ARBench: Algorithmic Reasoner or API Alchemist? Evaluating LLMs Beyond API Calls

Ren-Biao Liu[1,2], **Chao-Zeng Ma**[1,2], **Anqi Li**[1,2], **Hui Sun**[1,2], **Xin-Ye Li**[1,2], **Ming Li**[*1,2]

[1]National Key Laboratory for Novel Software Technology, Nanjing University, China
[2]School of Artificial Intelligence, Nanjing University, China
{liurb, macz, liaq, sunh, lixy, lim}@lamda.nju.edu.cn

Abstract

Large Language Models (LLMs) have demonstrated impressive capabilities in code generation. Like human programmers, LLMs tend to call high-level APIs and libraries to program efficiently. However, this shortcut may hinder LLMs from learning the essential algorithm reasoning, leading instead to rote memorization of API usage. As a result, LLMs often struggle to generalize to new or domain-specific algorithms that lack ready-made library support. In this work, we propose ARBench, a novel benchmark for evaluating LLMs' ability to generate machine learning algorithms from scratch, beyond merely invoking high-level APIs. It emphasizes algorithmic reasoning and implementation, distinguishing genuine understanding from superficial API usage. It covers fundamental and advanced machine learning tasks, rigorously assessing current LLMs' capacity to implement these algorithms from scratch. Our evaluation reveals the strengths and weaknesses of state-of-the-art LLMs in algorithmic reasoning and generalization, offering valuable insights to guide future research and development.

Introduction

Code generation seeks to automatically produce high-quality, executable programs that fulfill given problem specifications (Jiang et al. 2024; Roziere et al. 2023; Hui et al. 2024). Recently, Large Language Models (LLMs), trained on large-scale corpora of natural language and source code, have shown strong potential in understanding and generating complex programs (Brown et al. 2020; Askell et al. 2021; Grattafiori et al. 2024), garnering widespread attention.

Numerous benchmarks have been proposed to evaluate LLMs on general code generation tasks, where current LLMs have achieved pretty good performance. For example, LiveCodeBench (Jain et al. 2025), APPS (Hendrycks et al. 2021), and CodeContest (Li et al. 2022) primarily evaluate the correctness of generated code using test cases for isolated algorithmic tasks, often resembling competitive programming problems on platforms such as LeetCode.

There are two main limitations. First, prior benchmarks mainly focus on competitive programming and lack a comprehensive evaluation of ML algorithms in real-world scenarios. While these benchmarks help test correctness in solving non-ML tasks, they fail to provide deep insights into LLM development for ML-oriented code generation, including the bias-variance trade-off, appropriate model evaluation methods (e.g., cross-validation), and the theoretical conditions necessary for convergence. Consequently, they are insufficient for assessing the ability or readiness of LLMs for complex, real-world ML-oriented programming scenarios.

Second, and more importantly, like human programmers, LLMs often rely on high-level third-party APIs and libraries. For example, an LLM may directly import a `LinearRegression` from `scikit-learn` to solve a regression task. Although high-level API usage enhances LLMs, it may hinder generalization in code generation. Specifically, when generating ML algorithms, LLMs (and even human programmers) who rely solely on memorized API usage without understanding the underlying implementation often fail to truly understand the algorithms. As famously stated:

> *"What I cannot create, I do not understand."*
>
> — Richard P. Feynman

As a result, when faced with evolving or domain-specific requirements, LLMs may fail to improve and adapt the code of ML algorithms. Thus, relying solely on high-level APIs becomes a shortcut that ultimately impedes the generalization of LLMs. Consequently, prior benchmarks, such as those of Tang et al. (2023); Lai et al. (2023); Zhuo et al. (2025), which include ML coding tasks, do not constrain high-level API usage and therefore do not determine whether LLMs have truly learned the algorithmic principles underlying these APIs, which rely on rote memorization of black-box API calls.

Hence, our motivation is to investigate whether LLMs truly understand ML algorithms and can implement them from scratch with algorithmic reasoning, when restricted from invoking high-level APIs. We advocate that: *"What LLMs cannot implement from scratch, they do not understand. Know how to implement every high-level function that has been truly understood."*

In this study, we propose a new ML-oriented code generation benchmark, **ARBench**, designed to evaluate Algorithmic Reasoning ability in ML code generation rather than end-to-end problem-solving performance. It helps assess the

*Ming Li is the corresponding author.

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

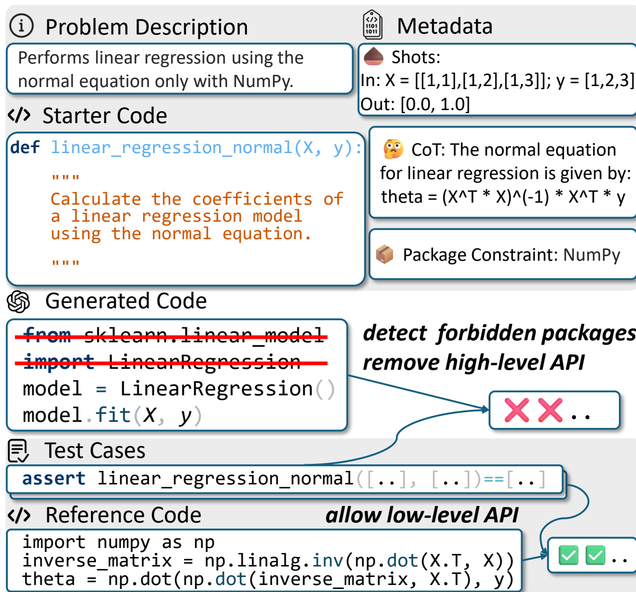


Figure 1: Illustration of the data composition and the evaluation procedure of ARBench. Content with a gray background constitutes a single task in the benchmark. Based on the content in the upper part, LLM generates ML code, which is later evaluated using the test cases shown in the lower part. The evaluation enforces restrictions on high-level API imports, allowing only a predefined set of packages.

potential of ML-oriented code generation to generalize to evolving and domain-specific ML tasks. As shown in Figure 1, each task includes a problem description, reference code, starter code, test cases, and metadata containing chain-of-thought (CoT) annotations (e.g., efficiency, usage, and hyperparameter analysis), and package constraints.

This benchmark consists of two parts: a human-annotated split and a model-synthesized split. The human-annotated split includes problems collected from online platforms and manually refined for clarity and accuracy. The model-synthesized split is constructed from academic and open-source code snippets, categorized and synthesized using LLMs. The benchmark comprises 300 unique ML algorithm tasks, featuring longer prompts, more detailed solutions, and more test cases than existing benchmarks. Analysis shows these tasks, especially in the ML algorithm code generation field, show higher (cyclomatic complexity) and broader (number of input parameters) complexity, demanding stronger algorithmic reasoning and coding skills.

We evaluate state-of-the-art (SOTA) LLMs on ARBench, as shown in Figure 1, the use of third-party libraries beyond specified package constraints is prohibited. Results show that while current LLMs perform well with access to high-level APIs and libraries, they struggle to implement ML methods from scratch, exhibiting weak algorithmic reasoning and frequent generation failures. Moreover, models with enhanced algorithmic reasoning, such as reasoning models trained with reinforcement learning, perform significantly better than models that rely primarily on code completion.

In summary, the contributions of this work are as follows:

- **Novel Algorithmic Reason Evaluation** We propose ARBench, a new benchmark specifically designed to evaluate the algorithmic reasoning of LLMs in ML code generation by requiring a from-scratch implementation and restricting high-level API usage, addressing a critical gap in existing evaluations.
- **Specialized Algorithmic Task Dataset** We construct a dataset of 300 unique ML algorithm tasks featuring detailed descriptions, CoT annotations, and explicit package constraints designed with higher complexity to test deep algorithmic understanding strictly.
- **Key LLM Capability Insights** We reveal that SOTA LLMs struggle to implement ML algorithms from scratch, indicating deficiencies in algorithmic reasoning. We find that models enhanced for such reasoning perform markedly better on ARBench.

Related Work

Large Language Models LLMs (Brown et al. 2020; Achiam et al. 2023; Zhao et al. 2023) have emerged as powerful tools in NLP, advancing language understanding and generation. Due to large-scale pre-training, LLMs (Radford et al. 2019; Askell et al. 2021; Bai et al. 2022) exhibit robust generalization capabilities. However, due to the distribution gap between code data and the general text corpora (Huang et al. 2025), their code generation performance remains limited, leading to difficulties in logical reasoning and syntactic precision (Dou et al. 2026). Consequently, CodeLLMs (Roziere et al. 2023; Bai et al. 2023) are trained for programming tasks through specific optimizations, such as syntax-aware learning (Gong, Elhoushi, and Cheung 2024) and specialized tokenization (Bavarian et al. 2022), during pre-training.

Code Generation with LLMs CodeLLMs, such as StarCoder (Li et al. 2023a), StarCoder2 (Lozhkov et al. 2024), and DeepSeek-Coder (Guo et al. 2024), exhibit strong foundational programming capabilities and excel in code generation. To further enhance these models, Supervised Fine-Tuning(SFT) has been employed to improve task-specific performance and model alignment (Luo et al. 2023; Wei et al. 2024; Zheng et al. 2024; Liu et al. 2025b). Code Alpaca (Chaudhary and Sahil 2023) adopts the Self-Instruct method (Wang et al. 2023), a technique that utilizes ChatGPT to generate high-quality instruction datasets. In contrast, Vicuna (Peng et al. 2023) is fine-tuned on daily conversational datasets collected from a data-sharing platform. Similarly, WizardLM (Luo et al. 2023) uses the Evol-Instruct method to iteratively enhance instruction datasets, resulting in more complex and diverse tasks.

Code Generation Benchmarks Many benchmarks have been developed to evaluate the code generation capabilities of LLMs. For basic Python programming, widely used benchmarks include HumanEval (Chen et al. 2021) and MBPP (Mostly Basic Programming Problems) (Austin et al. 2021). EvalPlus (Liu et al. 2023) provides a more rigorous evaluation variant, typically by extending the test cases of existing benchmarks. MultiPL-E (Cassano et al.

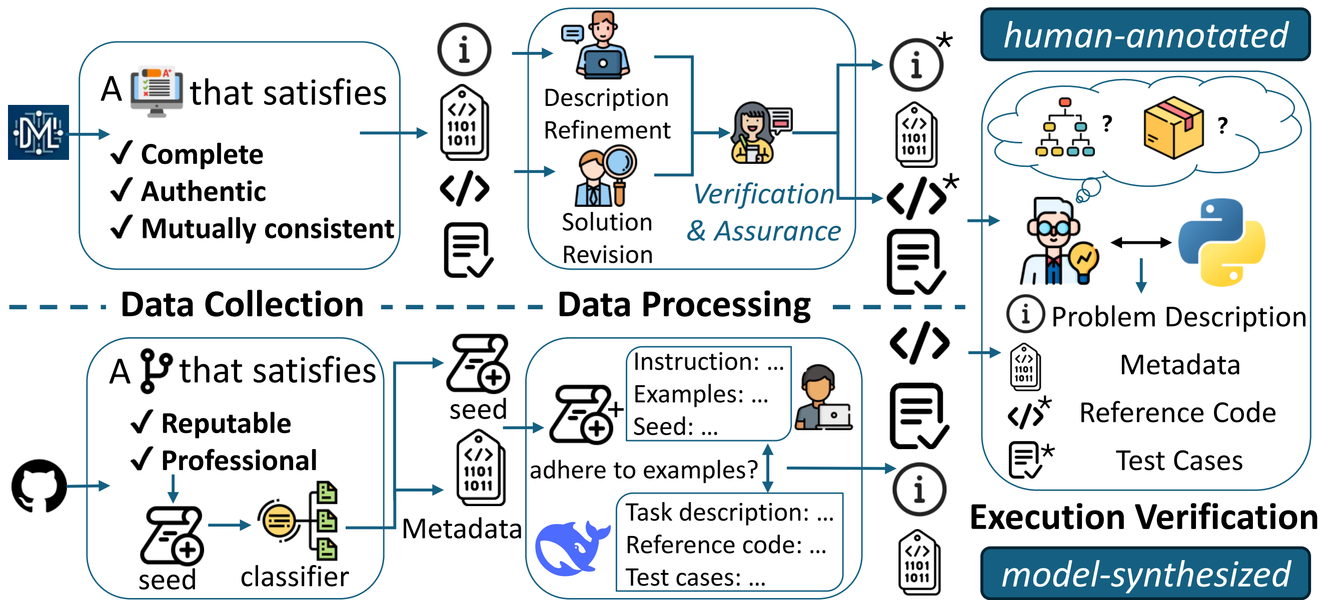


Figure 2: Overview of the ARBench construction pipeline. Both splits follow a three-stage process; however, differences in data sources result in variations in the first two stages (i.e., data collection and data processing). Asterisks (⋆) indicate steps involving human expert revision.

2022) extends these two popular benchmarks (HumanEval and MBPP) by translating them into 18 additional programming languages. As LLM capabilities advance, many of these benchmarks become too simplistic to accurately evaluate modern models. Some specialized benchmarks focus on competitive programming challenges. APPS (Hendrycks et al. 2021), CodeContests (Li et al. 2022), and TACO (Li et al. 2023b) source problems from platforms such as Codeforces and AtCoder. LiveCodeBench (Jain et al. 2025) provides holistic, contamination-free evaluations by dynamically updating coding challenges from platforms such as LeetCode (Xia et al. 2025) and AtCoder. CODEELO (Quan et al. 2025) aligns with the Codeforces platform by enabling direct submissions and developing an Elo rating calculation system. The CAPIR approach (Ma et al. 2024) proposes a composite API recommendation method that decomposes coarse-grained task descriptions into multiple subtasks and recommends relevant APIs for each.

Benchmark Construction

The novel benchmark, ARBench, comprises two complementary splits: human-annotated and model-synthesized, constructed through a unified, three-stage process: data collection, data processing, and execution verification. This approach ensures both conceptual rigor and diverse task coverage. An overview of the construction pipeline is shown in Figure 2. In the following sections, we detail each step of the ARBench construction process.

Data Collection

We design a comprehensive data collection framework with two distinct splits to support a comprehensive and unbiased evaluation of LLMs across various ML tasks. Each split is constructed from different sources, ensuring broad coverage and avoiding cross-contamination, while also enabling exploration of synthetic task generation.

ARB-Human denotes the human-annotated split of ARBench. We systematically collect tasks from leading open-source data science competition platforms, such as HackerRank[1], stratascratch[2], and Deep-ML[3]. Tasks are selected based on high community engagement and often reflect their practical significance. Human experts then review the description, reference solution, test cases, and metadata for each task. Only tasks with accurate and consistent components are retained.

ARB-Model refers to the split synthesized by the model. While human-annotated tasks ensure high-quality benchmarks, they are time-consuming and resource-intensive. To scale ARBench more efficiently, we use LLMs to automatically synthesize tasks, augmenting and extending the benchmark. The synthetic tasks included in this split are deliberately designed to address problems in real-world machine learning programming scenarios. Codes from reputable academic and professional sources represent the optimal choice as a task source.

In particular, we use exercises from machine learning textbooks and code from popular GitHub repositories, such as ML-From-Scratch[4], numpy-ml[5], and MLAlgorithms[6].

[1]https://www.hackerrank.com/domains/ai
[2]https://www.stratascratch.com/
[3]https://www.deep-ml.com/
[4]https://github.com/eriklindernoren/ML-From-Scratch
[5]https://github.com/ddbourgin/numpy-ml
[6]https://github.com/rushter/MLAlgorithms

Benchmark	Task	Test(Avg.)		Prompt(Avg.)		Solution(Avg.)	
		Num.	Cov(%).	Char.	Line.	Char.	Line.
HumanEval	164	7.8	98	450.6	13.7	180.9	6.8
MBPP+	378	3.1	98	91.5	1.0	121.1	5.1
DS-1000	1,000	1.6	98	871.8	29.1	138.1	5.1
ODEX	945	1.8	96	87.5	1.0	50.4	1.9
BigCodeBench	1,140	5.6	99	663.2	11.7	426.0	10.0
ARB-Human	128	3.9	98	712.8.7	14.9	498.7	15.8
ARB-Model	172	9.4	99	1823.9	40.3	2341.7	61.6

Table 1: Benchmark statistics of representative function-level Python programming. **Num.**: Number of test cases. **Cov.**: Branch Coverage. **Char.**: Number of Characters. **Line.**: Number of Lines.

We extract basic code snippets and algorithmic constructs from these sources that serve as foundational components of various ML techniques, and utilize them to synthesize task instances later. For each code snippet, the LLM is asked to select a difficulty level and a task category from a predefined set of multiple candidates. To ensure the accuracy of both attributes, we employ a majority-voting strategy across three independent LLM runs. We retain only code snippets with at least two matching votes to ensure clarity and structural quality. This pipeline produces thousands of valid seed examples for further processing.

Data Processing

To ensure the quality and diversity of ARBench, we apply two tailored processing pipelines for the two splits.

ARB-Human We follow a multi-stage process for the human-annotated split to ensure task clarity and correctness, as shown in the upper part of Figure 2. A team of domain experts with advanced training and practical experience in ML reviews tasks collected in the previous stage. To ensure quality and correctness, each problem-solution pair undergoes three independent annotation steps, each conducted by different expert groups:

- *Problem Description Refinement* The first expert group refines the problem description to enhance clarity, eliminate ambiguities, ensure precise terminology, and provide sufficient context to make the task well-defined and appropriately scoped.
- *Reference Solution Revision* Another expert group revises the reference solution by correcting factual errors and logical inconsistencies, and by adding clear explanations and methodological steps to ensure the solution is accurate and robust.
- *Consistency Verification and Quality Assurance* The third expert group reviews the refined problem description and the revised reference solution to verify their semantic and logical alignment. This step ensures that the solution fully addresses the task and that the final annotation is both internally consistent and of high quality.

ARB-Model To synthesize complete task instances, inspired by recent work (Wei et al. 2024; Luo et al. 2023), we adopt a prompt-driven approach, using canonical context and a multi-agent framework to guide LLMs in generating diverse and well-structured programming tasks from seed code. For each task instance, problem descriptions, metadata, initial reference solutions, and corresponding tests are generated. And the synthesis pipeline involves three key components:

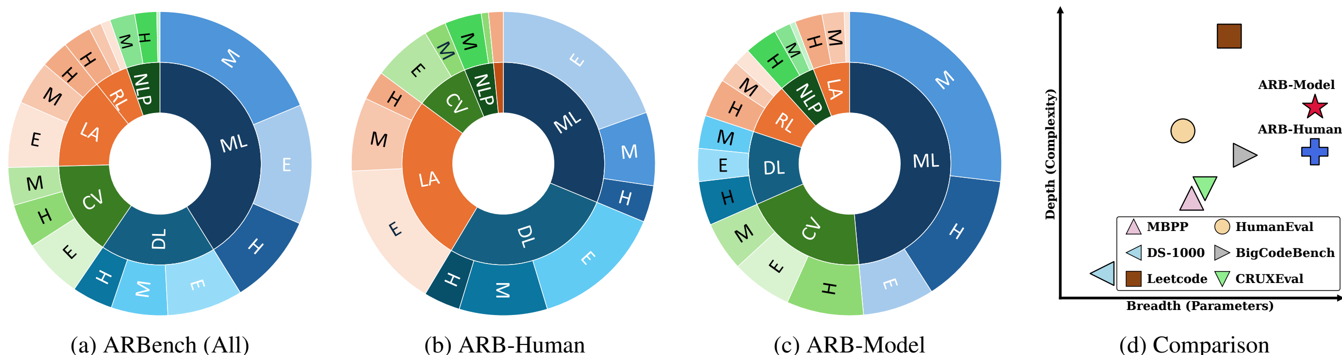
- *Few-shot Prompt Template* We construct few-shot prompts with three examples to take advantage of the LLM's capability of in-context learning. It guides the model to generate coherent, self-contained tasks from the provided seed examples.
- *Model Selection* Synthesizing the whole task instance is inherently challenging, even with reference examples. Thus, choosing a capable model is crucial. We conducted pilot experiments comparing candidate models on representative seed inputs. Based on their performance, we selected DeepSeek-R1 (Guo et al. 2025) as the primary generator. It provides transparent reasoning steps, allowing for direct inspection of its outputs and thereby enhancing the trustworthiness of the generated content.
- *Bias Mitigation* Prior studies (Zheng et al. 2023; Panickssery et al. 2024) show that LLMs often exhibit biases favoring their generations, which may skew benchmark evaluations. We introduce additional models, such as Qwen-2.5-Coder, as reviewers to validate synthesized tasks, thereby reducing this risk. Specifically, the models evaluate whether the generated problems align with the difficulty, category, and original knowledge concepts. Only tasks independently confirmed by at least one reviewer are included in the benchmark.

Execution Verification

During synthesis, we observe three recurring failure modes: (1) ML code generated by LLMs often contains critical flaws, such as incorrect optimization procedures, misused mathematical operations, or improper data handling. (2) LLMs may overuse high-level third-party libraries, which ignore algorithmic reasoning; (3) Due to numerical hallucinations of LLMs, the outputs of generated test cases are often unreliable. These issues compromise their reliability and render many outputs unusable for evaluation. To address these issues, we introduce the following execution-based verification framework to construct high-quality reference solutions and test cases.

Lexical Correctness We employ an Abstract Syntax Tree (AST) parser to verify the lexical correctness of LLM-generated code. This step ensures that only syntactically well-formed code (free from basic lexical and structural errors) proceeds to subsequent validation stages.

Syntactic Validity For syntactic validation, we analyze the code's dependency graph. Our method mandates a unique entry point for each code and extracts all reachable and relevant functions. It helps isolate the core logic and allows pruning of unrelated or unsafe components, such as unnecessary imports or disallowed library calls (Liu et al. 2023; Zhuo et al. 2025). This step is essential, as LLMs may generate code that appears correct but lacks necessary



(a) ARBench (All) (b) ARB-Human (c) ARB-Model (d) Comparison

Figure 3: Panels (a–c) show sub-domains (**LA**: Linear Algebra. **ML**: Machine Learning. **DL**: Deep Learning. **NLP**: Natural Language Processing. **CV**: Computer Vision. **RL**: Reinforcement Learning) and difficulty distributions (**E**: Easy. **M**: Medium. **H**: Hard) of ARBench, while (d) compares ARBench with existing benchmarks in argument breadth and reasoning depth.

imports or includes disallowed libraries. Without enforcing package constraints, the model may circumvent evaluation by taking shortcuts.

Semantic Accuracy Finally, domain experts manually revise the filtered reference solutions based on the synthesized problem descriptions. Using refined solutions, numerical inputs extracted from generated test cases are used to construct reference outputs, and the original outputs of numerical hallucinations are discarded. Given that many semantically accurate ML algorithm implementations produce numerically unstable results, we adopt tolerance-aware comparison when constructing the refined test cases, ensuring that the outputs of other implementations within an acceptable range pass the test.

Benchmark Statistics

Following the construction process described in the previous section, ARBench comprises **300** unique ML code generation tasks. Each task includes a well-defined problem description, consistent metadata, a reference implementation, and a comprehensive set of test cases. All tasks undergo a human-in-the-loop verification to ensure semantic accuracy and numerical correctness.

Table 1 compares key statistics of ARBench with prior benchmarks, including HumanEval(Chen et al. 2021), MBPP+(Austin et al. 2021), DS-1000(Lai et al. 2023), ODEX(Wang et al. 2022), and BigCodeBench(Zhuo et al. 2025), which shows that tasks in both ARB-Human and ARB-Model have significantly longer prompts and solutions, both in characters and lines, than previous benchmarks. This suggests that ARBench provides more detailed instructions and more complex reasoning implementations.

Besides, ARB-Model exhibits the highest average number of test cases, supporting a more rigorous execution-based evaluation. Consequently, ARBench (particularly its model-synthesized split) poses greater challenges in algorithmic reasoning and programming, thereby raising the standard for evaluating LLMs' ability to generate ML algorithm code.

Figure 3a-c presents ARBench statistics across various ML sub-domains and three difficulty levels. Additionally, Figure 3d compares the two splits with existing benchmarks, illustrating two dimensions: breadth (number of input function parameters, x-axis) and depth (cyclomatic complexity of the reference code, y-axis). ARB-Human exhibits a moderate number of input parameters but relatively high cyclomatic complexity, indicating more intricate solution logic. On the other hand, ARB-Model poses a greater challenge, requiring the management of larger input interfaces and the implementation of algorithms with more complex control flows than the other benchmarks.

Experiment

Experiment Setup

We evaluate various SOTA LLMs, including proprietary models accessed via official APIs, such as OpenAI's o-series (Ye et al. 2023) and GPT-4 variants (Achiam et al. 2023; Hurst et al. 2024), Gemini-2.5 (Team et al. 2023), DouBao-1.5 (ByteDance 2025), and the Claude family (Anthropic 2024). We also evaluate several open-source LLMs, including Qwen3 (Yang et al. 2025), Qwen2.5-Coder (Hui et al. 2024), LLaMA-4 (Meta 2025), LLaMA-3 (Grattafiori et al. 2024), DeepSeek V3 (Liu et al. 2025a), DeepSeek R1 (Guo et al. 2025), and DeepSeek-Coder-33B-Instruct (Guo et al. 2024). Each model was tasked with completing code generation problems from both ARBench splits. All models share the same prompt and context settings for consistency.

Evaluation Protocol

The primary evaluation metric is **Pass@1** using greedy decoding. To enforce constraints on high-level external package usage, disallowed import statements were replaced with pre-defined imports tailored to each benchmark task. This preprocessing step ensures that any code relying on disallowed packages will fail the test.

Experimental Analysis

Leaderboard of Main Results Our experimental evaluation on ARBench provides critical insights into the current capabilities of LLMs in generating ML algorithms from

Model	Easy		Medium		Hard		All		Overall
	Human	Model	Human	Model	Human	Model	Human	Model	
Proprietary Models									
o4-mini	80.28	94.29	62.50	86.11	29.41	26.15	67.97	65.12	66.33
Gemini-2.5-Pro-preview	81.69	94.29	60.00	80.56	17.65	29.23	66.41	63.95	65.00
o3-mini	83.10	91.43	62.50	76.39	23.53	23.08	68.75	59.30	63.33
o3	77.46	97.14	52.50	81.94	23.53	23.08	62.50	62.79	62.67
Doubao-1.5-Thinking-pro	83.10	91.43	57.50	69.44	11.76	22.58	66.41	63.95	60.57
o1	78.87	88.57	47.50	76.39	23.53	21.54	61.72	58.14	59.67
o1-mini	70.42	94.29	50.00	84.72	11.76	20.00	56.25	62.21	59.67
Claude-3.7-Sonnet	80.28	85.71	60.00	75.00	17.65	12.31	65.62	53.49	58.67
Open-Source Models									
Deepseek-R1	81.69	94.29	47.50	75.00	11.76	23.08	61.72	59.30	60.33
Qwen3-235B-A22B	76.06	82.86	45.00	68.06	23.53	18.46	59.38	52.33	55.33
Deepseek-V3	83.10	82.86	50.00	61.11	23.53	13.85	64.84	47.67	55.00
Qwen3-32B	74.65	88.57	50.00	65.28	23.53	12.31	60.16	50.00	54.33
Llama-4-Maverick	76.06	85.71	37.50	59.72	17.65	6.15	56.25	44.77	49.67
Qwen3-30B-A3B	78.87	80.00	47.50	48.61	17.65	4.62	60.94	38.37	48.00
Llama-3.3-70B-Instruct	73.24	68.57	47.50	47.22	17.65	4.62	57.81	35.47	45.00
Qwen3-14B	71.83	74.29	45.00	41.67	5.88	7.69	54.69	35.47	43.67
Llama-3.1-405B-Instruct	77.46	68.57	40.00	38.89	5.88	4.62	56.25	31.98	42.33
Qwen2.5-Coder-14B-Instruct	74.65	62.86	45.00	36.11	5.88	3.08	56.25	29.07	40.67
Qwen2.5-Coder-1.5B-Instruct	49.30	11.43	12.50	5.56	0.00	0.00	31.25	4.65	16.00
Qwen3-0.6B	22.54	0.00	5.00	1.39	0.00	3.08	14.06	1.74	7.00

Table 2: Leaderboard of top proprietary and open-source LLMs ranked by Pass@1 on ARBench.

scratch, as shown in Table 2. The results reveal that while leading proprietary models, such as o4-mini and Gemini-2.5-Pro, perform strongly overall, a significant performance gap emerges across different task difficulty levels. Specifically, many models achieve high accuracy on Easy tasks, where solutions align with common function usage patterns. However, a marked decrease in performance is observed for Medium and, more strikingly, for Hard tasks. For instance, even the top-performing model on Hard tasks, Gemini-2.5-Pro-preview, only achieved a Pass@1 score of 29.23. This significant drop in performance on more complex tasks, which demand deeper algorithmic reasoning and implementation from scratch, strongly supports our abstract's proposition that current LLMs, despite their proficiency with high-level APIs, struggle with genuine algorithmic reasoning and generalization when direct library support is unavailable. The different benchmark splits, particularly the challenges posed by ARBench-Model, underscore the gap in LLMs' ability to move beyond rote memorization of API usage toward a fundamental understanding of algorithms. These findings highlight the strengths and weaknesses of current LLMs in algorithmic reasoning, providing insights into the significant room for improvement in LLM capabilities for complex algorithmic reasoning, which constitutes a potential direction for future research.

Overall Performance across Models Figure 4 reveals the overall performance of all models on two splits of the ARBench. Overall, the maximum Pass@1 achieved by any model on either split is only approximately 70%. This indicates that the ARBench's difficulty level can challenge current state-of-the-art LLMs.

Performance Difference Between Splits Figure 4 shows that most proprietary and open-source models exhibit lower ARB-Model performance than ARB-Human. We posit that the disparate sources of the two splits are a critical factor for this result. Specifically, ARB-Model's seed data originate from code on GitHub that implements machine learning algorithms. In contrast, ARB-Human data is sourced from problems on popular programming platforms. Some tasks within ARB-Human may not require code integration, allowing LLMs to focus more on algorithm implementation and necessitating less interaction with other ML modules. Consequently, the implementation complexity of ARB-Human tasks may be lower than that of ARB-Model tasks, thereby contributing to the observed performance difference. Nevertheless, when assessing LLMs' capability to generate ML algorithms from scratch, regardless of the degree of code integration required by tasks, LLMs must possess core programming proficiency to accomplish them and to ensure robust performance across diverse scenarios. The results can guide future research.

Comparison across Problem Categories The performance of different models across various categories reveals notable variations in accuracy. As shown in Figure 5, the highest accuracy on the ARB-Human dataset is observed in CV, followed closely by LA and DL. Model performance in ML is generally moderate, while NLP and RL exhibit comparatively lower accuracy. For the ARB-Model dataset, o4-mini, Gemini-2.5, and DeepSeek-R1 perform well in ML and CV. In DL, Gemini-2.5 attains the highest accuracy,

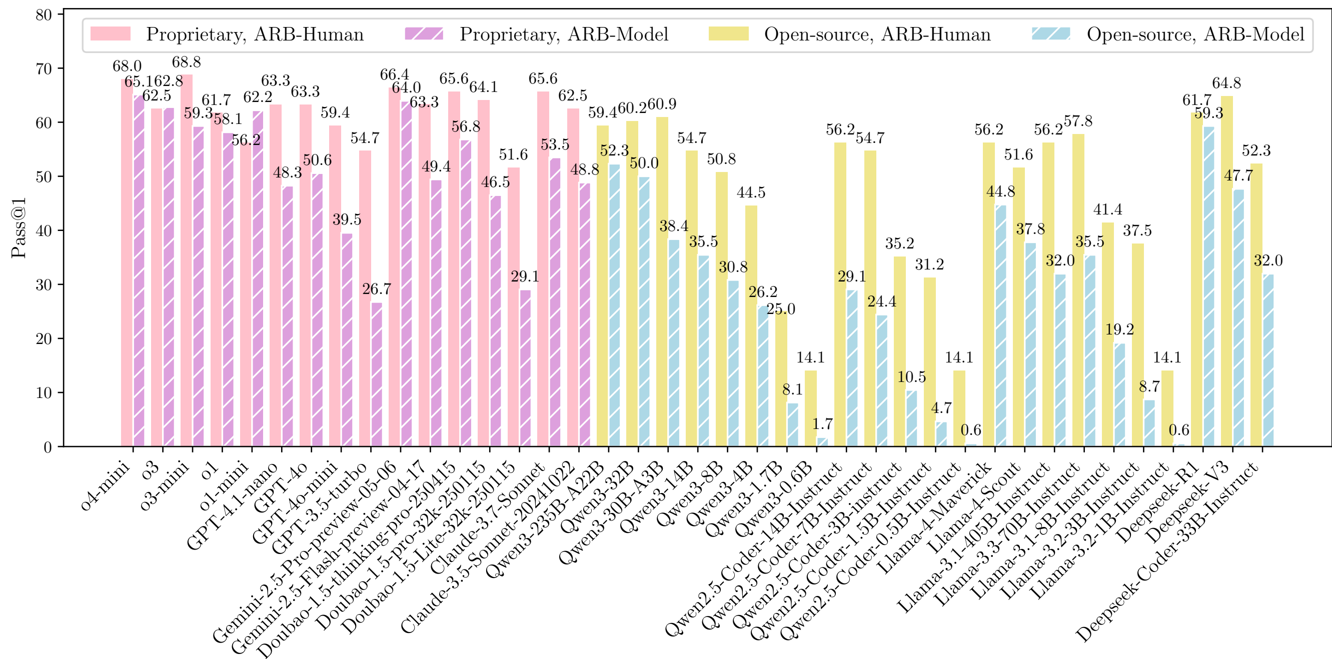


Figure 4: Comparison of Pass@1 scores for SOTA LLMs on ARBench across both splits. Family group models are ordered from left to right by version (newest to oldest) and size (largest to smallest).

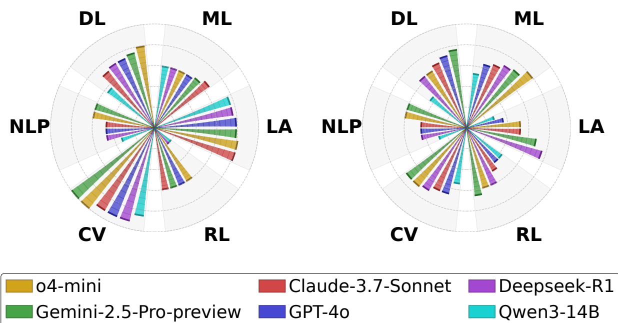


Figure 5: Pass@1 on ARB-Human and ARB-Model across various categories, including Linear Algebra (**LA**), Machine Learning (**ML**), Deep Learning (**DL**), Natural Language Processing (**NLP**), Computer Vision (**CV**), and Reinforcement Learning (**RL**). Each sector represents a domain, with radial length indicating model performance (longer = better). Dashed concentric circles mark reference levels at 20%, 40%, 60%, 80%, and 100%.

followed by GPT-4o. LA presents more variable outcomes, with DeepSeek-R1 achieving a notable accuracy of 70%. NLP and RL continue to pose challenges for current models, with the highest accuracies capped at 50% for NLP and 57.1% for RL, respectively. The benchmark produces consistent and meaningful results across diverse models, confirming its robustness and fairness. In summary, most models perform robustly in CV and ML on both datasets. However, NLP and RL remain comparatively weaker domains. Notably, models such as Gemini-2.5 and DeepSeek-R1 consistently perform strongly across multiple fields, whereas others tend to underperform in specific areas. Furthermore, o4-mini significantly outperforms many other models, while its performance in LA and DL is comparatively weaker.

Failure Case Compared with using high-level APIs, implementing ML algorithms from scratch requires meticulous control over low-level implementation details, such as precise data type management, the design of numerically stable operations, and the construction of correct control-flow logic. LLMs often struggle with these aspects, leading to silent failures that appear plausible but are semantically incorrect or numerically unstable when executed.

Conclusion

LLMs have achieved remarkable success in code generation, yet their reliance on high-level APIs often conceals a lack of genuine algorithmic reasoning. To address this limitation, we introduce ARBench, a novel benchmark specifically designed to evaluate LLMs' ability to implement ML algorithms from scratch. By restricting high-level API/library usage, ARBench shifts the focus from superficial API calls to core algorithmic reasoning. Our benchmark encompasses various ML tasks and provides rich metadata for nuanced evaluation. Empirical results reveal that while current LLMs excel when allowed to use high-level APIs, they struggle significantly when required to implement algorithms from scratch. These findings highlight the promise and current limitations of LLMs in generalizing to unseen or domain-specific ML challenges. ARBench will serve as a valuable evaluation tool for fostering the development of LLMs.

Acknowledgments

This paper is supported by the Major Program (JD) of Hubei Province (2023BAA024) and NSFC (62076121). The authors thank Dr. Wang-Zhou Dai and Hao-Yuan He for their insightful discussions regarding this work.

References

Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. 2023. Gpt-4 technical report. arXiv:2303.08774.

Anthropic. 2024. Claude 3.5 Sonnet.

Askell, A.; Bai, Y.; Chen, A.; Drain, D.; Ganguli, D.; Henighan, T.; Jones, A.; Joseph, N.; Mann, B.; DasSarma, N.; et al. 2021. A general language assistant as a laboratory for alignment. arXiv:2112.00861.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732.

Bai, J.; Bai, S.; Chu, Y.; Cui, Z.; Dang, K.; Deng, X.; Fan, Y.; Ge, W.; Han, Y.; Huang, F.; et al. 2023. Qwen technical report. arXiv:2309.16609.

Bai, Y.; Jones, A.; Ndousse, K.; Askell, A.; Chen, A.; DasSarma, N.; Drain, D.; Fort, S.; Ganguli, D.; Henighan, T.; et al. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. arXiv:2204.05862.

Bavarian, M.; Jun, H.; Tezak, N.; Schulman, J.; McLeavey, C.; Tworek, J.; and Chen, M. 2022. Efficient training of language models to fill in the middle. arXiv:2207.14255.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. In *Proc. of NeurIPS*, 1877–1901.

ByteDance. 2025. Doubao 1.5 Pro: Official Introduction.

Cassano, F.; Gouwar, J.; Nguyen, D.; Nguyen, S.; Phipps-Costin, L.; Pinckney, D.; Yee, M.-H.; Zi, Y.; Anderson, C. J.; Feldman, M. Q.; Guha, A.; Greenberg, M.; and Jangda, A. 2022. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. arXiv:2208.08227.

Chaudhary; and Sahil. 2023. Code alpaca: An instruction-following llama model for code generation.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Girish; et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

Dou, S.; Jia, H.; Wu, S.; Zheng, H.; Zhou, W.; Wu, M.; Chai, M.; Fan, J.; Huang, C.; Tao, Y.; et al. 2026. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. *SCIENCE CHINA Information Sciences*, 69(1): 112107–.

Gong, L.; Elhoushi, M.; and Cheung, A. 2024. AST-T5: Structure-Aware Pretraining for Code Generation and Understanding. In *Proc. of ICML*.

Grattafiori, A.; Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Vaughan, A.; et al. 2024. The llama 3 herd of models. arXiv:2407.21783.

Guo, D.; Yang, D.; Zhang, H.; Song, J.; Zhang, R.; Xu, R.; Zhu, Q.; Ma, S.; Wang, P.; Bi, X.; et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. arXiv:2401.14196.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; and Steinhardt, J. 2021. Measuring Coding Challenge Competence With APPS. In *Proc. of NeurIPS*.

Huang, S.; Cheng, T.; Liu, J. K.; Xu, W.; Hao, J.; Song, L.; Xu, Y.; Yang, J.; Liu, J.; Zhang, C.; et al. 2025. Opencoder: The open cookbook for top-tier code large language models. In *Proc. of ACL*.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; et al. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186.

Hurst, A.; Lerer, A.; Goucher, A. P.; Perelman, A.; Ramesh, A.; Clark, A.; Ostrow, A.; Welihinda, A.; Hayes, A.; Radford, A.; et al. 2024. GPT-4o System Card. arXiv:2410.21276.

Jain, N.; Han, K.; Gu, A.; Li, W.-D.; Yan, F.; Zhang, T.; Wang, S.; Solar-Lezama, A.; Sen, K.; and Stoica, I. 2025. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *Proc. of ICLR*.

Jiang; Juyong; Wang; Fan; Shen; Jiasi; Kim; Sungju; Kim; and Sunghun. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515.

Lai, Y.; Li, C.; Wang, Y.; Zhang, T.; Zhong, R.; Zettlemoyer, L.; Yih, W.-t.; Fried, D.; Wang, S.; and Yu, T. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *Proc. of ICML*, 18319–18345.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. 2023a. Starcoder: may the source be with you! arXiv:2305.06161.

Li, R.; Fu, J.; Zhang, B.-W.; Huang, T.; Sun, Z.; Lyu, C.; Liu, G.; Jin, Z.; and Li, G. 2023b. TACO: Topics in Algorithmic COde generation dataset. arXiv:2312.14852.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 1092–1097.

Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. 2025a. DeepSeek-V3 Technical Report. arXiv:2412.19437.

Liu, J.; Xia, C. S.; Wang, Y.; and Zhang, L. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Proc. of NeurIPS*.

Liu, R.-B.; Li, A.; Yang, C.; Sun, H.; and Li, M. 2025b. Revisiting Chain-of-Thought in Code Generation: Do Language Models Need to Learn Reasoning before Coding? In *Proc. of ICML*, 38809–38826.

Lozhkov, A.; Li, R.; Allal, L. B.; Cassano, F.; Lamy-Poirier, J.; Tazi, N.; Tang, A.; Pykhtar, D.; Liu, J.; Wei, Y.; et al. 2024. Starcoder 2 and the stack v2: The next generation. arXiv:2402.19173.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568.

Ma, Z.; An, S.; Xie, B.; and Lin, Z. 2024. Compositional API recommendation for library-oriented code generation. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 87–98.

Meta. 2025. Llama 4: Model Card and Prompt Format.

Panickssery; Arjun; Bowman; Samuel; Feng; and Shi. 2024. Llm evaluators recognize and favor their own generations. In *Proc. of NeurIPS*, 68772–68802.

Peng, B.; Li, C.; He, P.; Galley, M.; and Gao, J. 2023. Instruction tuning with gpt-4. arXiv:2304.03277.

Quan, S.; Yang, J.; Yu, B.; Zheng, B.; Liu, D.; Yang, A.; Ren, X.; Gao, B.; Miao, Y.; Feng, Y.; et al. 2025. CodeElo: Benchmarking Competition-level Code Generation of LLMs with Human-comparable Elo Ratings. arXiv:2501.01257.

Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I.; et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 9.

Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. 2023. Code llama: Open foundation models for code. arXiv:2308.12950.

Tang, X.; Liu, Y.; Cai, Z.; Shao, Y.; Lu, J.; Zhang, Y.; Deng, Z.; Hu, H.; An, K.; Huang, R.; et al. 2023. ML-Bench: Evaluating Large Language Models and Agents for Machine Learning Tasks on Repository-Level Code. arXiv:2311.09835.

Team, G.; Anil, R.; Borgeaud, S.; Alayrac, J.-B.; Yu, J.; Soricut, R.; Schalkwyk, J.; Dai, A. M.; Hauth, A.; Millican, K.; et al. 2023. Gemini: a family of highly capable multimodal models. arXiv:2312.11805.

Wang, Y.; Kordi, Y.; Mishra, S.; Liu, A.; Smith, N. A.; Khashabi, D.; and Hajishirzi, H. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proc. of ACL*, 13484–13508.

Wang, Z.; Zhou, S.; Fried, D.; and Neubig, G. 2022. Execution-Based Evaluation for Open-Domain Code Generation. arXiv:2212.10481.

Wei, Y.; Wang, Z.; Liu, J.; Ding, Y.; and Zhang, L. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. In *Proc. of ICML*.

Xia, Y.; Shen, W.; Wang, Y.; Liu, J. K.; Sun, H.; Wu, S.; Hu, J.; and Xu, X. 2025. LeetCodeDataset: A Temporal Dataset for Robust Evaluation and Efficient Training of Code LLMs. arXiv:2504.14655.

Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Gao, C.; Huang, C.; Lv, C.; et al. 2025. Qwen3 Technical Report. arXiv:2505.09388.

Ye, J.; Chen, X.; Xu, N.; Zu, C.; Shao, Z.; Liu, S.; Cui, Y.; Zhou, Z.; Gong, C.; Shen, Y.; Zhou, J.; Chen, S.; Gui, T.; Zhang, Q.; and Huang, X. 2023. A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models. arXiv:2303.10420.

Zhao, W. X.; Zhou, K.; Li, J.; Tang, T.; Wang, X.; Hou, Y.; Min, Y.; Zhang, B.; Zhang, J.; Dong, Z.; et al. 2023. A survey of large language models. arXiv:2303.18223.

Zheng, L.; Chiang, W.-L.; Sheng, Y.; Zhuang, S.; Wu, Z.; Zhuang, Y.; Lin, Z.; Li, Z.; Li, D.; Xing, E.; et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Proc. of NeurIPS*, 46595–46623.

Zheng, T.; Zhang, G.; Shen, T.; Liu, X.; Lin, B. Y.; Fu, J.; Chen, W.; and Yue, X. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. arXiv:2402.14658.

Zhuo, T. Y.; Vu, M. C.; Chim, J.; Hu, H.; Yu, W.; Widyasari, R.; Yusuf, I. N. B.; Zhan, H.; He, J.; Paul, I.; et al. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *Proc. of ICLR*.