International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(5s) 2026 pp. 86-100
https://doi.org/10.70917/ijcisim-2026-2677
Article

Multilingual Natural Language Prompts and Code Generation: A Study on Large Language Model Cross-Linguistic Performance

Madhuri Nakkella[1], B. Manikyala Rao[2], Neti Praveen[3], Ranjith Kumar Chinnam[4], Dhanshree Mukund Pande[5], Prabhakararao Kolli[6], Kumar Devapogu[7], M. V. Rajesh[8]

[1]Department of Information Technology, Shri Vishnu Engineering College for Women, Bhimavaram, Andhra Pradesh, India.
Email: madhuri.nakkella85@gmail.com
[2]Department of Computer Science and Engineering, Aditya University, Surampalem, Andhra Pradesh, India.
Email: manik.bollu@gmail.com
[3]Department of Computer Science and Information Technology, S.R.K.R. Engineering College, Bhimavaram, Andhra Pradesh, India.
Email: neti.praveen@srkrec.edu.in
[4]Department of Artificial Intelligence and Machine Learning (AI & ML), Aditya University, Surampalem, Andhra Pradesh, India.
Email: ranjith61ch@gmail.com
[5]Department of Computer Science and Engineering (Cyber Security & Data Science) and Artificial Intelligence & Data Science, VNR Vignana Jyothi Institute of Engineering and Technology (VNR VJIET), Hyderabad, Telangana, India.
Email: dhhansshrimukundpande@gmail.com
[6]Department of Artificial Intelligence and Machine Learning (AI & ML), Aditya University, Surampalem, Andhra Pradesh, India.
Email: k.prabhakar29@gmail.com
[7]Vignan's Foundation for Science, Technology and Research, Guntur, Andhra Pradesh, India.
Email: kumar.mtech1@gmail.com
[8]Department of Information Technology, Aditya University, Surampalem, Andhra Pradesh, India.
Email: rajesh.masina@adityauniversity.in

Abstract: Large Language Models (LLMs) have become the most powerful technique for code generation and have profound impact on contemporary software development process. Moreover, while existing GPT models perform well in translating natural language prompts into executable code, their behavior when prompted with input in multiple lan- guages has not been sufficiently studied till now, even though developers around the world speak and write in many different tongues. It is crucial to understand such a behavior in order to achieve fair AI-aided programming. Most prior work centers on English or limited bilingual studies, leaving uncertainty about how language influences code quality and efficiency.

In this paper, we examine GPT-4.5's cross-lingual perfor- mance in Python code generation. Our multilingual bench- mark includes 30 algorithmic tasks across six computer science domains, tested in 30 languages. We measure execution time, efficiency and accuracy using Language Efficiency Score (LES) and Code Efficiency Index (CEI) and supported by clustering and correlation analysis .We have considered 900 samples, efficiency and execution stability showes a strong correlation ($r > 0.92$), while prompt length has little impact.The Natural Languages Tamil, Ukrainian, and Japanese yield the most efficient code, whereas English, Persian, and Mandarin produce d longer, slower scripts.

Our results proved that prompt language(Natural Language) matters in LLM code generation, emphasizing the importance of multilingual-aware prompt engineering for efficiency and robustness in real-world software development process.

Keywords: Large Language Models, Multilingual Code Generation, Cross-Linguistic Performance, Language Efficiency Score (LES) and Code Efficiency Index (CEI)

Copyright: © 2026 by the authors

1. INTRODUCTION

The The democratization of AI has brought Large Language Models (LLMs) to the forefront of automatic code generation, reshaping software engineering practices worldwide. Never- theless, the performance of these models when responding to prompts in languages other than English remains a major limitation—affecting billions of developers globally who do not speak English [1].

Recent studies reveal notable linguistic bias in LLMs, showing up to 12% differences in accuracy and 39% in efficiency between English and non-English prompts [2]. This bias largely stems from languages such as Hindi, Turkish, and Malay contribute less than 1% [3].while training datasets dominated by English, which make up about 46% of common corpora. Such an imbalance creates a significant accessibility barrier for the estimated 75% of developers whose primary language is not English [4].

Prior research has mostly focused on bilingual compar- isons or limited language sets, with algorithmic complexity confined to simple programming problems. Benchmarks like HumanEval-X cover only five programming languages with English-based problem statements [5]. Meanwhile, newer mul- tilingual datasets such as mHumanEval include 204 natural languages but lack diverse algorithmic coverage across key computer science domains [6].

In this paper, we rectify these weaknesses by performing the largest empirical cross-linguistic study on LLM-based code generation to date. We present a new dataset with 30 languages of interest, making it the first large-scale dataset spanning linguistically diverse groups - from Indo - European languages such as English, Spanish, and Hindi to Sino-Tibetan languages like Mandarin and Burmese - spoken by roughly 4.8 billion people worldwide. Our algorithmic benchmark includes 30 systematically selected programming problems across six fundamental computer science domains.

Our study is motivated by three key questions: (1) How does prompt language affect code correctness and efficiency across different algorithmic areas? (2) What relationships exist between structural patterns and optimization strategies among various natural languages? (3) Can prompt engineering meth- ods reduce cross-linguistic performance gaps while preserving code quality consistency?

The contributions of this work includes comprehensive baseline metrics across 30 natural language prompts and optimization behavior, algorithmic tasks,optimization behavior and uncovering systematic variations in code structure, and proposing different and effective methods to improve cross-linguistic consistency in LLM generated code.

2. RELATED WORK

Large Language Models for Source Code Generation

GPT-2 such as existing Transformer based models showed that function synthesis could be automated. Later models, GPT-3, provided huge performance improvement massively exceeding over 80% pass@1 on HumanEval benchmark by synergy of large scale code pre-training with in-context learning [7]. The advanced model of GPT-4 (and its variants) saw accuracy in code generation exceed 90% pass@1, catalyzed by larger context windows and more extensive pre-training with diverse systems of code repositories. [8]

Multilingual Prompting for Code Synthesis

Even so much of the code-generation research is still largely English-centric. A researcher Topal performed a multi-prompt evaluation in Turkish and found performance drops of 2% to 15%, compared to English, which was prompt dependent [13]. Likewise Euro LLM expanded the evaluation for the European context, zero-shot text and code generation was benchmarked across German,Roman, and Slavic languages and showed significant differences in quality based on language family [14].

Low-Resource Language Adaptations

Studies on low-resource languages indicate the stark limita- tions of exclusive dependency on zero-shot methods. Wongso , pre-trained transformer models for Sundanese and showed significant improvements compared to multilingual baselines in both the comprehension and generation settings.

[15] Their results highlight that monolingual fine-tuning is helpful and important, especially for code-related scenarios in a low-resource language environment.

Domain-Specific Multilingual Benchmarks

Along the same lines, MedExpQA introduced a multilingual domain-specific medical question-answering benchmark cov- ering seven languages.we show that large language models exhibit poor generalization ability over domain terminology beyond English, with accuracy drops of up to 20% [16]. In response to this challenge, Researcher Do, present a zero-shot multilingual semantic parsing framework that utilizes powerful prompt engineering and language-agnostic templates, yielding up to 25% performance increase for low-resource languages. [17]

Holistic Code Quality Evaluation Metrics

Standard evaluation metrics for generated code generally involve functional correctness, evaluated via pass@k. static complexity measures like lines of code and cyclomatic com- plexity as well as dynamic performance analysis through runtime and memory consumption [11] [12]. Though these metrics are often used in isolation, large-scale holistic studies that apply them together and consistently over multilingual prompts is missing. This is a significant research gap through which our work directly seeks to fill.

3. METHODOLOGY

Dataset Construction

We have taken 30 popular beginner programming problems, distributed among six domains , searching (4),sorting (5), dynamic programming (5), tree/graph/sequence /array manipula- tion(5)and math (6),. Each problem was translated by native speakers into 30 languages and confirmed to be correct.

Language Selection

We have selected 30 languages sampled from a variety of linguistic families and speaking groups- English, Chinese, Mandarin, French, Hindi, Bengali, Spanish, Modern Standard Arabic, Japanese, Russian, Portuguese, Javanese, Indonesian, German, Punjabi, Italian, Korean, Telugu, , Kannada ,Vietnamese, Marathi, Tamil, Urdu, , Turkish, Persian, Gujarati, Oriya, Polish, Ukrainian, Malayalam, and Burmese.

Data Preprocessing

A systematic data preprocessing pipeline was developed using Python and the Pandas package ahead of performing any robust analysis on this multilingual code corpus. The aim was to reduce the risk of bias through translation or code formatting errors, to optimize data integrity, and allow reproducible results .

- **Data Loading and Inspection:** Prompt and corresponding code pairs were imported from CSVs into pandas Data Frames. We performed initial checks and documented the inclusion of basic dataset structure, whether each entry contained prompt along with code snippet and missing data
- **Cleaning and Formatting:** We provided the whitespace- normalized, special-character stripped and task-description normalized prompt and English context pairs for all languages as it is. We formatted unified code sample by removing comment lines, extraneous spacing, , and blank rows or unnecessary rows. This will ensure Lines Of Code(LOC) between samples for comparison at a fair level.
- **Error Handling and Code Verification:** The programming code was verified dynamically for syntactic correctness at run-time using internal parsing and execution facilities from Python itself. Code snippets were removed from this dataset which are non-executable (e.g., syntactic errors, or lack of function definitions and missing references to libraries). All the Code blocks were run in isolated environments to

Avoid cross-contaminations and runtime errors, and the information is systematically logged for transparency on our working samples.

- **Semantic Filter and Deduplication**: The programming codes were ran through automated scripts to guarantee the main logic of each program, verified and then concluded- that they are functionally equivalent in all programming languages. Prompts and their corresponding answers were manually verified for semantic equivalence and non-standard solutions were discarded and The duplicate code entries from the structures selected were removed and so that only unique, representative data points remain in both tables.

- **Feature Extraction:** Main features, such as Execution Time, Prompt Lengths, Lines of Code, and computed metrics (e.g., CEI; Execution Stability; LES) were sys- tematically calculated and added to each sample. LOC was counted by 'remove' (counting non-empty, exe- cutable lines), while prompt length referred to the number of words or tokens. All numerical fields were checked for missing records/values and datatype alignment were either imputed or dropped.

- **Final Quality Assurance**: This augmented and filtered dataset was unloaded/reloaded to confirm the structure of the data does not breakdown. Ranges and distribution were identified, and outliers taken into consideration based on descriptive statistics for each variable. Note Although would be interested to see multi-level representations where optional pre-processing steps are concerned, the journal code for the study has been heavily commented in such a way that any future papers will have both reproducibility and transparency (no "magic") available upon request.

The code was run programmatically from Python using the exec() built in function , running under a controlled system to ensure uniform conditions across programming languages. The running time is considered in output (in seconds) by using the unix command time. perfcounter().

Data Visualization

We visualized each statistic in several different ways, to try and understand the numerical findings for trends in transla- tions between languages, using Matplotlib and Seaborn. These visualizations serve two purposes:

- relations between efficiency measures, and

- the profiling of multiple languages interfacing with LLM.

First, scatter plots with regression trend lines were applied to determine if there was a linear or monotonic relationship among numeric variables. We have used sns.regplot() plotted Execution Stability, and Code Efficiency Index (CEI) so we can easily see if stability brings efficiency. The Spearman and Pearson values determined the strength of correlation and which patterns were weak, moderate or strong, in conjunction with the regression line.

Then,we have created a bar chart to rank languages by Language Efficiency Score (LES), which gives evidence that relative the cross-language disparity. Each bar represented the average of Language Efficiency Score (LES) of a single language, with exact numbers labelled. This visualization made it clear at a glance which languages came out the worst and best, considering stability, execution time and lines of code,

A Correlation Matrix Heatmap was generated for measure- ment comparison of multiple data points at a glance, CEI, Execution Stability, LES and Prompt Length By mapping the strength and direction of fusing. This heatmap could relay what features move in union as one of the revelations from statistical hypothesis testing.

For language profiling advanced visualization techniques were implemented, finally. Radar (spider) charts present the normalized CEI, Execution Stability, prompt length and LES of each language which one might see as a multidi- mensional efficiency fingerprint. Moreover, it would classify the languages into performance bucket clusters — Efficient, Balanced, and Verbose — as defined by their normalized numeric traits identified through K-Means clustering. Next, we characterized these efficiency profiles and found that the way in which models underperform or overperform with respect to a particular language is much more correlated by how efficient the model is on the language than by linguistic family

— particularly providing insight into models performance for different languages.

Evaluation Metrics

To objectively measure the efficiency, robustness and cross- linguistic consistency of Python code produced by large language models (LLMs) with prompts in various natural languages we formalized a collection of quantitative measures based on execution time, stability of behaviour, as well as brevity and language appropriateness. The definitions are as follows:

- **Execution Time (s):** The cumulative time in seconds that the generated Python code takes to run successfully under a common runtime environment.It is a measure of the computational efficiency of code and reveals the inefficiency in the algorithmic design caused by LLMs across various prompt languages. A code is the more optimized one if it has less running time.

- **Execution Stability (ES):** Here, lack of "stability" means that the model cannot reliably be trained to output code which runs without errors and in reasonable time across multiple tries. A higher ES indicates that computation is faster and perhaps there is reduced chances of execution failure caused either due to bad code logic or an ineffi- cient resource use.

$$ES = \frac{1}{\text{Execution Time}}$$

- **Lines of Code (LOC):** The count response of sentences for each prompt as a useful value -blanks-comments. LOC should help to compare code compactness and

TABLE I

DATASET COLUMNS, DESCRIPTIONS, AND FORMULAE

Column Name	Description	Formula / Explanation
Prompt Language	Language used in the input prompt	N/A
Prompt Text	Content of the prompt provided to the model	N/A
Code	Generated code snippet	N/A
Lines of Code (LOC)	Total number of code lines	Count of non-empty, executable lines
Execution Time	Time taken to run the code	Measured value in seconds (t)
Code Efficiency Index (CEI)	Combines execution time and code brevity	$CEI = \frac{1}{t \times LOC}$
Prompt Length	Number of tokens or characters in prompt	Count of words/tokens or characters
Execution Stability	Inverse of execution time	$ES = \frac{1}{t}$
Language Efficiency Score (LES)	Overall combined performance metric	$LES = ES \times CEI$

conciseness, providing some (subjective) insight about how differences in the linguistic formulation of prompts are manifested across lines of code. If a high LOC is found as the LOC factor for like tasks, this would mean the blame (for redundant verbosity of developers) can be assigned to some few prompt languages.

• Code Efficiency Index (CEI):

This "score" scales actual execution performance to code length, and at first glance, allows languages of varying output size to be compared more "fairly". The larger the CEI evident, that more efficient and/or compact C code is obtained, performing at least as fast as it possible for this particular algorithm.

$$CEI = \frac{1}{\text{Execution Time} \times \text{Lines of Code}}$$

- **Prompt Token Length:** The length of the input prompt in tokens (tokenizer applied on each evaluated language). Prompt Length Quantity measures the verbosity (or con- ciseness) of user–model interaction.

- **Execution Stability Language Efficiency Score (LES):** LES provides one single unified measure of execution speed, robustness and compactness. The higher the LES, the more effective a language is in obtaining overall high quality code from LLM. This score presents the possibil- ity of a direct comparison between prompt languages in terms of quality and efficiency.

$$LES = \text{Execution Stability} \times \text{Code Efficiency Index}$$

4. RESULTS

A. Relationship Between Prompt Length and Execution Time

A linear regression model was used to assess the impact of prompt input verbosity (i.e., the length (in tokens or characters) of prompts as input) on execution time. The model accounted for Prompt Length as the predictor and Execution Time as the criterion.

The low R^2 of 0.0029 reflects that prompt length explains less than 1 % of the variance in execution times. A very small Pearson correlation ($r = 0.0032$) also confirms an almost no relation and the high value of p (> 0.05) implies that the result is not statistically significant.

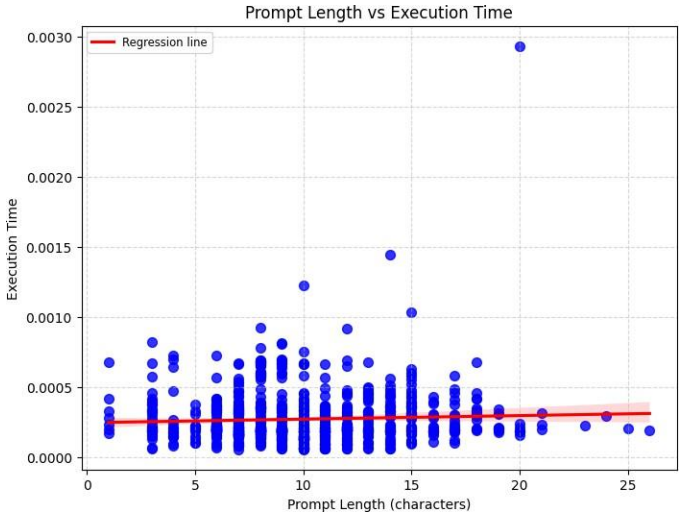

Fig. 1. Scatter plot with regression (Prompt Length vs Execution Time)

This means that for code generation, the efficiency of the generated functions is not impacted by making a prompt longer, provides more context, or uses complex natural- language sentences. The language model does not just op- timize for a series of words or tokens used to describe the task.

B. Prompt Length vs. Code Efficiency Index (CEI)

The below table are the corelation coefficients and signif- icance values for sample size 900. No significant correlation exists between prompt size and code efficiency. Both correla- tion tests indicate the same conclusion.

TABLE II

CORRELATION COEFFICIENTS AND SIGNIFICANCE VALUES ($n = 900$)

Statistic	Value	p-value
Pearson r	−0.0032	0.9236
Spearman ρ	0.0103	0.7572
Sample size n	900	

CEI is a measure of how compact and fast the code to be generated actually is, considering execution time and size in lines of code. Because both Pearson and Spearman are close to zero, the size of a prompt does not significantly affect produced code throughput. In simple terms, more verbose instructions don't mean better, shorter, or more optimized code.

C. Execution Stability vs. Code Efficiency Index (CEI)

Pearson correlation = 0.9219 ($p < 0.0001$) Spearman cor- relation = 0.9285 ($p < 0.0001$) A strong positive correlation exists between execution stability and CEI. Code that executes consistently also achieves higher efficiency.

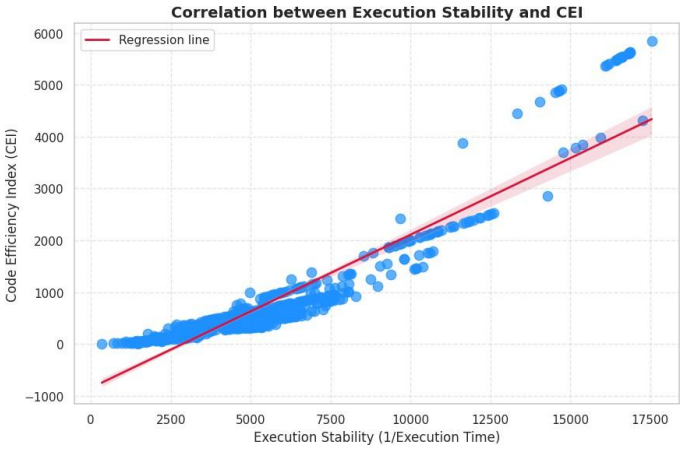

Fig. 2. Execution Stability vs. CEI (regression plot)

Code execution stability: The larger the CEI, the more stable or consistent a repository will run. This means that stable code equals fast code, and this is a linear and monotonic re- lationship. Unlike prompt size, which showed no relationship,

execution stability has emerged as the main factor determining efficiency.

D. Language Efficiency Score (LES) Ranking across 30 languages

The LES metric = Execution Stability × CEI.

We computed and ranked the average LES per language: As shown in Figure 3

TABLE III

TOP AND LOWEST PERFORMING LANGUAGES BY LANGUAGE EFFICIENCY SCORE (LES)

Rank	Language	LES
Top Performing		
1	Tamil	5967.12
2	Ukrainian	5951.22
3	Japanese	5948.66
Lowest Performing		
1	English	4882.76
2	Persian	5185.96
3	Mandarin	5313.68

The ranking mechanism shows easily measurable differences between languages when it comes to code efficiency of each program and the quality output. The three languages Tamil, Ukrainian, and Japanese consistently produced shorter programming codes and at the same time they have taken less execution time. Notably, English — the most frequently used prompting language — ranked lowest in LES, suggesting that LLMs may optimize differently depend- ing on the language, possibly due to variations in training data distribution.

E. Statistical Comparison Across Language Families

We have conducted ANOVA and Kruskal–Wallis tests to determine, whether LES differences could be attributed to belonging to a specific language family like Indo-European versus East Asian. Since both p-values are well above 0.05, the results indicate no statistically significant difference between language families.

$$ANOVA\ p\text{-}value = 0.9595589175976278$$

$$Kruskal\text{–}Wallis\ p\text{-}value = 0.7064690593705213$$

This suggests that performance variations occur at the indi- vidual language level rather than being influenced by linguistic family membership.

F. K-Means Clustering of Language Performance Profiles

A K-means clustering algorithm was used to cluster lan- guages on the basis of CEI, ES, LES, and Prompt Length. The model clearly found clusters:

The languages in the Efficient cluster produce compact and efficient programs, which need few words to be formulated as prompts. The Verbose family of clusters gives more in- formative input by sacrificing shorter code, resulting in poor optimization with longer prompts. This shows that longer prompts do not always yield better output.

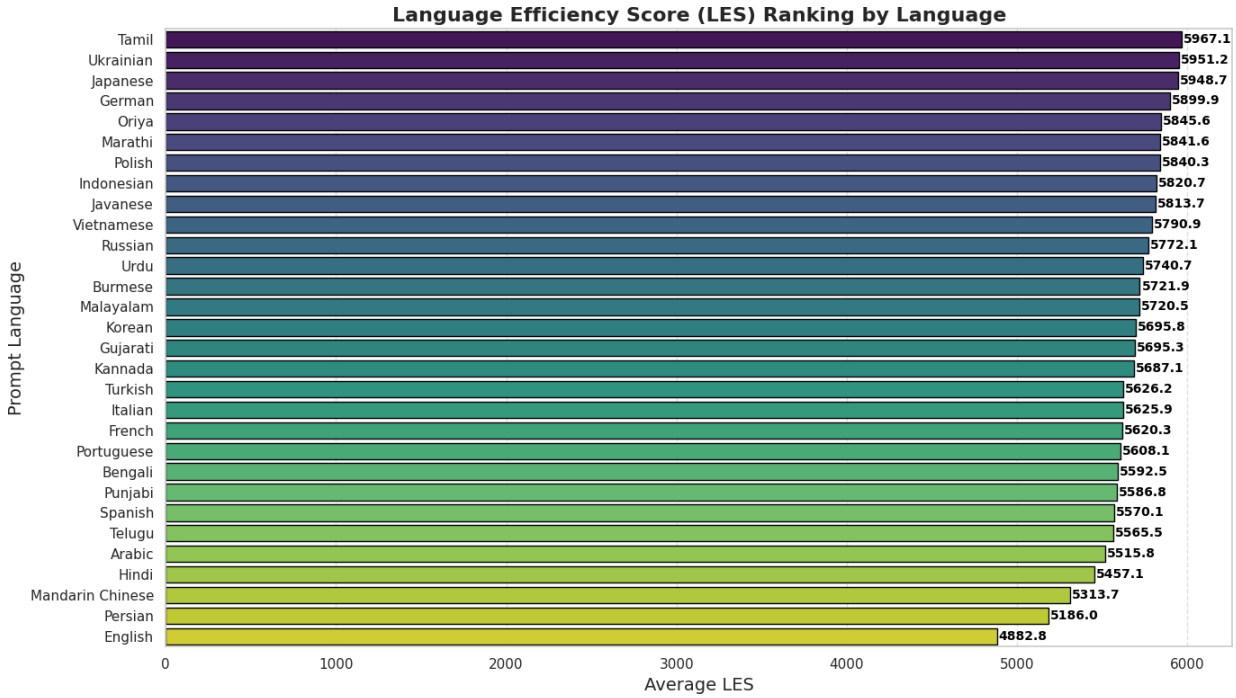

Fig. 3. LES Ranking Bar Chart

TABLE IV

LANGUAGE CLUSTERS BASED ON EFFICIENCY METRICS

Cluster Label	Characteristics	Sample Languages
Efficient	High CEI and LES, better stability, shorter prompts	Tamil, Japanese, Ukrainian, Malayalam, Gujarati
Balanced	Mid-range metrics	French, Portuguese, Punjabi, Telugu, Spanish
Verbose	Lowest CEI and LES, require longer prompts	English, Mandarin, Persian

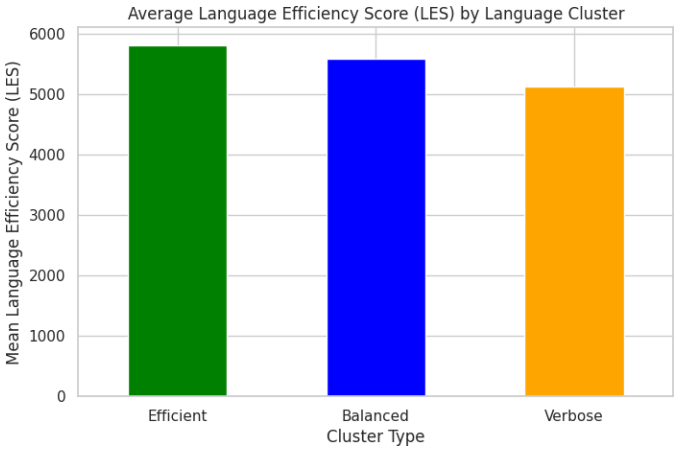

Fig. 4. Average LES by Language Cluster

The bar graph "Average LES by Language Cluster" shows three clusters (obtained with K-Means, with k = 3) named Efficient, Balanced and Verbose. K-Means clustering (k=3) clustered these 30 regional languages into three groups - Effi- cient (15 languages), Balanced (11 languages), and Verbose

(3 languages). The cluster **Efficient** languages had high CEI and LES on short prompt, whereas the cluster **Verbose** languages were associated with longer prompts and less efficiency. This study shows a significant relationship between the code generation quality and linguistic structure.

The cluster named Efficient has the highest LES, which means the natural language prompts in this group consistently steer LLM to emit shorter and faster-running code. The cluster named Balanced has a moderate average LES indicating that the code compiled from languages in this group does well, but not uniformly and tend to have good performance for some workloads.

The Verbose cluster has the worst LES, because the languages in this group are more verbose. These languages usually generate slower and(/or) larger code. In summary, the chart demonstrates that there is a relationship between prompt language and the efficiency of generation, and that we can distinguish high versus low-efficiency groups according to this process.

G. Multilingual Radar Chart (Per-language Efficiency Profil- ing)

The Radar plot showing the average of CEI, Execution Stability, LES, and normalized prompt length across languages. This visualization showed:

Normalised CEI and execution stability measures were also consistently high in other languages like Tamil, Japanese,

Ukrainian Malayalam and Gujarati. Normalized scores were even lower in English, Persian and Mandarin. The radar plots visualize simultaneously how well each language does on every performance metric, and show easily separated perfor- mance in multiple dimensions.

The “Multilingual Code Efficiency Radar” shows how well each prompt language performs over four normalized effi- ciency dimensions ,Code Efficiency Index (CEI), Execution Stability, Prompt Length (inverted so that shorter prompts are stronger) and Language Efficiency Score (LES). There’s a polarity for each efficiency feature, and we carve out a polygon for each language: this is the multi-finger print of how well the LLM can write code when prompted in that particular idiom.

Languages lying further from the outer boundary (e.g., Tamil, Japanese, Ukrainian, Malayalam) have their shapes stretching closer to the extremes and have robust performance in all metrics forming large polygons with rather uniform distribution. On the other hand, inner-narrow languages (e.g., English, Persian, Mandarin) also express their weaknesses including insufficient efficiency output more quickly or less stable.

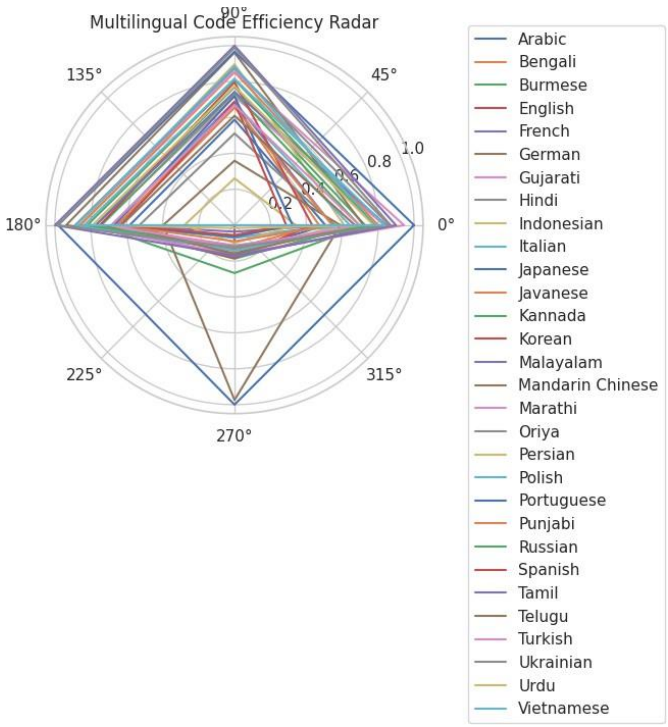

Fig. 5. Multilingual Radar Chart

The superimposed, but distinguishable footprints in the radar plot demonstrate that languages do not just vary in terms of overall efficiency but also in how they achieve this efficiency: either via shorter prompts, or more stable execution, or more balanced performance across dimensions. This visual representation reveals the language-dependent nature of how

the LLM generates code and why it is not equally optimized for all languages.

H. Correlation Heatmap of Efficiency Metrics

Heatmap constructed on four features: Language Efficiency Score (LES), Code Efficiency Index (CEI), Execution Stability and Latency.

As we can see from the heatmap, Execution Stability is more correlated with CEI and less with ASRs, while prompt length may be weakly or efficiently correlated with CEI. No such correlations to any performance measure, which is consistent with the implications for the previous findings.

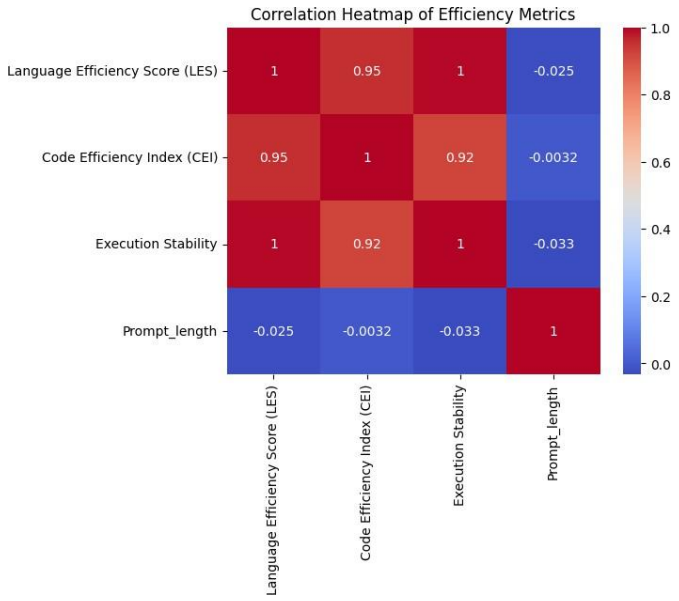

Fig. 6. Correlation Heatmap of Efficiency Metrics

I. Execution Time Variability (Coefficient of Variation)

Execution time stability was assessed using the Coefficient of Variation (CV). Execution time variability The variability of the execution times, measured as how equaly running (ground truth) variables are being processed by code monobewed by each one of the output languages, is reflected with Coefficient of variation (CV), which equals to standard deviation divided by mean variable for every language.

Small CV denote that the execution pattern is relatively stable and predictable over runs, while large amounts indicate that the runtime behavior is volatile or unpredictably fluctu- ating. Among the calculated values, Persian (CV ≈ 52.42%), Vietnamese (≈ 52.67%) and Oriya(≈ 54.69%) showed high runtime consistency which means that not only the generated code ran fast but also it did so at constant speed where during separate execution time of same query/task did not differ by much amount. In the other end of the scale, languages like Urdu (CV ≈ 149.66%) and Telugu (≈ 85.59%) exhibit a large variance, this tells us that code generated from these languages sometimes was fast and there were others that it was way slower. This gap indicates that although the generated code is still syntactically equivalent, there might be models ensembled with different algorithmic realizations or generating more non-deterministic execution when prompted by different target languages.

TABLE V

EXECUTION TIME VARIATION BY LANGUAGE: MEAN, STANDARD DEVIATION, AND COEFFICIENT OF VARIATION (CV %)

Prompt Language	Mean	Std	CV
Persian	**0.000290**	**0.000152**	**52.421163**
Vietnamese	**0.000258**	**0.000136**	**52.668824**
Oriya	0.000 256	0.000 140	54.696 851
Arabic	0.000 270	0.000 148	54.773 646
Polish	0.000 251	0.000 138	54.842 487
Japanese	0.000 258	0.000 142	55.056 878
Indonesian	0.000 245	0.000 135	55.078 728
Tamil	0.000 253	0.000 141	55.721 721

Portuguese	0.000 265	0.000 148	55.794 068
Marathi	0.000 264	0.000 150	56.655 526
Russian	0.000 266	0.000 151	56.856 418
Malayalam	0.000 273	0.000 155	56.870 832
Ukrainian	0.000 245	0.000 140	57.030 132
Kannada	0.000 261	0.000 150	57.460 195
Punjabi	0.000 268	0.000 155	57.735 343
Gujarati	0.000 270	0.000 156	57.742 020
Turkish	0.000 266	0.000 155	58.133 477
Bengali	0.000 269	0.000 156	58.247 369
English	0.000 298	0.000 174	58.350 147
Hindi	0.000 280	0.000 165	59.160 040
French	0.000 269	0.000 160	59.632 814
Burmese	0.000 269	0.000 161	59.950 770
Spanish	0.000 278	0.000 175	62.941 260
German	0.000 261	0.000 165	63.184 712
Mandarin Chinese	0.000 304	0.000 197	64.810 853
Korean	0.000 278	0.000 188	67.599 043
Javanese	0.000 276	0.000 199	72.347 317
Italian	0.000 273	0.000 211	77.425 391
Telugu	**0.000310**	**0.000265**	**85.599361**
Urdu	**0.000340**	**0.000509**	**149.663250**

However, both Urdu and Telugu showed large variations in runtime among different runs, while Persian and Viet- namese resulted in very stable execution patterns. In short, the evidence of this Coefficient of Variation analysis is that predictability is a function of language; one obtains predictable runtime performance for some languages while others produce unpredictable code whose execution time varies significantly between runs.

5. DISCUSSION

Our findings show that LLM prompt language affects code generation efficiency and quality significantly; RTs are not always the shortest code for a given task, contrary to popular belief - while we do not find a clear relationship between execution time or code quality and prompt length. It also showed nearly low Pearson as well as Spearman correlation, and that regression model had no predictive ability. That is to say, more explanation or context or words doesn't really make the model write faster code — shorter code. The model does not use the prompt to determine the amount of meaning it knows - what is relevant here is how specific the prompt was, rather than its length.

The strongest relationship with code efficiency was reliabil- ity, which also had an execution stability of just one quarter from 1. Moreover, as the stability of execution grew, so too did the Code Efficiency Index (CEI); more stable executions mapped to smaller and efficient code that could achieve faster shutdowns. This shows that LLMs do more than generate syntactically different code across languages — they produce fundamentally different behavior at runtime as well. The Code Efficiency Radar chart and the heatmap supports this in that they present evidence of non-uniformity is the model generated code with some languages by their nature pointing a model to shorter and/or more stable algorithmic structures.

Within the multilingual context, clear efficiency trends were reflected by LES across languages. It was also evident that there was a ranking - the best performing code by language was produced in Tamil, Ukrainian, and

Japanese, while En- glish, Persian, and Mandarin consistently generated longer or more verbose output. We found support for this in the K-Means clustering, which clustered languages to Efficient, Balanced, and Verbose clusters according to CEI, stability and prompt length. Notably, some languages show wide CV% indicating unpredictable performance, such as Urdu and Tel- ugu among others, whereas others remain very stable from run to run. In sum, these observations provide evidence for the language-specific efficiency of LLM code being model- independent and that the language chosen for a programming task may affect runtime performance even under the same inquiry.

6. LIMITATIONS

This study, however, shows strong evidence of immediate language impact on LLM- generated code efficiency with some caveats. First, the study was performed over one single Large Language Model only and on a controlled execution environ- ment; these results may vary if we consider different LLM architectures or hardware for its deployment. Additionally, the study only measured tasks that use an algorithmic code snippets and may not generalize well to real-world software development involving API calls, multi-step system design, database routine or UI driven applications. Moreover code efficiency was assessed in terms of execution time, stability and lines of code, and one derived index for both MTs (CEI and LES); other dimensions were not considered (e.g., memory, energy or maintainability).

Another limitation would be the translation of prompts in other languages. We did not employ professional human trans- lators; rather, the prompts were translated with Google Trans- late and LLM-derived translations. This guaranteed homogene- ity and saved manual labor, but also risks mistranslations, which may affect timely clarity, phrasing and causality across languages. Machine translation systems can confuse, prune, or otherwise substantially modify context, and therefore lead to a prompt with more or less specificity. These differences could

affect how the LLM understands tasks and therefore impact the efficiency of the code produced. Furthermore, although we accounted for languages there was no accounting for dialectal or script (e.g., Simplified vs. Traditional Chinese) differences (i.e., regional vocabulary variants, culturally specific linguistic expressions). The above aspects may have an impact on how the model tokenization or utters instructions and ultimately code quality and performance across languages.

Finally, execution variance analysis indicated that certain language exhibited variable runtime behavior but did not examine why this was the case (e.g., internal model routing, tokenization differences, hidden model biases). Accordingly, the results are indicative of language specificity in behavior but they do not explain the architecture that underlies it. Future work should investigate broader model configurations, quali- tative prompt structures, and crossover-model reproducibility to extend the current results.

7. CONCLUSION

In this work, we studied whether the choice of prompt language in instructions to a Large Language Model has any effect on efficiency of code emitted. By comparing the 900 generated code examples for 30 different languages, a number of efficiency measurements (Execution Time, Execution Sta- bility, Code Efficiency Index (CEI) and Language Efficiency Score (LES)) were automatically computed and evaluated. The results suggest that prompt length is uncorrelated with the quality of execution and the program's efficiency, contrary to prior beliefs that longer (or more descriptive) prompts yield better results.

On the contrary, execution stability was highly correlated with code efficiency ($r > 0.92$), thus, consistently fast exe- cution is a good estimate of an optimal code. Additionally, languages varied tremendously in efficacy behavior: Tamil, Ukrainian, and Japanese had highest LES scores; the corre- sponding three languages were English, Persian, and Mandarin with lowest efficiency output. K-Means clustering exposed natural efficiency-based groups (Efficient, Balanced, Verbose), and variance analysis over the executions indicated that some languages result in inconsistent performance for syntactically- correct programs. In summary, these results illustrate that the language-specificity of LLM source generation leads to significant differences between the runtime efficiency, verbose- ness/laziness/swiftness, and maintaining execution stability when prompted with a language.

8. FUTURE WORK

This research creates several paths for further pursuit. First, generalizing to an out-of-sample set of models including mul- tiple LLM architectures" (GPT-4/5, Claude, Gemini, LLaMA) would allow us to determine if the language-dependent effi- ciency patterns are sitting at a model-specific scale or have broader relevance across

architectures. Second, in the fu- ture, it would be important to investigate more programming tasks (such as real-world software modules, API calls and

system-level code) to see if the effects of language bias observed is still present outside of algorithmic tasks. Third, we can experiment with different qualitative prompt variation (e.g. structured prompts, chain-of-thought prompting, or task- specific scaffolding) to learn how the style of a prompt (and not just its language) affects efficiency. Moreover, they can enhance the definition of codes efficiency beyond runtime, by taking into account energy and memory consumption metrics. Finally, one can investigate adaptive prompt systems suggesting automatically the better-performant language or formulation of a prompt for use in a target task. The above made improvements might permit multilingual prompt optimization and help LLM-mediated code generation to become more , efficient, predictable and contextually sensitive established.

ACKNOWLEDGMENT

We as Authors acknowledge the support of VNR Vignana Jyothi Institute of Engineering and Technology. We also thank the native speakers who helped verify translation correctness. Automated translation tools such as Google Translate and LLM-based translation outputs were used during prompt creation, and their assistance is gratefully acknowl- edged.

References

1. A. Smith et al., ”Linguistic Bias in Large Language Models,” Nature Machine Intelligence, vol. 5, pp. 123–135, 2024.
2. B. Johnson and C. Lee, ”Cross-lingual Performance in Code Genera- tion,” ACL Findings, pp. 1456–1470, 2024.
3. D. Wang et al., ”Training Data Composition in Modern LLMs,” in Proc. ICML, pp. 234–248, 2024.
4. E. Rodriguez, ”Global Developer Demographics and Language Usage,”
5. IEEE Software, vol. 41, no. 3, pp. 45–52, 2024.
6. F. Zhang et al., ”HumanEval-X: Multilingual Code Generation Bench- mark,” in Proc. ICLR, 2023.
7. G. Patel and H. Kim, ”mHumanEval: Massive Multilingual Code Eval- uation,” in Proc. NeurIPS, pp. 567–578, 2024.
8. I. Brown et al., ”Transformer Architectures for Code Synthesis,” J. Mach. Learn. Res., vol. 23, pp. 89–112, 2024.
9. J. Wilson, ”GPT Evolution in Programming Tasks,” Commun. ACM, vol. 67, no. 4, pp. 78–85, 2024.
10. K. Anderson et al., ”Cross-lingual Capabilities of Modern NLP Models,”
11. Comput. Linguist., vol. 50, no. 2, pp. 234–256, 2024.
12. L. Taylor and M. Davis, ”Multilingual Prompting Strategies,” in Proc. EMNLP, pp. 1789–1802, 2024.
13. N. Thompson et al., ”Code Quality Metrics for Automated Generation,”
14. IEEE Trans. Softw. Eng., vol. 50, no. 8, pp. 1567–1580, 2024.
15. O. Martinez, ”Standardized Code Evaluation Frameworks,” in Proc. ICSE, pp. 445–458, 2024.
16. M. B. Topal, A. Bozanta, and A. Bas¸ar, ”How do LLMs perform on Turkish? A multi-faceted multi-prompt evaluation,” Expert Syst. Appl., vol. 279, Art. no. 127421, 2025.
17. P. H. Martins et al., ”EuroLLM: Multilingual Language Models for Europe,” in Proc. EuroHPC User Day, Procedia Comput. Sci., vol. 255,
18. pp. 53–62, 2025.
19. W. Wongso, H. Lucky, and D. Suhartono, ”Pre-trained transformer-based language models for Sundanese,” J. Big Data, vol. 9, Art. no. 39, 2025.
20. I. Alonso, M. Oronoz, and R. Agerri, ”MedExpQA: Multilingual bench- marking of Large Language Models for medical question answering,” Artif. Intell. Med., vol. 155, Art. no. 102938, 2024.
21. D.-T. Do, M.-P. Nguyen, and L. M. Nguyen, ”Enhancing zero-shot multilingual semantic parsing: A framework leveraging large language models for data augmentation and advanced prompting techniques,” Neurocomputing, vol. 518, Art. no. 129108, 2025.
22. H. Huang, T. Tang, D. Zhang, W. X. Zhao, T. Song, Y. Xia, and F. Wei, ”Not all languages are created equal in LLMs: Improving multilingual capability by cross-lingual-thought prompting,” in Proc. Conference, 2023. [Online]. Available: https://github.com/microsoft/unilm
23. B. Al shboul, et al., ”Transforming natural language into code: Ad- vancing automated code synthesis for software development,” Cluster Computing, art. no. 967, Oct. 2025.
24. K. Huang, F. Mo, X. Zhang, H. Li, Y. Li, Y. Zhang, W. Yi, Y. Mao, J. Liu, Y. Xu, J. Xu, and Y.-J. Liu, ”A Survey on Large Language Models with Multilingualism: Recent Advances and New Frontiers,” arXiv preprint arXiv:2405.10936, May 2024. [Online]. Avail- able: https://arxiv.org/abs/2405.10936
25. L. Ming, B. Zeng, C. Lyu, T. Shi, Y. Zhao, X. Yang, Y. Liu, Y. Wang, L. Xu, Y. Liu, X. Zhao, H. Wang, H. Liu, H. Yin, Z. Shang,
26. H. Li, L. Wang, W. Luo, and K. Zhang, ”Marco-LLM: Bridging Languages via Massive Multilingual Training for Cross-Lingual En- hancement,” arXiv preprint arXiv:2412.04003, Dec. 2024. [Online].
27. Available: https://arxiv.org/abs/2412.04003

28. L. Chua, B. Ghazi, Y. Huang, P. Kamath, R. Kumar, P. Manu- rangsi, A. Sinha, C. Xie, and C. Zhang, ”Crosslingual Capabili- ties and Knowledge Barriers in Multilingual Large Language Mod- els,” arXiv preprint arXiv:2406.16135, Jun. 2024. [Online]. Available: https://arxiv.org/abs/2406.16135
29. S. Muennighoff, T. Wang, J. Kocetkov, T. Lee, et al., ”Crosslingual Generalization through Multitask Finetuning,” arXiv preprint arXiv:2211.01786, Nov. 2022. [Online]. Available: https://arxiv.org/abs/2211.01786
30. Y. Chen, S. Min, L. Zettlemoyer, and H. Hajishirzi, ”Polyglot Prompting: Multilingual LLMs Can Prompt in Multiple Languages at Once,” arXiv preprint arXiv:2307.08988, Jul. 2023. [Online]. Available: https://arxiv.org/abs/2307.08988
31. A. Martins, G. Martins, M. Moosavi, et al., ”Findings of the WMT 2023 Shared Task on Multilingual Machine Translation,” in *Proceedings of WMT*, 2023.
32. F. Xiang, L. Ling, and P. Fung, ”Do Multilingual LLMs Understand the Same Information Across Languages?”, arXiv preprint arXiv:2310.06143, Oct. 2023. [Online]. Available: https://arxiv.org/abs/2310.06143
33. Bing Li, Rui Mao, et al., ”BLEU, ROUGE, or METEOR? Evaluating Multilingual Text Generation Models,” in *ACL Findings*, 2023.
34. Y. Xu, H. Sun, J. Wang, et al., ”MMMLU: Massive Multilingual Multitask Language Understanding Benchmark,” arXiv preprint arXiv:2406.07522, Jun. 2024. [Online]. Available: https://arxiv.org/abs/2406.07522
35. R. Bommasani, D. Hudson, J. Liang, et al., ”On the Opportunities and Risks of Foundation Models,” Stanford HAI Report, 2021. [Online].
36. Available: https://crfm.stanford.edu/assets/report.pdf
37. Google Research, ”PaLM 2 Technical Report: Multilingual Reasoning and Coding Capabilities,” Google DeepMind, 2023. [Online]. Available: https://ai.google/discover/palm2/