has highlighted the advantages of generating multiple outputs from LLMs and adopting the most frequent answer, a mechanism known as self-consistency (SC; [130]). Table 5.4 showcases the notable improvements realized through self-consistency (aggregating 10 outputs), with DP achieving an accuracy of 64.84% and PyAgent attaining 63.49%.

Self-Evaluation. Based on our error analysis in Section 5.4.2, different reasoning methods excel at specific tasks. For instance, symbolic reasoning tends to outperform textual reasoning in counting and column localization tasks. To optimize the choice between these methods, we strategically prompt the LLMs, which avoids directly validating answers against tables but guides the LLM to choose between the two reasoning approaches based on the question's nature and each answer's clarity. By weighing the problem against the known strengths and weaknesses of each reasoning strategy, this tactic mitigates potential bias towards textual reasoning by LLMs and enhances answer accuracy. As evidenced by Table 5.4, using self-evaluation boosts accuracy to 64.99%. Impressively, this method, using only two reasoning paths, matches the performance of using 10 paths of DP or PyAgent independently.

Mix Self-Consistency. According to Section 5.4.1, symbolic and textual reasoning exhibit distinct focuses but deliver similar performance. Consequently, we introduce *Mix Self-Consistency*, a method that selects a predetermined number of outputs for each type of inference, aiming for self-consistency. This approach hinges on the idea that multiple outputs can reflect the

confidence levels of LLMs in answer generation. In scenarios where LLMs are less proficient, they tend to produce a diverse set of answers. Conversely, for tasks that LLMs handle adeptly, consistent answers are often generated across multiple reasoning attempts, converging towards one answer. Such convergence allows for the aggregation of model outputs that align with areas where LLMs exhibit stronger reasoning capabilities, thereby substantially improving accuracy. The detailed mechanics of how this approach is operationalized within the framework of *Mix Self-Consistency*, including the aggregation and interpretation of these outputs. Table 5.4 demonstrates that using mix self-consistency (generating 5 outputs per inference type,[3] totaling 10) enhances performance substantially, achieving an impressive accuracy of 72.40%, which achieves SOTA performance on the sampled WTQ data.

5.5.2 Overall Evaluation

To evaluate our method thoroughly, we conduct a comprehensive pass of testing using the complete WTQ test set, integrating both NORM and Mix self-consistency mechanisms. Since re-sorting may change the answers of row index-related questions, we perform NORM without resorting in this evaluation.[4] However, it is noteworthy that re-sorting can be advantageous for questions not dependent on row indexes, particularly when dealing with tables that are initially unorganized or messy.

As illustrated in Table 5.6, our proposed method exhibits outstanding efficacy with an accuracy of 73.6%, significantly outperforming existing models to achieve SOTA performance on the complete WTQ test set. Importantly, our approach is conducted in a fully zero-shot manner. For a detailed analysis of how table size impacts method performance, see Section 5.6.

In response to the third research question, our findings reveal that **reasoning path**

[3]The choice of generating 5 outputs per inference type (5+5) is a hyperparameter selection influenced by the dataset's distribution. We conducted an ablation study regarding this in Secrion 5.7.1. We use an equal split (5+5) based on observed comparable performance between the two reasoning strategies.

[4]Originally, the NORM process included a re-sorting step to counteract the row-shuffling perturbation. However, re-sorting may inadvertently alter answers reliant on the initial sequence, as explored in the error analysis (Section 5.4.2).

Table 5.6. Comparison of various methods on all test data of WTQ. ★ denotes methods based on the GPT-3.5 [92]; ♠ denotes methods based on the Codex [91].

Method	Accuracy (%)
Fine-tuning Based Models	
TAPAS [47]	48.8
T5-3B [140]	49.3
TAPAX [80]	57.5
REASTAP [156]	58.7
OMNITAB [56]	63.3
LLMs Based Methods	
STRUCTGPT★ [55]	48.4
BINDER★ [19]	55.5
BINDER♠ [19]	64.6
LEVER♠ [89]	65.8
DATER♠ [145]	65.9
Ours★	**73.6**

aggregation significantly enhances LLMs' accuracy in table reasoning tasks. Notably, **the *Mix Self-Consistency* method achieves an accuracy of 73.6% on the WTQ dataset, surpassing the previous SOTA by a considerable margin**. The *Self-Evaluation* strategy also contributes to this remarkable performance by adeptly selecting between reasoning approaches.

5.6 Additional Results on impact of Table Size on WTQ Performance

This section presents an analysis of the impact of table size (quantified by row numbers) on the performance of different methods when applied to the WikiTableQuestions benchmark. Specifically, we examine how the average accuracy of DP, PyAgent, and the combination by applying mix self-consistency is affected by the number of rows in a table.

To systematically evaluate the impact, we segmented the row numbers into 10 ranges, each containing approximately 430 data points, and calculated the average accuracy within these intervals. Figure 5.3 visualizes the average accuracy across these ranges for each method. It

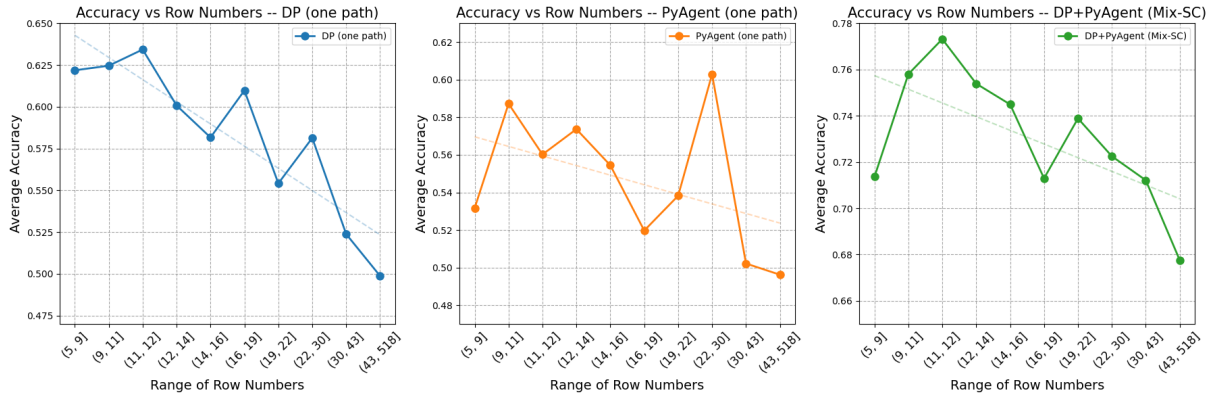

Figure 5.3. The impact of table size on the accuracy of DP, PyAgent, and Mix-SC on the all the test set of WIKITABLEQUESTIONS. The x-axis represents the row number ranges, and the y-axis shows the average accuracy for each method.

is evident that there is a shared trend of diminishing accuracy as the number of rows increases. This observation suggests that all methods are subject to decreased efficacy in the context of long tables.

The decline in performance with larger tables can be attributed to the complexity of handling long-context data and the abundance of potentially interfering information. This complexity often results in an increased error rate. The insights gained from this analysis point towards a need for the development of better symbolic methods for handling long tables, which might be capable of effectively narrowing down the scope of larger tables, either by selective attention to relevant segments or by intelligently summarizing the data, to mitigate the challenges posed by long-context information.

5.7 Additional Analysis of Mix Self-Consistency

5.7.1 Ablation Study of Output Selection

This section presents an ablation study conducted to elucidate the effect of various combinations of DP and PyAgent outputs on the performance of the *Mix Self-Consistency* method. For this experiment, we systematically explored different combinations while keeping the total output count constant at ten. Each combination was tested 100 times through random

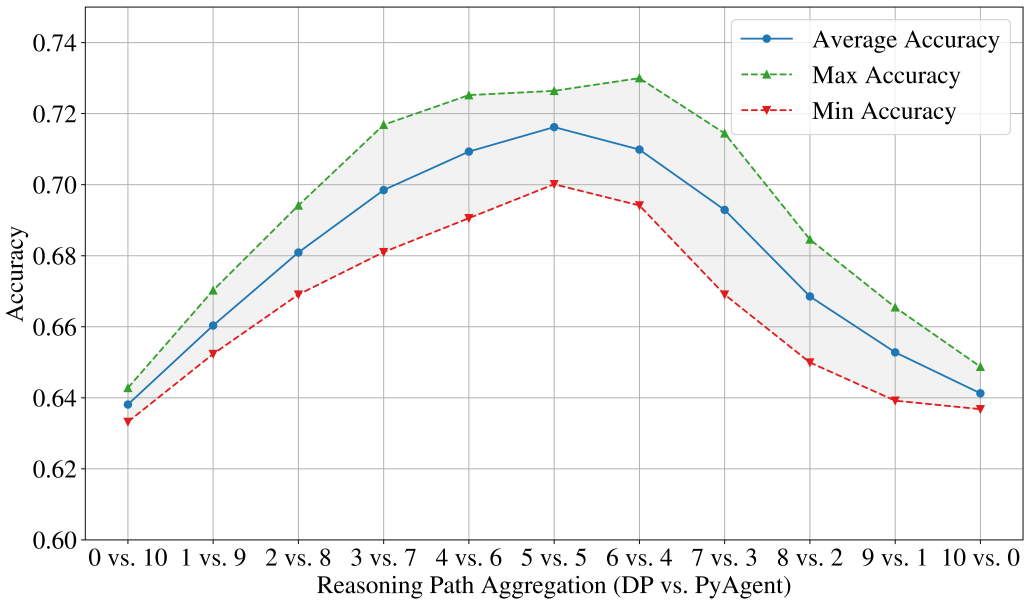

Figure 5.4. Accuracy results for the *Mix Self-Consistency* method applied to the sampled WTQ dataset, with varying combinations of DP and PyAgent outputs (depicted as DP vs. PyAgent on the x-axis). The combinations range from 10 DP vs. 0 PyAgent to 0 DP vs. 10 PyAgent. Each data point represents the maximum, minimum, and average accuracies obtained from 100 tests per combination, conducted using random sampling. Note that for the 10 DP vs. 0 PyAgent and 0 DP vs. 10 PyAgent combinations, there is no random sampling of paths. However, variance is observed due to the presence of multiple equally probable answer sets generated by the 10 paths, leading to different possible selections of answers even without sampling, thereby introducing randomness into the results.

shuffling. For each test, maximum, minimum, and average accuracies were recorded.

Figure 5.4 shows the results of the ablation study. The 5+5 combination (5 DP + 5 PyAgent) consistently gives the highest minimum and average accuracies among all tested combinations, making it a robust and reliable choice for this task. The 4+6 combination (4 DP + 6 PyAgent) secured the highest maximum accuracy in our tests.

Through this ablation study, we aim to provide insights into how different output selections influence the effectiveness of the *Mix Self-Consistency* method. Importantly, the choice of output combination should be considered as a hyperparameter that is intimately related to the distribution of the dataset being used. Given that different reasoning strategies exhibit unique

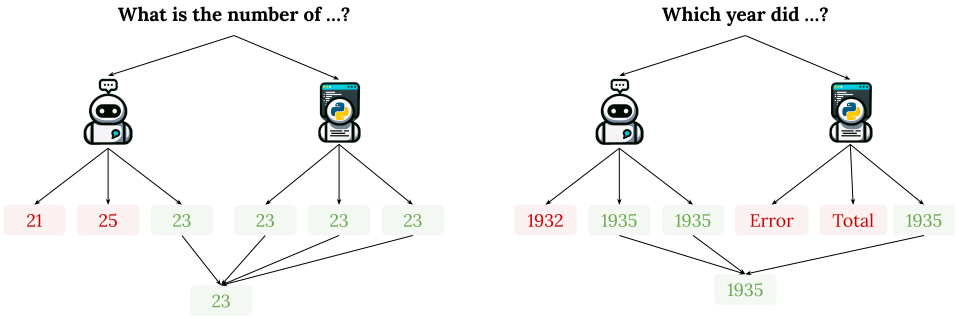

Figure 5.5. An illustration of *Mix Self-Consistency* by aggreagting outputs from multiple reasoning methods to form a unified, high-confidence prediction..

strengths and weaknesses, it is crucial to tailor the output combination to align with the characteristics of the specific tasks and datasets in question, thereby maximizing the performance of the *Mix Self-Consistency* method.

5.7.2 Mechanics of Mix Self-Consistency in Output Selection

The effectiveness of the *Mix Self-Consistency* method in achieving high accuracy largely stems from its ability to harness the strengths of different reasoning methods. Intuitively, the multiple outputs from certain reasoning method can be interpreted as the confidence score for the generated answers. In scenarios where a method excels, its outputs often tend to converge towards a common answer, signifying higher confidence and reliability. In contrast, a method less suited to the problem at hand tends to produce more diverse results, indicative of a lower level of confidence. By aggregating these outputs from different methods and applying majority voting, the *Mix Self-Consistency* method refines these variations into a more accurate prediction. As shown in Figure 5.5, This process leverages the strengths of the employed reasoning methods, thereby enhancing overall performance.

Chapter 6

Related Work

6.1 Related Work for ToolkenGPT

Fine-tuning LLMs to use tools. Early research relied heavily on fine-tuning to augment LMs with tools. In these works, LMs were mostly fine-tuned to use one or a few tools in a specific domain. For example, the retriever has been a crucial tool for augmenting LLMs with external knowledge sources [147]. The prominent works in this line include REALM [42], RAG [67], and RETRO [11]. More recently, WebGPT [87] fine-tuned GPT-3 on human web search behaviors to learn how to use the web browser. With the advancements in LLMs, there has also been growing interest in tuning these models on a collection of general tools, including the QA model, calculator, translator, etc. Example works include TALM [97] and Toolformer [111]. However, LLM fine-tuning is costly and these tuned LLMs struggle to generalize to emergent or updated tools. ToolkenGPT learns lightweight toolken embeddings for new tools, without any gradient calculation for the parameters of LLMs. This enables efficient adaption to new tools and maintains a minimal GPU memory overhead for training toolken embeddings, at a cost similar to LLM inference.

In-context learning for tools. LLMs exhibit a strong in-context learning ability [12], which becomes a prevalent method to use tools by showing tool descriptions and demonstrations in context [86, 102]. Building on this idea, reasoning chains can be incorporated to tackle more complex problems [144, 61, 96]. This paradigm has given rise to popular industry products such

as ChatGPT plugins and Langchain [15], along with many successful applications in important research topics. For instance, a code interpreter can effectively address the LLM's shortcomings in symbolic operations [18, 36, 45, 85, 141, 76]. Furthermore, by calling "tools" that have an effect on the virtual or physical world, the LLM is capable of guiding embodied agents to accomplish various household tasks [51, 1, 52, 117, 53]. Recent attempts to utilize LLMs as a controller to coordinate multiple neural models also achieve promising progress in multimodal reasoning tasks [112, 83]. Nevertheless, all methods based on in-context learning suffer from inferior performance in complex scenarios, where the tools are unfamiliar or numerous. One concurrent work, Li et al. [70] propose to retrieve the tools based on the text embedding of their documents, which may mitigate that issue. However, ToolkenGPT is fundamentally different from their method, in that the toolken embeddings can encode the implicit semantics of tools from extensive demonstrations, which can never be inferred from the surface text (A concrete example is shown in Figure 3.3). Also, note that ToolkenGPT is compatible with the recent advanced prompting techniques, e.g., Chain-of-Thought (CoT) [135], to improve the LLMs performance further.

Efficient tuning of large language models. Adapting pre-trained frozen LLMs efficiently to new tasks is an active research area, leading to a surge of interest in parameter-efficient fine-tuning (PEFT) methods [60, 73, 25, 81, 79]. The idea is to only fine-tune a small subset of parameters of the LLM while freezing most of its parameters, which bears similarity to our toolken embedding method. Which part of parameters to tune is the key to PEFT methods; for instance, Adapters [49] insert trainable layers, BitFit [149] tunes the bias parameters, prompt tuning [66, 133] appends parameters to the input embedding layer, and LoRA [50] learns low-rank matrices within specific dense layers, etc. However, existing PEFT methods have not proven suitable for efficient tool learning, and utilizing these methods on tool demonstrations may not efficiently capture the desired tool knowledge as ToolkenGPT does. To the best of our knowledge, we are the first to explore efficient tuning methods for predicting tools as tokens for tool learning of massive tools.

6.2 Related Work for RepoBench

LLMs for Code Completion Code completion, also referred to as auto-completion or intelligent code completion, is an essential feature provided by many modern Integrated Development Environments (IDEs) and code editors. It aids programmers in writing code more efficiently by predicting and automatically completing the next line or multiple next lines. The inception of Language Models (LMs) in code completion can be traced back to the usage of n-gram based LMs [126, 48], RNN models [138], and probabilistic grammar-models [9, 106, 46], which laid the foundation for the subsequent introduction of more advanced LMs in this field. With the advent of transformer-based models [128, 24, 103, 104, 12], decoder-only models trained on large-scale code datasets have been proposed to foster the advancements in code completion. For instance, GPT-C [120] and CodeGPT [84] following the underlying architecture of GPT-style models are pre-trained on vast amounts of code. UniXCoder [39] and CugLM [77] incorporates multi-task learning strategies, and leverages code structures to enhance pretraining. More recent LLMs, including Codex [16], PolyCoder [142], CodeGen [90], In-Coder [33], CodeGeeX [158], SantaCoder [3], StarCoder [71, 82], LongCoder [40], CodeLlama [110] and DeepSeekCoder [41] employ billions of parameters and excel in code generation tasks, benefiting from large-scale, high-quality code corpora. The scope of code completion has expanded with works like RLPG [114], CoCoMIC [26], and RepoCoder [152], emphasizing the integration of in-file and cross-file contexts and the importance of specialized benchmarks for evaluating repository-level code autocompletion systems.

Code Completion Datasets The task of code completion serves as a foundation for programming language models and plays a pivotal role in intelligent code completion systems. While public benchmarks like CodeXGLUE [84] with datasets *PY150* [105] and *Github Java Corpus* [4] play a key role in evaluating models within single-file contexts, they may not fully encapsulate the intricacies of real-world coding projects which often entail cross-file interactions. To address this, Ding et al. [26] proposed CoCoMIC, a model for cross-file completion and a

code completion dataset with retrieved cross-file context. Different from the CoCoMIC data, our benchmark extends beyond code completion and includes evaluation of retrieval and pipeline construction, thus can better capture the complexity of such cross-file code completion systems. RepoEval by Zhang et al. [152] serves as a project-oriented benchmark, focusing on 16 selected Python repositories to simulate real-world coding environments. However, its limitation arises from being integrated into the training data of StarCoder. RepoBench not only spans a wider range of repositories across Python and Java, but also offers a segmented evaluation into retrieval, completion, and end-to-end tasks.

Transitioning from file-based to repository-level code completion not only offers a more realistic representation of practical coding scenarios but also serves as a platform for evaluating the transfer learning capabilities of language models, as most models are not initially pre-trained with cross-file contexts included. This shift also introduces the challenge of handling longer prompts, a situation less common in single-file contexts, and a known limitation of many Transformer-based models. Recent research on long-range transformers [148] has shown promise in handling long sequences, with notable contributions from initial works like LongFormer [7] and Reformer [62], as well as more recent advancements like CoLT5 [2], UnlimiFormer [8], and Claude-100k [99], which has demonstrated their potential in effectively processing and generating code with much more cross-file context included.

6.3 Related Work for Tabular Data Reasoning

PLMs for Tabular Data Processing. Tabular reasoning presents unique challenges due to the fusion of free-form natural language questions with structured or semi-structured tabular data, for which PLMs jointly trained on tables and text are developed in the past few years, including TaBERT [146], TaPas [47], TAPEX [80], ReasTAP [156], and PASTA [38]. The recent development of TableLlama [154], an open-source model excelling in a variety of table-based tasks, adds a new dimension to the field. Despite these advancements, recent studies have

identified generalization issues under table perturbations [157, 14], raising concerns regarding the robustness of PLMs. Specific efforts like LETA [157] and LATTICE [129] have investigated and mitigated the vulnerabilities related to structural perturbations of tabular data, like row/column shuffling and table transpose, through various techniques, including data augmentation and order-invariant graph attention. However, these approaches require whitebox access to the models, limiting their applicability to SOTA LLMs with only blackbox accessibility, a limitation directly addressed in this work.

Tabular Data Processing with LLMs. Recent advancements in LLMs, notably within few-shot learning, have demonstrated their potential for tabular reasoning. Chen [17] leveraged the Chain-of-Thought (CoT) technique [136] to illustrate LLMs' effectiveness in this domain. Building upon CoT, Cheng et al. [19] and Ye et al. [145] introduced frameworks that incorporate symbolic reasoning for improved comprehension, with Ye et al. emphasizing their ability to adeptly decompose both evidence and questions. The advent of aligned models, such as ChatGPT, has enabled zero-shot table reasoning. However, these models often lack sensitivity to table structures, struggling with structural perturbations. StructGPT [55], while introducing a promising framework for LLMs to efficiently engage with structured data, has its effectiveness limited by not integrating symbolic reasoning, a critical aspect for enhancing the full capabilities of LLMs in tabular reasoning, which is the focal point of this study. Furthermore, while programming-based approaches can mitigate some challenges, they are limited in addressing free-form queries, creating a gap in the landscape. Innovations like AutoGPT [116] have sought to address this, spawning the development of tabular agents like LangChain [15], SheetCopilot [69], and DataCopilot [155]. These agents offer solutions unattainable through conventional programming but still require rigorous evaluation in various scenarios. In our study, we delve into addressing these challenges for enhancing LLMs' reasoning capabilities within structural perturbations, hence providing insights that facilitate improved accuracy in the current context.

Chapter 3 and 6, in part, is a reprint of the material as it appears in "ToolkenGPT: Augmenting Frozen Language Models with Massive Tools via Tool Embeddings." by Shibo Hao,

Tianyang Liu, Zhen Wang and Zhiting Hu, which was published at *Conference on Neural Information Processing Systems (NeurIPS)*, 2023. This thesis author was the co-primary investigator and author of this paper.

Chapter 4 and 6, in part, is a reprint of the material as it appears in "RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems." by Tianyang Liu, Canwen Xu and Julian McAuley, which was published at *The International Conference on Learning Representations (ICLR)*, 2024. This thesis author was the primary investigator and author of this paper.

Chapter 5 and 6, in part,is a reprint of the material as it appears in "Rethinking Tabular Data Understanding with Large Language Models." by Tianyang Liu, Fei Wang and Muhao Chen, which was published at *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2024. This thesis author was the primary investigator and author of this paper.

Chapter 7

Conclusion and Future Work

In summary, this thesis encompasses three works, focusing on augmenting and evaluating LLMs in the realm of NLP. The first work introduces ToolkenGPT, a novel method for integrating external tools into LLMs to enrich their functionality and adaptability. The second work presents RepoBench, a benchmark designed to assess the proficiency of LLMs in handling repository-level code auto-completion tasks. The third work delves into the challenges and intricacies of tabular data understanding and reasoning by LLMs, exploring how they interpret varied table structures and the effectiveness of textual versus symbolic reasoning. Together, these works contribute to a deeper understanding of LLM capabilities and limitations, while proposing innovative approaches to augment their utility in complex tasks.

Looking ahead, I have following questions regarding the future work:

1. **How Can We Rethink Benchmark Evaluation for More Authentic Insights?** As we strive to understand the true capabilities of Large Language Models (LLMs), the question arises: Are our current benchmarks truly reflective of real-world user experiences? The current benchmarks, while informative, may focus on aspects that are easier to measure but not entirely representative of real-world user experiences. This can lead to a skewed perception of model performance. Future efforts should aim to develop benchmarks that better capture long-form generation and long-context generation tasks, providing a more holistic view of LLM capabilities.

2. **What Strategies Can Maximize Learning from High-Quality Data?** The cornerstone of any high-performance model is the quality of its training data. The challenge lies in not just collecting vast amounts of information, but in ensuring the relevance, diversity, and accuracy of the data. This involves sophisticated data collection strategies, rigorous deduplication processes, and thorough cleaning procedures to eliminate noise and irrelevant information. Future research should delve into innovative methods for data curation that prioritize these aspects, enhancing the model's ability to learn efficiently and effectively from the best possible data sources.

3. **Can We Achieve Superior Model Performance with Smaller Sizes?** In an era where bigger often equates to better in the realm of LLMs, the challenge ahead lies in defying this norm by pioneering "anti-scaling" techniques. The objective is to cultivate models that maintain, or even surpass, the efficacy of their larger counterparts while being more compact and versatile. This approach promises to democratize access to cutting-edge LLMs, enabling a wider range of applications and users to harness the power of advanced AI without prohibitive resource requirements.

4. **What New Frontiers Await LLM Applications?** The potential applications for LLMs are expanding rapidly, opening doors to unexplored territories. From revolutionizing healthcare diagnostics and personalized education to innovating in creative industries, the possibilities are vast. Future endeavors should be bold, venturing into new areas where LLMs can provide significant value, and exploring innovative use cases that could redefine industries.

Bibliography

[1] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can, not as i say: Grounding language in robotic affordances, 2022.

[2] Joshua Ainslie, Tao Lei, Michiel de Jong, Santiago Ontañón, Siddhartha Brahma, Yury Zemlyanskiy, David Uthus, Mandy Guo, James Lee-Thorp, Yi Tay, Yun-Hsuan Sung, and Sumit Sanghai. Colt5: Faster long-range transformers with conditional computation, 2023.

[3] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. Santacoder: don't reach for the stars!, 2023. URL https://arxiv.org/abs/2301.03988.

[4] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216, 2013. doi: 10.1109/MSR.2013.6624029.

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

[6] Razvan Azamfirei, Sapna R Kudchadkar, and James Fackler. Large language models and the perils of their hallucinations. *Critical Care*, 27(1):1–2, 2023.

[7] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *ArXiv preprint*, abs/2004.05150, 2020. URL https://arxiv.org/abs/2004.05150.

[8] Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew R. Gormley. Unlimiformer: Long-range transformers with unlimited length input, 2023.

[9] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2933–2942. JMLR.org, 2016. URL http://proceedings.mlr.press/v48/bielik16.html.

[10] Michael Bommarito II and Daniel Martin Katz. Gpt takes the bar exam. *arXiv preprint arXiv:2212.14402*, 2022.

[11] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pages 2206–2240. PMLR, 2022.

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

[13] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023.

[14] Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. Dr.spider: A diagnostic evaluation benchmark towards text-to-sql robustness, 2023.

[15] Harrison Chase. LangChain, 10 2022. URL https://github.com/hwchase17/langchain.

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

[17] Wenhu Chen. Large language models are few(1)-shot table reasoners, 2023.

[18] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

[19] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. Binding language models in symbolic languages. *ICLR*, abs/2210.02875, 2023.

[20] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[21] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, ed-

itors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/d5e2c0adad503c91f91df240d0cd4e49-Paper.pdf.

[22] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models, 2022.

[23] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

[25] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Hai-Tao Zheng, Jianfei Chen, Yang Liu, Jie Tang, Juanzi Li, and Maosong Sun. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904*, 2022.

[26] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Cocomic: Code completion by jointly modeling in-file and cross-file context, 2022. URL https://arxiv.org/abs/2212.10007.

[27] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13042–13054, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/c20bb2d9a50d5ac1f713f8b34d9aac5a-Abstract.html.

[28] Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke,

Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, Roman Wang, Nikhil Singh, Taylor L. Patti, Jayson Lynch, Avi Shporer, Nakul Verma, Eugene Wu, and Gilbert Strang. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences*, 119(32):e2123433119, 2022.

[29] Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. Successive prompting for decomposing complex questions. *arXiv preprint arXiv:2212.04092*, 2022.

[30] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. Gpts are gpts: An early look at the labor market impact potential of large language models. *arXiv preprint arXiv:2303.10130*, 2023.

[31] Jacqueline Fagard, Lauriane Rat-Fischer, Rana Esseily, Eszter Somogyi, and JK O'Regan. What does it take for an infant to learn how to use a tool by observation? *Frontiers in psychology*, 7:267, 2016.

[32] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

[33] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis, 2022. URL https://arxiv.org/abs/2204.05999.

[34] Bin Fu, Yunqi Qiu, Chengguang Tang, Yang Li, Haiyang Yu, and Jian Sun. A survey on complex question answering over knowledge base: Recent advances and challenges. *arXiv preprint arXiv:2007.13069*, 2020.

[35] Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization, 2022.

[36] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

[37] Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.552. URL https://aclanthology.org/2021.emnlp-main.552.

[38] Zihui Gu, Ju Fan, Nan Tang, Preslav Nakov, Xiaoman Zhao, and Xiaoyong Du. PASTA: Table-operations aware fact verification via sentence-table cloze pre-training. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4971–4983, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.331. URL https://aclanthology.org/2022.emnlp-main.331.

[39] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.499. URL https://aclanthology.org/2022.acl-long.499.

[40] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. Longcoder: A long-range pre-trained language model for code completion, 2023.

[41] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

[42] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR, 2020.

[43] Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings, 2023.

[44] Shibo Hao, Bowen Tan, Kaiwen Tang, Bin Ni, Xiyan Shao, Hengzhe Zhang, Eric Xing, and Zhiting Hu. BertNet: Harvesting knowledge graphs with arbitrary relations from pretrained language models. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5000–5015, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.309. URL https://aclanthology.org/2023.findings-acl.309.

[45] Joy He-Yueya, Gabriel Poesia, Rose E Wang, and Noah D Goodman. Solving math word problems by combining language models with symbolic solvers. *arXiv preprint arXiv:2304.09102*, 2023.

[46] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.

[47] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian

Eisenschlos. TaPas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.acl-main.398. URL https://doi.org/10.18653%2Fv1%2F2020.acl-main.398.

[48] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[49] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[50] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2021.

[51] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR, 2022.

[52] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models, 2022.

[53] Wenlong Huang, Fei Xia, Dhruv Shah, Danny Driess, Andy Zeng, Yao Lu, Pete Florence, Igor Mordatch, Sergey Levine, Karol Hausman, and Brian Ichter. Grounded decoding: Guiding text generation with grounded models for embodied agents, 2023.

[54] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

[55] Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. Structgpt: A general framework for large language model to reason on structured data. 2023. URL https://arxiv.org/pdf/2305.09645.

[56] Zhengbao Jiang, Yi Mao, Pengcheng He, Graham Neubig, and Weizhu Chen. OmniTab: Pretraining with natural and synthetic data for few-shot table-based question answering. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 932–942, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.68. URL https://aclanthology.org/2022.naacl-main.68.

[57] Zhengbao Jiang, Frank F. Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. Active retrieval augmented generation, 2023.

[58] Qiao Jin, Yifan Yang, Qingyu Chen, and Zhiyong Lu. Genegpt: Augmenting large language models with domain tools for improved access to biomedical information. *ArXiv*, 2023.

[59] Jan-Christoph Kalo and Leandra Fichtel. Kamel: Knowledge analysis with multitoken entities in language models. In *Proceedings of the Conference on Automated Knowledge Base Construction*, 2022.

[60] Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. Compacter: Efficient low-rank hypercomplex adapter layers. *Advances in Neural Information Processing Systems*, 34:1022–1035, 2021.

[61] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022.

[62] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=rkgNKkHtvB.

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

[64] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023.

[65] Yann LeCun. A path towards autonomous machine intelligence version 0.9. 2, 2022-06-27. *Open Review*, 62, 2022.

[66] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, 2021.

[67] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[68] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[69] Hongxin Li, Jingran Su, Yuntao Chen, Qing Li, and Zhaoxiang Zhang. Sheetcopilot: Bringing software productivity to the next level through large language models, 2023.

[70] Minghao Li, Feifan Song, Bowen Yu, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.

[71] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[72] Shuang Li, Xavier Puig, Chris Paxton, Yilun Du, Clinton Wang, Linxi Fan, Tao Chen, De-An Huang, Ekin Akyürek, Anima Anandkumar, Jacob Andreas, Igor Mordatch, Antonio Torralba, and Yuke Zhu. Pre-trained language models for interactive decision-making, 2022.

[73] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, 2021.

[74] Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. Making large language models better reasoners with step-aware verifier, 2023.

[75] Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, Yun Wang, Linjun Shou, Ming Gong, and Nan Duan. Taskmatrix.ai: Completing tasks by connecting foundation models with millions of apis, 2023.

[76] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023.

[77] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 473–485, 2020.

[78] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023.

[79] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

[80] Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. Tapex: Table pre-training via learning a neural sql executor, 2022.

[81] Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Lam Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *arXiv preprint arXiv:2110.07602*, 2021.

[82] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024.

[83] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *arXiv preprint arXiv:2304.09842*, 2023.

[84] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan,

Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv preprint*, abs/2102.04664, 2021. URL https://arxiv.org/abs/2102.04664.

[85] Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning. *arXiv preprint arXiv:2301.13379*, 2023.

[86] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. Augmented language models: a survey, 2023.

[87] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2022.

[88] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2022.

[89] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning (ICML'23)*, 2023.

[90] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint*, 2022.

[91] OpenAI. Codex, 2022. URL https://openai.com/blog/openai-codex.

[92] OpenAI. Chatgpt, 2023. URL https://openai.com/blog/chatgpt.

[93] OpenAI. Dall·e 3, 2023. URL https://openai.com/dall-e-3.

[94] OpenAI. Gpt-4 technical report, 2023.

[95] Batu Ozturkler, Nikolay Malkin, Zhen Wang, and Nebojsa Jojic. Thinksum: Probabilistic reasoning over sets using large language models. *arXiv preprint arXiv:2210.01293*, 2022.

[96] Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

[97] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

[98] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables, 2015.

[99] Anthropic PBC. Introducing 100k context windows. https://www.anthropic.com/index/100k-context-windows, 2023. Accessed: 2023-05-27.

[100] Fabio Petroni, Tim Rocktäschel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, Alexander H Miller, and Sebastian Riedel. Language models as knowledge bases? *arXiv preprint arXiv:1909.01066*, 2019.

[101] Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8494–8502, 2018.

[102] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shi Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bo Li, Ziwei Tang, Jing Yi, Yu Zhu, Zhenning Dai, Lan Yan, Xin Cong, Ya-Ting Lu, Weilin Zhao, Yuxiang Huang, Jun-Han Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. Tool learning with foundation models. *ArXiv*, abs/2304.08354, 2023.

[103] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[104] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[105] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *SIGPLAN Not.*, 51(10):731–747, 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984041. URL https://doi.org/10.1145/3022671.2984041.

[106] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747, 2016.

[107] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese

bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[108] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[109] Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Kurt Shuster, Eric M. Smith, Y-Lan Boureau, and Jason Weston. Recipes for building an open-domain chatbot, 2020.

[110] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[111] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[112] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580*, 2023.

[113] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context, 2023.

[114] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code, 2022. URL https://arxiv.org/abs/2206.12839.

[115] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. Retrieval augmentation reduces hallucination in conversation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3784–3803, 2021.

[116] Significant Gravitas. Auto-GPT, 2023. URL https://github.com/Significant-Gravitas/Auto-GPT.

[117] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

[118] Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss,

Alec Radford, Dario Amodei, and Paul Christiano. Learning to summarize from human feedback, 2022.

[119] Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen tau Yih, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. One embedder, any task: Instruction-finetuned text embeddings, 2023.

[120] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer, 2020.

[121] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer, 2020.

[122] Alon Talmor and Jonathan Berant. The web as a knowledge-base for answering complex questions. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 641–651, 2018.

[123] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

[124] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[125] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao,

Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[126] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.

[127] Tim van Dam, Maliheh Izadi, and Arie van Deursen. Enriching source code with contextual data for code completion models: An empirical study, 2023.

[128] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[129] Fei Wang, Zhewei Xu, Pedro Szekely, and Muhao Chen. Robust (controlled) table-to-text generation with structure-aware equivariance learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2022.

[130] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023.

[131] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.

[132] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation, 2023.

[133] Zhen Wang, Rameswar Panda, Leonid Karlinsky, Rogerio Feris, Huan Sun, and Yoon Kim. Multitask prompt tuning enables parameter-efficient transfer learning. In *The Eleventh International Conference on Learning Representations*, 2023.

[134] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners, 2022.

[135] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

[136] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[137] Sean Welleck, Ilia Kulikov, Stephen Roller, Emily Dinan, Kyunghyun Cho, and Jason Weston. Neural text generation with unlikelihood training. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SJeYe0NtvH.

[138] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE, 2015.

[139] Jiannan Xiang, Tianhua Tao, Yi Gu, Tianmin Shu, Zirui Wang, Zichao Yang, and Zhiting Hu. Language Models Meet World Models: Embodied Experiences Enhance Language Models. *NeurIPS*, 2023.

[140] Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, Victor Zhong, Bailin Wang, Chengzu Li, Connor Boyle, Ansong Ni, Ziyu Yao, Dragomir Radev, Caiming Xiong, Lingpeng Kong, Rui Zhang, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. *EMNLP*, 2022.

[141] Yaqi Xie, Chen Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. Translating natural language to planning goals with large-language models. *arXiv preprint arXiv:2302.05128*, 2023.

[142] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A systematic evaluation of large language models of code. *ArXiv preprint*, abs/2202.13169, 2022. URL https://arxiv.org/abs/2202.13169.

[143] Sherry Yang, Ofir Nachum, Yilun Du, Jason Wei, Pieter Abbeel, and Dale Schuurmans. Foundation models for decision making: Problems, methods, and opportunities. *arXiv preprint arXiv:2303.04129*, 2023.

[144] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and

Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[145] Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. Large language models are versatile decomposers: Decompose evidence and questions for table-based reasoning, 2023.

[146] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. TaBERT: Pretraining for joint understanding of textual and tabular data. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.745. URL https://aclanthology.org/2020.acl-main.745.

[147] Wenhao Yu, Chenguang Zhu, Zaitang Li, Zhiting Hu, Qingyun Wang, Heng Ji, and Meng Jiang. A survey of knowledge-enhanced text generation. *ACM Computing Surveys*, 54 (11s):1–38, 2022.

[148] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html.

[149] Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1–9, 2022.

[150] Yuheng Zha, Yichi Yang, Ruichen Li, and Zhiting Hu. AlignScore: Evaluating factual consistency with a unified alignment function. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11328–11348, Toronto, Canada, July 2023. Association for Computational Linguistics. URL https://aclanthology.org/2023.acl-long.634.

[151] Yuheng Zha, Yichi Yang, Ruichen Li, and Zhiting Hu. Text alignment is an efficient unified model for massive nlp tasks. *NeurIPS*, 2023.

[152] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023.

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

[154] Tianshu Zhang, Xiang Yue, Yifei Li, and Huan Sun. Tablellama: Towards open large generalist models for tables, 2023.

[155] Wenqi Zhang, Yongliang Shen, Weiming Lu, and Yueting Zhuang. Data-copilot: Bridging billions of data and humans with autonomous workflow, 2023.

[156] Yilun Zhao, Linyong Nan, Zhenting Qi, Rui Zhang, and Dragomir Radev. ReasTAP: Injecting table reasoning skills during pre-training via synthetic reasoning examples. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 9006–9018, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.615. URL https://aclanthology.org/2022.emnlp-main.615.

[157] Yilun Zhao, Chen Zhao, Linyong Nan, Zhenting Qi, Wenlin Zhang, Xiangru Tang, Boyu Mi, and Dragomir Radev. Robut: A systematic study of table qa robustness against human-annotated adversarial perturbations, 2023.

[158] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023.

[159] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models, 2023.