L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models

Ansong Ni[†] **Pengcheng Yin**[♣] **Yilun Zhao**[†] **Martin Riddell**[†]
Troy Feng[†] **Rui Shen**[†] **Stephen Yin**[†] **Ye Liu**[♢] **Semih Yavuz**[♢] **Caiming Xiong**[♢]
Shafiq Joty[♢] **Yingbo Zhou**[♢] **Dragomir Radev**[†] **Arman Cohan**[†‡]
[†]Yale University, USA [‡]Allen Institute for AI, USA
[♣]Google DeepMind, USA [♢]Salesforce Research, USA
{ansong.ni, arman.cohan}@yale.edu

Abstract

Recently, large language models (LLMs), especially those that are pretrained on code, have demonstrated strong capabilities in generating programs from natural language inputs. Despite promising results, there is a notable lack of a comprehensive evaluation of these models' language-to-code generation capabilities. Existing studies often focus on specific tasks, model architectures, or learning paradigms, leading to a fragmented understanding of the overall landscape. In this work, we present **L2CEval**, a systematic evaluation of the language-to-code generation capabilities of LLMs on 7 tasks across the domain spectrum of semantic parsing, math reasoning, and Python programming, analyzing the factors that potentially affect their performance, such as model size, pretraining data, instruction tuning, and different prompting methods. In addition, we assess confidence calibration, and conduct human evaluations to identify typical failures across different tasks and models. **L2CEval** offers a comprehensive understanding of the capabilities and limitations of LLMs in language-to-code generation. We release the evaluation framework[1] and all model outputs, hoping to lay the groundwork for further future research.

1 Introduction

Language-to-code (**L2C**[2]) is a type of task that aims to automatically map natural language descriptions to programs, which are later executed to satisfy the user's demand (Yin and Neubig, 2017; Austin et al., 2021). As illustrated in Figure 1, language-to-code is the foundation of many applications in AI, such as task-oriented dialogue systems (Andreas et al., 2020), coding assistant (Agashe et al., 2019; Lai et al., 2023), language interfaces to databases (Pasupat and Liang, 2015; Yu et al., 2018), and robotic control (Zhou et al., 2022; Shridhar et al., 2020). It has also served as a great testbed for evaluating various language understanding capabilities of NLP systems, such as logical and math reasoning (Gao et al., 2023; Han et al., 2022), grounded language understanding (Xie et al., 2022; Huang et al., 2023), and tool use (Schick et al., 2024; Paranjape et al., 2023).

Recent progress on large language models (LLMs) (OpenAI, 2023; Chowdhery et al., 2023; Touvron et al., 2023a), especially those specifically trained for coding (Fried et al., 2023; Nijkamp et al., 2022; Chen et al., 2021; Li et al., 2023), has shown that LLMs trained on a mixture of text and code are able to perform language-to-code generation under few-shot or even zero-shot learning settings (Rajkumar et al., 2022; Ni et al., 2023b). However, the modeling factors that affect the performance of LLMs for such **L2C** tasks–including model size, training data mixture, prompting methods, and instruction tuning–are poorly understood. In addition, there lacks a consistent evaluation of different LLMs on the same spectrum of language-to-code tasks, making it difficult for the users to decide which models to use for certain tasks or if they should resort to finetuning their own models. Beyond model performance, model properties such as robustness to prompt and confidence calibration are also crucial for understanding the reliability of LLMs, but such properties have not been systematically studied for **L2C** tasks.

In this work, we present **L2CEval**, providing a systematic evaluation of the language-to-code generation capabilities of LLMs. **L2CEval** includes a wide range of state-of-the-art models,

[1]All future evaluations (*e.g.,* LLaMA-3, StarCoder2, etc) will be updated on the project website: https://l2c-eval.github.io/.

[2]We refer to ''natural language'' whenever we use the term ''language'' in this work.

Transactions of the Association for Computational Linguistics, vol. 12, pp. 1311–1329, 2024. https://doi.org/10.1162/tacl_a_00705
Action Editor: David Chiang. Submission batch: 11/2023; Revision batch: 9/2024; Published 10/2024.
© 2024 Association for Computational Linguistics. Distributed under a CC-BY 4.0 license.

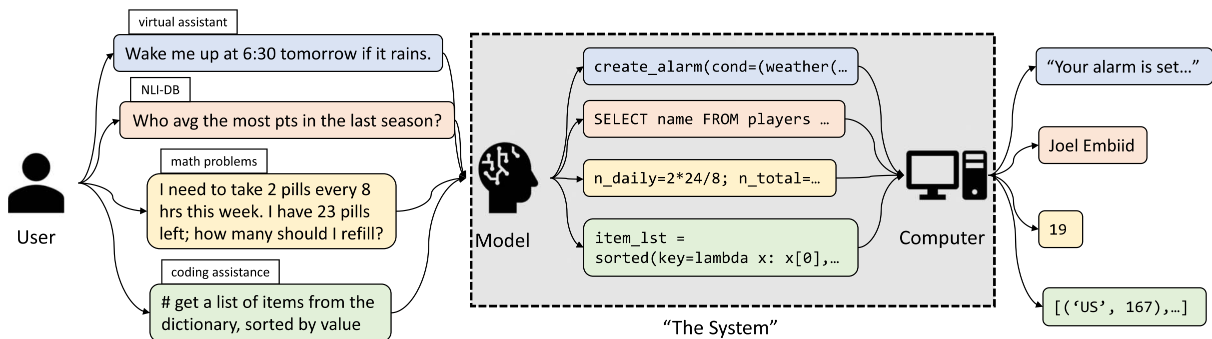

Figure 1: Language-to-code (**L2C**) generation is the cornerstone for many applications in AI. It is also the key to enabling direct communication between the users and the computers with natural language.

specifically 56 models from 13 different organizations, all evaluated on three core domains of language-to-code generation tasks: *semantic parsing*, *math reasoning*, and *Python programming*. Our **L2CEval** framework includes extensive evaluations of models as small as 1 billion parameters, to significantly larger ones such as Falcon-180B, as well as davinci and GPT-4 models from OpenAI. We also benchmark models that are trained on different mixtures of data of varying sizes (35B $\sim$ 3.5T tokens), as well as models that are instruction-tuned, from both open-source and open-access proprietary categories. Our work is the first to conduct extensive and thorough comparisons of LLMs for language-to-code generation across multiple dimensions of variation. To summarize, we release **L2CEval** and its main contributions are as follows:

- We standardize the evaluation (*e.g.,* prompts, metrics) of **7 L2C** tasks across domains of semantic parsing, math reasoning, and Python programming to allow controlled comparisons among **56** models from **13** organizations;
- We study the scaling effect of model size, pretraining compute/data mixture, as well as several modeling contributions (*e.g.,* instruction-tuning, zero/few-shot prompting) for **L2C** tasks;
- We analyze the robustness and calibration measurements of different models, and identify their common error cases;
- We release the code for our evaluation framework, and model outputs (*i.e.,* texts and logits) for reproducibility and future studies.

Through our work, we hope to provide insight into applying LLMs to **L2C** applications, as well as building future LLMs.

2 L2CEval

The main motivation behind **L2CEval** is to provide a *comprehensive* evaluation of language-to-code generation capabilities and understand what affects such **L2C** capabilities. In the following sections, we first discuss the key desiderata in design of **L2CEval** in § 2.1, then the formulation of **L2C** in § 2.2, then in § 2.3 we introduce the domains and specific tasks we consider for **L2CEval**, as well as the reasons for choosing such tasks. Finally, in § 2.4, we describe the models included in **L2CEval** and the model selection process.

2.1 Desiderata

To ensure that the **L2CEval** framework serves as a comprehensive resource for evaluating language-to-code (**L2C**) capabilities, we outline a set of key desiderata that guided its construction.

Task Inclusion. *Diverse Task Representation*: The benchmark aims to capture a wide scope of L2C tasks, specifically incorporating semantic parsing, Python programming, and math reasoning. *Task Complexity*: Under each of these domains, we include 2 to 3 sub-tasks to represent a combination of different levels of language understanding, reasoning, and programming abilities.

Model Evaluation. *Open Source and Commercial Models*: **L2CEval** is designed to accommodate both open-source and commercial models to provide a holistic view of available L2C

Domain	Dataset	Split	Size	Input	Output
Semantic Parsing	Spider (Yu et al., 2018)	Dev	1,032	DB schema + NL	SQL Query
	WikiTQ (Pasupat and Liang, 2015)	Dev	2,831	Table headers* + NL	SQL Query
Math Reasoning	GSM8k (Cobbe et al., 2021)	Dev[3]	1,495	Math problem in NL	Python solution
	SVAMP (Patel et al., 2021)	All	1,992	Math problem in NL	Python solution
Python Programming	MBPP (Austin et al., 2021)	Test	500	NL spec. + 1 test	Python function
	HumanEval (Chen et al., 2021)	All	164	NL spec. + 1–3 test	Python function
	DS-1000 (Lai et al., 2023)	All	1,000	NL spec.	Python lines

Table 1: A summary of all the benchmarks included for evaluation in **L2CEval**.

capabilities. Our focus is more on the open-source models, however, as proprietary models do *not* disclose certain basic information which makes drawing scientific conclusions difficult. *Model Size Variability*: The benchmark includes models of different sizes to explore any correlation between model size and performance. *Specialized vs General Models*: We examine the performance trade-offs between models exclusively trained on code and general language models to understand the advantages or disadvantages of specialization.

Evaluation Setup. *Standardized Prompts*: All tasks and models are evaluated using standardized prompts, overcoming the inconsistencies prevalent in prior work. *Reproducibility*: Our evaluation setup is clearly described to facilitate reproducible experiments. *Fair Comparison*: Universal evaluation metrics are employed across diverse tasks and models to enable equitable comparisons.

Transparency and Reusability. *Documentation*: The framework is thoroughly documented to promote community engagement and constructive feedback. *Interoperability*: **L2CEval** is built to be easily updated or extended, allowing for the incorporation of new tasks or models as the field evolves. We also share the code and model outputs to support future research in this domain.

By adhering to these desiderata, the **L2CEval** framework aims to be a comprehensive, fair, and practical evaluation resource for the community.

2.2 Language-to-Code Generation (L2C)

Problem Formulation. While language-to-code generation covers a wide range of tasks as shown in Figure 1, here we attempt to give a unified problem formulation. Given the user's intent described in natural language x (*e.g.,* description of a Python function) and optionally some programming context c (*e.g.,* existing function definitions, open test cases), an **L2C** model aims to automatically map the input to a program y (*e.g.,* a Python function). In the context of LLMs, this is typically modeled as a conditional generation problem with prompting:

$$\hat{y} = \arg\max_{y} P_{\text{LM}}(y \mid \textbf{prompt}) \qquad (1)$$

where the "**prompt**" consists of task-specific instructions I and optionally m exemplars $\{(x_i, y_i, c_i)\}_{i<m}$:

$$\textbf{prompt} = f(I, \{(x_i, y_i, c_i)\}_{i<m}, c, x)$$

To obtain the best candidate program $\hat{y}$, we use *greedy decoding* in all our evaluations.[4] Also for a fair comparison, we standardize the prompting methods by following previous work (Ni et al., 2023b; Ben Allal et al., 2022) and avoid prompts that are tailored for specific models.

Execution-based Evaluation. The generated program candidate $\hat{y}$, sometimes accompanied with additional execution context e (*e.g.,* connection to DB) is later executed by an executor $\mathcal{E}(\cdot)$ (*e.g.,* Python interpreter). We can evaluate *execution accuracy* by checking if it matches the gold execution results z^* upon execution:

$$\text{Acc.} = \mathbb{1}(\hat{z}, z^*) \;\text{ where }\; \hat{z} = \mathcal{E}(\hat{y}, e) \qquad (2)$$

We use execution accuracy as a proxy for whether the user's original intent is satisfied.[5] This is also consistent with previous work on **L2C** (Xie et al., 2022; Ni et al., 2023b; Shi et al., 2022).

2.3 Tasks

We evaluate the language-to-code capabilities of LLMs in three representative application scenarios

[3]Here we use the split from Ni et al. (2023a).

[4]We discuss the limitation of greedy decoding in § 4.

[5]See § 4 for the limitations of execution-based evaluation.

shown in Figure 1: *semantic parsing*, *math reasoning*, and *Python programming*. Particularly, these tasks collectively assess the capabilities of models in language-to-code generation to understand natural language in different contexts, reason about the steps for solving the problem, and convert it into executable code (see Figure 1). *Semantic parsing* focuses on the transformation of natural language queries into structured, domain-specific languages; *math reasoning* challenges the models' numerical and logical reasoning abilities by requiring them to solve problems that involve multiple steps of calculation and reasoning; and *Python programming* tests the models' proficiency in generating functional code that aligns with a user's intent, reflecting a real-world application of LLMs in software development. A summary of **`L2CEval`** benchmarks is shown as Table 1 and we discuss each of these tasks in detail as below.

Semantic Parsing. Semantic parsing considers the task of translating a user's natural language utterance (*e.g., who averaged the most pots in the last season?* in Figure 1) into machine-executable programs (*e.g.,* an SQL database query), and has been a long-standing problem in NLP (Zettlemoyer and Collins, 2005; Berant et al., 2013). A prompt to an LLM consists of an NL utterance and descriptions of relevant structured context, such as the schema information of a database (*e.g.,* columns in each table). The target output is a program defined in some domain-specific languages, such as SQL. Intuitively, semantic parsing challenges LLMs on grounded language understanding (Xie et al., 2022; Cheng et al.), where a model needs to associate NL concepts in utterances (*e.g., "last season"*) with relevant structured knowledge (*e.g.,* superlative operation on column `season`) in order to synthesize the program (Pasupat and Liang, 2015; Yu et al., 2018; Yin et al., 2020). In this work, we choose to use text-to-SQL as a representative task as it closely ties with applications such as natural language interface to databases (Androutsopoulos et al., 1995; Affolter et al., 2019). Recent work (Rajkumar et al., 2022; Ni et al., 2023b) shows that LLMs are effective in performing text-to-SQL parsing. In this work, we use two widely used text-to-SQL datasets, **Spider** (Yu et al., 2018) and **WikiTQ** (Pasupat and Liang, 2015), as our datasets for benchmarking semantic parsing capabilities of LLMs. Following Xie et al. (2022), we concatenate the natural language utterance with the database schema or table headers as LLM input.[6]

Math Reasoning. To solve a math word problem, a model needs to abstract the mathematical relations from the natural language description, and reason about the potential steps. Compared to semantic parsing where the target programs are table-lookup queries, programs for math reasoning tasks usually require multiple steps of calculation and numerical and logical reasoning. Because of this, math word problems are widely adopted as testbeds for evaluating the reasoning abilities of LLMs (Cobbe et al., 2021; Wei et al., 2022b; Ni et al., 2023a; Welleck et al., 2022). In this paper, we choose the **GSM8k** (Cobbe et al., 2021) and **SVAMP** (Patel et al., 2021) datasets, which are grade-school level math problems described in natural language. We chose these two benchmarks due to their moderate difficulty and popularity. Following Welleck et al. (2022) and Gao et al. (2023), we prompt the models to answer math word problems by generating Python programs as solutions, which are later executed by a Python interpreter to output the answer.

Python Programming. One of the most important applications for LLMs trained on code is to assist programmers in developing software. Typically, a model is given a developer's natural language intent (*e.g., write a merge sort function*) with optional additional specifications (Austin et al., 2021) such as input/output examples or unit tests (*e.g.,* `assert merge_sort([5,7,3])==[3,5,7]`), to generate the code (*e.g.,* a Python function) that implements the user's intent. To evaluate the basic programming skills of the LLMs, we use the **MBPP** (Austin et al., 2021) and **HumanEval** (Chen et al., 2021) datasets, for which the model needs to implement some basic Python functions to pass the test cases. Moreover, we also include **DS-1000** (Lai et al., 2023), which focuses on data-science-related questions and libraries.[7]

[6]While more challenging datasets exists for text-to-SQL (*e.g.,* BIRD-SQL (Li et al., 2024)), we believe the observations should be generalizable due to the similar task format.

[7]While APPS (Hendrycks et al., 2021) is another popular Python programming dataset, we decide not to use it due to various issues found in Li et al. (2022b).

Organization	Model Series	Variants	Sizes	# All Tokens	# Code Tokens	Context Length	Code Specific
Salesforce	CodeGen (Nijkamp et al., 2022)	multi/mono	6.1/16.1B	505∼577B	119∼191B	2K	✓
	CodeGen-2.5 (Nijkamp et al., 2023)	multi/mono/ instruct†	7B	1.4T	1.4T	2K	✓
Eleuther AI	GPT-J (Wang and Komatsuzaki, 2021)		6.1B	402B	46B	2K	✗
	GPT-NeoX (Black et al., 2022)		20.6B	472B	54B	2K	✗
	Pythia (Biderman et al., 2023)		1.4/6.9/12B	300B	35B	2K	✗
Databricks	Dolly-v2 (Conover et al., 2023)		6.9/12B	–	–	2K	✗
BigCode	SantaCoder (Allal et al., 2023)		1.1B	236B	236B	2K	✓
	StarCoder (Li et al., 2023)	base/plus	15.5B	1∼1.6T	1T	8K	✓
Meta	InCoder (Fried et al., 2023)		1.3/6.7B	52B	52B	2K	✓
	LLaMA (Touvron et al., 2023a)		7/13/30B	1∼1.4T	45∼63B	2K	✗
	LLaMA-2 (Touvron et al., 2023b)		7/13/70B	2T	–	4K	✗
	CodeLLaMA (Rozière et al., 2023)	base/instruct†	7/13/34B	2.5T	435B	16K	✓
Stanford	Alpaca† (Taori et al., 2023)		7/13/30B	–	–	2K	✗
Replit	Replit-v1-3b (rep)		3B	525B	525B	2K	✓
WizardLM	WizardCoder-v1 (Luo et al., 2023)		15B	–	–	2K	✓
MosaicML	MPT (Team, 2023a,b)	base/instruct†	7/30B	1T	135B	2K/8K	✗
MistralAI	Mistral-v0.1 (Jiang et al., 2023)	base/instruct†	7B	–	–	32K	✗
XLANG	Lemur-v1 (Xu et al., 2023)		70B	–	–	4K	✓
TII	Falcon (Almazrouei et al., 2023)	base/instruct†	7/40/180B	1∼3.5T	–	2K	✗
OpenAI	Codex (Chen et al., 2021)	code-cushman-001 code-davinci-002	12B/ –	400B	100B	2K/8K	✓
	InstructGPT† (Ouyang et al., 2022)	text-davinci-002/3	–	–	–	4K	✗
	ChatGPT† (OpenAI, 2022)	turbo-0301/0613	–	–	–		
	GPT-4† (OpenAI, 2023)	0314/0613	–	–	–	8K	✗

Table 2: Information table for the models evaluated in this work. –: no information on training data size is available, or the model is further tuned on top of other models. †: Instruction-tuned models.

2.4 Models

We evaluate 56 models that vary in size, training data mixture, context length, and training methods. Table 2 summarizes the open-source models we evaluated and several key properties.

Selection Criteria. While it is not possible to evaluate every single LLM on these tasks, we strive to provide a comprehensive evaluation of the current LLMs in **L2C** generation, by covering a diversified selection of LLMs of varying sizes and are trained on different mixtures of data. For example, the size of the models we consider ranges from 1B (*e.g.,* SantaCoder (Allal et al., 2023)) to 170B+ (*e.g.,* Falcon-180B (Almazrouei et al., 2023) and GPT-4 model from OpenAI). Though we prioritize the evaluation of code-specific models, which means that the majority of the training tokens are from code (*e.g.,* CodeLLaMA (Rozière et al., 2023), StarCoder (Li et al., 2023)), we also include the most competitive general LLMs such as LLaMA2-70B (Touvron et al., 2023a) and Falcon-180B for comparison. To evaluate the effect of instruction-tuning and its data mixtures on **L2C** tasks, we also include several instruct-tuned versions of the LLMs, such as Alpaca (Taori et al., 2023), Dolly (Conover et al., 2023), etc. We also prioritize the evaluation of open-source models and mainly present our findings with these models as we are unclear about the technical details of proprietary models (*e.g.,* model size, training data) and hesitate to speculate about them.

Model Access. For all the open-source models, we access them through huggingface model hub[8] and run them locally on a server with RTX A6000 48GB GPUs, using `Lightning`[9] as underlying framework. For proprietary OpenAI models we access them through the public API.[10]

3 Results and Analysis

We organize the experiment results and analysis as follows. We first discuss different

[8] `https://huggingface.co/models`.
[9] `https://lightning.ai/`.
[10] `https://api.openai.com/`.

Group	Model	Spider (2-shot)	WikiTQ (2-shot)	GSM8k (8-shot)	SVAMP (4-shot)	MBPP (3-shot)	HE (0-shot)	DS-1K (0-shot)	MWR
Other	gpt-4	79.2	56.7	88.5	92.8	74.2	80.5	24.0	100%
	text-davinci-003	68.3	45.4	64.1	80.7	63.6	52.4	15.3	95%
	gpt-3.5-turbo	72.7	38.4	74.7	80.7	66.6	39.0	11.0	92%
20 ~ 100B	*CodeLLaMA-base (34B)*	61.7	32.3	43.6	70.7	45.6	44.5	22.4	90%
	Lemur (70B)	68.0	44.9	57.5	47.9	51.4	41.5	20.7	88%
	LLaMA-2 (70B)	58.5	37.3	56.0	73.9	36.8	28.7	16.9	84%
10 ~20B	*CodeLLaMA (13B)*	58.5	35.6	30.7	64.9	44.0	34.2	18.8	86%
	StarCoder (15.5B)	52.1	27.4	22.1	48.8	46.6	34.2	19.8	78%
	LLaMA-2 (13B)	35.7	24.6	26.1	58.9	27.0	17.7	9.1	59%
2 ~10B	Mistral-v0.1 (7B)	53.3	31.4	38.4	69.4	37.8	25.0	14.1	79%
	CodeLLaMA-base (7B)	54.3	29.5	25.5	52.8	40.0	31.1	16.0	76%
	CodeGen2.5-multi (7B)	53.8	29.6	14.9	43.1	38.2	31.1	16.9	71%
<2B	*SantaCoder (1.3B)*	19.0	11.4	2.8	0.0	26.2	17.7	1.1	24%
	InCoder (1.1B)	13.4	6.2	1.0	3.5	13.8	8.5	2.9	11%
	Pythia (1.4B)	5.7	4.4	1.5	9.3	5.8	3.7	1.8	6%

Table 3: Top-3 models at different size ranges. All models are of the "base" variant (*i.e.,* w/o instruction-tuning or RLHF), except the "Other" group, which is for reference purposes only. MWR: Mean Win Rate (see definition in § 3.1). The best performance for each group is highlighted with color shades indicating the relative performance across different groups. Code-specific LLMs are noted in *italics*.

scaling effects in § 3.1, then in § 3.2 and § 3.3, we analyze how pretraining data mixture and instruction-tuning affects the models for `L2C` tasks. To study model robustness, in § 3.4, we measure the sensitivity of the models on the few-shot demonstrations, and show model confidence calibration in § 3.5. Finally, we present an error analysis in § 3.6.

3.1 Scaling

We examine the correlation between model performance and its parameter count, as well as the pretraining compute. While most of our findings align with previous work on scaling laws (Kaplan et al., 2020; Hoffmann et al., 2022), we focus on properties that are more related to `L2C` tasks.

Model Size. We present the top-3 models across different size ranges based on Mean Win Rate (MWR) in Table 3. MWR is defined as the fraction of times a model outperforms other models, averaged across all 7 tasks. More specifically, given a set of N models for comparison $\mathcal{P} = \{P_1, \ldots, P_N\}$ and and a set of M tasks $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_M\}$, MWR of a model P_i is defined as:

$$\mathrm{MWR}(P_i) = \sum_{k \leq M} \frac{|\{P_j | \mathcal{D}_k(P_j) \leq \mathcal{D}_k(P_i), j \leq N\}|}{M \times N}$$

where $\mathcal{D}_k(P_j)$ denotes the performance of model P_j on task $\mathcal{D}_k$. From this table, we can observe clear performance gaps between models of different size groups. While this is especially the case within the same model series, smaller models can also beat bigger ones across different series (*e.g.,* Mistral 7B outperforms LLaMA-2 13B), showing that other factors such as training data is also important.[11] However, such a scaling effect also varies across domains: for math reasoning tasks, the performance gaps between models of different size groups are much larger than those of Python programming and semantic parsing tasks. Given the programs to solve these math problems are relatively simple in syntax, we hypothesize that the bottleneck lies in the planning and reasoning capabilities, which correlate with model size. This hypothesis is consistent with previous findings (Wei et al., 2022a,b).

Pretraining Compute. To study the scaling of compute resources during pretraining, we plot the average model performance across all 7 tasks against the estimated FLOPS of compute[12] needed during training, as shown in Figure 2. From the figure, we can see that code LMs are much more

[11] Evaluation of more recent smaller models like LLaMA-3 (8B) shows that it achieves avg. performance of 45.1, outperforming CodeLLaMA-base (13B).

[12] Here we base our estimation on Kaplan et al. (2020): FLOPS ≈ 6 * model size (B) * training tokens (B).

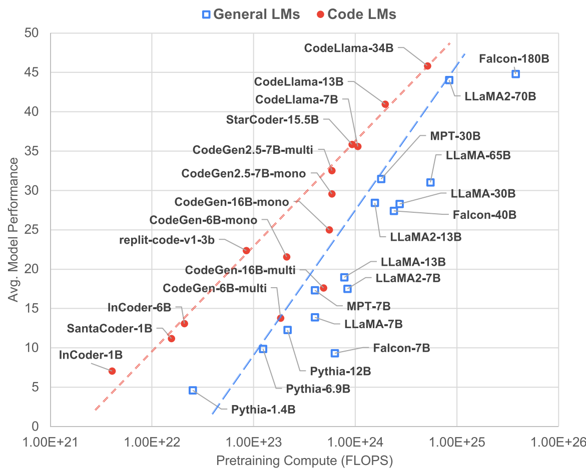

Figure 2: Pretraining compute scaling for code-specific and general LMs. Dashed lines denote the trend line where the optimal compute is achieved for models of each category.

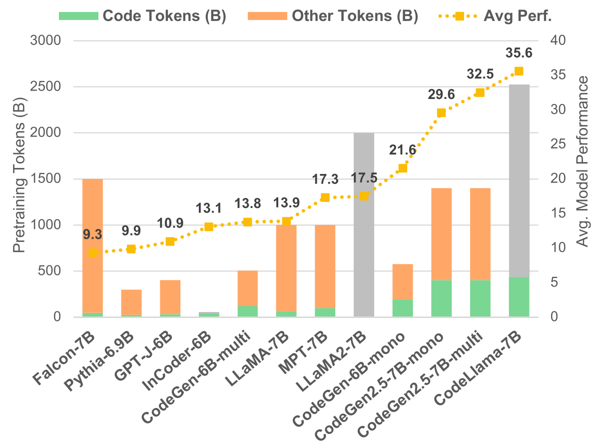

Figure 3: Pretraining data mixture for models of similar sizes ($6 \sim 7$B), ranked by performance. LLaMA-2 paper (Touvron et al., 2023b) only shares the size but not the distribution of the pretraining data, and CodeLLaMA is trained on top of LLaMA-2.

compute efficient than general LMs in terms of **L2C** performances. This is particularly noticeable when the compute budget is constrained; for example, SantaCoder-1B outperforms Pythia-6.9B with an order of magnitude less compute, and InCoder-1B outperforms Pythia-1.4B using only 1/5 of the pretraining compute. While code LMs generally require fewer FLOPS to achieve comparable performance on **L2C** tasks,[13] it is expected as well, given that general LMs are also optimized for many other natural language tasks unrelated to coding. Moreover, such a trend also seems to be diminishing when scaling up (*e.g.,* comparing CodeLLaMA-34B and LLaMA-2-70B), which suggests that when model and pretraining data size gets larger, it is possible that general LMs will be as compute efficient as code LMs for **L2C** tasks.

3.2 Data Mixture

While all of the models we evaluate in **L2CEval** have seen code during pretraining, the distributions of their training data mixture vary significantly, as illustrated in Table 2. In Figure 3, we show 12 different models that are around 7B, ordered by their average model performance on all 7 tasks, and plot the amount of the code and non-code tokens in their pretraining data. As we can see from the figure, the number of code tokens in the pretraining data affects the model performance much more than the amount of non-code tokens, and the model performance is almost monotonically increasing with the amount of code tokens in the training data. Given that CodeLLaMA models are further pretrained on code tokens on top of LLaMA-2, by comparing the performance of their 7B and 13B versions in Figure 3 and Table 3, we can see that training on more code tokens not only drastically improves text-to-sql parsing and Python programming, but also math reasoning tasks. As mentioned in § 3.1, since the generated Python solutions for GSM8K and SVAMP are both simple in syntax (*i.e.,* straight-line programs with numeric operations), we hypothesize that training on more code tokens improves the reasoning abilities of LMs in general. This is also consistent with previous findings (Fu et al., 2022; Mishra et al., 2022; Wei et al., 2022b).

3.3 Instruction Tuning

Instruction tuning (Ouyang et al., 2022) is a type of method that enhances the ability of LLMs to follow instructions written in natural language. Here we compare the instruction-tuned models and their base models and show their few/zero-shot results in Table 4.

Zero-shot Results. Firstly, we observe that the zero-shot results are generally improved after instruction tuning for all models on Spider and MBPP. This is perhaps a little surprising for Dolly and MPT-instruct models as

[13] The only exceptions are CodeGen-multi/mono models, which are trained with far less amount of code tokens compared with other code LLMs.

Models	Few-Shot						Zero-Shot					
	Spider (2)		GSM8k (8)		MBPP (3)		Spider (0)		GSM8k (0)		MBPP (0)	
	Base	IT	Base	IT	Base	IT	Base	IT	Base	IT	Base	IT
Pythia/Dolly-6.9B	12.5	*13.1*	2.6	2.6	13.2	12.0	3.0	*5.2*	0.0	0.0	0.4	*9.4*
Pythia/Dolly-12B	16.2	13.0	2.6	2.6	19.0	15.0	2.8	*6.5*	0.0	0.0	1.2	*3.8*
MPT-7B	27.3	25.5	10.9	10.4	21.0	*24.0*	19.5	*27.0*	1.5	*6.2*	0.2	*10.2*
MPT-30B	43.3	42.8	30.7	29.1	29.2	28.4	34.9	*44.1*	0.0	*1.9*	0.8	*23.4*
LLaMA/Alpaca-7B‡	13.1	*16.1*	8.0	3.5	16.6	14.4	5.7	*20.5*	0.0	0.0	5.0	*13.2*
LLaMA/Alpaca-13B‡	15.2	*24.3*	15.7	*18.5*	22.8	*23.4*	15.2	*26.1*	0.0	0.0	2.2	*7.2*
LLaMA/Alpaca-30B‡	38.5	*46.2*	15.9	*19.4*	26.6	*32.0*	41.8	*46.3*	0.0	0.0	20.6	*27.2*
CodeLlama-7B‡	54.3	*55.9*	25.5	*26.5*	40.0	*41.2*	51.8	*56.1*	0.0	*7.2*	10.0	*15.2*
CodeLlama-13B‡	58.5	*63.0*	30.7	*48.2*	44.0	*48.2*	64.2	*66.2*	17.0	10.6	18.6	*19.0*
CodeLlama-34B‡	61.7	*68.7*	43.6	*52.8*	45.6	*52.8*	69.6	*69.7*	15.1	5.8	34.0	*42.8*

Table 4: How instruction-tuning affects few/zero-shot **L2C** performances. Model names shown as {base}/{IT}-{size}. ‡: instruction-tuning includes code-related tasks. "IT" denotes the instruction-tuned version of the base model. Performance *improvements* and degradations are marked accordingly.

their instruction-tuning data does not explicitly include coding tasks. Besides the fact that instruction-tuning trains the models to focus more on the instruction, we also hypothesize that instruction-tuning generally improves language understanding abilities, which is essential for **L2C** tasks. We also note that the zeros-shot performances for GSM8k are all zeros for the selected models. By inspecting the model outputs, we find that the models fail to follow the instructions and provide the answer by ending the Python solution with `answer = x`.

Few-shot Results. As for few-shot performance numbers, those that are instruction-tuned with coding tasks, such as Alpaca and CodeLLaMA-instruct models, yield much more consistent improvements than Dolly and MPT-instruct across all tasks. Notably, CodeLLaMA-34B-instruct improves 7.0, 9.2, and 7.2 points over the base model on Spider, GSM8K, and MBPP datasets, respectively. Though we observe that some few-shot results deteriorate for Dolly and MPT-instruct, it should also be noted that such performance degradations are quite minimal, as half of them are within 2%. It is suggested in Ouyang et al. (2022) that instruction tuning generally decreases few-shot performance, as it shifts the attention of the model from the few-shot exemplars to the instructions, but from these results, we believe that whether instruction-tuning improves few-shot performance largely depends on how similar the instruction-tuning tasks are to tasks for evaluation.

3.4 Sensitivity to Prompt

Here we study how sensitive are the models to the number of few-shot demonstrations or different examples in the prompt.

Number of Few-shot Demonstrations. Figure 4 illustrates the correlation between model performance and the number of exemplars in the prompt.[15] While increasing the number of few-shot exemplars in the prompt generally improves execution accuracy, such improvement is not consistent with different models and tasks. For example, on the MBPP dataset, increasing from 3 to 8 exemplars in the prompt actually decreases the performance for most of the selected models, *e.g.*, by 4.0% for codex-cushman. We hypothesize that this is because the programs in the prompt will bias the model towards generating similar programs and ignore the specification. Supporting

[14] To be consistent with previous work, we use the evaluation harness from the BigCode project (`https://github.com/bigcode-project/bigcode-evaluation-harness`) for evaluating HumanEval and DS-1000. However, it does not output logits which are essential for calculating calibration scores.

[15] While the range of number of shots are different for each task due to different task prompt lengths (*e.g.*, database schema encoding for Spider), we keep it consistent across different models on the same task for a fair comparison.

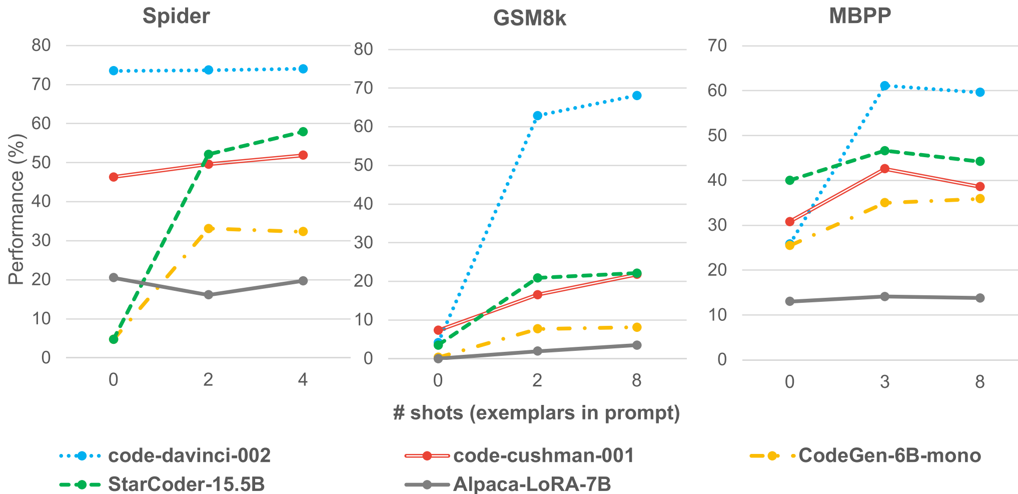

Figure 4: Models perf. with different # of exemplars in the prompt.

Models	Spider (2-shot)	GSM8k (2-shot)	MBPP (3-shot)
code-davinci	73.7±0.3	66.4±1.0	59.0±1.9
code-cushman	50.4±0.7	24.2±1.1	39.3±3.3
CodeGen-6B-mono	32.4±0.6	13.8±0.2	35.5±0.5
StarCoder-15.5B	54.9±2.7	32.3±0.8	44.1±2.2
Alpaca-7B	20.1±3.5	7.3±1.2	13.6±0.6

Table 5: Mean and std for (n)-shot performance over 3 runs with different random exemplars.

evidence of this is in Table 5, as codex-cushman is shown to be more sensitive to the exemplars. This effect has also been observed in Li et al. (2022b).

Different Examples as Demonstrations. Moreover, we also show the sensitivity of the models to different exemplars and present the results in Table 5 by showing the variance of model performance across different runs using different exemplars in the prompt. While the variances differ for different models and tasks, none of them are significant enough to alter the ranking of the models, nor impact the conclusions presented in this work.

3.5 Model Calibration

A good model not only produces high-quality outputs, but also should be well-calibrated, meaning that it should be uncertain about its predictions when such predictions are wrong. Following recent work (Liang et al., 2022), we evaluate model calibration using *expected calibration error* (ECE) (Naeini et al., 2015; Guo et al., 2017). For a model P_{LM} and a dataset $\mathcal{D}$, this is defined as:

$$\text{ECE}(P_{\text{LM}}, \mathcal{D}) = \mathbb{E}_{(x,y^*)\sim\mathcal{D}}[P_{\text{LM}}(\hat{y}|x) - \text{Acc}(\hat{y}, y^*)]$$
$$\text{s.t. } \hat{y} = \arg\max_y P_{\text{LM}}(y|x)$$

where $\text{Acc}(\cdot)$ denotes the execution accuracy metric described as Equation (2). From the results shown in Figure 5, we can observe that while model calibration generally correlates with model performance, the best-performing models are not necessarily the ones with the best calibration. Note that with a well-calibrated model, methods such as voting (Li et al., 2022a; Xuezhi Wang et al., 2023) and confidence-based reranking (Ni et al., 2023b) may be used to further improve their performance. Moreover, a better-calibrated model is more reliable in practical applications, such as coding assistants, where its confidence levels can serve as indicators of generation quality.

3.6 Error Modes

In Figure 6, we present an error analysis on the four best models, by manually[16] examining a fixed set of 100 examples from the GSM8k and MBPP datasets across 4 selected models. We categorize the errors into 5 cases:

1) *execution error*, where deformed programs are generated;

2/3) *missing/extra steps*, where some key steps are missing or extraneous lines are generated in predicted code;

4) *wrong steps*, where the model only makes subtle mistakes in certain steps in the code;

[16] Two of the authors performed this annotation.

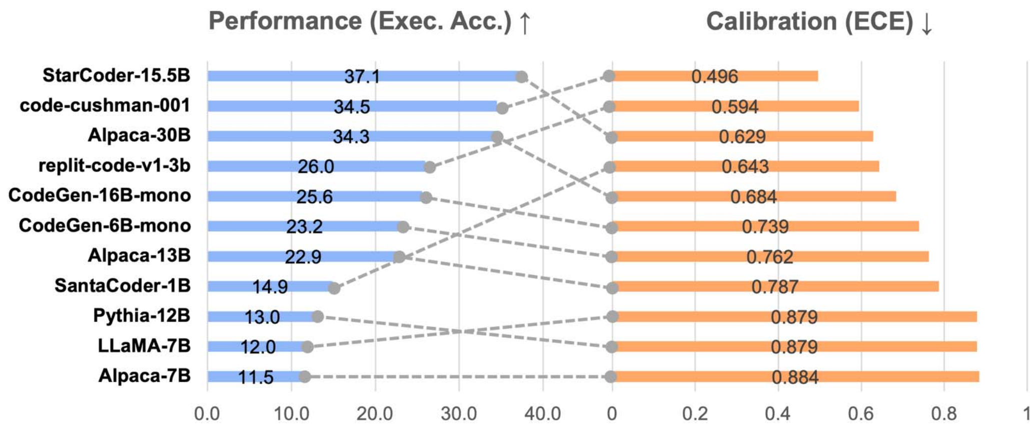

Figure 5: Avg. perf. across selected datasets (*i.e.,* Spider, WikiTQ, GSM8k and MBPP)[14] and their calibration score rankings.

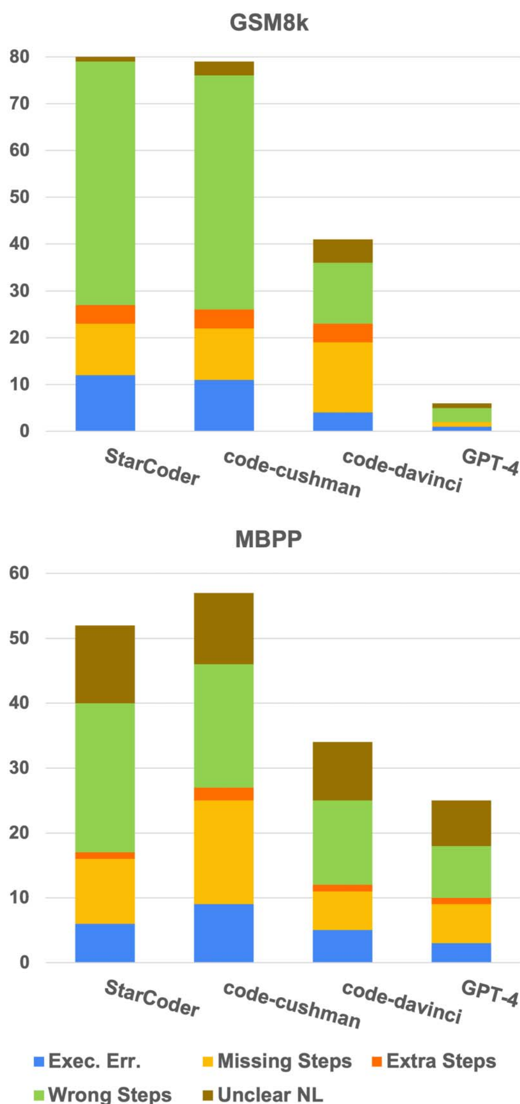

Figure 6: Error analysis on 100 examples for GSM8k and MBPP.

5) when the NL specification itself is ambiguous and *unclear*.

From the results shown in Figure 6, we can see that for GSM8k, compared with stronger models (*e.g.,* code-davinci and GPT-4), while a similar number of errors are made for missing and generating extra steps for solving the math problem, StarCoder and code-cushman make more mistakes in predicting intermediate steps, or generating deformed programs. On MBPP, however, weaker models are prone to miss crucial steps in the implementation, which shows a lack of understanding of the problem as well as planning abilities. Hallucination (Ji et al., 2023) is a common issue in natural language generation, while we find it to be rare for models to generate lines of code that are extraneous, hallucination can also exhibit as using wrong operators or introducing variable values that do not exist in the natural language description, which would be categorized as ‘‘wrong steps’’ in Figure 6.

4 Limitations

While we strive to provide a comprehensive and fair evaluation of LLMs in **`L2C`** tasks, here we also discuss the limitations of **`L2CEval`**.

Generation Using Greedy Decoding. In this work, we use greedy decoding to generate a single program for each example as the models' output. While this is the most efficient way of generation and ensures fair comparison for different models as it is not affected by factors like sampling temperature, it is also relatively noisy (Nijkamp et al., 2022; Chen et al., 2021). For tasks such as MBPP or Python programming in general, sampling k solutions then measure *pass@k* (any of the k programs being correct) or *n@k* (*i.e.,* # the k programs being correct) are better as they give

the model k tries to generate the correct program to lower the variance. For Python programming tasks, such methods are closer to practical use cases as we typically have test cases that can filter out some incorrect programs in the samples. For other tasks, having a better *pass@k* also provides opportunities for post-generation reranking methods such as (Shi et al., 2022; Zhang et al., 2023; Ni et al., 2023b). However, the cost for evaluating *pass@k* or *n@k* is k times of the compute compared with greedy decoding, thus we choose to only evaluate greedy decoding in this work and leave sampling-based evaluation to future work.

Execution-based Evaluation. Moreover, we mainly rely on execution-based evaluation (*i.e.,* execution accuracy) for this work. However, such evaluation may produce spurious programs, *i.e.,* false-positive programs that achieve the correct execution result by chance (Zhong et al., 2020; Ni et al., 2020). In this work, we adopt human evaluation to measure the problem of spuriousness and found non-trivial portion of ''correct'' programs being spurious for Spider but not for other datasets. In addition, execution may not always be straightforward in practice, especially when complex dependencies and potentially harmful programs are considered (Chen et al., 2021).

Confounding Factors During Comparison. When comparing models, especially across different model series, there are typically multiple performance-impacting factors that are in effect at the same time, some of which are not studied in this work, such as model architecture and pretraining objective. Such confounding factors may limit the validity of the conclusions that we draw from model comparisons. In this work, we try to mitigate this by fixing as many variables about the models as possible during a comparison, such as making observations within the same model series. While the general trend can still be observed across different model series, we should also note that when interpreting the results, readers should be mindful of such confounding factors when comparing different models.

Lack of Information for Proprietary Models. For the open-access proprietary LLMs (*e.g.,* OpenAI models), due to the lack of basic information and mismatches between the models described in the papers and the actual API engines, very few scientific conclusions can be drawn from these results. We evaluate such proprietary models mainly to provide baselines and in the hope of helping practitioners in choosing models for their use cases. We also present human evaluations of some of the strong models to discuss differences in common error modes. However, when making our findings, we generally rely on open-source models instead, to avoid being misled by speculative model details of such closed-source models.

5 Related Work

Code Generation Evaluation. Several code generation benchmarks are collected from raw data from GitHub and StackOverflow, and involve professional annotators to enhance the quality of the data (Iyer et al., 2018; Agashe et al., 2019; Yin et al., 2018). While such benchmarks focus more on lexical-based evaluation, ODEX (Xuezhi Wang et al., 2023) introduces execution-based evaluation, which has also been widely applied in recent code generation evaluation benchmarks, such as DS-1000 (Lai et al., 2023), HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021). More recently, there has been an increasing focus on assessing the generalization capabilities of code generation models across multiple programming languages (Athiwaratkun et al., 2023), and benchmarks such as CodeGeeX (Zheng et al., 2023) and MultiPL-E (Cassano et al., 2023). In our work, we focus on studying whether LLMs can map natural language instructions to code using the *most popular* programming languages for each domain (*i.e.,* SQL for semantic parsing and Python for math reasoning and programming). While the study of different programming languages are orthogonal to our work, we refer the readers to these existings works on multi-lingual evaluation benchmarks.

Other Code-related Tasks. Large language models have also shown significant success in other code-related directions. One popular direction is code understanding. For example, CodeXGLUE (Lu et al., 2023) comprises three widely used code understanding tasks including defect detection, clone detection, and code search. However, CONCODE (Iyer et al., 2018) is the only language-to-code task included in CodeXGLUE and it uses surface-form based evaluation metrics such as BLEU. BigCloneBench (Krinke and Ragkhitwetsagul 2022) tasks to measure the similarity between code pairs to predict whether

they have the same functionality. CodeSearchNet (Husain et al., 2019) is a benchmark of semantic code search given natural language queries. Besides code understanding, there have been other tasks such as code translation (Roziere et al., 2020) and program repair (Gupta et al., 2017). We leave systematic evaluation of LLMs on those tasks as important future work.

6 Conclusions

In this paper, we present **L2CEval**, a comprehensive evaluation framework for natural language to code generation, and we evaluate 56 models from 13 organizations, on 7 tasks from 3 core domains. **L2CEval** investigates models' performance on a variety of axes such as model scale, training data mixture, sensitivity to few-shot exemplars as well as the impact of instruction tuning, *inter alia*. We also present an analysis on the model calibration and conduct a human evaluation of common error modes across different models. We hope our study will provide useful insights for the community into applying LLMs for downstream code applications and future model development efforts.

Acknowledgments

We would like to thank Rui Zhang and Tao Yu for the initial discussions about this work, and Hailey Schoelkopf and Zhangir Azerbayev for their helpful discussions and suggestions. The authors would also like to thank the TACL reviewers and action editor David Chiang for the careful reviews and valuable feedbacks. This work is supported in part by a gift from Salesforce Research.

References

Replit model. https://huggingface.co/replit/replit-code-v1-3b. Accessed: 2023-09-30.

Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. 2019. A comparative survey of recent natural language interfaces for databases. *The VLDB Journal*, 28:793–819. https://doi/10.1007/s00778-019-00567-8

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5436–5446. Hong Kong, China. Association for Computational Linguistics. https://doi.org/10.18653/v1/D19-154

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. 2023. Santacoder: Don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Maitha Alhammadi, Mazzotta Daniele, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. The falcon series of language models: Towards open frontier models.

Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, et al. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571. https://doi.org/10.1162/tacl_a_00333

Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases–an introduction. *Natural Language Engineering*, 1(1):29–81. https://doi.org/10.1017/S135132490000005X

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan

Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2023. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. 2022. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*. https://doi.org/10.18653/v1/2022.bigscience-1.9

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* pages 1–17. https://doi.org/10.1109/TSE.2023.3267446

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

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. *ArXiv preprint arXiv:arXiv:2210.02875*.

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. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. 2023. Free dolly: Introducing the world's first truly open instruction-tuned llm.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis.

Yao Fu, Hao Peng, and Tushar Khot. 2022. How does gpt obtain its ability? Tracing emergent abilities of language models to their sources. *Yao Fu's Notion*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. 2017. On calibration of modern neural networks. In *International Conference on Machine Learning*, pages 1321–1330. PMLR.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing common C language errors by deep learning. In *AAAI Conference on Artificial Intelligence*. https://doi.org/10.1609/aaai.v31i1.10742

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Luke Benson, Lucy Sun, Ekaterina Zubova, Yujie Qiao, Matthew Burtell, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Alexander R. Fabbri, Wojciech Kryscinski, Semih Yavuz, Ye Liu, Xi Victoria Lin, Shafiq Joty, Yingbo Zhou, Caiming Xiong, Rex Ying, Arman Cohan, and Dragomir Radev. 2022. Folio: Natural language reasoning with first-order logic. *arXiv preprint arXiv:2209.00840*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training compute-optimal large language models.

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. 2023. Inner monologue: Embodied reasoning through planning with language models. In *Conference on Robot Learning*, pages 1769–1782. PMLR.

Hamel Husain, Hongqi Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *ArXiv*, abs/1909.09436.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652. https://doi.org/10.18653/v1/D18-1192

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023.

Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38. https://doi.org/10.1145/3571730

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Jens Krinke and Chaiyong Ragkhitwetsagul. 2022. Bigclonebench considered harmful for machine learning. *2022 IEEE 16th International Workshop on Software Clones (IWSC)* pages 1–7. https://doi.org/10.1109/IWSC55060.2022.00008

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2024. Can llm already serve as a database interface? A big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.

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. 2023. Starcoder: May the source be with you! *arXiv preprint arXiv:2305.06161*.

Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. 2022a. On the advance of making language models better reasoners. *arXiv preprint arXiv:2206.02336*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022b. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard,

Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin 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. 2023. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct.

Swaroop Mishra, Matthew Finlayson, Pan Lu, Leonard Tang, Sean Welleck, Chitta Baral, Tanmay Rajpurohit, Oyvind Tafjord, Ashish Sabharwal, Peter Clark, and Ashwin Kalyan. 2022. Lila: A unified benchmark for mathematical reasoning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5807–5832. https://doi.org/10.18653/v1/2022.emnlp-main.392

Mahdi Pakdaman Naeini, Gregory Cooper, and Milos Hauskrecht. 2015. Obtaining well calibrated probabilities using Bayesian binning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29. https://doi.org/10.1609/aaai.v29i1.9602

Ansong Ni, Jeevana Priya Inala, Chenglong Wang, Alex Polozov, Christopher Meek, Dragomir Radev, and Jianfeng Gao. 2023a. Learning math reasoning from self-sampled correct and partially-correct solutions. In *The Eleventh International Conference on Learning Representations*.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023b. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

Ansong Ni, Pengcheng Yin, and Graham Neubig. 2020. Merging weak and active supervision for semantic parsing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8536–8543. https://doi.org/10.1609/aaai.v34i05.6375

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue.

OpenAI. 2023. Gpt-4 technical report.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

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

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1470–1480. Beijing, China. Association for Computational Linguistics. https://doi.org/10.3115/v1/P15-1142

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are nlp models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2080–2094. https://doi.org/10.18653/v1/2021.naacl-main.168

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, 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. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546. https://doi.org/10.18653/v1/2022.emnlp-main.231

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 2020. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10737–10746. IEEE. https://doi.org/10.1109/CVPR42600.2020.01075

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca

MosaicML NLP Team. 2023a. Introducing mpt-30b: Raising the bar for open-source foundation models. Accessed: 2023-06-22.

MosaicML NLP Team. 2023b. Introducing mpt-7b: A new standard for open-source, commercially usable llms. Accessed: 2023-05-05.

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. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

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. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 billion parameter autoregressive language model. https://github.com kingoflolz/mesh-transformer-jax

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1271–1290. https://doi.org/10.18653/v1/2023.findings-emnlp.89

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022a. Emergent abilities of large language models. *Transactions on Machine Learning Research*.

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

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2023. Generating sequences by learning to self-correct. In *The Eleventh International Conference on Learning Representations*.

Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, et al. 2022. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 602–631. https://doi.org/10.18653/v1/2022.emnlp-main.39

Yiheng Xu, SU Hongjin, Chen Xing, Boyu Mi, Qian Liu, Weijia Shi, Binyuan Hui, Fan Zhou, Yitao Liu, Tianbao Xie, Zhoujun Cheng, Siheng Zhao, Lingpeng Kong, Bailin Wang, Caiming Xiong, and Tao Yu. 2023. Lemur: Harmonizing natural language and code for language agents. In *The Twelfth International Conference on Learning Representations*.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486. https://doi.org/10.1145/3196398.3196408

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450. Vancouver, Canada. Association for Computational Linguistics.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. Tabert: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426. https://doi.org/10.18653/v1/2020.acl-main.745

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921. Brussels, Belgium. Association for Computational Linguistics. https://doi.org/10.18653/v1/D18-1425

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26–29, 2005*, pages 658–666. AUAI Press.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for

code generation. In *International Conference on Machine Learning*, pages 41832–41846. PMLR.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shanshan Wang, Yufei Xue, Zi-Yuan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *ArXiv*, abs/2303.17568.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for text-to-sql with distilled test suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 396–411. https://doi.org/10.18653/v1/2020.emnlp-main.29

Shuyan Zhou, Pengcheng Yin, and Graham Neubig. 2022. Hierarchical control of situated agents through natural language. In *Proceedings of the Workshop on Structured and Unstructured Knowledge Integration (SUKI)*, pages 67–84. https://doi.org/10.18653/v1/2022.suki-1.8