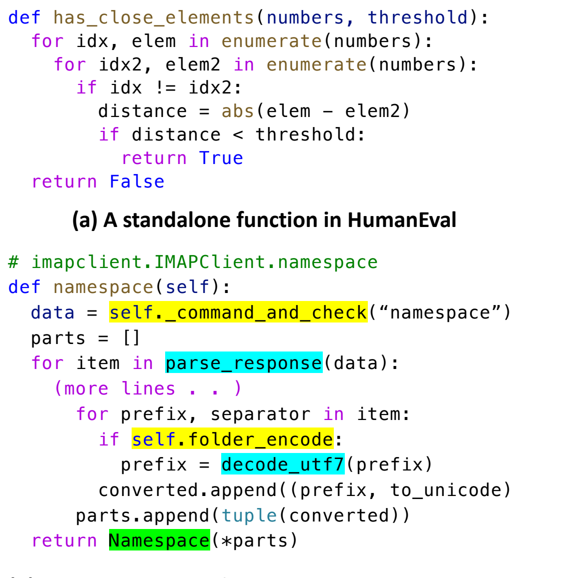
DevEval: A Code Generation Benchmark for Practical Software Projects

Anonymous ACL submission

Abstract

How to evaluate Large Language Models (LLMs) in code generation is an open question. There is currently no benchmark for practical software projects. In this paper, we propose a new benchmark named DevEval, which aligns with Developers' experiences in practical projects. DevEval is collected through a rigorous pipeline, containing 2,690 samples from 119 practical projects and covering 10 domains. Compared to previous benchmarks, DevEval aligns to practical projects in multiple dimensions, *e.g.,* real program distributions, sufficient dependencies, and enough-scale project contexts. We assess 12 popular LLMs on DevEval (*e.g.,* gpt-4, gpt-3.5-turbo, Claude 2, GLM-4, CodeLLaMa, StarCoder, and Mistral) and reveal their actual abilities in code generation. **For instance, the highest Pass@1 of gpt-3.5-turbo only is 42.97% in our experiments.** We also discuss the challenges of code generation in practical projects. We open-source DevEval[1] and hope it can facilitate the development of code generation in practical projects.



```
def has_close_elements(numbers, threshold):
  for idx, elem in enumerate(numbers):
    for idx2, elem2 in enumerate(numbers):
      if idx != idx2:
        distance = abs(elem - elem2)
        if distance < threshold:
          return True
  return False
```

(a) A standalone function in HumanEval

```
# imapclient.IMAPClient.namespace
def namespace(self):
  data = self._command_and_check("namespace")
  parts = []
  for item in parse_response(data):
    (more lines . . )
      for prefix, separator in item:
        if self.folder_encode:
          prefix = decode_utf7(prefix)
        converted.append((prefix, to_unicode)
      parts.append(tuple(converted))
  return Namespace(*parts)
```

(b) A non-standalone function in a real-world project

Figure 1: Examples of standalone and non-standalone functions Dependencies are highlighted, *i.e.,* yellow: intra-class dependencies, green: intra-file dependencies, and blue: cross-file dependencies.

1 Introduction

Code generation with Large Language Models (LLMs) has attracted lots of researchers' attention (Li et al., 2023c,a), and some commercial products have been produced, such as GitHub Copilot (GitHub, 2023). How to evaluate LLMs on code generation is an open question. Many code generation benchmarks have been proposed, but there are gaps between them and practical software projects. The gaps result in the development of code generation technologies being inconsistent with the experience of developers. To clarify the gaps, we analyzed over 1 million functions from 500 practical projects and summarized the gaps as follows.

Gap 1: Existing benchmarks differ from real program distributions, especially the proportion of non-standalone programs. As shown in Figure 1, a standalone function solely uses built-in or public libraries, while a non-standalone one contains project-specific *dependencies*. A project-specific dependency refers to an invocation of elements defined in projects, like `parse_response` in Figure 1. Out of 500 practical projects, 73.8% of functions are non-standalone, and 26.2% are standalone. However, existing benchmarks focus on standalone programs, with few or no non-standalone programs. For example, a popular benchmark - HumanEval (Chen et al., 2021) does not contain non-standalone functions, and the latest benchmark, CoderEval (Yu et al., 2023) only includes 146 non-standalone programs.

Gap 2: Dependencies within existing benchmarks are insufficient compared to practical projects. On average, each non-standalone function in 500 practical projects contains 3.22 depen-

[1] https://figshare.com/articles/dataset/DevEval/25231922

Table 1: The comparison between popular code generation benchmarks and DevEval. SA: Standalone. L(Re): the average lengths (tokens) of requirements.

Benchmark	Program Distribution				Dependency				Project Contexts		#L(Re)
	#Project	#Total	SA (%)	Non-SA (%)	#Type	#Total	#Per Sample	Path	#File	#Line	
CoNaLA (Yin et al., 2018)	–	500	100%	0%	0	0	0	✗	0	0	13.1
HumanEval (Chen et al., 2021)	–	164	100%	0%	0	0	0	✗	0	0	58.8
MBPP (Austin et al., 2021)	–	974	100%	0%	0	0	0	✗	0	0	16.1
PandasEval (Zan et al., 2022)	–	101	100%	0%	0	0	0	✗	0	0	29.7
NumpyEval (Zan et al., 2022)	–	101	100%	0%	0	0	0	✗	0	0	30.5
AixBench (Li et al., 2023b)	–	175	100%	0%	0	0	0	✗	0	0	34.5
ClassEval (Du et al., 2023)	–	100	100%	0%	0	0	0	✗	0	0	–
Concode (Iyer et al., 2018)	–	2,000	19.9%	80.1%	1	2,455	1.23	✗	0	0	16.8
CoderEval (Yu et al., 2023)	43	230	36%	64%	3	256	1.73	✗	71	14,572	41.5
DevEval	**119**	**2,690**	**26.2%**	**73.8%**	**3**	**5,849**	**2.95**	✓	**243**	**45,941**	**91.5**
500 Practical Projects	500	1M	26.2%	73.8%	3	3M	3.22	–	238	46,313	–

dencies, including intra-class, intra-file, and cross-file dependencies. However, existing benchmarks have insufficient dependencies. For instance, Concode (Iyer et al., 2018) and ClassEval (Du et al., 2023) solely contain intra-class dependencies, and CoderEval averages only 1.73 dependencies per non-standalone function.

Gap 3: The project contexts in existing benchmarks are small-scale compared to practical projects. In practical projects, developers rely on the project contexts (*e.g.,* relevant programs in projects) to continually write new programs. The contexts contain lots of project-specific knowledge (*e.g.,* private objects). The project contexts in 500 practical projects average 239 code files and around 43k lines. However, the project contexts in existing benchmarks often are small-scale. For instance, CoderEval's project contexts average only 71 code files and approximately 1.4k lines.

To address the above gaps, we propose a new code generation benchmark named DevEval, which aligns with Developers' experiences in practical projects. DevEval comprises 2,690 testing samples from 119 practical projects, collected by a rigorous pipeline and annotated by 13 developers. Each sample consists of a manually crafted natural language requirement, project contexts, reference code, reference dependencies, and multiple test cases. Table 1 compares DevEval with existing benchmarks, highlighting its three advances.

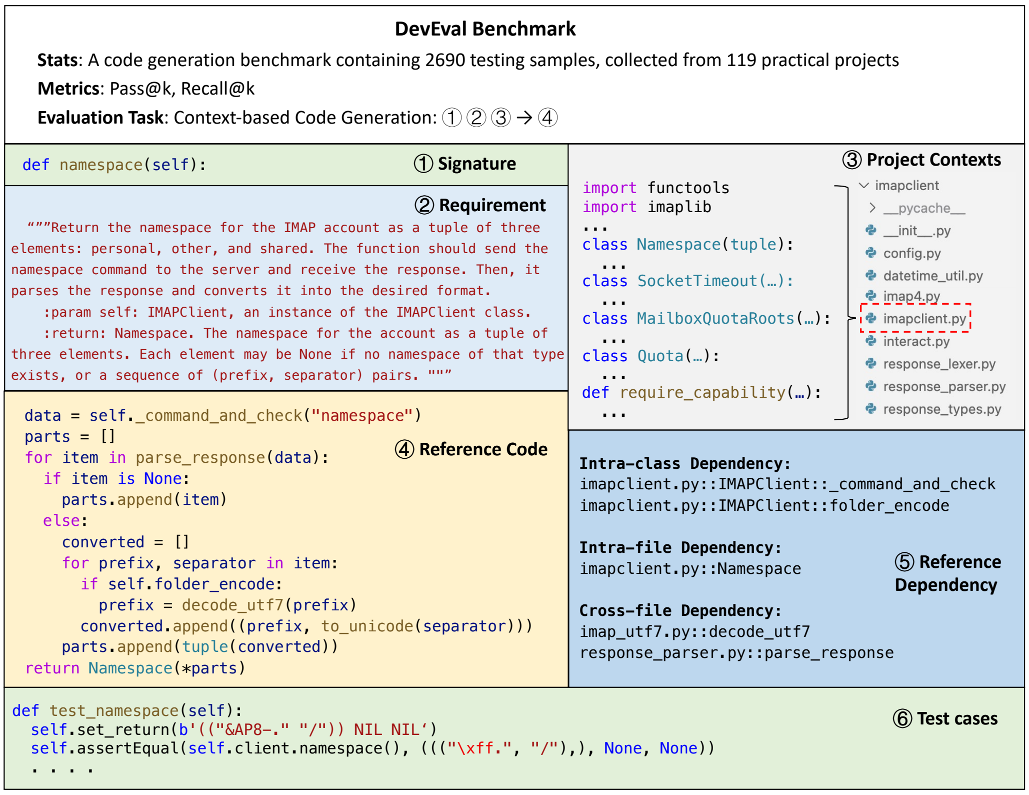
Advance 1: Real program distribution. DevEval features a real program distribution, comprising 1,984 (73.8%) non-standalone and 706 (26.2%) standalone programs, aligning the distribution observed in 500 practical projects.

Advance 2: Sufficient Dependencies. DevEval includes 5,849 dependencies, around 23 times more than CoderEval. Non-standalone programs in DevEval average 2.95 dependencies, close to the average value (*i.e.,* 3.22) of 500 practical projects. Besides, previous work (*i.e.,* CoderEval) only provides dependencies' names (*e.g.,* `close`). Because many functions with the same name in practice, it is hard to identify whether the dependencies generated by LLMs are correct by relying on names. DevEval labels dependencies with paths (*e.g.,* `A.py::ClassB::close`), addressing ambiguity and biases.

Advance 3: Enough-scale project contexts. DevEval contains enough-scale project contexts, averaging 243 code files with 45,941 lines per sample. Compared to previous benchmarks (*e.g.,* CoderEval: 71 files with 14k lines), DevEval's project contexts are closer to the average in 500 practical projects (238 files with 46,313 lines).

DevEval also has advantages in requirements, diversity, and evaluation metrics. ❶ *Requirements.* Original comments of programs often are short and vague and are not suitable for code generation. Thus, we engaged 13 developers to manually write detailed and accurate requirements for all programs. Our requirements average 91.5 tokens, approximately 2.2 times that of CoderEval. ❷ *Diversity.* DevEval contains 11 times more samples than CoderEval (2,690 vs. 230), collected from 119 projects across 10 domains (*e.g.,* Text Processing, Internet, Database). It covers diverse programming topics to comprehensively assess LLMs. ❸ *Evaluation Metrics.* DevEval leverages Pass@k (functional correctness) and Recall@k (recall of reference dependencies) to comprehensively assess generated programs.

We evaluate 12 popular LLMs upon DevEval (*i.e.,* gpt-4-turbo-1106 (OpenAI, 2023b), gpt-3.5-turbo-{0613, 1106} (OpenAI, 2023a), CodeLLaMa-{70B, 34B, 13B, 7B} (Rozière et al.,



Stats: A code generation benchmark containing 2690 testing samples, collected from 119 practical projects

Metrics: Pass@k, Recall@k

Evaluation Task: Context-based Code Generation: ①②③ → ④

```
def namespace(self):
```

```
"""Return the namespace for the IMAP account as a tuple of three
elements: personal, other, and shared. The function should send the
namespace command to the server and receive the response. Then, it
parses the response and converts it into the desired format.
    :param self: IMAPClient, an instance of the IMAPClient class.
    :return: Namespace. The namespace for the account as a tuple of
three elements. Each element may be None if no namespace of that type
exists, or a sequence of (prefix, separator) pairs. """
```

```
data = self._command_and_check("namespace")
parts = []
for item in parse_response(data):
  if item is None:
    parts.append(item)
  else:
    converted = []
    for prefix, separator in item:
      if self.folder_encode:
        prefix = decode_utf7(prefix)
      converted.append((prefix, to_unicode(separator)))
    parts.append(tuple(converted))
return Namespace(*parts)
```

```
import functools
import imaplib
...
class Namespace(tuple):
  ...
class SocketTimeout(…):
  ...
class MailboxQuotaRoots(…):
  ...
class Quota(…):
  ...
def require_capability(…):
  ...
```

```
Intra-class Dependency:
imapclient.py::IMAPClient::_command_and_check
imapclient.py::IMAPClient::folder_encode

Intra-file Dependency:
imapclient.py::Namespace

Cross-file Dependency:
imap_utf7.py::decode_utf7
response_parser.py::parse_response
```

```
def test_namespace(self):
  self.set_return(b'(("&AP8-." "/")) NIL NIL')
  self.assertEqual(self.client.namespace(), ((("\xff.", "/"),), None, None))
  . . . .
```

Figure 2: An overview of DevEval. It contains 2,690 samples, and each sample consists of six components.

2023), StarCoder (Li et al., 2023d), and Mistral-{7B, MoE} (Jiang et al., 2023; Mistral.AI, 2023)) and obtain surprising findings. First, LLMs exhibit low performance on DevEval, especially compared to their performance on previous benchmarks. **For example, gpt-3.5-turbo-1106 achieves a Pass@1 score of 73% on HumanEval, while its highest Pass@1 on DevEval is only 42.97%.** Our results reveal the actual abilities of LLMs in code generation. Second, LLMs benefit from project contexts but struggle with comprehensively understanding long contexts. Even if we input oracle contexts, the Pass@1 of LLMs is still below 50%.

In summary, our contributions are as follows:

- We identify gaps (*e.g.,* unreal program distributions, insufficient dependencies, and small-scale project contexts) between existing benchmarks and practical projects. We propose a new benchmark named DevEval with 2,690 samples, addressing these gaps.

- DevEval closely aligns practical projects, including real program distributions, sufficient dependencies, and enough-scale project contexts.

- We evaluate 12 popular LLMs on DevEval, analyzing their strengths and shortcomings in code generation for practical projects.

We hope DevEval can align with actual experiences of developers during the practical development process. By DevEval, practitioners can pick up superior LLMs and facilitate the application of code generation techniques in practical projects.

2 Benchmark - DevEval

2.1 Overview

DevEval contains 2,690 samples derived from 119 real-world open-source projects. As shown in Figure 2, each sample consists of six components. ❶ **Function Signature:** The signature of the code to be generated. ❷ **Requirement:** An English description detailing the functionality of the code to be generated. ❸ **Project Contexts:** Existing programs (*e.g.,* hundreds of code files) in the current project. ❹ **Reference Code:** A reference implementation of the code to be generated, crafted by developers. This code invokes dependencies defined in project contexts. ❺ **Reference Dependency:** The dependencies invoked in the reference code, include intra-class, intra-file, and cross-file dependencies. ❻ **Test Cases:** Test cases are used to check the functional correctness of the code.

Table 2: The distribution of dependency types. The values in parentheses are the corresponding percentages in all dependencies.

Dependency Type	HumanEval	Concode	CoderEval	DevEval	500 Projects
Intra-class	0	2,455 (100%)	117 (46%)	**2,383 (41%)**	939k (42%)
Intra-file	0	0	90 (35%)	**1,833 (31%)**	597k (29%)
Cross-file	0	0	49 (19%)	**1,633 (28%)**	611k (30%)

2.2 Benchmark Characteristics

Compared to existing benchmarks (*e.g.,* CoderEval (Yu et al., 2023)), DevEval aligns practical projects due to three key advances.

❶ **Real program distributions.** As shown in Table 1, it contains 1,984 (73.8%) non-standalone programs and 706 (26.2%) standalone programs, aligning the observed ratio in 500 practical projects.

❷ **Sufficient dependencies.** DevEval covers three dependency types and contain sufficient dependencies. As shown in Table 1, each sample in DevEval averages 2.95 dependencies, surpassing the averages in previous benchmarks (*e.g.,* CoderEval: 1.75) and closely approaching that of 500 practical projects (*i.e.,* 3.22).

Table 2 shows the distribution of dependency types, *i.e.,* intra-class, intra-file, and cross-file dependencies. DevEval outperforms previous benchmarks in all types, showing a more real distribution that is close to the distribution in 500 practical projects. For instance, cross-file dependencies constitute 28% in DevEval compared to the meager 19% in CoderEval.

❸ **Enough-scale project contexts.** As shown in Table 1, previous benchmarks' project contexts are notably small-scale, *e.g.,* CoderEval: 14k lines versus practical projects: 46k lines. In contrast, DevEval introduces more large-scale project contexts, averaging 45k lines.

Moreover, DevEval has advantages in other aspects compared to existing benchmarks, such as requirements and test cases.

Requirements. We engaged 13 developers to manually write requirements, costing approximately 674 person-hours. As depicted in Figure 2, each requirement encapsulates the code's purpose and input-output parameters. The average length of requirements in DevEval (91.5 tokens) more than doubles that of CoderEval (41.5 tokens).

Test cases. Each sample in DevEval is equipped with 7.95 test cases on average. These test cases are rigorously validated through code reviews and are capable of achieving high line-level and branch-level coverages. In comparison, the average number of test cases in HumanEval and MBPP are 7.7 and 3.0, respectively. Therefore, DevEval provides a reliable evaluation environment.

2.3 Task Definition

We define the **Context-based Code Generation** task upon DevEval. It aims to generate code based on a function signature, a requirement, and the project contexts. We also design a baseline setting, which generates code based on the signature and requirement. The baseline is used to evaluate LLMs' coding ability without project contexts.

2.4 Evaluation Metrics

Pass@k (Functional Correctness). Following previous studies (Chen et al., 2021; Austin et al., 2021; Yu et al., 2023), we assess the functional correctness of programs by executing test cases and compute the unbiased Pass@k. Specifically, we generate $n \geq k$ programs per requirement, count the number of correct programs $c \leq n$ that pass test cases, and calculate the Pass@k:

$$\text{Pass@}k := \mathop{\mathbb{E}}_{\text{Requirements}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (1)$$

Recall@k (Recall of Reference Dependency). Besides the functional correctness, we expect LLMs to invoke relevant dependencies defined in contexts. Hence, we propose Recall@k, which gauges the recall of reference dependencies in generated programs.

Specifically, LLMs generate k programs per requirement. For the i-th program, we employ a parser[2] to extract its dependencies as $\mathbb{P}_i$. Subsequently, we compare $\mathbb{P}_i$ with reference dependencies $\mathbb{R}$ and compute the Recall@k:

$$\text{Recall@}k := \mathop{\mathbb{E}}_{\text{Requirements}} \left[\max_{i \in [1,k]} \frac{|\mathbb{R} \cap \mathbb{P}_i|}{|\mathbb{R}|} \right] \quad (2)$$

where $|\cdot|$ means the number of elements of a set.

3 Benchmark Collection

As shown in Figure 3, the collection of DevEval consists of four steps.

❶ **Project Selection.** We crawl high-quality projects from an open-source community - PyPI

[2]We develop the parser based on an open-source static analysis tool - Pyan (Pyan, 2023).

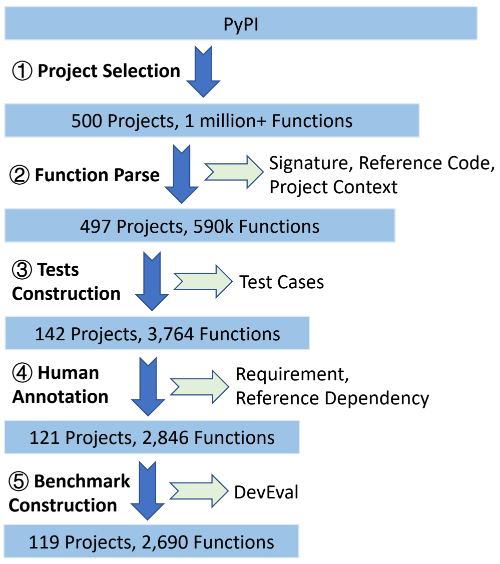

PyPI
① Project Selection
500 Projects, 1 million+ Functions
② Function Parse
Signature, Reference Code, Project Context
497 Projects, 590k Functions
③ Tests Construction
Test Cases
142 Projects, 3,764 Functions
④ Human Annotation
Requirement, Reference Dependency
121 Projects, 2,846 Functions
⑤ Benchmark Construction
DevEval
119 Projects, 2,690 Functions

Figure 3: The process of collecting DevEval.

(PyPI). To ensure a broad diversity, we identify the top 50 projects with open-source licenses in the top 10 popular programming topics. The 10 topics are shown in Appendix A. We download the latest released versions in November 2023 and obtain 500 practical projects (10 topics * 50 projects).

❷ **Function Parse.** We extract all functions within projects and parse their *signatures* and bodies. Unparseable or empty functions are excluded. The function bodies, crafted by developers and subjected to rigorous code reviews, are deemed as the *reference code*. Subsequently, we extract other programs within the current project as *project contexts*. Finally, we obtain 590,365 functions.

❸ **Tests Construction.** For each function, we extract test cases invoking it from its project. We use a popular testing framework - `Pytest`[3] to organize these test cases. We leverage a public framework - `setuptools`[4] to automatically build the running environments for each project. Functions without successful test cases are excluded. In summary, we retain 3,764 functions, each equipped with both successful test cases and running environments.

❹ **Human Annotation.** We engage 13 annotators to manually annotate requirements and reference dependencies for each function. All annotators obtain adequate payments given their countries of residence.

Through discussions with annotators, we establish two criteria for requirements. *Naturalness*–ensuring the requirement reads like a natural description from the perspective of a real-world developer. *Functionality*–demanding clear descriptions of the code's purposes and input-output parameters. Each requirement undergoes a dual-annotation process, with one annotator assigned to its initial drafting and another responsible for a meticulous double-check. Trivial functions (*e.g.,* shortcut functions) and functions violating the ethical code (*e.g.,* malware) are excluded. Subsequently, the same 13 annotators review the reference code and label dependencies within it. Finally, we retain 2,846 functions with high-quality requirements and labeled reference dependencies.

❺ **Benchmark Construction.** Among the retained functions, 862 are standalone, and 1,984 are non-standalone. We follow the program distribution in 500 practical projects to construct DevEval, ensuring the data size is maximized. Retaining all 1,984 (73.8%) non-standalone functions, we randomly sample 706 (26.2%) standalone functions, resulting in the DevEval with 2,690 samples.

4 Experiments

4.1 Studied LLMs

We evaluate 12 popular LLMs, including closed-source LLMs (*i.e.,* gpt-4-1106 (OpenAI, 2023b), gpt-3.5-turbo-{0613, 1106} (OpenAI, 2023a), Claude 2 (Anthropic, 2023), and GLM-4 (Zhipu.AI, 2024)) and open-source LLMs (*i.e.,* CodeLLaMa-{70B, 34B, 13B, 7B} (Rozière et al., 2023), StarCoder (Li et al., 2023d), and Mistral-{7B, MoE} (Jiang et al., 2023; Mistral.AI, 2023)). We use official interfaces or implementations to reproduce these LLMs. The details of LLMs can be found in Appendix B.

4.2 Experimental Setup

The prompt template in our experiments is shown in Appendix C. We use Pass@k and Recall@k (see Section 2.4) to assess generated programs. In this paper, $k \in [1, 3, 5, 10]$. When $k = 1$, we use the greedy search and generate a single program per requirement. When $k > 1$, we use the nucleus sampling with a temperature 0.4 and sample 20 programs per requirement. We set the top-p to 0.95 and the max generation length to 500.

4.3 Main Results

Baseline. Table 3 shows Pass@k and Recall@k of LLMs in the baseline setting (*i.e.,* without the project contexts). As some closed-source LLMs

[3]https://docs.pytest.org/en/8.0.x/
[4]https://github.com/pypa/setuptools

Table 3: Pass@k and Recall@k on the baseline setting *i.e.,* generating code based on the function signature and requirement.

LLMs	Size	Context Window	Pass@1	Pass@3	Pass@5	Pass@10	Recall@1	Recall@3	Recall@5	Recall@10
gpt-4-1106	N/A	128,000	**22.31**	–	–	–	18.64	–	–	–
gpt-3.5-turbo-0613	N/A	4,096	18.48	21.38	22.50	23.93	12.51	14.25	14.75	15.85
gpt-3.5-turbo-1106	N/A	16,385	17.88	20.39	21.55	22.99	11.72	13.28	14.01	14.9
Claude 2	N/A	100,000	17.73	–	–	–	15.78	–	–	–
GLM-4	N/A	128,000	17.14	–	–	–	13.43	–	–	–
CodeLLaMa-Instruct	70B	16,384	21.60	**26.44**	**28.87**	**31.83**	**19.31**	**22.31**	**23.89**	**25.87**
CodeLLaMa-Python	13B	16,384	17.88	22.86	25.32	28.39	16.80	20.43	22.02	24.27
CodeLLaMa-Python	7B	16,384	17.21	22.20	24.61	27.67	14.77	19.00	20.87	23.32
StarCoder	15.5B	8,192	17.10	21.89	24.33	27.26	17.73	21.36	23.57	25.75
CodeLLaMa-Instruct	34B	16,384	16.25	19.34	20.69	22.77	15.35	17.23	17.82	18.79
Mistral-Instruct-MoE	8*7B	32,768	13.23	17.06	18.75	20.76	10.44	13.66	15.26	16.67
Mistral-Instruct	7B	32,768	10.89	13.68	15.35	17.51	11.97	13.49	14.92	16.08

Table 4: Pass@k and Recall@k on context-based code generation, *i.e.,* generating code based on a signature, a requirement, and project contexts.

Settings	gpt-3.5-turbo-1106		CodeLLaMa-Instruct-34B		Mistral-Instruct-MoE	
	Pass@1	Recall@1	Pass@1	Recall@1	Pass@1	Recall@1
W/o Contexts	17.88	11.72	**21.60**	**15.35**	13.23	10.44
Local File	38.88	40.32	34.61	38.43	21.77	26.16
Local File + Sibling Files	41.26	45.91	39.70	46.36	23.56	28.14
Local File + Similar Files	42.34	46.29	39.78	47.63	24.65	30.95
Local File + Imported Files	**42.97**	47.48	40.15	**49.35**	28.51	30.95
Local File + Oracle	44.64	51.13	44.61	53.54	29.26	37.92

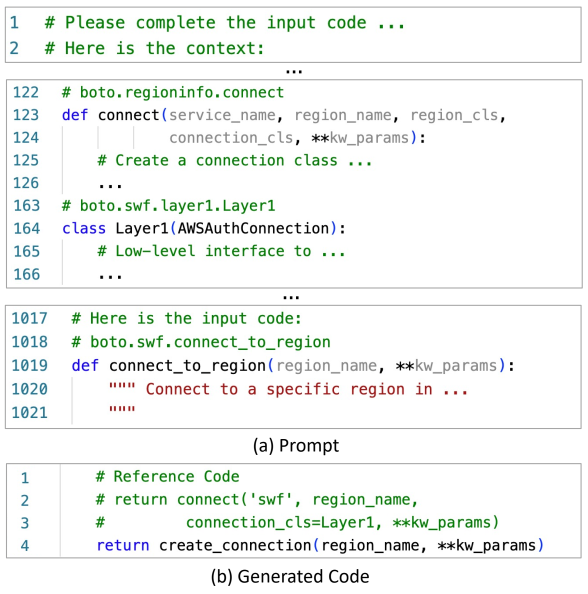
are expensive, we report the Pass@1 with greedy search. We can see that gpt-4 and CodeLLaMa-Instruct-70B achieve the highest Pass@1 and Recall@1 among all LLMs, respectively. However, all LLMs exhibit relatively low Pass@k and Recall@k values, compared to their performance on previous benchmarks. For instance, gpt-4 achieves a Pass@1 score of 88.4 on HumanEval, whereas it scores 22.31 on Pass@1 in this setting. The decreases validate our motivation that existing benchmarks can not comprehensively assess the capabilities of LLMs in practical projects. Furthermore, the results emphasize the importance of project contexts.

Interestingly, LLMs can successfully generate several dependencies without project contexts. A manual inspection of these dependencies reveals that they are mainly simple dependencies that can be reasoned from the requirements, *e.g.,* initialization functions of returned objects. It is hard for LLMs to generate more intricate dependencies without project contexts.

Context-based Code Generation. We further take the project contexts into considerations. Project contexts are typically very long, surpassing context windows of existing LLMs. Inspired by related work (Shrivastava et al., 2023), we extract parts of contexts as inputs. ❶ *Local file*: The code file where the reference code is in. We only take programs above the reference code. We consider the local file as a fundamental context and progressively add other contexts. ❷ *Sibling files*: Files within the same sub-folder as the local file. ❸ *Imported files*: Files imported by the local file. ❹ *Similar files*: Files with names similar to the local file. We split the names based on underscore or camelcase formatting and then match the tokens of names. If one or more parts match, two files are considered to have similar names. ❺ *Oracle*: Code implementations corresponding to reference dependencies. It consists of many code snippets from different files.

We input different contexts to three LLMs (*i.e.,* gpt-3.5-turbo-1106, CodeLLaMa-Instruct-34B, and Mistral-Instruct-MoE), and the results are presented in Table 4. After introducing the contexts, Pass@k and Recall@k values increase significantly. gpt-3.5-turbo-1106 achieves the highest Pass@1 under different contexts, although it is worse than other LLMs without contexts. It shows that gpt-3.5-turbo-1106 has a stronger ability to understand contexts compared to other LLMs.

We further inspect a few successful cases and attribute the improvements to the synergy of contexts and our requirements. On the one hand, the contexts provide lots of project-specific knowledge. For example, the local file contains essen-



```
# Please complete the input code ...
# Here is the context:
```
...
```
# boto.regioninfo.connect
def connect(service_name, region_name, region_cls,
            connection_cls, **kw_params):
    # Create a connection class ...
    ...
# boto.swf.layer1.Layer1
class Layer1(AWSAuthConnection):
    # Low-level interface to ...
    ...
```
...
```
# Here is the input code:
# boto.swf.connect_to_region
def connect_to_region(region_name, **kw_params):
    """ Connect to a specific region in ...
    """
```
(a) Prompt

```
    # Reference Code
    # return connect('swf', region_name,
    #         connection_cls=Layer1, **kw_params)
    return create_connection(region_name, **kw_params)
```
(b) Generated Code

Figure 4: A failed case on DevEval-CGen.

Table 5: Lengths (#Tokens) of different contexts.

Context	Average length	Max length
Local File	2,468	51,716
Local File+Imported Files	14,913	771,644
Local File+Similar Files	17,589	2,038,908
Local File+Sibling Files	53,412	579,237
All Files	1,147,282	9,263,195

tial local environments (*e.g.,* current classes, imported libraries) and a majority of dependencies (*e.g.,* intra-class and intra-file: 72% in DevEval). Similar and sibling files typically contain programs pertinent to the requirements (*e.g.,* parent classes). Imported files offer many cross-file dependencies that are likely to be invoked. Recent work (Zhang et al., 2023) in code completion also proved the importance of contexts. On the other hand, our manually written requirements elaborate the purposes of code and the background knowledge of projects. Thus, our requirements help LLMs understand long contexts and locate relevant dependencies.

Although promising, LLMs' abilities in context-based code generation are not satisfying. Even if we input oracle contexts, the highest Pass@1 only is 44.64%. A manual inspection of failed cases reveals LLMs struggle with understanding contexts. Figure 4 illustrates a failed case. LLMs invoke a non-existent function - `create_connection`, even though a valid function `connect` is present in the contexts. We think this problem is caused by two reasons.

First, the contexts are too long. Table 5 shows

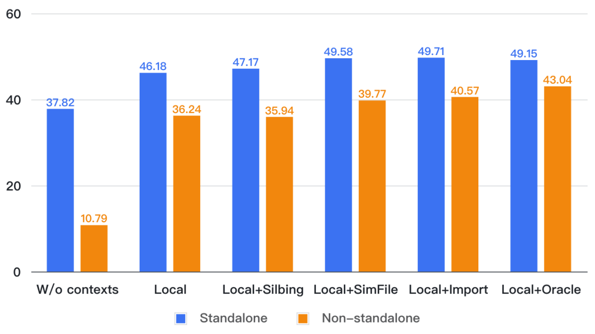


Figure 5: The Pass@1 of gpt-3.5-turbo-1106 on different program types.

the lengths of different contexts. The complete project contexts are lengthy, approximately 9 times the context window of the state-of-the-art LLM - gpt-4-1106. Even when partial contexts are considered, their lengths match or even exceed the context window of most current LLMs. Recent works (Liu et al., 2023) have found that LLMs often ignore relevant information in the middle of long contexts. This finding is consistent with our results. Second, the contexts are heterogeneous. In other words, the contexts are composed of discrete code snippets from different files rather than a continuous file. As shown in Figure 4, the programs within contexts come from multiple files, *e.g.,* `boto.regioninfo.py` and `boto.swf.layer1.py`. However, LLMs are typically trained to predict the next tokens based on the continuous contexts. The gap between training and inference objectives leads to a poor understanding of LLMs in contexts. Recent work (Shi et al., 2023) also found similar gaps in reading comprehension and question answering.

♡ **Takeaway:**

(1) Project contexts play a pivotal role in code generation; without them, LLMs exhibit subpar performance.

(2) Inputting relevant contexts benefits code generation. With limited context windows, local files and imported files can bring obvious improvements.

(3) Detailed and accurate requirements not only help LLMs know the purposes of programs but also understand long contexts.

(4) Existing LLMs struggle with understanding long and heterogeneous contexts. It causes LLMs to disregard the knowledge in contexts and even generate hallucinations (e.g., non-existent functions).

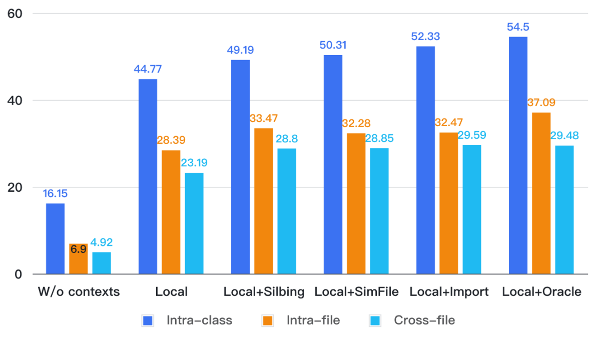


Figure 6: The Recall@1 of gpt-3.5-turbo-1106 on different dependency types.

4.4 Discussion

Results on different program types. Figure 5 shows Pass@1 of gpt-3.5-turbo-1106 on different program types (*i.e.,* standalone and non-standalone). The results reveal three observations. ❶ Project contexts are crucial to generating non-standalone functions. For example, adding local files improves the Pass@1 on non-standalone functions from 10.84 to 36.24. ❷ Project contexts also benefit standalone functions. This is attributed to the project-specific knowledge within contexts, aiding LLMs in understanding requirements. Thirdly, there exists considerable room for improving LLMs on both types of programs. How to effectively utilize contexts is a key problem.

Results on different dependency types. Figure 6 shows the Recall@1 of gpt-3.5-turbo-1106 on different dependency types (*i.e.,* intra-class, intra-file, and cross-file). The results yield two insights. ❶ Without project contexts, LLMs exhibit low Recall@1 values across three dependency types. However, LLMs demonstrate the ability to infer reason about easy dependencies based on requirements, *e.g.,* initialization functions of returned objects. ❷ With contexts, LLMs exhibit an improvement in generating dependencies. Nevertheless, LLMs have yet to grapple with generating dependencies, especially cross-file dependencies. As illustrated in Figure 4, LLMs often ignore available dependencies defined in contexts.

Data leakage. Theoretically, all open-source projects may be included in the training data for LLMs. Consequently, there is a risk of data leakage where several projects used to build DevEval appear in the training data. We think this risk has only a slight impact on DevEval due to four reasons. ❶ DevEval contains new data, *i.e.,* manually written requirements. These requirements are never included in the training data. ❷ Existing LLMs do not show overfitting tendencies to DevEval. Based on the release dates of 12 LLMs (see Section 4.1), we divide DevEval into two groups: unseen projects released later than LLMs and potentially seen projects released earlier than LLMs. The average difference of Pass@1 between the two groups is around 0.42. Compared to the average variations between LLMs (*e.g.,* 3.38 in Table 3 and 10.98 in Table 4), 0.42 is slight. ❸ DevEval is geared toward evaluating future LLMs. We release the links to our selected projects in Appendix A and encourage practitioners to omit these projects when collecting the training data for future LLMs.

The bias of Recall@k. As stated in Section 2.4, we develop a static analysis-based parser to automatically extract dependencies in generated programs. Because Python is a dynamically typed language, certain dependencies only are determined at run-time and may elude our parser. It may lead to lower Recall@k than actual values.

To gauge the above bias, we manually annotate dependencies within 100 programs generated by gpt-3.5-turbo-1106. Simultaneously, we employ our parser to extract dependencies in the same 100 programs. Based on the human-annotated and auto-extracted dependencies, we compute two Recall@1 values. The bias of two Recall@1 values is 0.23. Compared to the average variations between LLMs (3.47 in Table 3 and 11.08 in Table 4), 0.23 is slight. Consequently, we believe that Recall@k can effectively rank different LLMs, notwithstanding its slight bias.

5 Conclusion and Future Work

In this paper, we propose a new code generation benchmark named DevEval. Collected through a meticulous pipeline, DevEval aligns practical software projects in multiple dimensions, *e.g.,* real program distributions, sufficient dependencies, and enough-scale project contexts. We evaluate 12 popular LLMs on DevEval. The results reveal the strengths and weaknesses of LLMs in practical projects. Compared to previous benchmarks, DevEval offers a more challenging and practical evaluation scenario. We hope DevEval can facilitate the applications of LLMs in practical projects.

In the future, we will continue to update DevEval, *e.g.,* multilingual testing samples, more projects, and more test cases. Besides, we will explore how to improve the performance of LLMs in context-based code generation, *e.g.,* retrieval-augmented and tool-augmented generation.

6 Limitations

In this paper, we propose a new code generation benchmark - DevEval, which aligns with practical software projects. Based on DevEval, we evaluate 12 popular LLMs and analyze their strengths and shortcomings.

We believe that DevEval itself has four limitations. ❶ DevEval is a monolingual benchmark (*i.e.,* requirements in English and code in Python) and ignores other languages. In practice, LLMs require understanding requirements in different natural languages (*e.g.,* Chinese, Spanish) and generating programs in various programming languages (*e.g.,* Java, C). Thus, we plan to build a multilingual DevEval in future work. ❷ Test cases in DevEval may be insufficient. As stated in Section 2.2, each sample in DevEval is equipped with 7.95 test cases on average. These test cases are collected from practical projects and pass rigorous reviews. However, there is a risk where several samples lack corner test cases. This results in some incorrect programs being mistaken for correct ones. To alleviate this threat, we plan to leverage test case generation tools (Lukasczyk and Fraser, 2022; Dinella et al., 2022) to produce corner test cases and improve the coverage of test cases. ❸ As shown in Table 1 and 2, DevEval is closer to practical projects, but still has slight differences in the number and distribution of dependencies. In the future, we will introduce more dependencies into DevEval and alleviate the difference. ❹ As stated in Section 4.4, Recall@k values in DevEval may have slight biases, *i.e.,* they may be slightly less than actual values. Because Python is a dynamically typed language, certain dependencies can only be identified at runtime and may elude our parser. To gauge the bias introduced by our parser, we manually annotate dependencies within 100 programs generated by gpt-3.5-turbo-1106. Simultaneously, we employ the parser to extract dependencies in the same 100 programs. Based on the human-annotated and auto-extracted dependencies, we compute two Recall@1 values. The bias of two Recall@1 is 0.23. Compared to the average variations between LLMs (3.47 in Table 3 and 11.08 in Table 4), 0.23 is slight. Consequently, the Recall@k can effectively rank different LLMs, notwithstanding its slight bias.

Besides, our evaluation experiments can be further improved in three aspects. ❺ More LLMs. Due to the limited computing budgets, we mainly evaluate 12 mainstream LLMs (see Section B). It is worthy to evaluate some recently proposed LLMs (Guo et al., 2024; DeepMind, 2023). ❻ More investigations of project contexts. As shown in Table 4, we explore 5 straightforward approaches to utilizing project contexts. In the future, we will introduce more advanced techniques (*e.g.,* retrieval-augmented or tool-augmented generation) to investigate how to utilize contexts. ❼ Tuning hyper-parameters. It is known that LLMs are sensitive to sampling hyper-parameters and prompts. We ensure all LLMs are evaluated under the same experimental settings. Due to the limited computing budgets, we do not carefully tune hyper-parameters and prompts. Thus, there may be better hyper-parameters and prompts to further improve the performance of LLMs.

7 Ethics Consideration

DevEval is collected from open-source projects from the real world. We manually check all samples in DevEval. We ensure all samples do not contain private information or offensive content. We ensure all programs in DevEval are behaving normally and exclude any malicious programs.

References

Anthropic. 2023. Claude 2. https://www.anthropic.com/news/claude-2.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of artificial general intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712.

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

DeepMind. 2023. Gemini ultra. `https://blog.google/technology/ai/google-gemini-ai/#sundar-note`.

Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A neural method for test oracle generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2130–2141. ACM.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *CoRR*, abs/2308.01861.

Deep Ganguli, Liane Lovitt, Jackson Kernion, Amanda Askell, Yuntao Bai, Saurav Kadavath, Ben Mann, Ethan Perez, Nicholas Schiefer, Kamal Ndousse, Andy Jones, Sam Bowman, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Nelson Elhage, Sheer El Showk, Stanislav Fort, Zac Hatfield-Dodds, Tom Henighan, Danny Hernandez, Tristan Hume, Josh Jacobson, Scott Johnston, Shauna Kravec, Catherine Olsson, Sam Ringer, Eli Tran-Johnson, Dario Amodei, Tom Brown, Nicholas Joseph, Sam McCandlish, Chris Olah, Jared Kaplan, and Jack Clark. 2022. Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned. *CoRR*, abs/2209.07858.

GitHub. 2023. Github copilot. `https://github.com/features/copilot`.

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. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

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, Brussels, Belgium, October 31 - November 4, 2018*, pages 1643–1652. Association for Computational Linguistics.

Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. 1991. Adaptive mixtures of local experts. *Neural Comput.*, 3(1):79–87.

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. *CoRR*, abs/2310.06825.

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

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023a. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*.

Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023b. Skcoder: A sketch-based approach for automatic code generation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2124–2135. IEEE.

Jia Li, Yunfei Zhao, Li Yongmin, Ge Li, and Zhi Jin. 2023c. Acecoder: Utilizing existing code to enhance code generation. *arXiv preprint arXiv:2303.17780*.

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 V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-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. 2023d. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *CoRR*, abs/2307.03172.

Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 168–172. ACM/IEEE.

Mistral.AI. 2023. Mistral-moe. https://mistral.ai/news/mixtral-of-experts/.

OpenAI. 2023a. gpt-3.5-turbo. https://platform.openai.com/docs/models/gpt-3-5.

OpenAI. 2023b. GPT-4 technical report. *CoRR*, abs/2303.08774.

Pyan. 2023. Pyan. https://github.com/davidfraser/pyan.

PyPI. Pypi. https://pypi.org/.

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. *CoRR*, abs/2308.12950.

Weijia Shi, Sewon Min, Maria Lomeli, Chunting Zhou, Margaret Li, Xi Victoria Lin, Noah A. Smith, Luke Zettlemoyer, Scott Yih, and Mike Lewis. 2023. In-context pretraining: Language modeling beyond document boundaries. *CoRR*, abs/2310.10638.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 31693–31715. PMLR.

Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Dongdong Zhang, and Furu Wei. 2022. Deepnet: Scaling transformers to 1, 000 layers. *CoRR*, abs/2203.00555.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 476–486. ACM.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. Codereval: A benchmark of pragmatic code generation with generative pretrained models. *CoRR*, abs/2302.00288.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: continual pre-training on sketches for library-oriented code generation. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 2369–2375. ijcai.org.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Zhiyuan Liu, Peng Zhang, Yuxiao Dong, and Jie Tang. 2023. GLM-130B: an open bilingual pre-trained model. In *ICLR*. OpenReview.net.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Zhipu.AI. 2024. Glm-4. https://open.bigmodel.cn/pricing.