~~DF1.APPEND(DF2)~~ PD.CONCAT([DF1,DF2])

GITCHAMELEON: UNMASKING THE VERSION-SWITCHING CAPABILITIES OF CODE GENERATION MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

The rapid evolution of software libraries presents a significant challenge for code generation models, which must adapt to frequent version updates while maintaining compatibility with previous versions. Existing code completion benchmarks often overlook this dynamic aspect, and the one that does consider it relies on static code prediction tasks without execution-based evaluation, offering a limited perspective on a model's practical usability. To address this gap, we introduce **GitChameleon**, a novel, manually curated dataset comprising 116 Python code completion problems, each conditioned on specific library versions and accompanied by executable unit tests. **GitChameleon** is designed to rigorously assess the ability of modern large language models (LLMs) to generate version-specific code that is not only syntactically correct but also functionally accurate upon execution. Our comprehensive evaluations reveal that state-of-the-art LLMs struggle with this task; for instance, **GPT-4** achieves a pass@10 of only 39.9% (43.7% when provided with error feedback), highlighting the complexity of the problem and the limitations of current models. By providing an execution-based benchmark that emphasizes the dynamic nature of code libraries, **GitChameleon** serves as a critical tool for advancing the development of more adaptable and reliable code generation models. We release the dataset and evaluation framework to encourage further research in this vital area.

1 INTRODUCTION

Large Language Models (LLMs) have become highly popular in code completion, to the extent that they are now deployed as virtual coding assistants within popular code editors[1], enhancing the overall coding workflow. Code, being a dynamic and constantly evolving environment, necessitates a continuous process of adaptation to stay in sync with the rapidly shifting paradigms, frameworks, and methodologies within the software development domain. The inherent variability in coding styles, the emergence of new programming languages, and the continuous evolution of libraries and packages underscore the imperative for an active approach in updating code generation models.

In response to the needs of practical coding environments, several large language models (LLMs) have been introduced, including StarCoder (Li et al., 2023), DeepSeek-Coder (Guo et al., 2024), CodeLlama (Rozière et al., 2023), among others. Despite these advancements, existing LLMs often struggle to keep pace with the rapid changes in codebases, particularly when tasked with generating version-specific code that is both syntactically and functionally accurate. This issue is especially critical, as developers increasingly depend on AI-assisted coding tools to boost productivity and maintain code quality. A recent Stack Overflow survey revealed that 70% of the participants are using or planning to integrate AI coding tools, 33% citing increased productivity as the primary motivation to incorporate these tools into their workflows[2].

[1] https://github.com/features/copilot
[2] https://stackoverflow.co/labs/developer-sentiment-ai-ml/

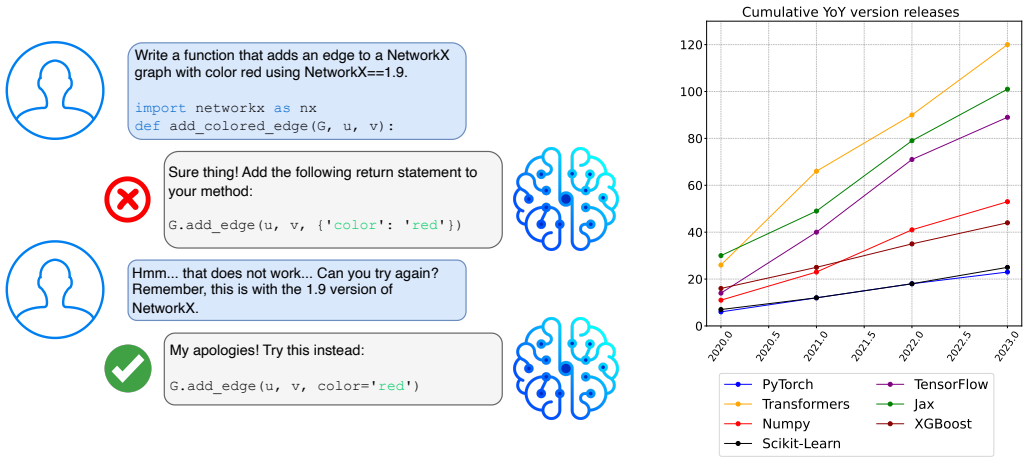


Figure 1: **Left:** Modern LLMs often struggle with generating version-accurate code, highlighting the need for benchmarks that specifically assess their ability to handle versioning. **Right:** Cumulative year-over-year version releases of popular Python-based machine learning libraries show a consistent upward trend, reflecting the rapid pace of development and version updates of code libraries and packages.

Given the rapid development and release cycles of popular libraries, as shown in Figure 1 (right), the need for code generation models to continually adapt to changing API's is more pressing than ever. For example, prominent machine learning and deep learning libraries like `PyTorch` (Paszke et al., 2019), `NumPy` (Harris et al., 2020), and `Scikit-Learn` (Buitinck et al., 2013) undergo frequent updates, which are reflected in a consistent upward trend in user downloads and version releases. This dynamic nature of code requires models that can adapt and generate code that adheres to the latest versions and practices, a need that current models often fail to meet comprehensively. In addition, certain hardware is restricted to compatibility with specific versions of commonly used packages, which adds an additional layer of complexity beyond merely updating the knowledge base of a code LLM to the latest library versions.

In response to these challenges, our work introduces a novel benchmark designed to assess the ability of LLMs to generate version-specific code. We propose **GitChameleon**, a benchmark that evaluates state-of-the-art code models by requiring them to produce executable code based on version-specific prompts. This code is then executed to verify its correctness against expected outputs. By highlighting the limitations of current models in generating accurate version-specific code, **GitChameleon** provides a structured approach to enhance these models and ensure their practical utility in real-world coding environments.

In summary, our contributions are highlighted as follows: 1) we introduce a novel code completion benchmark **GitChameleon** consisting of 116 Python-based version conditioning problems including human written unit tests; 2) we systematically analyse the version-specific performance of state-of-the-art code generation LLMs on API change type, version release year, and specific libraries. 3) we demonstrate the effectiveness of utilizing error log feedback as a way to improve version conditioning performance of code generation LLMs.

2 GitChameleon Benchmark

We introduce **GitChameleon**, a benchmark comprising 116 Python-based version conditioning problems focused on popular code libraries. To evaluate LLM performance on GitChameleon, each problem is accompanied by handwritten assertion-based unit tests, enabling a thorough execution-based assessment of the outputs generated by the code LLMs. This structured approach enables a thorough understanding and categorization of LLM failures in common scenarios involving version-specific code generation problems. In the following sections, we detail the benchmark statistics, data collection methodology, and sample verification process.

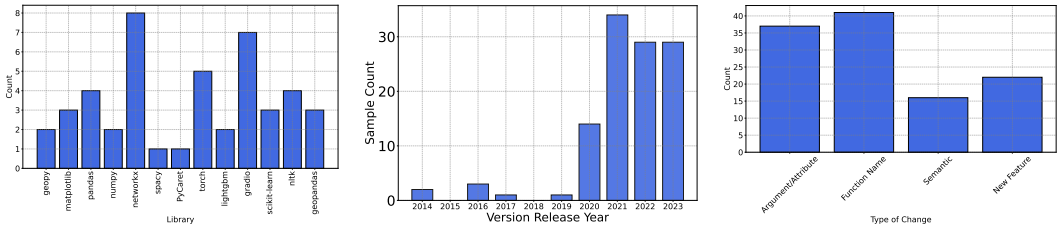
Table 1: Compared to other popular code generation benchmarks, including those evaluating version conditioning, **GitChameleon** features library- and version-specific problems with unit tests based on real version changes, closely aligning with practical settings.

Dataset	Problems	Data Source	Library Specific	Version Specific	Execution based	Real
HumanEval (Chen et al., 2021)	164	Hand-Written	✗	✗	✓	-
MBPP (Austin et al., 2021)	974	Hand-Written	✗	✗	✓	-
MTPB (Nijkamp et al., 2022)	115	Hand-Written	✗	✗	✓	-
APPS (Hendrycks et al., 2021)	10000	Competitions	✗	✗	✓	-
CodeContests (Li et al., 2022)	117	Competitions	✗	✗	✓	-
JulCe (Agashe et al., 2019)	1518049	Notebooks	✗	✗	✓	-
DSP (Chandel et al., 2022)	1119	Notebooks	✓	✗	✓	-
CoNaLa (Yin et al., 2018)	2879	StackOverflow	✓	✗	✗	-
DS-1000 (Lai et al., 2023)	1000	StackOverflow	✓	✗	✓	-
BigCodeBench (Zhuo et al., 2024)	1140	Expert-Curated	✓	✗	✓	-
Versicode (Wu et al., 2024b)	98692	GitHub, StackOverflow	✓	✓	✗	✓
CodeUpdateArena (Liu et al., 2024)	670	LLM-Generated	✓	✓	✓	✗
Wang et al. (2024)	28125	API change logs	✓	✓	✗	✓
GitChameleon (Ours)	116	Handwritten and LLM-assisted	✓	✓	✓	✓

2.1 STATISTICS

GitChameleon consists of 116 python-based version conditioned problems based on 11 libraries: `PyTorch` (Paszke et al., 2019), `Geopandas` (Jordahl et al., 2020), `NLTK` (Bird & Loper, 2004), `NetworkX` (Hagberg et al., 2008), `GeoPy`[3], `Gradio` (Abid et al., 2019), `Scikit-Learn` (Buitinck et al., 2013), `Matplotlib` (Hunter, 2007), `PyCaret`[4], `Pandas` (pandas development team, 2020; Wes McKinney, 2010) and `NumPy` (Harris et al., 2020). The samples were collected from version releases over a period from the year 2014 to 2023 and excludes legacy and yanked version releases.

Using the `cl100k_base` tokenizer, we analyzed the token counts of the GitChameleon samples. The problem statements average 20.4 tokens, and the starter code averages 47.4 tokens, leading to a combined average of 67.8 tokens per sample. Including the prompt template utilized for evaluating instruction-tuned LLMs, the total token count across all samples is 19,409 tokens.



(a) Number of unique versions per library in **GitChameleon**.

(b) Number of samples by version release year.

(c) Number of samples by type of change.

Figure 2: Fine-grained statistics of the **GitChameleon** benchmark. (a) The library with the most unique versions in the dataset is networkx with 8, whereas only 1 version of spacy and PyCaret are represented in **GitChameleon**. (b) Most versions in the dataset were released between 2021-2023, with a few versions released in earlier years. (c) The most common type of changes between versions were function name changes and argument/attribute changes, while semantic/output changes were least common.

As demonstrated in Fig. 2b, most of the samples in GitChameleon are from versions of libraries released in the years 2021, 2022 and 2023, with 2021 released version samples accounting for 35% of the total sample count in the benchmark. Since some of the models evaluated on GitChameleon have disclosed their training data cutoff dates, we have ensured that most, if not all, samples fall within the training window of these models. This approach helps to ensure that the models during their training period are likely exposed to the versions on which the samples are based. Fig. 2a shows that `NetworkX` (Hagberg et al., 2008) and `Gradio` (Abid et al., 2019) have the most versions in our

[3] `https://pypi.org/project/geopy/`
[4] `https://pycaret.org/`

benchmark (8 and 7, respectively). Meanwhile, `PyTorch` (Paszke et al., 2019) and `NumPy` (Harris et al., 2020) have the highest number of samples (18 and 15, respectively), together accounting for 34% of the total sample count. Further, we annotate each sample with the type of change that is classified into the following categories:

- **Argument or Attribute change**: The API call to a function, method, or class has a change in arguments (e.g. name, order, new, deprecated argument) between versions.
- **Function Name change**: The name of the API call has changed between versions (e.g. `pandas.append` to `pandas.concat`).
- **Semantics or Function Behaviour change**: The semantic / runtime behaviour of the API call changed between versions (e.g. returning a different type).
- **New feature or additional dependency based change**: A feature was introduced in a specific version, therefore, to execute the same functionality, a model using an older version should make use of an additional dependency (e.g. `torch.special` was introduced in TORCH 1.10, previously one could use NUMPY for the same).

Most samples in the GitChameleon benchmark fall under the Argument or Attribute and Function Name change category, as these are the most frequent and expected types of changes in mature and stable libraries.

Differentiating factor Several datasets examine LLM interactions with version-specific code, including Versicode (Wu et al., 2024b), CodeUpdateArena (Liu et al., 2024), and the dataset by Wang et al. (2024). While these datasets are valuable, our dataset offers a unique and complementary perspective by focusing on the real-world scenario where developers are often constrained to specific library versions due to technical debt. CodeUpdateArena investigates model adaptation to synthetic API changes, we focus our evaluation on real API changes to assess how effectively an LLM can generate code for version-specific changes of library versions that they have been trained with. In contrast, Versicode and Wang et al. (2024)'s datasets, while addressing library evolution, primarily rely on string matching for evaluation. Our approach diverges by incorporating executable tests, providing a more practical and rigorous assessment of code generation capabilities.

2.2 COLLECTION FRAMEWORK

Task Description

```
# Write a function that checks if all
    elements in an array are true.
```

Expected Result

```
np.all(arr)
```

Starter Code

```
import numpy as np
def alltrue_fn(arr):
    return
```

Assertion Test

```
arr = np.array([1, 1, 1, 1])
result = alltrue_fn(arr)
assert result == np.all(arr)
```

Table 2: Example of a problem statement derived from a changelog entry from Numpy 1.25.0

The examples were manually crafted by the authors, who divided the task among themselves. We compiled a list of popular Python libraries, focusing on those with which at least one author was familiar and that had detailed changelogs documenting changes between versions. For each library, we reviewed the changelogs to identify deprecated functions, argument changes, alterations in behavior, and newly introduced functions.

For each identified change, we create a concise problem statement, write the starter code, define the expected solution, and develop an assertion test. For instance, in Table 2, we illustrate an example based on the changelog for version 1.25.0 of `NumPy` (Harris et al., 2020), a library for scientific computing in Python. This changelog notes that "`np.alltrue` is deprecated. Use `np.all` instead." We used this change to craft a problem statement that tests the LLMs' ability to recognize and adapt to version-specific updates.

Unit test and evaluation framework verification To assess the correctness of the evaluation framework of GitChameleon, we needed to verify three key aspects:

- **Compilation**: Ensure that the starter code compiles successfully.
- **Assertion unit tests**: Confirm that the assertion tests function correctly.
- **Dependencies**: Verify that all necessary external dependencies are installed, excluding the ones being tested.

We used `venv` to create and manage virtual environments for testing. This process involved installing the appropriate library version and any additional dependencies. We then combined the starter code, expected result, and the assertion test into a single script, which was executed to verify all three criteria. We provide pseudocode for our verification process in appendix A.2.

3 Empirical Study

We evaluate state-of-the-art large language models (LLMs) using the **GitChameleon** benchmark to assess their ability to generate version-specific, executable code. This study highlights how well current models adapt to dynamic library versions and produce functionally correct code that passes the provided unit tests.

3.1 Experimental setup

For each open-source LLM, we downloaded the corresponding Hugging Face (HF) weights and served the models using Text Generation Inference (TGI). We used a single NVIDIA 95GB H100 GPU for models with fewer than 70 billion parameters, two GPUs for models more than 70 billion parameters.

We configured the generation parameters with a `top_p` value of 0.95, `top_k` of 50, and a temperature of 0.3 for **Pass@1** and 0.8 for **Pass@10**, in addition to finding the optimal temperature for each model. The maximum number of new tokens generated was set to 256. Additionally, we enabled flash attention (Dao et al., 2022) for all models to enhance inference efficiency. A list of all the models and there cutoff dates if available is provided in appendix A.3.

3.2 Evaluation Metrics

To comprehensively evaluate the performance of code generation models using the **GitChameleon** dataset, we employ a range of execution-based metrics. These metrics assess not only the correctness of the generated code but also its efficiency and adaptability to different versions.

`Pass@k` measures the proportion of problems for which at least one of the top k generated solutions passes all assertion tests. This metric provides insight into the model's ability to generate functionally correct code. For each problem, we generate n code samples, and compute the pass at k metric by the corrected formula:

```
    def corrected_pass_at_k(n, c, k=10):
    if n - c < k: return 1.0
    return 1.0 - np.prod(1.0 - k / np.arange(n - c + 1, n + 1))
```

For instruct models, we run the model's parsed output as standalone code, and for base models, the concatenation of the starting code and model's parsed output (completion).

`Greedy` refers to the standard greedy decoding method, where the most probable token from the next-token distribution is deterministically chosen. This is analogous to setting the temperature to 0.

`Error Feedback` adds the error log to the prompt (after executing the generated code from the model with the initial prompt). Then, the pass@k metric is recalculated based on the model's generated code using the prompt with error feedback. See appendix A.1 for an example.

3.3 Main Results

We report the performance of both base and instruct-tuned models on the **GitChameleon** benchmark in Tables 3 and 4, respectively. Our analysis reveals a strong positive correlation between

model size and performance in version-specific code generation tasks. For base models, Spearman's rank correlation coefficients are 0.82 for Pass@1 and 0.69 for Pass@10 (both p-values <0.01), indicating that larger models generally perform better. Specifically, DeepSeek-Coder 33B achieved the highest Pass@1 score of 35.7%, highlighting its proficiency in generating correct solutions on the first attempt, while CodeLlama 34B outperformed others at Pass@10 with a score of 42.8%, demonstrating its ability to produce correct solutions given multiple attempts.

Model	**Size**	**Pass@1** T=0.3	**Pass@10** T=0.8
CodeLlama	7B	20.4±1.6	36.1±5.5
	13B	25.8±1.0	36.4±2.0
	34B	30.6±1.4	**42.8±1.4**
Starcoder2	3B	11.9±1.9	27.1±1.9
	7B	15.5±1.1	23.1±2.6
	15B	13.7±1.7	27.0±3.4
Llama-3	8B	22.3±1.0	32.0±2.1
	70B	27.2±3.0	41.3±2.5
Qwen2	7B	27.4±1.2	37.7±1.8
	72B	33.2±2.1	39.7±5.5
Starcoderbase	1B	13.3±1.0	20.3±1.2
	3B	15.5±1.2	26.5±1.5
	7B	20.0±0.9	31.3±4.1
	15B	16.9±1.8	30.8±2.6
Starcoder	15B	16.0±1.2	35.9±1.9
Deepseek-coder	1.3B	22.0±2.5	28.0±1.9
	6.7B	31.0±1.8	36.1±0.7
	33B	**35.7±3.0**	37.9±4.9

Table 3: **Base Model Performance Metrics.** Deepseek-coder-33B is the strongest model for Pass@1 (temperature 0.3), while CodeLlama-34B is the strongest model when we compute Pass@10 with an increased number of generations (20) sampled at temperature 0.8. We observe that there is a strong positive correlation between model size and performance, with Spearman's rank correlation coefficients of 0.82 for Pass@1 and 0.69 for Pass@10.

Similarly, for instruct-tuned models, we observe Spearman's rank correlation coefficients of 0.52 for Pass@1 and 0.70 for Pass@10 (both with p-values under 1%), confirming the positive correlation between model size and performance. Phi-3.5-MoE (16×3.8B) achieved the highest baseline Pass@1 score of 30.9% (33.6% greedy) and Pass@10 (40.5%). GPT-4o outperformed others at Pass@10 with error feedback with a score of 43.7%. Incorporating error feedback led to average improvements of 4.47% in Pass@1 and 3.51% in Pass@10 across instruct-tuned models. Additionally, Pass@10 with n=20 samples showed an average performance improvements of 10.6% for base models and 14.8% for instruct-tuned models over Pass@1 with n=5. These findings highlight that scaling up model size, utilizing error feedback, and allowing multiple solution attempts are effective strategies for enhancing the ability of LLMs in handling version-specific code generation tasks.

3.4 In-Depth Analysis of Findings

In this section, we delve deeper into the results obtained from our experiments, analyzing model performance across various dimensions, including model size, year of library release, the type of API changes encountered and sample difficulty.

Analysis of Performance by Release Date At the top of Figure 4, we present the year-over-year performance of a subset of the instruction-finetuned models. The average performance of all models dropped significantly from 87.7% in 2019 (not shown) to 28.2% in 2023, with intermediate values of 79.1%, 45.2%, and 21.3%. This decline is likely due to the fact that the training datasets contain more data from earlier years, underscoring the need for code LLMs to better adapt to the evolving

Model	Size (Context) / Version	Pass@1 ($T = 0.3$)			Pass@10 ($T = 0.8$)	
		Baseline	+ Error Feedback	Greedy†	Baseline	+ Error Feedback
Starcoder2-v0.1	15B	22.4±0.9	27.9±1.2	21.6	37.4±1.0	38.1±1.3
CodeLlama	7B	16.5±1.0	19.6±0.9	19.0	27.2±2.2	32.2±1.7
	13B	20.3±0.6	25.6±1.3	22.4	35.7±0.8	41.2±1.0
Llama-3.1	8B	15.7±0.5	20.0±0.3	16.4	28.8±1.6	35.1±1.3
Llama-3.2	1B	9.3±0.4	12.0±0.7	9.5	16.2±0.7	20.1±0.7
	3B	10.4±0.6	14.0±0.4	12.1	20.2±0.6	25.4±0.7
CodeQwen1.5-Chat	7B	20.9±0.4	25.4±0.5	21.6	40.2±0.8	42.4±0.8
Qwen2	7B	17.8±0.4	24.8±1.5	18.1	38.4±1.0	40.7±1.0
	72B	26.0±0.6	29.0±0.4	26.7	38.2±0.7	40.8±0.3
Qwen2.5-Coder	1.5B	19.7±0.9	22.9±1.2	19.8	34.1±0.4	37.6±0.4
	7B	21.2±0.4	24.0±0.7	22.4	35.4±1.2	41.3±0.9
Codestral-v0.1	22B	25.1±0.6	31.6±0.4	25.0	37.4±0.3	41.5±0.3
Yi-Chat	6B	17.4±0.4	23.2±1.0	15.5	33.6±0.9	36.6±0.8
	9B	19.9±0.6	24.8±0.7	20.7	30.6±0.5	39.1±0.3
	34B	20.8±0.5	26.3±1.0	21.6	35.4±0.5	38.4±0.8
codegemma	7B	17.8±0.7	22.6±1.0	16.4	33.9±0.6	38.0±0.5
stable-code	3B	14.6±0.7	16.3±0.9	14.7	23.9±1.4	25.9±0.9
granite-code	3B (128k)	23.6±1.1	27.0±1.4	22.4	33.7±0.3	34.8±0.5
	8B (4k)	24.8±0.5	28.4±1.0	24.1	39.3±1.2	41.2±0.6
	8B (128k)	23.4±0.6	27.7±1.0	25.9	35.5±1.7	38.8±1.1
	20B (8k)	28.7±0.5	30.0±0.8	29.3	37.0±0.9	37.3±0.5
	34B (8k)	29.6±0.9	31.4±1.0	30.2	37.3±1.0	40.9±0.8
Phi-3.5-mini	3.8B	24.2±0.8	29.9±1.2	26.7	35.2±1.5	37.4±0.9
Phi-3.5-MoE	16x3.8B	**30.9**±0.8	**34.9**±0.7	**33.6**	40.5±0.5	43.2±0.1
Nxcode-CQ-orpo	7B	20.8±0.7	25.0±1.1	21.6	38.9±1.0	42.4±0.7
GPT	3.5	19.6±1.0	27.2±0.8	19.8	33.3±1.0	37.8±1.7
	4o (2024-05-13)	23.6±2.7	34.1±1.2	25.0	39.9±2.4	**43.7**±2.4

Table 4: **Instruct Model performance metrics.** (Top) OSS models, (bottom) closed-sourced models. We observe a 4.47% and 3.51% improvement with error feedback in Pass@1 and Pass@10, respectively. Additionally, there is a strong positive correlation between model size and performance, with Spearman's rank correlation coefficients of 0.52 for Pass@1 and 0.70 for Pass@10.

.

nature of code libraries and their versions. Interestingly, many models appear to improve with error feedback disproportionately across versions released in the years 2021-2023. For example, Qwen2-72B and Llama-3.2 3B improve more with feedback in 2022 compared to 2021 or 2023, while GPT-4o improves more with feedback in 2023. This raises a question about the extent to which models' training data influences the effectiveness of error feedback.

Analysis of Performance by Type of API Change At the bottom of Figure 4, our analysis of model performance across different types of API changes in **GitChameleon** revealed significant variations. The models struggled the most with **Semantics or Function Behaviour** changes, achieving an average Pass@1 score of only 7.34%. **Argument and Attribute** changes were the second most challenging, with an average Pass@1 score of 18.5%. In contrast, the models performed better on **Function Name** changes and **New Feature or additional dependency based** changes, with average Pass@1 scores of 50.5% and 48.6%, respectively.

In general, larger models are more robust to the name changes, argument/attribute changes, and new feature. However, all models perform very poorly on semantic changes, regardless of the availability of error feedback. This indicates a weakness of SotA code generation models, and an area for further investigation. Furthermore, error feedback appears to have a more significant impact in argument/attribute changes compared to the other types of changes. This indicates that the models

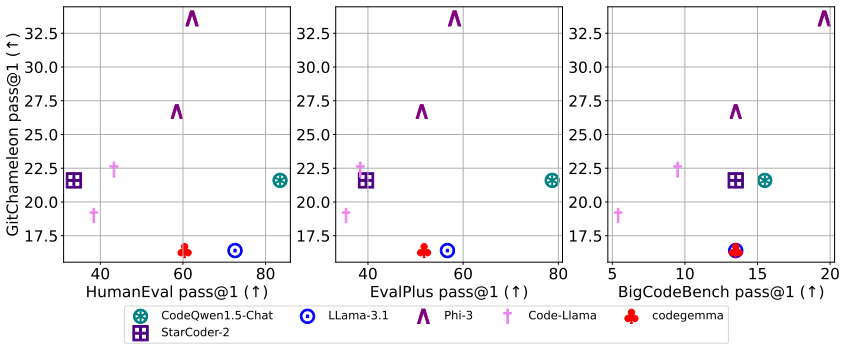


Figure 3: Correlation of **GitChameleon** with the representative code benchmarks (HumanEval, EvalPlus, and BigCodeBench-Hard Complete split). For each benchmark, the spearman correlation coefficient was -0.08, 0.07, and 0.35, respectively. While HumanEval and EvalPlus showed very weak correlations, BigCodeBench-Hard showed a positive correlation (+0.35) with **GitChameleon**.

may be using the error feedback to directly address failures in version-conditioned code generation, rather than non-specific errors such as syntax errors.

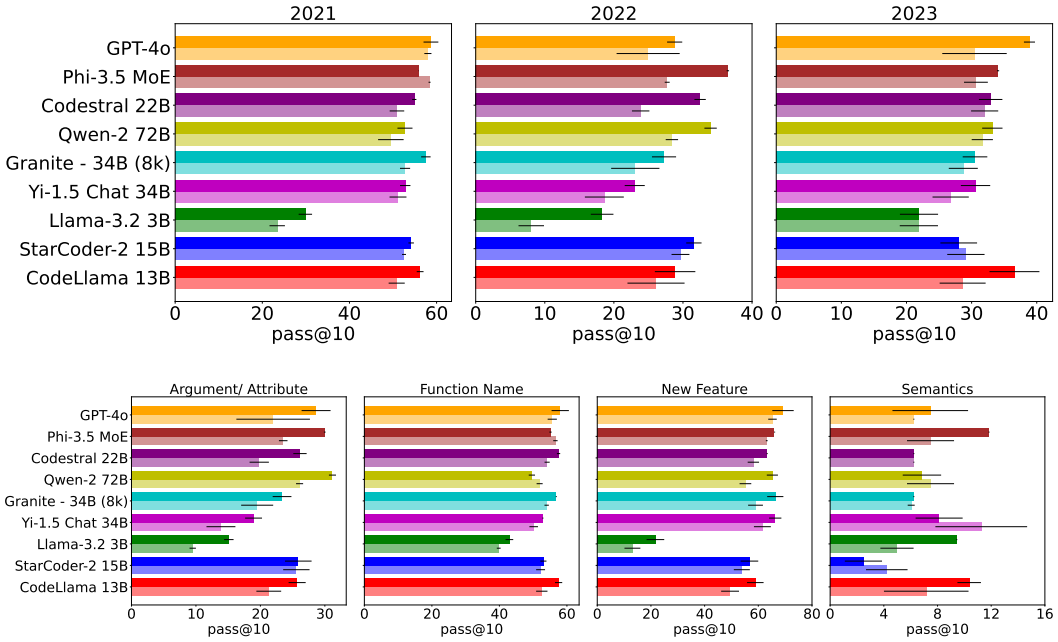


Figure 4: **Instruct-tuned model performance breakdown by version release year (top) and type of change (bottom)**: We analyze model performance in terms of `pass @ 10` for baseline and with error feedback generation across two dimensions: version release year and type of changes. Darker shaded bars represent values obtained via error feedback generation. Standard deviation is drawn as a black line, obtained from 5 random seeds. (Top) Many models perform poorly on 2022, and generally perform worse on more recent versions. (Bottom) All models perform very poorly at semantic changes, indicating an potential area for massive improvement. Most models perform well on function name changes and new feature (with the exception of Llama-3.2-3B, which is also the smallest model in this comparison).

Sample difficulty analysis Figure 3.4 shows the distribution of sample difficulty. Notably, individual models (right panel) often display bimodal distributions, meaning they tend to perform consistently well or poorly on specific problems. In contrast, the aggregate distribution (left panel) is

not bimodal, indicating that different models perform well on different sets of problems. The availability of error feedback shifts the distribution of the sample-wise difficulty to the right, as expected. Interestingly, some samples are not solved at all across models, even with feedback, and no samples are solved consistently by all models. As a further investigation, we plan to qualitatively examine samples that shift from unsolved to solved given error feedback. Finally, the right panel shows that many samples are either "easy" or "hard", however larger models tend to have more "medium" difficulty samples, indicating that scale can, at least partially, improve version-conditioned generation from unsolved to solved. We qualitatively demonstrate some of these examples in A.1.

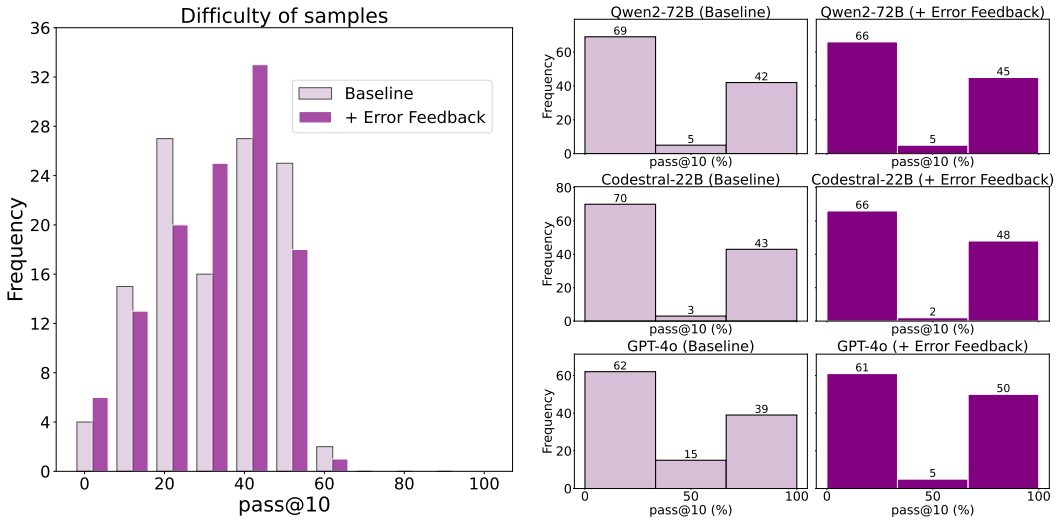


Figure 5: **Comparison of sample and model differences.** The left panel shows the distribution of sample difficulty, measured by the frequency of pass@10 scores across all models and seeds. The right panel presents the same distribution, but averaged for specific models across their seeds. Interestingly, individual models tend to exhibit bimodal distributions, indicating they are either consistently good or bad at specific problems. However, the aggregate distribution is not bimodal, suggesting that different models excel at different problems.

4 Related Work

Code LLM training and evaluation protocols Code LLM evaluations mainly revolve around code completion (Zhang et al. (2023); van Dam et al. (2023); Lu et al. (2021)). Existing benchmarks emphasize generic code completion, yet a recognized limitation is the inability of code LLM to generate and complete code that requires library and project-level knowledge (Xu & Zhu, 2022), let alone version-level knowledge, which is vital for real-world software applications.

Recent initiatives address repository-level code understanding by LLMs (Bairi et al. (2023); Shrivastava et al. (2023a;b); Liu et al. (2023); Guo et al. (2024)). Attempts at library-level code generation (Zan et al. (2022)) and consideration of dependencies between files (Guo et al. (2024)) have been made. However, these efforts do not directly address the challenge of accommodating version-sensitive changes, adding complexity.

The core issue arises from models being trained on library code without explicit knowledge of library versions or their functional changes. Consequently, when tasked with generating code specifically compatible with a particular library version, there is a significant risk models often encounter failures.

Datasets Existing datasets like HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and MTPB (Nijkamp et al., 2022) provide sets of handwritten prompts and test cases to evaluate code generated by code LLM. However, these datasets are relatively small and lack context regarding a model's comprehension of repositories. APPS (Hendrycks et al., 2021) and CodeContest (Li

et al., 2022) offer challenging datasets with coding competition questions, providing insights into a model's performance on difficult problems but without a focus on library-specific challenges. DSP (Chandel et al., 2022) and DS-1000 (Lai et al., 2023) concentrate on the top data science libraries in Python, while JulCe (Agashe et al., 2019) uses Jupyter Notebooks for training and evaluation, but these notebooks do not necessarily need to be repository-specific. CoNaLa (Yin et al., 2018) contains problems collected from StackOverflow across multiple programming languages, including both library-specific questions and non-library-specific code. More recently, BigCodeBench (Zhuo et al., 2024) is constructed to evaluate the comprehensive capabilities of code generation with tool use and instruction following, which poses a great challenge for existing models. Several datasets include version-specific code, such as Versicode (Wu et al., 2024b), CodeUpdateArena (Liu et al., 2024), and the dataset by Wang et al. Versicode's dataset, compiled from academic papers, Stack Overflow, and library source code, supports tasks like token, line, and block-level code completion and code editing. Unlike our dataset, Versicode evaluates using exact matches. Wang et al.'s dataset collects API mappings, such as "torch.lstsq() is deprecated in favor of torch.linalg.lstsq()," and evaluates LLMs using exact match, edit similarity, and fixed rate metrics. Although Versicode and Wang et al.'s datasets address the evolving nature of libraries, their evaluations are limited to string matching.

In contrast, CodeUpdateArena evaluates LLMs' ability to adapt to API changes, such as adding a boolean flag, by running tests. However, the dataset is synthetic and are not extracted from real-life version changes. For CodeUpdateArena, they also take the approach of training LLMs using the updated API function –using docstrings or examples–. It then tests if without access to the update during inference, the LLM's reflects the synthetic changes. While these datasets provide valuable resources for training and evaluating models, our **GitChameleon** dataset advances research into version-conditioned code generation by LLMs. Runnable tests offer insights into LLM adaptability, and **GitChameleon** further assesses a model's ability to differentiate between library versions, and successfully use a specific version.

Implications for Lifelong Learning Continual/lifelong learning in code generation models is in its early stages (Yadav et al., 2023; Weyssow et al., 2023; Wu et al., 2024a; Gao et al., 2023). However, current efforts often focus on artificial sequential tasks rather than utilizing the natural distribution shift in the chronological evolution of code. Notably, continual learning mainly targets mitigating catastrophic forgetting and balancing forward- and backward-transfer on a data stream, which may not align optimally with coding environment demands. In coding environments, obsolete or legacy libraries may prompt selective forgetting of irrelevant knowledge, particularly at the library/package level. Previous work, such as Caccia et al. (2021) may serve as a foundation for developing continual learning in Large Language Models for code.

5 Limitations

We consider the lack of prompt optimization done for the instruct models as a considerable limitation of our analysis. Furthermore, the dataset consists of 116 problems, which is relatively small compared to other code benchmarks. Finally, we do not explore approaches such as RAG, chain of thought, or finetuning on a split of our benchmark to observe an upper bound of performance on this task. Future work could explore such approaches using our dataset.

6 Conclusion

Recognizing the crucial need for code LLM adaptation to evolving code environments, particularly in widely used libraries, we introduce a novel and extensive Python-based version-specific benchmark named **GitChameleon**. By effectively leveraging **GitChameleon**, we expose the shortcomings of existing state-of-the-art (SoTA) models in producing version-specific code, representing an inaugural effort to draw attention to this challenge. While our work exposes this shortcoming, we acknowledge the dataset's limitations. In future endeavors, we aim to enhance the dataset's comprehensiveness across various programming languages and frameworks. Additionally, we plan to introduce new tasks that can benefit research on code LLM models using **GitChameleon**.

REFERENCES

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone, 2024. URL https://arxiv.org/abs/2404.14219.

Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan, Abdulrahman Alfozan, and James Zou. Gradio: Hassle-free sharing and testing of ml models in the wild, 2019. URL https://arxiv.org/abs/1906.02569.

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. Juice: A large scale distantly supervised dataset for open domain context-based code generation. *arXiv preprint arXiv:1910.02216*, 2019.

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

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-level Coding using LLMs and Planning, September 2023. URL http://arxiv.org/abs/2309.12499. arXiv:2309.12499 [cs].

Steven Bird and Edward Loper. NLTK: The natural language toolkit. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*, pp. 214–217, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://aclanthology.org/P04-3031.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

Massimo Caccia, Pau Rodriguez, Oleksiy Ostapenko, Fabrice Normandin, Min Lin, Lucas Caccia, Issam Laradji, Irina Rish, Alexandre Lacoste, David Vazquez, and Laurent Charlin. Online Fast Adaptation and Knowledge Accumulation: a New Approach to Continual Learning, January 2021. URL http://arxiv.org/abs/2003.05856. arXiv:2003.05856 [cs].

Shubham Chandel, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. Training and evaluating a jupyter notebook data science assistant. *arXiv preprint arXiv:2201.12901*, 2022.

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. Evaluating large language models trained on code. 2021.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai,

Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco Guzmán, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vítor Albiero, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. Keeping pace with ever-increasing data: Towards continual learning of code intelligence models, 2023.

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. URL https://arxiv.org/abs/2401.14196.

Aric Hagberg, Pieter J Swart, and Daniel A Schult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2008.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard,

Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

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

J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3): 90–95, 2007. doi: 10.1109/MCSE.2007.55.

Kelsey Jordahl, Joris Van den Bossche, Martin Fleischmann, Jacob Wasserman, James McBride, Jeffrey Gerard, Jeff Tratner, Matthew Perry, Adrian Garcia Badaracco, Carson Farmer, Geir Arne Hjelle, Alan D. Snow, Micah Cochran, Sean Gillies, Lucas Culbertson, Matt Bartos, Nick Eubank, maxalbert, Aleksey Bilogur, Sergio Rey, Christopher Ren, Dani Arribas-Bel, Leah Wasser, Levi John Wolf, Martin Journois, Joshua Wilson, Adam Greenhall, Chris Holdgraf, Filipe, and François Leblanc. geopandas/geopandas: v1.0.1, July 2020. URL https://doi.org/10.5281/zenodo.3946761.

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

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.

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. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems, October 2023. URL http://arxiv.org/abs/2306.03091. arXiv:2306.03091 [cs].

Zeyu Leo Liu, Shrey Pandit, Xi Ye, Eunsol Choi, and Greg Durrett. Codeupdatearena: Benchmarking knowledge editing on api updates, 2024. URL https://arxiv.org/abs/2407.06249.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

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. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

Meta. Llama 3: Meta's latest large language model. https://github.com/meta-llama/llama3, 2024. Accessed: 2024-06-03.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, Yan Koyfman, Boris Lublinsky, Maximilien de Bayser, Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Yi Zhou, Chris Johnson, Aanchal Goyal, Hima Patel, Yousaf Shah, Petros Zerfos, Heiko Ludwig, Asim Munawar, Maxwell Crouse, Pavan Kapanipathi, Shweta Salaria, Bob Calio, Sophia Wen, Seetharami Seelam, Brian Belgodere, Carlos Fonseca, Amith Singhee, Nirmit Desai, David D. Cox, Ruchir Puri, and Rameswar Panda. Granite code models: A family of open foundation models for code intelligence, 2024. URL https://arxiv.org/abs/2405.04324.

MistralAI. Codestral: Hello, world! 2024.

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

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky,

Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL https://doi.org/10.5281/zenodo.3509134.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, and Nathan Cooper. Stable code 3b. URL https://huggingface.co/stabilityai/stable-code-3b.

Alibaba Qwen. Qwen2 technical report. 2024.

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. Code Llama: Open Foundation Models for Code, August 2023. URL http://arxiv.org/abs/2308.12950. arXiv:2308.12950 [cs].

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. RepoFusion: Training Code Models to Understand Your Repository, June 2023a. URL http://arxiv.org/abs/2306.10998. arXiv:2306.10998 [cs].

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-Level Prompt Generation for Large Language Models of Code, June 2023b. URL http://arxiv.org/abs/2206.12839. arXiv:2206.12839 [cs].

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, Jane Fine, Tris Warkentin, Ale Jakse Hartman, Bin Ni, Kathy Korevec, Kelly Schaefer, and Scott Huffman. Codegemma: Open code models based on gemma, 2024. URL https://arxiv.org/abs/2406.11409.

Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024a. URL https://arxiv.org/abs/2403.05530.

Qwen Team. Code with codeqwen1.5, April 2024b. URL https://qwenlm.github.io/blog/codeqwen1.5/.

Qwen Team. Qwen2.5: A party of foundation models, September 2024c. URL https://qwenlm.github.io/blog/qwen2.5/.

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

Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. How and why llms use deprecated apis in code completion? an empirical study, 2024. URL https://arxiv.org/abs/2406.09834.

Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman (eds.), *Proceedings of the 9th Python in Science Conference*, pp. 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.

Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. On the usage of continual learning for out-of-distribution generalization in pre-trained language models of code. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1470–1482, 2023.

Tongtong Wu, Linhao Luo, Yuan-Fang Li, Shirui Pan, Thuy-Trang Vu, and Gholamreza Haffari. Continual learning for large language models: A survey, 2024a.

Tongtong Wu, Weigang Wu, Xingyu Wang, Kang Xu, Suyu Ma, Bo Jiang, Ping Yang, Zhenchang Xing, Yuan-Fang Li, and Gholamreza Haffari. Versicode: Towards version-controllable code generation, 2024b. URL https://arxiv.org/abs/2406.07411.

Yichen Xu and Yanqiao Zhu. A Survey on Pretrained Language Models for Neural Code Intelligence, December 2022. URL http://arxiv.org/abs/2212.10079. arXiv:2212.10079 [cs].

Prateek Yadav, Qing Sun, Hantian Ding, Xiaopeng Li, Dejiao Zhang, Ming Tan, Xiaofei Ma, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Mohit Bansal, and Bing Xiang. Exploring Continual Learning for Code Generation Models, July 2023. URL http://arxiv.org/abs/2307.02435. arXiv:2307.02435 [cs].

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, pp. 476–486, 2018.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. CERT: Continual Pre-Training on Sketches for Library-Oriented Code Generation, June 2022. URL http://arxiv.org/abs/2206.06888. arXiv:2206.06888 [cs].

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation, October 2023. URL http://arxiv.org/abs/2303.12570. arXiv:2303.12570 [cs].

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A Appendix

A.1 Qualitative Samples of Error Feedback

Listing 1: Original output from Codestral-22B-v0.1

```
# [problem]
Calculate the natural logarithm of the absolute value of the gamma
    function using pytorch special functions if available in this version
    , otherwise you may use another library.
# [starting code]
import torch
input_tensor = torch.linspace(0, 10, steps=10)
output =
# [sample solution]
torch.special.gammaln(input_tensor)

# [model output]
```

```
import torch
input_tensor = torch.linspace(0, 10, steps=10)
output = torch.log(torch.abs(torch.special.gamma(input_tensor)))
# [Error trace]
Traceback (most recent call last):
    output = torch.log(torch.abs(torch.special.gamma(input_tensor)))
AttributeError: module 'torch.special' has no attribute 'gamma'. Did you
    mean: 'digamma'?
```

Listing 2: Regenerated solution with Codestral-22B-v0.1 given error feedback

```
import torch
def natural_log_abs_gamma(input_tensor):
    gamma_tensor = torch.lgamma(input_tensor)
    natural_log_abs_gamma_tensor = torch.abs(gamma_tensor)
    return natural_log_abs_gamma_tensor
input_tensor = torch.linspace(0, 10, steps=10)
output = natural_log_abs_gamma(input_tensor)
```

Listing 3: Original output from Phi-3.5-MoE-instruct

```
# torch-1.10.0
# [problem]
Calculate the error function using pytorch special functions if available
     in this version, otherwise you may use another library.
# [starting code]
import torch\ninput_tensor = torch.linspace(0, 10, steps=10)\noutput =
# [sample solution]
torch.special.erf(input_tensor)

# [model output]
import torch
from scipy.special import erf
input_tensor = torch.linspace(0, 10, steps=10)
def calculate_erf(tensor):
    # Convert tensor to numpy array
    numpy_array = tensor.numpy()
    # Calculate erf using scipy.special.erf
    output = erf(numpy_array)
    # Convert numpy array back to tensor
    output_tensor = torch.from_numpy(output)
    return output_tensor
output = calculate_erf(input_tensor)
# [Error trace]
Traceback (most recent call last):
    from scipy.special import erf
ModuleNotFoundError: No module named 'scipy'
```

Listing 4: Regenerated solution with Phi-3.5-MoE-instruct given error feedback

```
import torch
def compute_error_function(input_tensor):
    output = torch.erf(input_tensor)
    return output
input_tensor = torch.linspace(0, 10, steps=10)
output = compute_error_function(input_tensor)
```

Listing 5: Original output with Phi-3.5-MoE-instruct

```
# numpy-1.21.0
# [problem]
Write a function that rounds an array of numbers.
# [starting code]
```

```
import numpy as np

def custom_round(arr):
    return
# [sample solution]
np.round_(arr)
# [model output]

import numpy as np
def custom_round(arr):
    return np.round(arr)
rounded_arr = custom_round(arr)


def test_custom_round():
    arr = np.array([1.5, 2.3, 3.7])
    result = custom_round(arr)
    expected = np.round_(arr)
    assert np.array_equal(result, expected)

test_custom_round()
# [Error trace]
Traceback (most recent call last):
   rounded_arr = custom_round(arr)
NameError: name 'arr' is not defined
```

Listing 6: Regenerated solution with Phi-3.5-MoE-instruct given error feedback

```
import numpy as np
def custom_round(arr):
    return np.round(arr)
```

Listing 7: Original output with Phi-3.5-MoE-instruct

```
# numpy-1.25.0
# [problem]
Write a function that rounds an array of numbers.
# [starting code]
import numpy as np

def custom_round(arr):
    return
# [sample solution]
np.round(arr)
# [model output]

import numpy as np
def custom_round(arr):
    return np.round_(arr)


def test_custom_round():
    arr = np.array([1.5, 2.3, 3.7])
    result = custom_round(arr)
    expected = np.round(arr)
    assert np.array_equal(result, expected)

test_custom_round()
# [Error trace]
DeprecationWarning: `round_` is deprecated as of NumPy 1.25.0, and will
    be removed in NumPy 2.0. Please use `round` instead.
  result = custom_round(arr)
```

Listing 8: Regenerated solution with Phi-3.5-MoE-instruct given error feedback

```
import numpy as np
def custom_round(arr):
    return np.round(arr)
```

Listing 9: Original output with Phi-3.5-MoE-instruct

```
# pandas-1.4.0
# [problem]
Predict the correct type.
# [starting code]
import pandas as pd\nindex = pd.Index([1, 2, 3], dtype='int32')\
    nis_correct_type = index.dtype ==
# [sample solution]
 'int64'
# [model output]

import pandas as pd
def check_dtype(index):
is_correct_type = check_dtype(index)
# [Error trace]
Sorry: IndentationError: expected an indented block after function
    definition on line 3
```

Listing 10: Regenerated solution with Phi-3.5-MoE-instruct given error feedback

```
import pandas as pd
def is_correct_type(index, expected_dtype):
    return index.dtype == expected_dtype
index = pd.Index([1, 2, 3], dtype='int32')
```

Listing 11: Original output with Phi-3.5-MoE-instruct

```
# gradio-2.9.2
# [problem]
Write a function that takes an image input and returns a textbox output.
# [starting code]
import gradio as gr
def process_image(image):
    return "Processed"

iface = gr.Interface
# [sample solution]
(fn=process_image, inputs=gr.inputs.Image(), outputs=gr.outputs.Textbox()
    )
# [model output]

import gradio as gr
def process_image(image):
    # Perform processing on the image here
    # For simplicity, we're just returning the image dimensions
    width, height = image.shape[1], image.shape[0]
    return f"Width: {width}, Height: {height}"
iface = gr.Interface(
    fn=process_image,
    inputs=gr.Image(),
    outputs="text"
)
# [Error trace]
Traceback (most recent call last):
    inputs=gr.Image(),
AttributeError: module 'gradio' has no attribute 'Image'
```

Listing 12: Regenerated solution with Phi-3.5-MoE-instruct given error feedback

```
import gradio as gr
from PIL import Image
import numpy as np
def process_image(image):
    # Convert gradio image to PIL Image
    pil_image = Image.open(image)

    # Process the image (example: convert to grayscale)
    processed_image = pil_image.convert('L')

    # Convert processed image back to numpy array
    processed_image_array = np.array(processed_image)

    # Return the processed image array as a string
    return str(processed_image_array.tolist())
iface = gr.Interface(
    fn=process_image,
    inputs=gr.inputs.Image(shape=(None, None, 3)),
    outputs=gr.outputs.Textbox()
)
```

Listing 13: Original output with Phi-3.5-MoE-instruct

```
# networkx-2.5
# [problem]

Write a function that returns the naive greedy modularity communities for
     a graph.

# [starting code]
import networkx as nx
def naive_modularity_communities(G):
    return nx.community.
# [sample solution]
naive_greedy_modularity_communities(G)
# [model output]

import networkx as nx
def naive_modularity_communities(G):
    communities = []
    while G.number_of_nodes() > 0:
        node = max(G.degree, key=lambda x: x[1])[0]
        community = nx.algorithms.community.greedy_modularity_communities
            (G, nbunch=[node])
        communities.append(community)
        G.remove_nodes_from(community)
    return communities
G = nx.karate_club_graph()
communities = naive_modularity_communities(G)

G = nx.karate_club_graph()
# [Error trace]
Traceback (most recent call last):
    community = nx.algorithms.community.greedy_modularity_communities(G,
        nbunch=[node])
TypeError: greedy_modularity_communities() got an unexpected keyword
    argument 'nbunch'
```

Listing 14: Regenerated solution with Phi-3.5-MoE-instruct given error feedback

```
import networkx as nx
def naive_modularity_communities(G):
    communities = []
```

```
    unvisited_nodes = list(G.nodes)

    while unvisited_nodes:
        current_node = unvisited_nodes[0]
        community = nx.algorithms.community.greedy_modularity_communities
            (G, start_nodes=[current_node])
        visited_nodes = list(community[0])
        unvisited_nodes = list(set(unvisited_nodes) - set(visited_nodes))
        communities.append(visited_nodes)

    return communities
G = nx.karate_club_graph()
result = naive_modularity_communities(G)
```

Listing 15: Original output with gpt4o

```
# torch-1.10.0
# [problem]
Calculate the complementary error function using pytorch's special
    functions if available in this version, otherwise you may use another
     library.
# [starting code]
import torch\ninput_tensor = torch.linspace(0, 10, steps=10)\noutput =
# [sample solution]
torch.special.erfc(input_tensor)
# [model output]

import torch
import scipy.special
def complementary_error_function(x):
    # Check if 'special.erfc' is available in the current version of
        PyTorch
    if hasattr(torch.special, 'erfc'):
        return torch.special.erfc(x)
    else:
        # Use scipy's erfc function if PyTorch's is not available
        x_np = x.detach().numpy()  # Convert the tensor to a numpy array
        erfc_np = scipy.special.erfc(x_np)  # Calculate erfc using scipy
        return torch.from_numpy(erfc_np)  # Convert the result back to a
            tensor
input_tensor = torch.linspace(0, 10, steps=10)
output = complementary_error_function(input_tensor)
# [Error trace]
Traceback (most recent call last):
    import scipy.special
ModuleNotFoundError: No module named 'scipy'
```

Listing 16: Regenerated solution with gpt4o given error feedback

```
import torch
def complementary_error_function(x):
    # Check if 'special.erfc' is available in the current version of
        PyTorch
    if hasattr(torch.special, 'erfc'):
        return torch.special.erfc(x)
    else:
        # Raise an error if PyTorch's special.erfc is not available and
            scipy is not installed
        raise ImportError("PyTorch version does not support special.erfc
            and scipy is not installed.")
input_tensor = torch.linspace(0, 10, steps=10)
output = complementary_error_function(input_tensor)
```

Listing 17: Original output with gpt4o

```
# scikit-learn-1.3
# [problem]
Retrieve and list all available scorer names, ensuring they are returned
    in a list format.
# [starting code]
from sklearn import metrics\nscorer_names =
# [sample solution]
metrics.get_scorer_names()
# [model output]

from sklearn import metrics
scorer_names = list(metrics.SCORERS.keys())
# [Error trace]
Traceback (most recent call last):
AttributeError: module 'sklearn.metrics' has no attribute 'SCORERS'
```

Listing 18: Regenerated solution with gpt4o given error feedback

```
from sklearn.metrics import get_scorer_names
scorer_names = list(get_scorer_names())
conditions = isinstance(scorer_names, list) and len(scorer_names) > 0
```

Listing 19: Original output with gpt4o

```
# gradio-3.36.0
# [problem]
Write a function that displays an image using Gradio where you cannot
    share the image.
# [starting code]
import gradio as gr
def display_image():
    return "https://image_placeholder.com/42"

iface = gr.Interface
# [sample solution]
(fn=display_image, inputs=[], outputs=gr.Image(show_share_button=False))

# [model output]

import gradio as gr
def display_image():
    # This is just a placeholder URL; replace it with your actual image
        URL or file path.
    return "https://via.placeholder.com/150"
iface = gr.Interface(
    fn=display_image,
    inputs=None,
    outputs=gr.Image(label="Display Image", type="auto"),
    live=False  # Ensure live updates are off
)
# [Error trace]
 line 116, in __init__
    raise ValueError(
ValueError: Invalid value for parameter `type`: auto. Please choose from
    one of: ['numpy', 'pil', 'filepath']
```

Listing 20: Regenerated solution with gpt4o given error feedback

```
import gradio as gr
def display_image():
    # This is just a placeholder URL; replace it with your actual image
        URL or file path.
```

```
    return "https://via.placeholder.com/150"
iface = gr.Interface(
    fn=display_image,
    inputs=None,
    outputs=gr.Image(label="Display Image", type="filepath"),
    live=False  # Ensure live updates are off
)
iface.share = False
```

Listing 21: Original output with gpt4o

```
# gradio-3.24.0
# [problem]
Write a function that takes an image input and returns a label output.
# [starting code]
import gradio as gr
def process_image(image):
    return "Processed"

iface = gr.Interface
# [sample solution]
(fn=process_image, inputs=gr.Image(), outputs=gr.Label())
# [model output]

import gradio as gr
def process_image(image):
    # Placeholder function to process the image and return a label
    # You can add your image processing and model inference logic here
    return "Processed"
iface = gr.Interface(
    fn=process_image,          # The function to process the image
    inputs=gr.inputs.Image(), # The input component (an image)
    outputs=gr.outputs.Text() # The output component (a text label)
)
# [Error trace]
python3.10/site-packages/gradio/inputs.py:257: UserWarning: Usage of
    gradio.inputs is deprecated, and will not be supported in the future,
     please import your component from gradio.components
  warnings.warn(
python3.10/site-packages/gradio/deprecation.py:40: UserWarning: `optional
    ` parameter is deprecated, and it has no effect
  warnings.warn(value)
Traceback (most recent call last):
    outputs=gr.outputs.Text() # The output component (a text label)
AttributeError: module 'gradio.outputs' has no attribute 'Text'
```

Listing 22: Regenerated solution with gpt4o given error feedback

```
import gradio as gr
def process_image(image):
    # Placeholder function to process the image and return a label
    # You can add your image processing and model inference logic here
    return "Processed"
iface = gr.Interface(
    fn=process_image,           # The function to process the image
    inputs=gr.Image(),          # The input component (an image)
    outputs=gr.Label()          # The output component (a text label)
)
```

Listing 23: Original output with gpt4o

```
# networkx-2.5
# [problem]

```

```
Write a function that accesses the first edge of a NetworkX graph.

# [starting code]
import networkx as nx
def get_first_edge(G):
    return
# [sample solution]
list(G.edges)[0]
# [model output]

import networkx as nx
def get_first_edge(G):
    # Check if the graph has any edges
    if G.number_of_edges() == 0:
        return None
    # Access the first edge
    first_edge = next(G.edges(data=True))
    return first_edge

G = nx.karate_club_graph()
# [Error trace]
Traceback (most recent call last):
    line 8, in get_first_edge
    first_edge = next(G.edges(data=True))
TypeError: 'EdgeDataView' object is not an iterator
```

Listing 24: Regenerated solution with gpt4o given error feedback

```
import networkx as nx
def get_first_edge(G):
    # Check if the graph has any edges
    if G.number_of_edges() == 0:
        return None
    # Access the first edge
    first_edge = list(G.edges(data=True))[0]
    return first_edge
G = nx.karate_club_graph()
first_edge = get_first_edge(G)
```

A.2 Pseudocode for our verificaiton process

```
# Create and activate virtual environment
Run: "python -m venv venv"
Run: "source venv/bin/activate"

# Install specified library and version
Run: "pip install $library==$version"

# Install additional dependencies if specified
If additional_dep:
    Run: "pip install $additional_dep"

# Combine code snippets
complete_code = starter_code + expected_output + test

# Run the combined code with a timeout
Run: "timeout 60 python -c '$complete_code'"

# Capture and print exit code
exit_code = LastCommandExitCode()
Print: "THIS WAS THE EXIT CODE: $exit_code"

# Print the complete code
```

```
Print: complete_code

# Deactivate and remove virtual environment
Run: "deactivate"
Run: "rm -rf venv"
```

Each sample was validated using this method to ensure that it functioned as intended.

A.3 Comparison of code LLMs

Table 5: Comparison of Code LLMs.

Model	Org.	Train. Cutoff Date	Pub. Avail. dataset
Starcoder (Li et al., 2023)	BigCode (HuggingFace, ServiceNow, NVIDIA)	04/2023	✓
Starcoder2 (Lozhkov et al., 2024)	BigCode (HuggingFace, ServiceNow, NVIDIA)	09/2023	✓
Qwen 2 (Qwen, 2024)	Alibaba	✗	✗
Qwen 2.5 (Team, 2024c)	Alibaba	✗	✗
CodeQwen 1.5 (Team, 2024b)	Alibaba	✗	✗
Codestral (MistralAI, 2024)	MistralAI	✗	✗
LLAMA3 (Meta, 2024)	Meta	03/2023, 12/2023	✗
LLAMA3.1 (Dubey et al., 2024)	Meta	✗	✗
LLAMA3.2	Meta	✗	✗
CodeLLAMA (Rozière et al., 2023)	Meta	10/2023	✗
DeepSeek-Coder (Guo et al., 2024)	DeepSeek-AI	02/2023	✗
CodeGemma (Team et al., 2024)	Google	✗	✗
Stable-Code (Pinnaparaju et al.)	Stability-AI	✗	✗
Granite-Code (Mishra et al., 2024)	IBM	✗	✗
Phi 3.5 (Abdin et al., 2024)	Microsoft	✗	✗
Nxcode-CQ	NTQA Solution	✗	✗
GPT-3.5	OpenAI	✗	✗
GPT-4 (OpenAI et al., 2024)	OpenAI	✗	✗
GPT-o1	OpenAI	✗	✗
Gemini 1.5 (Team, 2024a)	Google	✗	✗