

---

# CodeMixBench: Evaluating Large Language Models on Code Generation with Code-Mixed Prompts

---

**Manik Sheokand**

Department of Computer Science  
Chandigarh University  
Punjab, India  
21bcs11856@cuchd.in

**Parth Sawant**

Department of Technology Management and Innovation  
New York University  
New York, USA  
ps5624@nyu.edu

## Abstract

Large Language Models (LLMs) have achieved remarkable success in code generation tasks, powering various applications like code completion, debugging, and programming assistance. However, existing benchmarks such as HumanEval, MBPP, and BigCodeBench primarily evaluate LLMs on English-only prompts, overlooking the real-world scenario where multilingual developers often use code-mixed language while interacting with LLMs. To address this gap, we introduce *CodeMixBench*, a novel benchmark designed to evaluate the robustness of LLMs on code generation from code-mixed prompts. Built upon BigCodeBench, CodeMixBench introduces controlled code-mixing (CMD) into the natural language parts of prompts across three language pairs: Hinglish (Hindi-English), Spanish-English, and Chinese Pinyin-English. We comprehensively evaluate a diverse set of open-source code generation models ranging from 1.5B to 15B parameters. Our results show that code-mixed prompts consistently degrade Pass@1 performance compared to their English-only counterparts, with performance drops increasing under higher CMD levels for smaller models. CodeMixBench provides a realistic evaluation framework for studying multilingual code generation and highlights new challenges and directions for building robust code generation models that generalize well across diverse linguistic settings.

## 1 Introduction

Large Language Models (LLMs) have revolutionized code generation Bielik et al. (2016); Yin and Neubig (2017), enabling tasks such as automated synthesis, completion, bug fixing, and summarization. Models like Codex Chen et al. (2021), AlphaCode Li et al. (2022), StarCoder Li et al. (2023), and DeepSeek-Coder Guo et al. (2024) have achieved strong results by training on large-scale code datasets. However, these advancements typically assume monolingual (English-only) interactions between developers and LLMs—an assumption that does not reflect the reality of global software development.

In practice, developers frequently blend their native language with English when writing comments, docstrings, or giving natural language instructions—a phenomenon known as code-mixing or code-switching Myers-Scotton (1993); Solorio and Liu (2008); Bali et al. (2014). This behavior is common in multilingual communities, where using native expressions alongside English code constructs is both natural and intuitive. Despite the increasing adoption of LLMs, their ability to handle such code-mixed prompts remains underexplored.

Existing code generation benchmarks such as HumanEval Chen et al. (2021), MBPP Austin et al. (2021), APPS Hendrycks et al. (2021), and BigCodeBench Zhuo et al. (2025) evaluate models using only English prompts, failing to account for the linguistic diversity found in real-world programming.

To address this gap, we introduce *CodeMixBench*, a benchmark designed to evaluate LLMs on code generation tasks where prompts are written in a code-mixed format combining English with another language. Built on top of BigCodeBench, CodeMixBench utilizes the concept of controllable code-mixing degree (CMD) introduced by Gupta et al. (2024) spanning English-Hindi (Hinglish), Spanish-English, and Chinese Pinyin-English. We employ an LLM-driven augmentation pipeline using Gemini-2.0-Flash-Lite for translation and Controlled Generation (CG) Gupta et al. (2024) to modulate code-mixing while preserving task semantics.

We evaluate a diverse set of open-source code generation models (1.5B–15B parameters) across CMD levels and languages. Our results show that code-mixing consistently degrades performance, especially at high CMD values; larger and instruction-tuned models like OpenCoder-8B-Instruct Huang et al. (2024) and DeepSeek-Coder Guo et al. (2024) are more robust, while smaller models suffer significantly. CodeMixBench thus provides a new lens into the multilingual generalization capabilities of code LLMs and highlights the importance of evaluating model robustness beyond monolingual assumptions.

## 2 Related Work

### 2.1 Code Generation Benchmarks

Several benchmarks have been proposed to evaluate the code generation capabilities of LLMs, primarily using English-only prompts. HumanEval Chen et al. (2021) introduced a set of hand-crafted Python tasks focused on functional correctness, while MBPP Austin et al. (2021) extended this with 974 entry-level tasks accompanied by unit tests. APPS Hendrycks et al. (2021) contributed a large-scale benchmark with 10,000 problems from competitive programming platforms, and xCodeEval Khan et al. (2024) emphasized multilingual executable code across 11 programming languages.

BigCodeBench Zhuo et al. (2025) builds on these efforts with 1,140 realistic tasks requiring function calls from 139 Python libraries across diverse domains. It emphasizes tool use and compositional reasoning, requiring multi-step synthesis and validation via high-coverage unit tests. However, like its predecessors, BigCodeBench assumes monolingual (English-only) instructions and does not address LLM robustness to multilingual or code-mixed prompts.

### 2.2 Multilingual Code Models and Code-Mixing

Recent LLMs such as CodeLLaMA Rozière et al. (2024), StarCoder2 Li et al. (2023), and DeepSeek-Coder Guo et al. (2024) have introduced multilingual support for code understanding and generation. These models are typically trained on datasets containing multiple programming languages and are evaluated on tasks such as completion and translation. However, their focus remains on code syntax diversity rather than handling code-mixed natural language instructions embedded within prompts.

In parallel, code-mixing has been extensively studied in NLP for tasks such as text classification Chaturanga et al. (2021), translation Vavre et al. (2022), and sentiment analysis Perera et al. (2024). Data augmentation techniques such as controlled generation Gupta et al. (2024), and semantic evaluation methods like Gold-standard Agnostic Measure for Evaluation (GAME) Gupta et al. (2024) and Metric-Independent Pipeline for Evaluation (MIPE) Garg et al. (2021), have advanced the quality of code-mixed datasets and their evaluation. However, code-mixing in the context of natural language-to-code generation remains underexplored, particularly in relation to preserving task semantics and ensuring executable correctness.

### 2.3 Evaluation Metrics for Code Generation

Pass@ $k$  Chen et al. (2021) has become the standard metric for evaluating the functional correctness of generated code, measuring the likelihood that at least one of the top- $k$  outputs passes all associated test cases. While suitable for execution-based validation, it does not capture whether code-mixed prompts retain their intended meaning. Metrics like BLEU Papineni et al. (2002) are poorly suited for this purpose due to lexical variability and the absence of gold-standard references in code-mixed text. To address this, we adopt the GAME score, an embedding-based metric designed for code-mixed

evaluation. GAME compares sentence embeddings of back-translated prompts to the original using cosine similarity, providing a semantic fidelity score on a 0–100 scale.

Our work is the first to integrate these threads — multilingual NLP, code generation benchmarks, and semantic evaluation — into a unified benchmark that systematically evaluates code generation LLMs under code-mixed prompt conditions.

### 3 Dataset: CodeMixBench

CodeMixBench is constructed as an augmentation of BigCodeBench Zhuo et al. (2025), a benchmark for complex function-level code generation. BigCodeBench comprises two splits: *complete split*, containing full docstrings for function synthesis, and *instruct split*, containing concise task instructions for instruction-tuned models. We retain both splits and introduce multilingual variations by modifying only the natural language components—prompt instructions and docstrings—while preserving the executable code structure. Table 7 (see Appendix 8.7) shows examples of each.

#### 3.1 Target Languages and Code-Mixing Scope

We support three realistic code-mixing configurations based on prevalence in multilingual developer communities: Hinglish (Hindi-English), Spanish-English, and Chinese Pinyin-English. Table 1 summarizes the rationale for their inclusion. Code-mixing is applied only to the prompt instructions and docstrings, leaving code unchanged.

Table 1: Target language mixes and motivation for inclusion in CodeMixBench.

Language mix	Rationale
Hinglish (Hindi-English)	Common in South Asia, especially India; widely observed in informal and technical communication.
Spanish-English	Prevalent in Latin America and the US; large bilingual developer base.
Chinese Pinyin-English	Reflects common use in East Asia and among Chinese developers using romanized Chinese in prompts.

#### 3.2 Code-Mixing Methodology

We adopt a two-stage augmentation strategy: (1) translation to a matrix language (e.g., Hindi) using Gemini-2.0-Flash-Lite while preserving programming tokens, and (2) controlled code-mixing using a  $CMD \in [0, 1]$  to determine the proportion of embedded English tokens retained. For example,  $CMD = 0.6$  retains 60% of switchable words in English, while  $CMD = 0.9$  retains more. Details of this procedure are shown in the example prompt in Table 5 in Appendix 8.2.

##### 3.2.1 Implementation Pipeline for CodeMixBench

The complete pipeline used to construct CodeMixBench is composed of the following four components:

**1. Base Translation and Word Dictionary Construction** We begin by prompting *Gemini-2.0-Flash-Lite* to translate both the `instruct_prompt` and the doc string(`doc_struct`) into the target matrix language (e.g., Hindi), while preserving programming-specific tokens such as `args` and `list`. To identify potential switch points for controlled code-mixing, we apply Part-of-Speech (PoS) tagging Toutanova et al. (2003) to the original English sentence using spaCy. This enables us to extract a filtered list of content words—typically nouns, adjectives, and verbs—that are syntactically appropriate for substitution. Using the same LLM prompt, Gemini is then asked to identify the corresponding words in the translated matrix-language sentence for each selected English token. The model returns a bilingual dictionary of aligned word pairs:

$$W = \{(w_i^{\text{eng}}, w_i^{\text{mtx}})\}_{i=1}^n$$

where  $w_i^{\text{eng}}$  is an English word and  $w_i^{\text{mtx}}$  is its matrix-language equivalent. Additionally, for each  $w_i^{\text{mtx}}$ , Gemini provides three romanized transliterations to capture spelling variability found in real-world code-mixed usage:

$$R_i = \{\text{roman}_i^{(1)}, \text{roman}_i^{(2)}, \text{roman}_i^{(3)}\}.$$

This romanization is necessary to simulate informal orthographic patterns prevalent in platforms like Twitter, where Hinglish is often written with inconsistent spellings. As a result, this translation step produces two key artifacts: (1) a fully translated matrix-language version of the prompt with code-related tokens intact, and (2) a structured dictionary mapping

English → Matrix Language → Romanized Forms,

**2. Controlled Code-Mix Injection** To simulate realistic patterns of code-mixing, we apply a replacement strategy based on lexical trends observed in real-world code-mixed text. Specifically, we use a code-mixed Twitter corpus Nayak and Joshi (2022) to estimate the frequency of both English words and their Romanized Hindi variants.

For each replaceable word  $w_i$ , we calculate a replacement score  $s_i$  that captures how likely it is for the Hindi variant to be substituted back into English. This score is defined as:

$$s_i = \frac{f(\text{eng}_i)}{f(\text{hi}_i)}, \quad f(\text{hi}_i) = \sum_{j=1}^3 f(\text{roman}_i^{(j)})$$

where  $f(\cdot)$  denotes frequency in the corpus, and the denominator aggregates the frequencies of up to three Romanized variations for the Hindi translation of  $w_i$ . A higher score implies that the English form is more dominant or natural in real-world code-mixed usage. If  $f(\text{hi}_i) = 0$ , we assign  $s_i = \infty$ , indicating a preference to retain the English word due to the rarity or unnaturalness of its Hindi form.

Given a specified CMD  $\in [0, 1]$ , we calculate the number of words to be replaced using  $\text{floor}(N \cdot \text{CMD})$ , where  $N$  is the number of eligible switch points—i.e., content words (typically nouns, verbs, and adjectives) identified via PoS tagging in the original English sentence. We then sort the candidate words by their score  $s_i$  in descending order and select the top words for replacement. Words with infinite scores are always prioritized and replaced first, even if they exceed the quota implied by the CMD value.

For example, with CMD = 0.7 and three replaceable words, we replace only two of them.

$$3 \times 0.7 \approx 2$$

**3. Romanization** After injecting the desired level of code-mixing, we apply a Romanization step to convert any remaining Hindi tokens into the Roman script. This is accomplished via a follow-up prompt to the same LLM, which is instructed to romanize only the Hindi words while leaving programming-specific tokens, punctuation, and English subwords untouched. This selective Romanization ensures that the resulting code-mixed prompts remain structurally intact and syntactically parseable, while being fully compatible with LLMs trained predominantly on Roman script input.

**4. Docstring Replacement for Complete Prompts** For prompts in the complete split, we apply the same translation and controlled code-mixing to the function-level docstring. The original English docstring is first translated into the matrix language and then processed using PoS tagging, frequency-based scoring, CMD-controlled replacement, and Romanization — exactly as described in earlier steps. The final code-mixed and romanized version is returned as a structured dictionary and injected back into the source code, replacing the default English docstring.

### 3.3 Semantic Quality Verification: GAME Score

To evaluate whether code-mixed prompts preserve semantic intent, we use the **GAME** score. Each prompt is back-translated to English using Gemini-2.0-Flash-Lite, and compared to the original using cosine similarity of sentence embeddings (via all-MiniLM-L6-v2). Cosine similarity is clipped to  $[0, 1]$  and scaled to a final GAME score in  $[0, 100]$ . Across CMD levels 0.6 and 0.9, we observe a mean GAME score of **90%**, confirming high semantic fidelity in the generated prompts.

### 3.4 Dataset Statistics

CodeMixBench includes 1,140 unique tasks, each augmented across three language mixes and two CMD levels (0.6, 0.9), using both `instruct` and `complete` variants. This results in 6,840 total prompts (3 languages  $\times$  2 CMDs  $\times$  1140). Table 2 summarizes the distribution.

Table 2: Codemixbench example counts across languages and CMD levels.

Language mix	CMD = 0.6	CMD = 0.9
Hinglish	1140	1140
Spanish-English	1140	1140
Pinyin-English	1140	1140
<b>Total</b>	3420	3420

Detailed base translation, romanization and all are giving under Appendix 8.5 and 8.4. Code-mix algorithm is given under Appendix 8.3. All data, preprocessing scripts, and evaluation tools will be released on Hugging Face and GitHub under a CC-BY 4.0 license upon acceptance.

## 4 Experimental Setup

All experiments were conducted using a modified version of the official BigCodeBench harness. Code was executed inside isolated E2B sandbox containers (Python 3.10, preinstalled libraries). Inference ran on a cloud instance hosted by Lightning AI with 16 vCPUs, 128 GB RAM, and a single NVIDIA L40s GPU (48 GB VRAM). Code execution was CPU-based. Full evaluation per model on our dataset completed in under one hour.

### 4.1 Models Evaluated

We evaluate 17 open-source code generation models ranging from 1B to 15B parameters, including instruction-tuned, distilled, and multilingual variants. These span multiple architectures and training objectives (e.g., Qwen2.5, DeepSeek, StarCoder2, OpenCoder, Phi-4). See Appendix 8.1 (Table 4) for the full list of models and parameter sizes.

All models were evaluated in a zero-shot setting using standardized prompts. For `complete` tasks, we supplied full docstrings; for `instruct` tasks, concise natural instructions were used. We adopted the default prompting technique provided by the BigCodeBench library to ensure consistency with prior evaluation protocols. Prompt examples across CMD values are provided in Appendix 8.6 (Table 6).

### 4.2 Evaluation Metric: Pass@1

We adopt the standard **Pass@1** metric Chen et al. (2021), which checks whether the top-1 generated solution passes all associated test cases. Each model-task pair uses greedy decoding to produce a single code snippet, which is then executed securely in the E2B sandbox and evaluated against BigCodeBench’s test suite.

## 5 Results and Analysis

We report Pass@1 performance across English-only prompts (baseline), and code-mixed prompts at CMD = 0.6 (light mixing) and CMD = 0.9 (heavy mixing). Models are evaluated on the `complete` split of CodeMixBench, with code-mixed prompts evaluated exclusively in the Hindi-English code-mix subset.

### 5.1 Overall Pass@1 Performance

Code-mixed prompts consistently reduce Pass@1 performance, with more severe degradation observed at CMD = 0.9. Larger, instruction-tuned models like OpenCoder-8B-Instruct and Phi-4 maintain strong performance across settings, while smaller models (e.g., LLaMA-3.2-1B, StarCoder2-3B) degrade sharply under heavy mixing.

Table 3: Pass@1 performance on the complete split across English and code-mixed prompts.

Model	English (%)	CMD = 0.6 (%)	CMD = 0.9 (%)
DeepSeek-R1-Distill-Qwen-1.5B	7.9	4.1	4.8
StarCoder2-3b	21.4	9.3	9.1
Llama-3.2-1B	11.3	5	4.5
Qwen2.5-Coder-1.5B-Instruct	32.7	31.8	22.2
Gemma-3-4b-it	37.8	28.2	28.4
OpenCoder-8B-Instruct	50.9	51.3	39.0
Phi-4-multimodal-instruct	46.5	46.4	33.1
DeepSeek-R1-Distill-Llama-8B	15.3	15.9	11.1
Hermes-2-Theta-Llama-3-8B	36.4	36.4	29.1
CodeLlama-7b-Instruct-hf	25.0	17.7	17.7
Qwen2.5-Coder-7B-Instruct	48.8	41.2	41.3
Llama-3.1-8B-Instruct	40.5	38.8	31.4
StarCoder2-7b	27.7	6.8	8.9
Phi-4	55.4	46.7	47.1
DeepSeek-R1-Distill-Qwen-14B	48.4	38.8	40.3
DeepSeek-Coder-V2-Lite-Instruct	47.6	37.7	38.1
StarCoder2-15b-instruct-v0.1	45.1	33.1	32.7

## 5.2 Analysis of CMD Level, Language, and Model Size

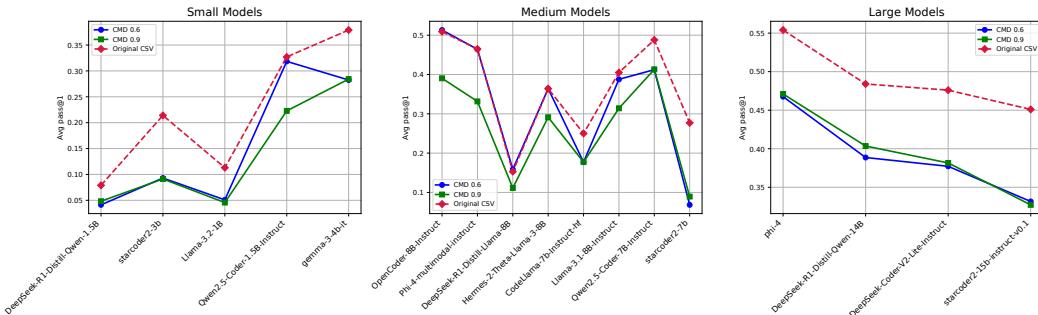


Figure 1: Comparison of CMD 0.6, 0.9 and original English prompt values for pass@1

Despite the general downward trend, a few models stand out for their robustness to increasing CMD. In particular, OpenCoder-8B-Instruct, DeepSeek-R1-Distill-Llama-8B, and Qwen2.5-Coder-1.5B-Instruct retain near-baseline performance even under heavy code-mixing. OpenCoder-8B Huang et al. (2024) exhibits virtually no loss at CMD 0.6 (51.3% Pass@1 vs. 50.9% on English). This model’s resilience can be attributed to its training on a massive and diverse dataset (2.5 trillion tokens, 90% code and 10% code-related web text) across languages, combined with strong instruction-tuning. Indeed, the *CodeMixBench* findings note that OpenCoder-8B’s “nearly identical performance on English and code-mixed prompts” likely due to its multilingual, noisy pretraining and robust fine-tuning regimen.

Similarly, DeepSeek-R1-Distill-Llama-8B DeepSeek-AI et al. (2025) shows only minor degradation as CMD increases. This 8B model maintains high Pass@1 across CMD levels in the medium sized model bracket. Its robustness is shown in the *DeepSeek-Coder* training strategy and distillation technique. The DeepSeek models was trained from scratch on 2 trillion tokens of data comprising 87% code and 13% code-mixed natural language. The R1-Distill Llama-8B model in particular was obtained by knowledge distillation from a larger DeepSeek model onto a Llama-based 8B architecture.

Qwen2.5-Coder-1.5B-Instruct is a particularly noteworthy case of a smaller model remaining robust to code-mixing. Despite having only 1.5B parameters, Qwen2.5-Coder shows only a modest drop in Pass@1 when moving to CMD 0.9, especially compared to other models in the 1–3B range

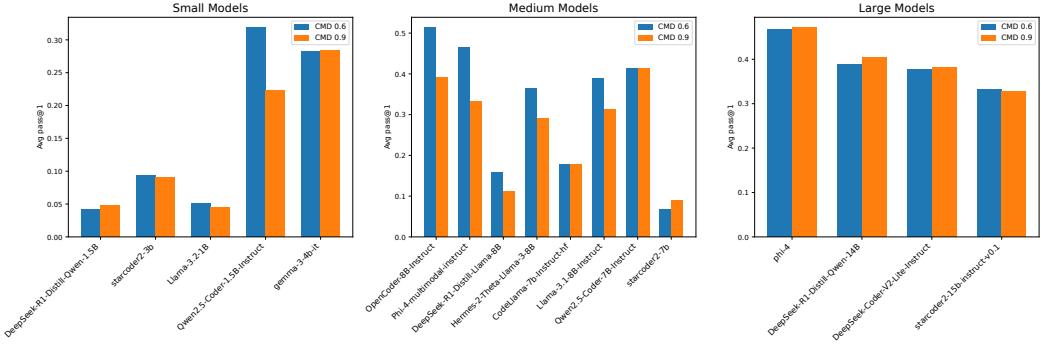


Figure 2: Comparison of CMD 0.6, 0.9 and model size for pass@1

(which often collapse under heavy mixing). We attribute Qwen2.5’s strong performance to its highly effective training. It is built on Alibaba’s *Qwen2.5* Hui et al. (2024) architecture and was continued-pretrained on an enormous 5.5 trillion token code corpus. This scale of training far exceeds that of most models in its size class.

## 6 Discussion

### 6.1 Key Findings and Takeaways

Our study presents the first systematic benchmark evaluating the robustness of large language models (LLMs) to code-mixed prompts in code generation. Through comprehensive experimentation across multiple models, languages, and levels of code-mixing, we identify several key takeaways. First, code-mixed prompts consistently degrade Pass@1 performance, with higher CMD levels (e.g., CMD = 0.9) producing the most severe drops. Model size and instruction tuning significantly influence robustness, with larger models (7B+) demonstrating greater resilience than smaller models (1B–3B), and instruction-tuned variants consistently outperforming their base counterparts. Notably, OpenCoder-8B-Instruct emerged as exceptionally robust, with minimal performance degradation across all code-mixed settings—highlighting the impact of multilingual and noisy pretraining. Furthermore, our results indicate that models trained on large-scale, diverse corpora—particularly those containing noisy and multilingual natural language—consistently perform better under code-mixed prompts. For instance, Qwen2.5-Coder-1.5B and DeepSeek-R1-Distill-Llama-8B both exhibit strong Pass@1 accuracy despite their smaller or distilled sizes, owing to the breadth and diversity of their training datasets.

### 6.2 Why Does Performance Drop on Code-Mixing?

We hypothesize that several factors contribute to performance degradation under code-mixed conditions. A primary reason is training data bias: most LLMs are trained on English-dominant code datasets, with minimal exposure to multilingual or code-mixed content. Tokenizer limitations also play a role—non-English tokens, especially Romanized text, inflate sequence lengths and lead to fragmented tokenization. Additionally, semantic ambiguity in code-mixed prompts can make it harder for models to ground instructions, and frequent mixing breaks the language modeling priors on which these models rely.

### 6.3 Practical Implications

These findings have practical implications for LLM deployment in multilingual developer environments. Without explicit multilingual or code-mixed pretraining, LLM-based assistants may underperform for non-English or mixed-language users. Instruction tuning alone appears insufficient; instead, dataset diversification with noisy or code-mixed content is necessary to build more inclusive and robust systems. Developers and LLM providers alike should consider refining tokenizer strategies and incorporating multilingual data to better support global usage patterns.

## 6.4 Limitations of Our Study

Despite its contributions, CodeMixBench has several limitations. It currently focuses on three language pairs—Hinglish, Spanish-English, and Chinese-English—leaving many others unexplored. Our code-mixing is applied only to natural language instructions and docstrings, whereas real-world mixing often occurs in comments, variable names, and inline annotations. Furthermore, we focus solely on functional correctness (Pass@1), without evaluating code quality, readability, or adherence to stylistic conventions. Additionally, due to computational and resource constraints, we were limited in the number and variety of models evaluated across multiple code-mixing degrees and language sets. As a result, several models and code-mixing scenarios remain unexplored in this study.

## 6.5 Future Work Directions

Future research can extend CodeMixBench in several directions. Expanding to additional language pairs such as Tamil-English, Bengali-English, and Korean-English would offer broader multilingual coverage. Enhancing realism by introducing code-mixing in comments, variable names, and inline explanations would further reflect developer behavior. Constructing human-authored code-mixed datasets would help eliminate translation artifacts and improve benchmark authenticity. Future work can also explore multilingual-aware tokenizers optimized for Romanized non-English words, and assess the impact of fine-tuning or instruction-tuning on code-mixed generation. Finally, incorporating new metrics—such as code quality, runtime efficiency, and maintainability—would provide a more comprehensive evaluation of model performance in multilingual coding environments.

## 7 Conclusion

In this work, we introduced **CodeMixBench**, a novel benchmark for evaluating code generation capabilities of large language models (LLMs) under realistic code-mixed prompting scenarios. Built as a controlled augmentation of BigCodeBench, CodeMixBench incorporates graded code-mixing (CMD) across three high-impact multilingual pairs—Hinglish (Hindi-English), Spanish-English, and Chinese Pinyin-English—while ensuring the preservation of functional correctness and task semantics.

Our evaluation of 17 open-source models, spanning parameter sizes from 1.5B to 15B, reveals a consistent degradation in Pass@1 performance with increasing code-mixing, particularly at higher CMD levels (e.g., CMD = 0.9). We find that smaller and base models are disproportionately affected, often failing to generalize in heavily code-mixed settings. In contrast, instruction-tuned and larger models demonstrate greater robustness, though not without limitations. Crucially, we observe that model resilience is not solely a function of scale—models such as Qwen2.5-Coder-1.5B and DeepSeek-R1-Distill-Llama-8B outperform larger counterparts due to their training on diverse, multilingual, and noisy corpora.

CodeMixBench serves as a critical step toward closing the gap between monolingual code benchmarks and the real-world multilingual interactions of software developers. As global software engineering increasingly incorporates code-mixed communication, our benchmark provides the first rigorous framework to assess and improve LLMs under these settings.

All datasets, augmentation scripts, and evaluation pipelines will be released publicly to ensure transparency, reproducibility, and continued progress in this emerging area of multilingual code intelligence.

## References

- Jacob Austin, Augustus Odena, Maxwell Nye, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Kalika Bali, Jatin Sharma, Monojit Choudhury, and Yogarshi Vyas. "i am borrowing ya mixing?" an analysis of english-hindi code-mixing in facebook. In *Proceedings of the First Workshop on Computational Approaches to Code Switching*, pages 116–126, 2014.
- Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: Probabilistic model for code generation. In *International Conference on Machine Learning (ICML)*, pages 2933 – 2942, 2016.

- Shanaka Chaturanga, Surangika Ranathunga, et al. Classification of code-mixed text using capsule networks. *Proceedings of Recent Advances in Natural Language Processing*, page 256–263, 2021.
- Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- DeepSeek-AI, Daya Guo, Dejian Yang, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL <https://arxiv.org/abs/2501.12948>.
- Ayush Garg, Sammed Kagi, Vivek Srivastava, et al. Mipe: A metric independent pipeline for effective code-mixed nlg evaluation. *Proceedings of the 2nd Workshop on Evaluation and Comparison of NLP Systems*, 2021.
- Daya Guo, Qihao Zhu, Dejian Yang, et al. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
- Ayushman Gupta, Akhil Bhoga, Kripabandhu Ghosh, et al. Multilingual controlled generation and gold-standard-agnostic evaluation of code-mixed sentences. *arXiv preprint arXiv:2410.10580*, 2024.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, et al. Measuring coding competence with apps. *Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2021.
- Siming Huang, Tianhao Cheng, Jason Liu, et al. Opencoder: The open cookbook for top-tier code large language models. 11 2024. doi: 10.48550/arXiv.2411.04905.
- Bin yuan Hui, Jian Yang, Zeyu Cui, et al. Qwen2.5-coder technical report, 2024. URL <https://arxiv.org/abs/2409.12186>.
- Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, et al. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.
- Raymond Li, Loubna Ben Allal, Niklas Muennighoff, et al. Starcoder: May the source be with you! *Transactions on Machine Learning Research*, 2023.
- Yuju Li, David Choi, Junyoung Chung, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.
- Carol Myers-Scotton. *Social Motivations for Codeswitching: Evidence from Africa*. Oxford University Press, 1993.
- Ravindra Nayak and Raviraj Joshi. L3Cube-HingCorpus and HingBERT: A code mixed Hindi-English dataset and BERT language models. In Girish Nath Jha, Sobha L., Kalika Bali, and Atul Kr. Ojha, editors, *Proceedings of the WILDRE-6 Workshop within the 13th Language Resources and Evaluation Conference*, pages 7–12, Marseille, France, June 2022. European Language Resources Association. URL <https://aclanthology.org/2022.wildre-1.2/>.
- Kishore Papineni, Salim Roukos, Todd Ward, et al. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 311–318, 2002.
- Anne Perera, Amitha Caldera, et al. Sentiment analysis of code-mixed text: A comprehensive review. *Journal of Universal Computer Science*, 2024.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2024.
- Thamar Solorio and Yang Liu. Learning to predict code-switching points. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 973–981. ACL, 2008.

Kristina Toutanova, Dan Klein, Christopher D Manning, et al. Feature-rich part-of-speech tagging with a cyclic dependency network. *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180, 2003.

Aditya Vavre, Abhirut Gupta, Sunita Sarawagi, et al. Adapting multilingual models for code-mixed translation. *Findings of the Association for Computational Linguistics: EMNLP*, page 7133–7141, 2022.

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

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, et al. Bigcodebench: Evaluating language models on tool-augmented code generation. *Proceedings of the 13th International Conference on Learning Representations (ICLR 2025)*, 2025.

## 8 Appendix

### 8.1 Full list of evaluated models

Table 4: Full list of evaluated models and their parameter sizes.

Model name	Size (B)
StarCoder2 (base)	3B, 7B, 15B
Qwen2.5-Coder (instruct)	1.5B, 7B
DeepSeek-Coder-V2 (lite-instruct)	14B
DeepSeek-R1-Distill (Qwen, LLaMA variants)	1.5B, 8B, 14B
CodeLLaMA (instruct)	7B
Phi-4 / Phi-4-multimodal	7B, 14B
LLaMA-3.1 / LLaMA-3.2 (base)	1B, 8B
OpenCoder-8B-Instruct	8B
Gemma-3	4B
Hermes-2-Theta (LLaMA3)	8B

### 8.2 CMD 0.9 Code-Mixed prompt example (Hinglish)

Table 5: Comparison of original English and code-mixed Hinglish prompt at CMD = 0.9.

Original English prompt	Hinglish prompt (CMD = 0.9)
<pre>""" Given a list of numbers, compute the average sum of absolute differences across all permutations.  Args:     numbers (list): List of integers.  Returns:     float: The average sum of     differences.  """ </pre>	<pre>""" Diye gaye list ke sabhi permutations ke liye, har consecutive pair ke beech ke absolute difference ka average nikalo.  Args:     numbers (list): Sankhyaoon ki ek     list.  Returns:     float: Sabhi permutations ke     absolute antar ka average.  """ </pre>

### 8.3 CodeMix injection algorithm

The following pseudocode describes how we apply controlled code-mixing at a specified CMD level.

---

**Algorithm 1** CMD-Based word replacement (adapted from Nayak and Joshi Nayak and Joshi (2022))

---

**Require:** Translated sentence  $S$ , sorted list `sorted_words` of tuples (`eng`, `lan`, `score`), code-mix degree CMD in  $[0, 1]$

**Ensure:** Code-mixed sentence  $S^*$

```
1:  $S^* \leftarrow S$ 
2:  $words\_replaced \leftarrow 0$ 
3:  $k \leftarrow \lfloor \text{CMD} \times |\text{sorted\_words}| \rfloor$ 
4: for  $i = 0$  to  $|\text{sorted\_words}| - 1$  do
5:    $(weng, wlan, score) \leftarrow \text{sorted\_words}[i]$ 
6:    $remaining \leftarrow \max(0, k - words\_replaced)$ 
7:   if  $remaining = 0$  then
8:     break
9:   end if
10:  if  $score = \infty$  then
11:     $S^* \leftarrow \text{REPLACEWORD}(S^*, wlan, weng)$ 
12:     $words\_replaced \leftarrow words\_replaced + 1$ 
13:    continue
14:  end if
15:  while  $remaining > 0$  do
16:     $S^* \leftarrow \text{REPLACEWORD}(S^*, wlan, weng)$ 
17:     $words\_replaced \leftarrow words\_replaced + 1$ 
18:     $remaining \leftarrow remaining - 1$ 
19:  end while
20: end for
21: return  $S^*$ 
```

*Note:* `lan` refers to the matrix-language word corresponding to the English token `eng`.

---

## 8.4 GAME validation

### GAME validation prompt

I will give you a text in roman-`{lan}` and English below:-

-----

`{processed_cand_cm_romanized}`

-----

You have to follow these steps below to translate the text into English:

1. First translate the sentence exactly into non-roman `{lan}` representation of characters.
2. Now translate the sentence into its English translation and remember it as SE.
3. Now for the sentences that are English words and are common in both the SE in step 2 and the original text, POS tag them.
4. Now remembering the POS tags of words, translate the original sentence into `{lan}`, remembering the meaning of POS words and cross-language homonyms.
5. Now translate the final `{lan}` sentence in step 4 to its English translation.

Only give me the translated sentence in step 5 as your response in this template below so that I can extract the sentence using regular expression:

\*\*\*\*\*Final Translated Sentence\*\*\*\*\*

[put the final sentence from step 5.]

\*\*\*\*\*End\*\*\*\*\*

*Note:* The variable `lan` denotes the target matrix language, and `processed_cand_cm_romanized` refers to the Roman-script code-mixed prompt being validated.

## 8.5 LLM prompt templates

### A.1 Prompt for matrix language translation and dictionary construction

Exact prompt used

```
Translate this "instruct" sentence in {lan}
-----
{prompt['instruct_prompt']}
-----
Do not translate the code and programming terms (args, list etc) in the prompt. Make it more like
human written.
And here is some of the important English PoS tags: {i_imp_eng}
Look for the corresponding meaning of these PoS in {lan} and look for the {lan} word in the
translated sentence.
Now translate each {lan} word in the dictionary in Roman {lan} in three ways or spellings, all must
be strictly different in spelling. Remember that the translation must be only in Anglo-Saxon script.
Create a JSON dictionary which contains the eng PoS word as eng_word and the corresponding {lan}
word in the translated sentence as {lan}_word if exists in {lan}, and the Roman Anglo-Saxon script
translations of the {lan}_word.
Format above as RFC8259-compliant JSON dictionary, in the format:
["eng": <eng_word>, "{lan)": <{lan}_word>, "roman_{lan}":
<transliterations>]

Translate this "doc_struct" prompt in {lan}
-----
{prompt['doc_struct']}
-----
These are the requirements:-
Do not translate the keys and do not translate the word 'Args'.
Do not translate the code.
Do not translate programming terms like args, list etc in the prompt.
Do not translate any abbreviation.
Do not translate 'reqs','raises','examples'.
Return the whole prompt.
Do not miss to include any docstring element.
Only output the docstring dictionary with no other text at all.
And here is some of the important English PoS tags for this sentence: {d_imp_eng}
Look for the corresponding meaning of these PoS in {lan} and look for the {lan} word in the
translated sentence.
Now transliterate each {lan} word in the dictionary in Roman {lan} in three ways or spellings, all
must be strictly different in spelling. Remember that the translation must be only in Anglo-Saxon script.
Create a JSON dictionary which contains the eng PoS word as eng_word and the corresponding {lan}
word in the translated sentence as {lan}_word if exists in {lan}, and the Roman Anglo-Saxon script
translations of the {lan}_word.
Format above as RFC8259-compliant JSON dictionary, in the format:
["eng": <eng_word>, "{lan)": <{lan}_word>, "roman_{lan}":
<transliterations>]

Only return the translated prompt and the Roman dictionary in this format structure and nothing else:-
*****translated_instruct_prompt
[translated_prompt]
*****roman_instruct_dictionary
[roman_dictionary]
*****translated_doc_struct_prompt
[translated_prompt]
*****roman_doc_struct_dictionary
[roman_dictionary]
```

## A.2 Prompt for romanization of translated tokens

Romanization prompt

I want you to romanize the following from {lan} to its roman translation while keeping the words from English as it is.  
 Remember that the translation must be only in roman Anglo-Saxon script. The translated prompt should not contain any {lan} words and all the spellings for English words must be correct.

---

{prompt}

---

Only return the roman translation in the exact format and structure as the original prompt.

## 8.6 Prompt examples across CMD levels

Table 6: Prompt translations across CMD levels for a realistic logging-related task. Differences between the CMD levels—especially extra English tokens retained in CMD = 0.9—are highlighted in **bold**.

Prompt level	Prompt text
<b>English (original)</b>	<p>Find the latest log file in a specified directory that matches a given regex pattern. This function searches through all files in the specified directory, filters them based on the provided regex pattern, and returns the path to the most recent log file based on modification time. If no files match the pattern or the directory is empty, the function returns None.</p> <p>The function should output with:</p> <p><b>str or None:</b> The path to the most recent log file that matches the pattern, or None if no matching files are found.</p> <p>You should write self-contained code starting with:</p> <pre>import os import re def task_func(pattern, log_dir='/var/log/'): </pre>
<b>CMD = 0.6 (Hinglish)</b>	<p>Ek diye gaye regex pattern se mel khaane vaali, ek specified directory mein sabse <i>latest log file</i> dhoondhe. Yeh <i>function</i> specified <i>directory</i> mein sabhi <i>files</i> ko khojata hai, unhein pradaan kiye gaye <i>regex pattern</i> ke aadhaar par filter karta hai, and <i>modification time</i> ke aadhaar par sabse <i>recent log file</i> ke <i>path</i> ko lautaata hai. Yadi <i>pattern</i> se koi <i>files</i> mel nahi khaate or <i>directory empty</i> hai, to <i>function None</i> lautaata hai.</p> <p>Function ko output karna chahiye:</p> <p><b>str or None:</b> Sabse recent log file ka path jo pattern se mel khaata hai, or None yadi koi matching files nahi paaye jaate hain.</p> <p>Aapko yahaan se shuru hone vaala self-contained code likhna chahiye:</p> <pre>import os import re def task_func(pattern, log_dir='/var/log/'): </pre>
<b>CMD = 0.9 (Hinglish)</b>	<p>Ek specified directory mein latest log file khojen jo diye gaye regex pattern se <b>matching</b> khati hai. Yeh function specified directory mein sabhi files ki khj karta hai, unhein pradaan kiye gaye regex pattern ke aadhaar par filter karta hai, and modification time ke aadhaar par <b>most</b> recent <i>log file</i> ka path lautata hai. Yadi koi bhi file pattern se <b>matching</b> nahi khati hai or directory <b>khali</b> hai, to function None lautata hai.</p> <p>Function ko output karna chahiye:</p> <p><b>str or None:</b> <b>Most</b> recent log file ka path jo pattern se <b>matching</b> khata hai, or yadi koi matching files nahi mili to <b>None</b>.</p> <p>Aapko is tarah se self-contained code likhna chahiye:</p> <pre>import os import re def task_func(pattern, log_dir='/var/log/'): </pre>

## 8.7 Prompt split examples

Table 7: Example prompt formats from the complete and instruct splits of BigCodeBench.

Complete split	Instruct split
<pre> import numpy as np import math  def task_func(data, target, k):     """     Calculate the 'k' nearest neighbors by     geographic coordinates using a dataset     and a target data point.      The function returns a list of the     'k' nearest neighbors, sorted in     ascending order of their distances     from the target.      Parameters:     data (DataFrame): The dataset         containing geographical coordinates         with columns ['Latitude',         'Longitude'].     target (list): The target data         point as [Latitude, Longitude].     k (int): The number of nearest         neighbors to return. Must be a         non-negative integer.      Returns:     list: List of the 'k' nearest         neighbors as [Latitude, Longitude].      Raises:     ValueError: If 'k' is negative or         not an integer      Constants:     Radius of Earth = 6371 km      Example:     &gt;&gt; data = pd.DataFrame([         [14, 25], [1, 22], [7, 8]],         columns=['Latitude', 'Longitude'])     &gt;&gt; target = [10, 15]     &gt;&gt; k = 2     &gt;&gt; task_func(data, target, k)     [[7, 8], [14, 25]]     """ </pre>	<p>Calculate the 'k' nearest neighbors by geographic coordinates using a dataset and a target data point. The function returns a list of the 'k' nearest neighbors, sorted in ascending order of their distances from the target.</p> <p>Constants: radius of earth is 6371 km. The function should raise the exception for:</p> <p><code>ValueError: If 'k' is a negative integer or not an integer.</code></p> <p>The function should output with:</p> <p><code>list: List of the 'k' nearest neighbors as [Latitude, Longitude].</code></p> <p>You should write self-contained code starting with:</p> <pre> ``` python import numpy as np import math def task_func(data, target, k): ``` </pre>