SWE-bench: Can Language Models Resolve Real-World GitHub Issues?

Abstract

Language models have outpaced our ability to evaluate them effectively, but for their future development it is essential to study the frontier of their capabilities. We consider real-world software engineering to be a rich, sustainable, and challenging testbed for evaluating the next generation of language models. We therefore introduce SWE-bench, an evaluation framework including 2,294 software engineering problems drawn from real GitHub issues and corresponding pull requests across 12 popular Python repositories. Given a codebase along with a description of an issue to be resolved, a language model is tasked with editing the codebase to address the issue. Resolving issues in SWE-bench frequently requires understanding and coordinating changes across multiple functions, classes, and even files simultaneously, calling for models to interact with execution environments, process extremely long contexts and perform complex reasoning that goes far beyond traditional code generation. Our evaluations show that both state-of-the-art proprietary models and our fine-tuned model SWE-Llama can resolve only the simplest issues. Claude 2 and GPT-4 solve a mere 4.8% and 1.7% of instances respectively, even when provided with an oracle retriever. Advances on SWE-bench represent steps towards LMs that are more practical, intelligent, and autonomous.

Keywords: Software Evaluation, Language Models, Code Resolution

1 Introduction

Language models (LMs) are rapidly being deployed in commercial products such as chatbots and coding assistants. At the same time, existing benchmarks have become saturated and fail to capture the frontier of what state-of-the-art LMs can and cannot do. There is a need for challenging benchmarks that more accurately reflect real-world applications of LMs to help shape their future development and usage.

Building a good benchmark is difficult since tasks must be challenging enough to stump existing models, but model predictions must also be easy to verify. Coding tasks are appealing as they pose challenging problems to LMs and generated solutions can be easily verified by running unit tests. However, existing coding benchmarks mostly involve self-contained problems that can be solved in a few lines of code.

In the real world, software engineering is not as simple. Fixing a bug might involve navigating a large repository, understanding the interplay between functions in different files, or spotting a small error in convoluted code. Inspired by this, we introduce SWE-bench, a benchmark that evaluates LMs in a realistic software engineering setting. Models are tasked to resolve issues (typically a bug report or a feature request) submitted to popular GitHub repositories. Each task requires generating a patch describing changes to apply to the existing codebase. The revised codebase is then evaluated using the repository's testing framework.

2 Related works

2.1 Evaluation of LMs

Evaluation of LMs. Several recent works for evaluating LMs have either proposed a collection of mutually distinct tasks spanning across multiple domains or turned to the web as an interactive setting featuring tasks that require multiple steps to solve [9]. There are several drawbacks with such a "potpourri" style setup. First, each task tends to narrowly focus on one or a few skills, resulting in challenges that are typically too simple, pigeonhole the model into a reduced role, and do not provide models with the bandwidth to exercise their versatility or potentially demonstrate new abilities. Consequently, a model's performance on such task conglomerations may not yield actionable, deep insights regarding its capabilities and how to improve them [7]. SWEbench addresses these shortcomings, as our work demonstrates that it is significantly challenging, presents a wide range of possibilities for improving LMs to solve this task, and is easy to refresh over time with new task instances, each of which introduce novel, nuanced, and practical challenges.

2.2 Code Generation Benchmarks

HumanEval is the current standard in a longstanding pursuit of synthesizing code from natural language descriptions [12]. In the past year, subsequent benchmarks have sought to augment HumanEval with extensions to different languages [6], variations in edit scope, similar but novel code completion tasks [5], and more testing. Simultaneously, separate works have sought to introduce new coding paradigms [10] or design library-specific problems [11]. Instead of partitioning problems into siloed datasets and curtailing them for simplicity's sake, SWE-bench's collection procedure transforms the source code with minimal post-processing, preserving a much broader set of challenges grounded in real-world software engineering beyond closed form completion, such as patch generation, reasoning over long contexts, navigating a codebase directory, and capturing dependency-based relationships across modules.

2.3 ML for Software Engineering

To overcome traditional program analysis techniques that may not scale or incorporate natural language, one direction of current software engineering research has is to use neural networks, including LMs, to automate real-world software development processes. Use cases include automating commit generation [3], PR review [8], bug localization, testing , and program repair [4]. Most relevant to SWE-

bench are works that have sought to apply LMs towards automated program repair, guiding code editing with commits [1]. However, none of the existing datasets present code context at the scale of SWE-bench. Moreover, SWE-bench isolates the changes at the function level, and can be easily extended to new programming languages and other software modalities. SWE-bench is compatible with such works, but provides a significantly more realistic and challenging arena to carry out future experiments towards augmenting LMs with software engineering tools and practices.

3 Method

SWE-bench is a benchmark featuring GitHub issues from popular repositories that report bugs or request new features, and pull requests that make changes to the repository to resolve these issues. The task is to generate a pull request that addresses a given issue and passes tests related to the issue.

3.1 Benchmark construction

GitHub is a rich data source for software development, but repositories, issues, and pull requests can be noisy, ad-hoc, or poorly documented or maintained. To find high-quality task instances at scale, we use a 3-stage pipeline as follows [2].

Stage I: Repo selection and data scraping. We start by collecting pull requests (PRs) from 12popular open-source Python repositories on GitHub, producing about 90,000 PRs in total. We focus on popular repositories as they tend be better maintained, have clear contributor guidelines, and have better test coverage. Each PR has an associated codebase, which is the state of the repository before the PR was merged.

Stage II: Attribute-based filtering. We create candidate tasks by selecting the merged PRs that (1) resolve a GitHub issue and (2) make changes to the test files of the repository, which indicates that the user likely contributed tests to check whether the issue has been resolved.

Stage III: Execution-based filtering. For each candidate task, we apply the PR's test content, and log the associated test results before and after the PR's other content is applied. We filter out task instances without at least one test where its status changes from a fail to pass (henceforth referred to as fail-to-pass test). We also filter out instances that result in installation or runtime errors.

Through these stages of filtering, the original 90,000 PRs are filtered down to the 2,294 task instances which comprise SWE-bench. A final breakdown of these task instances across repositories is presented in Figure 1, and Table 1 highlights the key features of SWE-bench task instances. We highlight that the codebases are large with thousands of files, and the reference pull requests often make changes to multiple files at once.

3.2 Task Formulation

Model input. A model is given an issue text description and a complete codebase. The model is then tasked to make an edit to the codebase to resolve the issue. In practice, we represent edits as patch files, which specify which lines in the codebase to modify in order to resolve the issue.

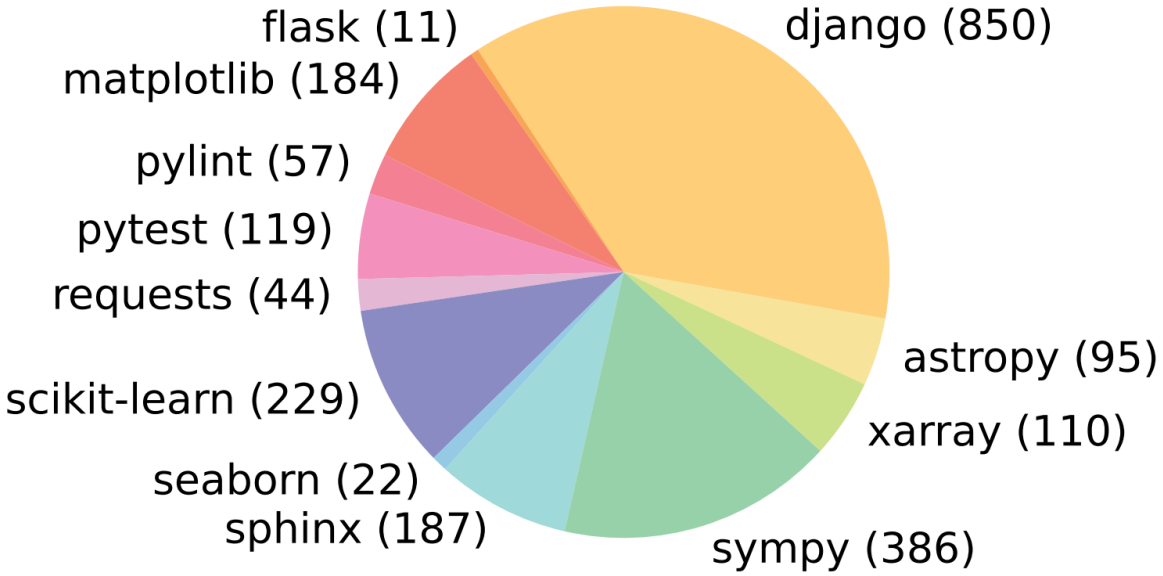

Figure 1. Distribution of SWE-bench tasks (in parenthesis) across 12 open source GitHub repositories that each contains the source code for a popular, widely downloaded PyPI package.

Evaluation metrics. To evaluate a proposed solution, we apply the generated patch, using unix'spatch program, to the codebase and then execute the unit and system tests associated with the task instance. If the patch applies successfully and all of these tests pass we consider the proposed solution to have successfully resolved the issue. The metric for our benchmark is the percentage of task instances that are resolved.

3.3 Features of SWE-bench

Traditional benchmarks in NLP typically involve only short input and output sequences and consider somewhat "contrived" problems created specifically for the benchmark. In contrast, SWE-bench's realistic construction setting imbues the dataset with unique properties, which we discuss below.

Real-world software engineering tasks. Since each task instance in SWE-bench consists of a large and complex codebase and a description of a relevant issue, solving SWE-bench requires demonstrating sophisticated skills and knowledge possessed by experienced software engineers but are not commonly evaluated in traditional code generation benchmarks.

Continually updatable. Our collection process can be easily applied to any Python repository on GitHub and requires almost no human intervention. Therefore, we can extend SWE-bench with a continual supply of new task instances and evaluate LMs on issues created after their training date, which ensures that the solution was not included in their training corpus.

Diverse long inputs. Issue descriptions are typically long and detailed (195 words on average), and codebases regularly contain many thousands of files. Solving SWE-bench requires identifying the relatively small number of lines that need to be edited to solve the issue amongst a sea of context.

Robust evaluation. For each task instance, there is at least one fail-to-pass test which was used to test the reference solution, and 40% of instances have at least two fail-to-pass tests. These tests evaluate

whether the model addressed the problem in the issue. In addition, a median of 51 additional tests run to check whether prior functionality is properly maintained.

Cross-context code editing. Unlike prior settings that may constrain scope to a function or class or provide cloze-style fill-in blanks, SWE-bench does not provide such explicit guidance. Rather than merely having to produce a short code snippet, our benchmark challenges models to generate revisions in multiple locations of a large codebase. SWE-bench's reference solutions average editing1.7 files, 3.0 functions, and 32.8 lines (added or removed).

Wide scope for possible solutions. The task of repository-scale code editing can serve as a level playing field to compare approaches ranging from retrieval and long-context models to decisionmaking agents, which could reason and act in code. SWE-bench also allows creative freedom, as models can generate novel solutions that may deviate from the reference PR.

3.4 SWE-Llama: Fine-tuning CodeLlama for SWE-bench

Benchmarking the performance of open models on SWE-bench against proprietary models is essential. Currently, the CodeLlama models are the only ones capable of handling the extensive contexts necessary for this task. However, it has been observed that the standard variants of CodeLlama models struggle with following detailed instructions to generate comprehensive code edits across repositories, often resulting in placeholder responses or irrelevant code.

To evaluate these models' capabilities more accurately, supervised fine-tuning was performed on both the 7 billion and 13 billion-parameter CodeLlama-Python models. This process transformed them into specialized repository editors that are capable of running on consumer hardware and resolving GitHub issues effectively.

For training data, a specific data collection procedure was followed to gather 19,000 issue-pull request pairs from an additional 37 popular Python package repositories. Unlike the initial approach described in the earlier section, the pull requests in this phase of data collection did not need to contribute test changes. This adjustment allowed for the creation of a much larger training set for supervised fine-tuning. To minimize any risk of data contamination, the repositories used for training were carefully chosen to be distinct from those included in the evaluation benchmark.

Training details included using the instructions, issue texts from GitHub, and the relevant code files as prompts for fine-tuning SWE-Llama. The goal was to enable the model to generate the patch that solved the given issue, referred to as the "gold patch". To enhance memory efficiency, only the weights of the attention sublayer were fine-tuned using LoRA, and training sequences were limited to those with no more than 30,000 tokens. This limitation reduced the effective size of the training corpus to 10,000 instances.

3.5 Retrieval-Based Approach

SWE-bench instances provide an issue description and a codebase as input to the model. While issues descriptions are usually short (195 words on average as shown in Table 1), codebases consist of many more tokens (438K lines on average) than can typically be fit into an LMs context window.

Then the question remains of exactly how to choose the relevant context to provide to the model during generation?

To address this issue for our baselines, we simply use a generic retrieval system to select the files to insert as context. In particular, we evaluate models under two relevant context settings: 1) sparse retrieval and 2) an oracle retrieval.

Sparse retrieval. Dense retrieval methods are ill-suited to our setting due to very long key and query lengths, and especially the unusual setting of retrieving code documents with natural language queries. Therefore, we choose to use BM25 retrieval to retrieve relevant files to provide as context for each task instance. We experiment with three different maximum context limits, and simply retrieve as many files as fits within the specified limit. We evaluate each model on all limits that fit within its context window and report the best performance. "Oracle" retrieval. We additionally consider a setting where we only use all files edited by the reference patch that solved the issue on GitHub. This "oracle" retrieval setting is less realistic, since a software engineer working on addressing an issue does not know a priori which files may need to be modified. However, this setting is also not necessarily comprehensive since edited files alone may not include all the required context to understand exactly how software will behave when interacting with unseen parts of the code.

We compare the BM25 retrieval results against the "oracle" retrieval setting in Table 3, where we see that BM25 retrieves a superset of the oracle files in about 40% of instances with the 27,000 token context limit but only also excludes all of the oracle files in over half of instances.

Table 1. Average and maximum numbers characterizing different attributes of a SWE-bench task instance. Statistics are micro-averages calculated without grouping by repository.

		Mean	Max
Issue Text	Length (Words)	195.1	4477
Codebase	# Files (non-test)	3,010	5,890
	# Lines (non-test)	438K	886K
Gold Patch	# Lines edited	32.8	5888
	# Files edited	1.7	31
	# Func. edited	3	36
Tests	# Fail to Pass	9.1	1633
	# Total	120.8	9459

4 Implementation details

4.1 Comparing with the released source codes

During the replication process of this research, due to the complexity of the project and the costliness of the computational resources required, direct innovation and significant technical improvement were

not the focus of this work. Instead, the main contribution lies in the successful replication of an existing open-source project, ensuring its stability and feasibility in new datasets and environments. Specifically, the project successfully deployed the service and created a stable virtual environment to support the inference process of large models. After the model generated patches, a thorough verification process was established to ensure the correctness and effectiveness of the patches. This process involved not only a technical replication but also precise control of the environment and strict verification of the results, ensuring the reliability and practicality of the entire system. The process required an in-depth understanding of the original code and the adaptation and adjustment of various configurations to ensure the accuracy of the results.In the replication and deployment process, I referred to the following open-source code repository:

Princeton NLP's SWE-bench(https://github.com/princeton-nlp/SWE-bench/): This repository provided the basic architecture and necessary guidance that allowed me to set up a complete service in the local environment. Based on the understanding and application of these resources, I was able to effectively customize and extend the service.

Completing the replication of such a complex project under resource-constrained conditions is itself a technical challenge. Through this process, our research team demonstrated a deep understanding of existing technologies and provided detailed implementation details, which are of reference value for subsequent research and engineering practices. Moreover, although there was no direct technical innovation in this work, we have accumulated valuable experience in project configuration and code debugging. These experiences will lay a solid foundation for future innovative research in environments with more abundant resources.

4.2 Experimental environment setup

SWE-bench instances provide an issue description and a codebase as input to the model. While issues descriptions are usually short (195 words on average), codebases consist of many more tokens (438K lines on average) than can typically be fit into an LMs context window. Then the question remains of exactly how to choose the relevant context to provide to the model during generation? To address this issue for our baselines, we simply use a generic retrieval system to select the files to insert as context. In particular, we evaluate models under two relevant context settings: 1) sparse retrieval and 2) an oracle retrieval.

Once the retrieved files are selected using one of the two methods above, we construct the input to the model consisting of task instructions, the issue text, retrieved files and documentation, and finally an example patch file and prompt for generating the patch file.

Due to the need to process long sequence lengths, there are only a few models that are currently suitable for SWE-bench. In the original, they evaluate ChatGPT-3.5 (gpt-3.5-turbo-16k-0613), GPT-4 (gpt-4-32k-0613), Claude 2, and SWE-Llama with their context limits shown in Table 2.

Due to resource constraints, this replication work only conducted partial inference on SWE-Llama and used the Claude 2 API for additional inference. The retrieval method employed was "Oracle" Retrieval. In this experiment, SWE-Llama 13b was run on a server equipped with 4 A100 GPUs, while Claude 2 was tested through the API provided by its service provider.

Table 2. We compare the different context lengths and proportion of the "oracle" retrieval setting covered. Models with shorter context lengths are thus inherently disadvantaged. Note that descriptions of token-lengths is a relative non-standard measure (e.g. Llama-tokenized sequences are 42% longer on average than the equivalent sequence tokenized for GPT-4).

	ChatGPT-3.5	GPT-4	Claude 2	SWE-Llama
Max. Tokens	16,385	32,768	100,000	$\geq 100,000$
% of Instances	58.1%	84.1%	96.4%	$\geq 94.8\%$

5 Results and analysis

In the original text, results are reported for models in a multitude of settings with different retrieval mechanisms and prompting styles, providing analysis and insight into model performance and difficulty. Performance of the models on both the BM25 and "oracle" retrieval settings is summarized in Table 3. Across the board, models struggle significantly to resolve issues. The best performing model, Claude 2, achieves only a 4.8% pass rate using the "oracle" retrieval context. Evaluated in the BM25 retrieval setting, the performance of Claude 2 drops to 1.96%. The performance in the BM25 retrieval setting underscores the importance of choosing appropriate context, which becomes a theme in the analysis that is discussed further below.

Difficulty differs across repositories. When breaking down performance by repository, it is observed that all models show similar trends across different repositories. Despite this, the issues resolved by each model do not necessarily overlap extensively. For instance, in the "oracle" setting Claude 2 and SWE-Llama 13b perform comparably, with each model resolving 110 and 91 instances respectively. Yet of these instances, Claude 2 only solves 42% of the instances solved by SWE-Llama.

Table 3. We compare models against each other using the BM25 and oracle retrieval settings as described in Section 4. *Due to budget constraints we evaluate GPT-4 on a 25% random subset of SWE-bench in the "oracle" and BM25 27K retriever settings only.

	BM25 Retrieval		"Oracle" Retrieval	
Model	% Resolved	% Apply	% Resolved	% Apply
ChatGPT-3.5	0.20	10.50	0.52	12.38
Claude 2	1.96	29.86	4.80	47.00
GPT-4*	0.00	4.50	1.74	13.20
SWE-Llama 7b	0.70	37.84	3.00	54.80
SWE-Llama 13b	0.70	39.41	4.00	52.10

Due to various constraints, I have conducted experiments with SWE-Llama 13b and Claude 2 only on a subset of cases. Details of the experimental results are presented in Table 4.

In this experiment, the performance of the SWE-Llama 13b and Claude 2 models was evaluated

Table 4. The results were based on "Oracle" Retrieval, and due to computational resource limitations, only 1551 cases were generated under the SWE-Llama 13b model. Owing to budget constraints, only 23 cases were generated under the Claude 2 model.

Model	Generated	With Logs	Applied	Resolved
SWE-Llama-13b	1551 (67.61%)	1453(93.68%)	480(30.94%)	3(0.19%)
Claude 2	23(1%)	11(47.8%)	6(26%)	1(4.42%)

using two retrieval mechanisms—BM25 and "Oracle" retrieval. The results have revealed significant challenges that even the most advanced language models face in resolving real-world software engineering issues.

Within the "Oracle" retrieval setup, the Claude 2 model exhibited the best performance, yet it achieved a mere 4.8% pass rate, underscoring that problem resolution remains a difficult task even under ideal information retrieval conditions. Similarly, when using the BM25 retrieval mechanism, the performance of all models declined significantly, with Claude 2's pass rate dropping to 1.96%, further highlighting the importance of selecting the appropriate context.

In terms of the number of cases generated, SWE-Llama 13b produced 1551 cases, accounting for 67.61% of the total, with 93.68% of these cases having logs. However, the actual application of cases was only 30.94%, and the cases successfully resolved were less than 0.2%. This indicates that despite generating a large number of cases, the cases that were ultimately resolved were very limited.

The Claude 2 model generated only 23 cases in the experiment, of which 47.8% had logs, 26% were applied, and the resolution rate was 4.43%. Although the number of cases was limited, the resolution rate was relatively high, which may suggest that under certain conditions, the Claude 2 model could perform better in solving specific types of issues.

Overall, these results demonstrate the limitations of existing language models in dealing with complex software engineering problems and also highlight potential areas for future research to improve model performance and retrieval mechanisms.

6 Conclusion and future work

In the original research, the challenging benchmark SWE-bench and the large model SWE-Llama designed to solve real-world coding problems were introduced. Through the evaluation of the performance of the SWE-Llama 13b and Claude 2 models under BM25 and "Oracle" retrieval mechanisms, a series of meaningful conclusions were drawn. Although the SWE-Llama 13b model was able to generate a large number of cases within the "Oracle" retrieval environment, the resolution rate was extremely low, highlighting the significant challenges that language models still face in solving complex software engineering problems, even under ideal conditions. Meanwhile, Claude 2 demonstrated a relatively better resolution rate in limited case tests, which may suggest the potential of the model under specific settings. The SWE-bench benchmark and the SWE-Llama model proposed in the original research, although not yet achieving the desired level in problem resolution rate, have provided a reliable evaluation

framework and baseline for subsequent research.

Future work will focus on training more effective large language models (LLMs) to improve the task pass rate on platforms like SWE-bench. It is anticipated that significant improvements in model performance can be achieved through the integration of more advanced model architectures, the optimization of training algorithms, and the expansion of training datasets. Moreover, future research will explore the development of LLM Agent, an automated tool that can work in conjunction with existing models to further improve the success rate of problem resolution by providing more accurate problem contexts and more effective code patch suggestions. Through these efforts, we hope not only to enhance the absolute performance of the models but also to increase their usability and reliability in real-world applications.

References

[1] Saikat Chakraborty and Baishakhi Ray. On multi-modal learning of editing source code. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Nov 2021.

[2] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2023.

[3] Shangqing Liu, Yanzhou Li, and Yang Liu. Commitbart: A large pre-trained model for github commits. Aug 2022.

[4] Martin Monperrus. Automatic software repair. ACM Computing Surveys, page 1–24, Jan 2019.

[5] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue, Zhuo Swayam, SinghXiangru Tang, LeandroVon Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models.

[6] Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishah Singh, and Michele Catasta. Measuring the impact of programming language distribution. Feb 2023.

[7] David Schlangen. Language tasks and language games: On methodology in current natural language processing research. Cornell University - arXiv,Cornell University - arXiv, Aug 2019.

[8] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. Mining the modern code review repositories. In Proceedings of the 13th International Conference on Mining Software Repositories, May 2016.

[9] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. Jul 2022.

[10] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. Natural language to code generation in interactive data science notebooks. Dec 2022.

[11] 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.

[12] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. Large language models meet nl2code: A survey. Dec 2022.