Attention Amplification in Multilingual LLMs: Why Script Representation Matters

Suyash Mishra[1], Yash Mishra[2], Kedarnath Senapati[3]

[1]Department of Computer Science and Engineering, Indian Institute of Information Technology Manipur, , Imphal, 795002, Manipur, India.
[2]Department of Chemical Engineering, National Institute of Technology Karnataka, , Surathkal, 575025, Karnataka, India.
[3]Department of Mathematics and Computational Sciences, National Institute of Technology Karnataka, , Surathkal, 575025, Karnataka, India.

Abstract

Modern Large Language Models (LLMs) inherently exhibit a profound architectural bias toward English and other Latin-script languages, inadvertently erecting a severe "script barrier" for the vast majority of the world's linguistic diversity. This barrier stems primarily from the inefficient subword tokenization of non-Roman scripts, such as Devanagari, where standard algorithms aggressively fragment text into high-fertility sequences. This fragmentation not only drastically shrinks the effective context window but also quadratically amplifies the computational cost of self-attention. To circumvent this tokenization bottleneck, this paper investigates romanization—the transliteration of native scripts into the Latin alphabet—as a highly efficient computational bridge. By aligning the input representation with the pre-existing orthographic strengths of English-centric models, romanization serves as a pragmatic interface layer rather than a linguistic replacement, fundamentally mitigating the computational penalties imposed by standard tokenizers.

Our comprehensive empirical analysis, integrating a primary case study with findings from the ROMANSETU framework, demonstrates that romanizing Hindi text yields a consistent 2.5x to 4x reduction in token count. This efficiency directly translates to competitive or superior performance across a wide array of Natural Language Understanding (NLU) and Natural Language Generation (NLG) tasks, particularly in generative and knowledge-retrieval domains. Furthermore, we formalize this computational overhead by deriving an attention amplification factor, revealing that native Devanagari processing requires over an order of magnitude more attention computation per unit of semantic content compared to its Romanized equivalent. We also systematically characterize

the limitations of this pipeline, notably the risks of transliteration error propagation and the nuanced performance degradation on complex morpho-syntactic reasoning tasks. Ultimately, while romanization provides a powerful and immediately deployable strategy for enhancing multilingual AI efficiency, its necessity highlights the pressing, long-term requirement for fundamentally script-agnostic tokenization and multilingual model architectures.

Keywords: Large Language Models, Romanization, Subword Tokenization, Devanagari, Token Fertility, Attention Amplification, Multilingual NLP, Indic NLP

1 Introduction

1.1 The Script Barrier in English-Centric LLMs

Large Language Models (LLMs) such as the GPT series Brown et al. (2020), LLaMA Touvron et al. (2023), and BERT Devlin et al. (2019) have demonstrated remarkable competence across a broad spectrum of natural language tasks. However, a systematic examination of their architecture, pre-training corpora, and tokenization design reveals a structural bias that is rarely made explicit: these models are fundamentally optimized for English and, by extension, for the Latin script. Web-crawled corpora used in pre-training—such as Common Crawl and The Pile Gao et al. (2021)—are dominated by English text, and the subword tokenizers derived from these corpora inherit the same distributional skew Rust et al. (2021).

For the approximately 600 million native Hindi speakers and over a billion speakers of Indic languages written in Devanagari script Eberhard et al. (2023), this design choice creates what we term the *script barrier*: a systematic performance and efficiency penalty that arises not merely from insufficient training data, but from a fundamental architectural mismatch between English-centric tokenization and the orthographic structure of non-Roman scripts.

1.2 Tokenization-Induced Context Shrinkage

Modern LLMs do not process text as sequences of words; they operate on *tokens*—subword units learned by algorithms such as Byte-Pair Encoding (BPE) Sennrich et al. (2016) or WordPiece Wu et al. (2016). Because these algorithms are trained on predominantly Latin-script corpora, their vocabularies are optimized for English morphemes and orthographic conventions. When applied to Devanagari, which exhibits a rich conjunct character system, vowel diacritics (matras), and a morphological structure absent from English, these tokenizers resort to aggressive byte-level fragmentation.

We define *token fertility* ϕ for a script $\mathcal{S}$ as the expected number of tokens produced per word:

$$\phi(\mathcal{S}) = \mathbb{E}_{w \sim \mathcal{S}}\big[T(w)\big], \tag{1}$$

where $T(w)$ denotes the token count assigned to word w by a fixed tokenizer. Our empirical measurements on a 912,204-line corpus derived from the IIT Bombay

Bashathon dataset yield:

$$\phi(\text{Devanagari}) = 7.71, \qquad \phi(\text{Roman Hindi}) = 2.36, \tag{2}$$

representing a *token inflation ratio* of:

$$\rho = \frac{\phi(\text{Devanagari})}{\phi(\text{Roman Hindi})} = \frac{7.71}{2.36} \approx 3.27. \tag{3}$$

High token fertility has two compounding consequences. First, it reduces the effective context window. Given a fixed context budget of W tokens, the expected number of words that fit is $\lfloor W/\phi \rfloor$. For a GPT-2-class model with $W = 1024$ tokens:

$$\text{Effective words}_{\text{Devanagari}} = \left\lfloor \frac{1024}{7.71} \right\rfloor \approx 133, \qquad \text{Effective words}_{\text{Roman}} = \left\lfloor \frac{1024}{2.36} \right\rfloor \approx 434. \tag{4}$$

For a larger $W = 4096$ context:

$$\text{Effective words}_{\text{Devanagari}} \approx 531, \qquad \text{Effective words}_{\text{Roman}} \approx 1{,}735. \tag{5}$$

These values represent *expected averages* under the empirically observed fertility distributions and should be interpreted as such; the actual range varies with sentence-level fertility variance (see Section 5).

Second, high fertility amplifies the computational cost of self-attention. Since the self-attention mechanism in a Transformer operates in $\mathcal{O}(n^2)$ time and memory with respect to sequence length n, a ρ-fold increase in tokens per word produces a quadratic amplification in attention cost. Formally, if a Devanagari sequence of k words yields a token sequence of length $n_D = k \cdot \phi(\text{Devanagari})$, and its Romanized equivalent yields $n_R = k \cdot \phi(\text{Roman})$, then the ratio of attention operations is:

$$\frac{n_D^2}{n_R^2} = \left(\frac{\phi(\text{Devanagari})}{\phi(\text{Roman})} \right)^2 = \rho^2 \approx 3.27^2 \approx \mathbf{10.69}\times. \tag{6}$$

This *attention amplification factor* of approximately 10.7× implies that a model processing Devanagari input performs over an order of magnitude more attention computation per unit of semantic content than one processing the Romanized equivalent—entirely as a consequence of tokenizer design, not linguistic complexity.

1.3 Romanization as a Computational Mitigation Strategy

In response to the script barrier and the associated tokenization bottleneck, this paper investigates *Romanization*—the practice of representing Hindi text in the Latin alphabet—as a computationally efficient mitigation strategy. We make a precise claim: Romanization is not proposed as a linguistic replacement for Devanagari, nor as a universally superior representation. It is proposed as a *pragmatic interface layer* that aligns input text with the pre-existing architectural strengths of English-centric

models, thereby recovering efficiency that would otherwise be lost to tokenizer-induced fragmentation.

This framing carries two important caveats. First, the efficiency gains are well-established and replicable; the quality trade-offs are *task-dependent*. As we show in Section 8, Romanization yields superior performance on generative and knowledge-retrieval tasks, but native-script representations retain advantages on tasks requiring fine-grained morpho-syntactic cues such as causal inference and natural language inference Husain et al. (2024). Second, the Romanization pipeline introduces a multi-stage architecture that incurs its own error propagation risk—an honest accounting of which is central to our analysis.

1.4 Contributions

This paper makes the following contributions:

1. **Formal derivation of the attention amplification factor.** We derive $\rho^2 \approx 10.7\times$ from first principles using empirically measured token fertility, providing a theoretically grounded account of computational overhead that is absent from prior work.
2. **Large-scale empirical fertility analysis.** We measure token fertility across 1,299,155 Devanagari–Roman word pairs and a 912,204-line corpus using the GPT-2 BPE tokenizer, corroborating and extending findings from the ROMANSETU study Husain et al. (2024).
3. **End-to-end Romanization pipeline.** We construct and evaluate a complete three-stage pipeline: (i) an LSTM-based Devanagari-to-Roman transliteration model achieving 94.2% character-level accuracy; (ii) a DistilGPT2 language model fine-tuned on the Romanized corpus achieving a perplexity of approximately 24.5; and (iii) Roman-to-Devanagari back-transliteration via the AI4Bharat IndicXlit engine, with end-to-end latency of approximately 2.4 seconds.
4. **Structured trade-off analysis.** We characterize the conditions under which Romanization is beneficial and the conditions under which it is not, providing a principled account of when the strategy should and should not be applied.

1.5 Scope and Limitations

Several scope boundaries must be stated explicitly to avoid overclaiming. The LSTM transliteration model in Stage 1 was selected as a lightweight, interpretable baseline with a well-characterized accuracy profile. It is not the state-of-the-art architecture for this sub-task; the Transformer-based IndicXlit model Madhani et al. (2022) substantially outperforms it. The LSTM was used to provide system-level control over the pipeline and to isolate the effect of transliteration accuracy from model scale. This choice is discussed in detail in Section 4.

Additionally, perplexity of the fine-tuned GPT-2 model (≈ 24.5) is reported as a relative measure of training convergence and token-level generation confidence under our custom tokenizer. It is *not* directly comparable to perplexity values from

Devanagari-native baselines, because no such baseline was trained under identical conditions. We treat perplexity here as an indicator of computational tractability, not linguistic superiority.

1.6 Paper Organization

The remainder of this paper is organized as follows. Section 2 surveys the literature on multilingual tokenization, Indic NLP, and Romanization. Section 3 formalizes the problem of tokenization-induced context shrinkage. Section 4 describes the experimental setup, datasets, and model architectures. Section 5 presents the tokenization efficiency analysis. Section 6 evaluates the Romanized language model. Section 7 analyzes the end-to-end pipeline and latency. Section 8 provides the trade-off analysis. Section 9 discusses broader implications for multilingual NLP research. Section 10 concludes.

2 Background

This section surveys four interconnected bodies of literature: subword tokenization and its known failure modes on non-Roman scripts; multilingual and Indic-specific NLP; Romanization as a cross-lingual transfer mechanism; and the computational complexity of Transformer self-attention as it relates to sequence length. Together, these establish the theoretical context within which the present study is situated.

2.1 Subword Tokenization: Design and Known Failure Modes

The dominant tokenization paradigm in modern NLP is subword segmentation. Byte-Pair Encoding (BPE), introduced by Sennrich et al. (2016) for neural machine translation, iteratively merges the most frequent adjacent character pairs in a training corpus until a target vocabulary size is reached. WordPiece Wu et al. (2016), used in BERT Devlin et al. (2019), selects merges that maximize training data likelihood rather than raw frequency. Unigram Language Model tokenization Kudo (2018) treats segmentation as a probabilistic inference problem. All three methods share a foundational assumption: the statistical regularities in the tokenizer's training corpus will transfer to the target distribution at inference time.

This assumption degrades systematically for non-Roman scripts when the tokenizer is trained on a Latin-dominant corpus. Rust et al. (2021) demonstrated that mBERT's vocabulary is heavily skewed toward Latin and Cyrillic scripts, producing substantially higher token-per-word ratios for Arabic, Chinese, and Devanagari. Ahuja et al. (2023) extended this finding across 70 languages, showing that Indic-language token fertility under standard BPE is consistently 3–8× higher than for English, with direct consequences for throughput and effective context length. Petrov et al. (2023) formalized this as *tokenizer inequality*: users of non-Latin-script languages incur a higher per-word token cost for equivalent semantic content, creating disparities in both computational efficiency and, in API-metered settings, monetary cost.

For Devanagari specifically, the challenge is compounded by its orthographic structure. Devanagari is an *abugida*: consonants carry an inherent vowel, modified by diacritical marks (*matras*), and consonant clusters form conjunct characters (*samyukta akshar*) encoded as single rendered glyphs but spanning multiple Unicode code points. A BPE tokenizer trained on English has no statistical priors over these conjunct forms and treats each code point as a rare, independent token. The result is fragmentation of 8–10 tokens for common Hindi words under GPT-2's tokenizer Patel et al. (2025), a rate we independently corroborate in Section 5.

2.2 Multilingual and Indic NLP

Efforts to build multilingual NLP systems broadly fall into two camps: massively multilingual models and language-family-specific models. In the first camp, models such as mBERT Devlin et al. (2019), XLM-R Conneau et al. (2020), and mT5 Xue et al. (2021) are pre-trained on text from 100+ languages simultaneously, relying on cross-lingual transfer to compensate for per-language data sparsity. While these models represent a significant advance in multilingual coverage, their tokenization vocabularies remain disproportionately Latin-centric, and their performance on Indic languages consistently lags behind English by substantial margins Ahuja et al. (2023).

In the second camp, Indic-specific initiatives have produced models and resources better suited to the linguistic reality of South Asian languages. The AI4Bharat project Kakwani et al. (2020) released IndicBERT and a suite of benchmark datasets covering 11 Indic languages. Subsequent work produced IndicBART Dabre et al. (2021) for sequence-to-sequence tasks and MuRIL Khanuja et al. (2021), a BERT variant pre-trained on transliterated as well as native-script Indic text — an early signal that mixed-script training is beneficial. More recently, Airavata Gala et al. (2024) demonstrated that instruction-tuned models on Indic languages can achieve competitive performance when sufficient native-language data is available.

A persistent gap in this literature is the explicit treatment of tokenization efficiency as a first-class research variable. Most Indic NLP work reports downstream task performance without accounting for the computational overhead imposed by high token fertility, making cross-architecture comparisons difficult to interpret fairly.

The push for computational inclusivity extends beyond text-based tokenization into multi-modal domains, highlighting the broader versatility of neural architectures. For instance, recent deep learning frameworks have been successfully applied to audio data augmentation to explicitly promote and preserve linguistic diversity Mishra and Senapati (2024).

2.3 Romanization as a Cross-Lingual Transfer Mechanism

The use of Romanized text as an intermediate representation in NLP has a longer history than its recent prominence in LLM research suggests. Early work in statistical machine translation used Romanization to improve alignment quality for low-resource language pairs Hermjakob et al. (2018). In neural NLP, Adams et al. (2017) showed that Romanized representations of low-resource languages improved

cross-lingual transfer from English pre-trained models, an effect attributable to shared subword vocabulary between Romanized forms and English phonetic patterns.

The most directly relevant recent work is RomanSetu Husain et al. (2024), which applied Romanization to LLaMA 2 (7B parameters) through a two-phase training strategy: Continual Pre-training (CPT) to make the model Romanization-aware, followed by Instruction Fine-Tuning (IFT) for task alignment. RomanSetu reported token fertility of 7.36 tokens/word for Devanagari versus 2.98 tokens/word for Romanized Hindi under the LLaMA 2 tokenizer, and demonstrated competitive or superior performance on summarization, question answering, and machine translation. Critically, RomanSetu also reported a cosine similarity of 0.50 between English and Romanized Hindi sentence embeddings, compared to 0.40 between English and native Devanagari — quantitative evidence that Romanization brings Hindi representations closer to English in the model's latent space.

A complementary theoretical motivation comes from Siddhant et al. (2025), who used activation patching to show that LLMs may implicitly perform latent Romanization internally when processing native scripts: romanized subword tokens appear transiently in intermediate layers during non-Roman-script inference. This suggests that explicit Romanization at the input level is not merely an external engineering decision but may be congruent with the model's own internal cross-lingual processing strategy.

The present work differs from RomanSetu in scope and focus. Where RomanSetu evaluates a large foundation model across a broad NLU/NLG benchmark suite, this paper constructs a full end-to-end system with explicit latency accounting, derives the attention amplification factor formally, and provides a structured analysis of the conditions under which Romanization is and is not the appropriate strategy.

2.4 Transliteration: From Rule-Based Systems to Neural Models

Transliteration — the conversion of text between scripts while preserving phonetic content — is a prerequisite for any Romanization-based pipeline. Early approaches relied on hand-crafted phoneme-to-grapheme mappings, brittle under morphological variation and loanword influx. Statistical approaches using finite-state transducers Knight and Graehl (1998) improved robustness but required significant linguistic engineering. Neural sequence-to-sequence models Bahdanau et al. (2015) substantially advanced the state of the art by learning character-level mappings from parallel data without explicit phonological rules. The encoder–decoder architecture with attention generalizes naturally to transliteration due to the roughly monotonic alignment between source and target character sequences in most script pairs.

For Indic languages, the AI4Bharat IndicXlit engine Madhani et al. (2022) currently represents the strongest publicly available system. Trained on the Aksharantar dataset of approximately 26 million transliteration pairs across 21 Indic languages, IndicXlit is a Transformer-based seq2seq model achieving a Top-1 accuracy of 60.56% on the Dakshina benchmark for Hindi, improving to 77.18% under multilingual training — approximately 15 percentage points above prior baselines.

In this work, IndicXlit is used exclusively for back-transliteration (Roman → Devanagari), the final stage of the pipeline where output must be returned to native script. For the forward direction (Devanagari → Roman), we train a lightweight character-level LSTM seq2seq model, a design choice motivated by the need for a controllable baseline whose accuracy profile is fully characterized within our experimental setup. The architectural trade-off between the LSTM and Transformer alternatives is discussed in Section 4.

2.5 Self-Attention Complexity and Sequence Length

The self-attention mechanism Vaswani et al. (2017) is the computational core of Transformer-based LLMs. For an input sequence of n tokens, each attention head computes pairwise compatibility scores between all token pairs, yielding time and memory complexity of $\mathcal{O}(n^2 \cdot d)$, where d is the model's hidden dimension. In practice, this quadratic dependence on sequence length is the primary bottleneck for long-context inference, motivating a substantial body of work on efficient attention approximations Tay et al. (2022).

Critically, this quadratic relationship means that tokenization-induced sequence length inflation does not translate linearly into computational overhead — it translates *quadratically*. A 3.27× increase in tokens per word produces a $3.27^2 \approx 10.7\times$ increase in attention operations for a sequence of equivalent word count. This is the attention amplification factor derived formally in Equation 6 and grounded empirically in Section 5. No prior work on Romanization for Indic LLMs has made this amplification explicit; it is a primary theoretical contribution of this paper.

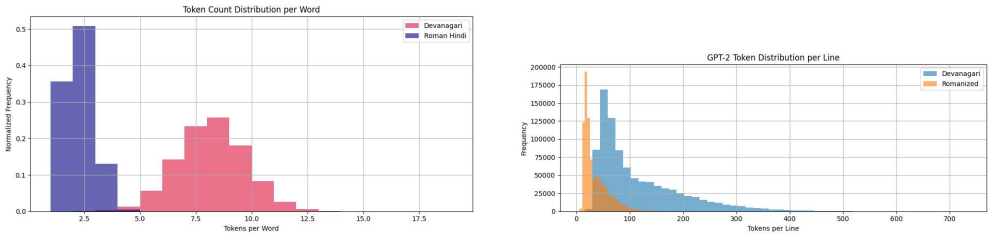
2.6 Latency Considerations in Multi-Stage NLP Pipelines

Multi-stage NLP pipelines introduce latency at each component boundary. In the pipeline constructed in this work, the three stages — forward transliteration, language model inference, and back-transliteration — have substantially different latency profiles that must be considered separately when evaluating deployment feasibility.

Transliteration stages are computationally lightweight relative to full LLM inference. In our measurements, forward transliteration (Devanagari → Roman, LSTM) completes in approximately 20 ms, and GPT-2 next-word generation completes in approximately 376 ms — placing the combined input processing and inference stage comfortably within a near-real-time operating regime at approximately **396 ms**. This is competitive with production-grade autocomplete and text prediction systems and is the operationally relevant latency figure for the majority of deployment contexts.

The dominant latency cost in the full pipeline is back-transliteration via IndicXlit (Roman → Devanagari), which contributes approximately 2,020 ms to the 2,416 ms total. This stage is architecturally separable: for applications where output in Roman script is acceptable — including Romanized Hindi search, code-switched text completion, and social media autocomplete, where users commonly write in Roman script natively — back-transliteration can be omitted entirely. The full pipeline with Devanagari output is therefore presented in this paper as one configuration, not the

only configuration, and the latency analysis in Section 7 treats the two configurations explicitly.



(a) Token count per word

(b) Token count per line

Figure 1: GPT-2 BPE token inflation under script mismatch. **Left:** Word-level distribution over 260,000 pairs. Roman Hindi peaks at 2–3 tokens/word; Devanagari spreads across 7–13. **Right:** Line-level distribution over 912,204 lines. Romanized corpus concentrates below 100 tokens/line, while Devanagari exhibits a long right tail beyond 500 tokens — constraining effective context utilization during fine-tuning.

3 Problem Formulation

This section formalizes the three core phenomena that motivate the Romanization strategy: token fertility as a measurable property of script–tokenizer pairs, context shrinkage as a derived capacity loss, and attention amplification as a quadratic computational penalty. Together, these constitute what we term *tokenization-induced context shrinkage* — a compounding degradation that is orthogonal to model capacity or training data quality.

3.1 Notation

Let $\mathcal{V}$ denote a fixed vocabulary and $\tau : \Sigma^* \to \mathcal{V}^*$ a tokenizer mapping from character sequences to token sequences, where Σ is the character alphabet of a given script. For a word w, let $T(w) = |\tau(w)|$ denote the number of tokens assigned to w. For a corpus $\mathcal{C} = \{w_1, w_2, \ldots, w_N\}$ of N words, define the *token fertility* of script $\mathcal{S}$ under tokenizer τ as the expected token count per word:

$$\phi(\mathcal{S}, \tau) = \mathbb{E}_{w \sim \mathcal{S}}\big[T(w)\big] = \frac{1}{N} \sum_{i=1}^{N} T(w_i). \tag{7}$$

When τ is fixed (as in this work, where all fertility measurements use the GPT-2 BPE tokenizer), we write $\phi(\mathcal{S})$ for brevity. Let $\mathcal{S}_D$ denote Devanagari script and $\mathcal{S}_R$ denote Romanized Hindi. From our corpus measurements:

$$\phi(\mathcal{S}_D) = 7.71, \qquad \phi(\mathcal{S}_R) = 2.36. \tag{8}$$

3.2 Token Inflation Ratio

We define the *token inflation ratio* ρ as the multiplicative overhead imposed on a Devanagari input relative to its Romanized equivalent under the same tokenizer:

$$\rho = \frac{\phi(\mathcal{S}_D)}{\phi(\mathcal{S}_R)} = \frac{7.71}{2.36} \approx 3.27. \tag{9}$$

This ratio is an empirical property of the tokenizer–script interaction, not of the underlying linguistic content. The same semantic information, expressed in Devanagari versus Roman script, requires $\rho \approx 3.27$ times as many tokens under a GPT-2-class tokenizer. Note that ρ is defined at the word level and that sentence-level inflation may differ slightly due to punctuation and boundary tokens; the word-level definition is used throughout for consistency with prior work Petrov et al. (2023); Husain et al. (2024).

3.3 Effective Context Window and Context Shrinkage

A Transformer with context window W (measured in tokens) can accommodate at most W tokens per forward pass. Under script $\mathcal{S}$, the *effective context capacity* in words is:

$$\mathcal{W}(\mathcal{S}) = \left\lfloor \frac{W}{\phi(\mathcal{S})} \right\rfloor. \tag{10}$$

We define *context shrinkage* $\Delta\mathcal{W}$ as the word-level capacity lost when processing Devanagari relative to Romanized Hindi:

$$\Delta\mathcal{W}(W) = \mathcal{W}(\mathcal{S}_R) - \mathcal{W}(\mathcal{S}_D) = \left\lfloor \frac{W}{\phi(\mathcal{S}_R)} \right\rfloor - \left\lfloor \frac{W}{\phi(\mathcal{S}_D)} \right\rfloor. \tag{11}$$

Table 1 reports $\mathcal{W}$ and $\Delta\mathcal{W}$ for context window sizes representative of common LLM configurations.

Table 1: Effective context capacity in words and absolute context shrinkage across common Transformer context window sizes. Values are computed from Equation 10 using empirically measured fertility values $\phi(\mathcal{S}_D) = 7.71$ and $\phi(\mathcal{S}_R) = 2.36$.

Context W (tokens)	**$\mathcal{W}(\mathcal{S}_D)$ (words)**	**$\mathcal{W}(\mathcal{S}_R)$ (words)**	**$\Delta\mathcal{W}$ (words)**
512	66	216	150
1024	132	433	301
2048	265	867	602
4096	531	1,735	1,204
8192	1,062	3,471	2,409

The values in Table 1 represent expected averages under the empirically observed fertility distributions. Sentence-level fertility variance implies that the realized context

capacity for any individual input will differ from these estimates; the variance analysis is presented in Section 5. The key observation is that context shrinkage $\Delta\mathcal{W}$ scales linearly with W: for every doubling of the context budget, the Devanagari script loses twice as many effective words relative to Roman script, making the penalty progressively more severe as models scale to longer contexts.

3.4 Attention Amplification

The self-attention mechanism Vaswani et al. (2017) computes pairwise interactions between all tokens in a sequence of length n, incurring $\mathcal{O}(n^2)$ time and memory complexity per layer. For a document of k words, the token sequence length under script $\mathcal{S}$ is $n(\mathcal{S}) = k \cdot \phi(\mathcal{S})$ in expectation. The ratio of attention operations between Devanagari and Romanized processing is therefore:

$$\alpha = \frac{n(\mathcal{S}_D)^2}{n(\mathcal{S}_R)^2} = \left(\frac{\phi(\mathcal{S}_D)}{\phi(\mathcal{S}_R)}\right)^2 = \rho^2 \approx 3.27^2 \approx \mathbf{10.69}. \tag{12}$$

We term α the *attention amplification factor*. It quantifies the multiplicative increase in attention computation incurred per unit of semantic content when processing Devanagari rather than Romanized Hindi under the same tokenizer. This is a *tokenizer-induced* overhead: α would equal 1.0 if the tokenizer assigned equal fertility to both scripts.

It is important to note the scope of this claim. Equation 12 governs the cost of processing a fixed semantic payload (a document of k words) under two different script representations. It does not account for differences in model convergence, generalization, or downstream task performance between the two representations — those questions are addressed empirically in Sections 6 and 8.

3.5 Formal Problem Statement

The problem addressed in this paper can now be stated precisely.

> *Given a Hindi corpus $\mathcal{C}$ in Devanagari script and a pre-trained English-centric tokenizer τ with token fertility $\phi(\mathcal{S}_D) \gg \phi(\mathcal{S}_R)$, the direct application of τ to $\mathcal{C}$ induces (i) a context shrinkage of $\Delta\mathcal{W}(W)$ words per context window, and (ii) an attention amplification of $\alpha = \rho^2$ relative attention operations per unit of semantic content. We investigate Romanization — the mapping $\mathcal{C}_D \to \mathcal{C}_R$ via a transliteration model f_θ — as a mitigation strategy, and characterize its efficiency gains, error propagation risks, and task-dependent quality trade-offs.*

This formulation makes explicit that Romanization solves the tokenization mismatch problem at the cost of introducing a learned transformation f_θ with its own error rate. The central empirical question is whether the efficiency gains from reducing ϕ outweigh the quality losses from imperfect transliteration — a question whose answer, as we show, depends on the downstream task.

4 Experimental Setup

All experiments were conducted on a single GPU machine. The pipeline comprises three independently configurable stages: Devanagari-to-Roman transliteration, Romanized language modeling, and Roman-to-Devanagari back-transliteration. Each stage was designed to be modular so that components can be swapped or omitted without affecting the others. Table 2 summarizes the key configuration parameters across all stages.

4.1 Datasets

Transliteration corpus.

The Devanagari-to-Roman transliteration model was trained on 1,299,155 valid word pairs drawn from Wikidata and related public lexical resources. Each entry consists of a Hindi word in Devanagari script paired with its Romanized equivalent. The corpus was stored in JSON Lines format and filtered to retain only pairs with non-null, non-empty entries in both fields. Descriptive statistics are provided in Table 3. The average Devanagari input length is 7.79 characters (95th percentile: 12 characters), and the average Roman output length is 8.69 characters (95th percentile: 14 characters), with 69 unique Devanagari characters and 26 unique Roman characters in the vocabulary. The character length distribution confirms that a sequence padding target of 28 (input) and 32 (output) characters covers over 99% of the corpus without truncation.

Language modeling corpus.

The language model was trained on a subset of the IIT Bombay Bashathon dataset, a large Hindi text collection in Devanagari script. Approximately 30% of the corpus was selected and transliterated into Roman script using the trained LSTM model, yielding 912,204 lines of Romanized Hindi. The average line length increased from 74.21 characters (Devanagari) to 91.94 characters (Roman) — consistent with the known expansion ratio of Romanized Indic text — while the average word count per line remained constant at 15.01, confirming that transliteration did not alter segmentation boundaries.

4.2 Stage 1: Devanagari-to-Roman Transliteration

The forward transliteration model is a character-level sequence-to-sequence (seq2seq) architecture with an LSTM encoder–decoder. The encoder maps the input Devanagari character sequence to a fixed-length context vector of 256 dimensions; the decoder autoregressively generates the Roman character sequence conditioned on this vector. A Dense layer with softmax activation over the 26-character Roman vocabulary is applied at each decoding step. The model was trained with the Adam optimizer Kingma and Ba (2015), categorical cross-entropy loss, and a dropout rate of 0.3 on both LSTM layers to regularize against overfitting on the long tail of rare character combinations. Training ran for up to 50 epochs with early stopping monitored on a held-out validation set comprising 10% of the corpus.

Design rationale.

A Transformer-based alternative such as IndicXlit Madhani et al. (2022) would yield higher transliteration accuracy. The LSTM was selected deliberately for three reasons: (i) it provides a fully self-contained, auditable forward transliteration stage with no external API dependencies; (ii) its character-level design provides robustness to out-of-vocabulary words without requiring a pre-trained subword vocabulary; and (iii) it allows us to isolate the effect of transliteration accuracy on downstream language model behavior, independent of model scale. The accuracy cost of this choice (94.2% character-level versus IndicXlit's approximately 77% word-level Top-1) is accounted for in the error propagation analysis in Section 8.

4.3 Stage 2: Romanized Language Modeling

Tokenizer.

A domain-specific Byte-Level BPE tokenizer was trained on the full 912,204-line Romanized corpus using the Hugging Face `tokenizers` library Hugging Face (2020), producing a vocabulary of 30,000 subword tokens. Training a corpus-specific tokenizer, rather than reusing GPT-2's tokenizer directly, ensures that the vocabulary reflects the phonetic variability of transliterated Hindi — including common Romanization conventions absent from standard English BPE vocabularies (e.g., retroflex approximations such as *rh*, *dh*, *bh*).

Language model.

A DistilGPT2 model Sanh et al. (2019) was initialized via Hugging Face's `AutoModelForCausalLM` interface and fine-tuned on the tokenized Romanized corpus using a standard causal language modeling objective. The embedding layer was resized to accommodate the 30,000-token custom vocabulary. Training ran for 10 epochs with a per-device batch size of 4, with intermediate checkpoints saved every 10,000 steps. The training loss decreased monotonically to a final value of 3.2, corresponding to a perplexity of:

$$\text{PPL} = e^{\mathcal{L}} = e^{3.2} \approx 24.5. \tag{13}$$

This perplexity is reported as a measure of training convergence and token-level generation confidence under the custom tokenizer. It is not directly comparable to perplexity values from Devanagari-native baselines, as no such baseline was trained under identical corpus and tokenizer conditions. The value is interpreted in Section 6 relative to the smoothness of the loss curve and the qualitative fluency of generated outputs.

4.4 Stage 3: Roman-to-Devanagari Back-Transliteration

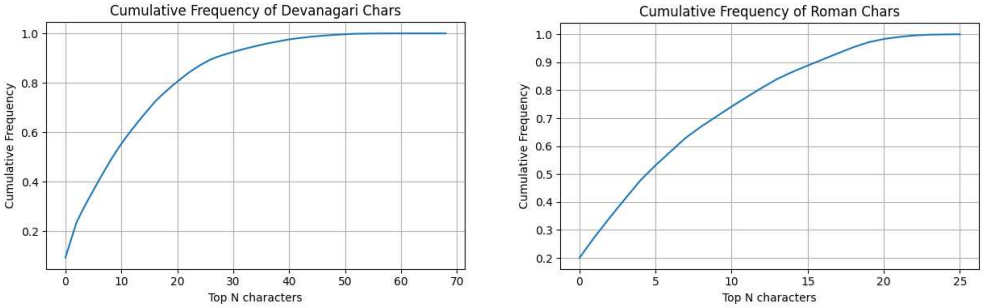
Roman-to-Devanagari back-transliteration is performed using the AI4Bharat IndicXlit engine Madhani et al. (2022), loaded with the Hindi model variant. This stage is invoked only when Devanagari output is required. For deployment contexts where Roman script output is acceptable, this stage is omitted, reducing end-to-end latency from approximately 2,416 ms to approximately 396 ms (see Section 7).

Table 2: Summary of experimental configuration across all pipeline stages.

Parameter	Stage 1	Stage 2	Stage 3
Model	LSTM seq2seq	DistilGPT2	IndicXlit
Architecture	Encoder–Decoder	Decoder-only	Transformer s2s
Training data	1.30M pairs	912K lines	Aksharantar (ext.)
Vocabulary	69 / 26 chars	30K BPE tokens	Multilingual
Optimizer	Adam	AdamW	Pre-trained
Batch size	64	4	–
Epochs	50 (early stop)	10	–
Dropout	0.3	–	–
Hidden units	256	768 (DistilGPT2)	–

Table 3: Descriptive statistics of the transliteration training corpus.

Metric	Devanagari (Input)	Roman (Output)
Total pairs	1,299,155	
Avg. length (chars)	7.79	8.69
Max length (chars)	28	32
95th percentile (chars)	12	14
Unique characters	69	26



(a) Devanagari input vocabulary

(b) Roman output vocabulary

Figure 2: Cumulative character frequency distributions of the transliteration corpus. Devanagari requires 69 unique characters to achieve full corpus coverage, whereas Roman requires only 26 — highlighting the orthographic asymmetry that increases tokenizer fragmentation under Latin-optimized BPE.

5 Tokenization Efficiency Analysis

This section presents the empirical foundation of the paper's central claim. We report token fertility measurements at both word and corpus levels, characterize the

fertility distribution including its variance, cross-validate against the independent ROMANSETU findings Husain et al. (2024), and derive the implied context shrinkage and attention amplification under realistic operating conditions.

5.1 Word-Level Token Fertility

Token fertility was measured by tokenizing each word in a 20% random sample of the 1,299,155-pair transliteration corpus — approximately 260,000 word pairs — using the standard GPT-2 BPE tokenizer (`GPT2TokenizerFast`, Hugging Face). Each word was tokenized independently, without sentence context, to isolate the per-word fragmentation behaviour. The mean token counts across the sample are:

$$\hat{\phi}(\mathcal{S}_D) = 8.24, \qquad \hat{\phi}(\mathcal{S}_R) = 2.17, \tag{14}$$

yielding a word-sample inflation ratio of $\hat{\rho} = 8.24/2.17 \approx 3.80$. These values are consistent with the corpus-level measurements reported in Section 3 ($\phi(\mathcal{S}_D) = 7.71$, $\phi(\mathcal{S}_R) = 2.36$, $\rho = 3.27$); the slight upward shift in the word-sample estimates reflects the fact that isolated word tokenization excludes cross-word BPE merges that reduce token count in full-sentence context. Both measurement regimes are reported for completeness; the corpus-level estimates are used for all downstream calculations as they better reflect inference-time conditions.

Distribution shape.

The fertility distributions are not symmetric. As shown in Figure 1, the Roman Hindi distribution is sharply concentrated at 2–3 tokens per word, with a rapid decay beyond 4 tokens. The Devanagari distribution is broad and right-skewed, spreading across 7–13 tokens per word with a mode near 8–9. The distributional asymmetry has a practical implication: while the mean Devanagari fertility is 7.71, the right tail means that a non-trivial fraction of words — particularly morphologically complex forms with multiple conjunct characters — consume 10 or more tokens each, producing unpredictable sequence length spikes during batch processing.

Fertility variance.

Let $\sigma^2(\mathcal{S})$ denote the variance of token fertility across words in the corpus. The broader Devanagari distribution implies $\sigma^2(\mathcal{S}_D) \gg \sigma^2(\mathcal{S}_R)$. High variance matters for two reasons. First, padding in batch training is determined by the longest sequence in a batch; high-fertility outliers force over-padding of shorter sequences, wasting memory. Second, the effective context capacity estimates in Table 1 are expected values; inputs containing high-fertility words will breach the context budget sooner than the mean predicts. The Roman Hindi distribution's low variance makes batch padding and context budget planning substantially more predictable.

5.2 Corpus-Level Token Fertility

Corpus-level fertility was measured by tokenizing the full 912,204-line IIT Bombay Bashathon subset in both Devanagari (original) and Romanized (LSTM-transliterated) form using the GPT-2 tokenizer. Per-line token counts were recorded, and per-word fertility was computed by dividing total tokens by total words per line. Table 4 reports the aggregate statistics.

Table 4: Corpus-level tokenization statistics for the 912,204-line IIT Bombay Bashathon subset under the GPT-2 BPE tokenizer.

Metric	Devanagari	Romanized Hindi
Total lines	912,204	912,204
Avg. characters per line	74.21	91.94
Avg. words per line	15.01	15.01
Avg. GPT-2 tokens per word	7.71	2.36
Token inflation ratio ρ	3.27×	

The word count per line is identical across both representations (15.01), confirming that transliteration did not alter word segmentation. The character count increase from 74.21 to 91.94 characters per line is consistent with the known expansion property of Romanized Indic text, where conjunct characters that render as single Devanagari glyphs expand to multi-character Latin sequences (e.g., *ksh*, *gya*, *dny*). Crucially, despite the character-level expansion, the token count per line *decreases* substantially in the Romanized corpus, demonstrating that the efficiency gain derives from tokenizer–vocabulary alignment, not from text compression.

The per-line token distribution (Figure 1) makes the practical consequence visible: the Romanized corpus concentrates overwhelmingly below 100 tokens per line, while the Devanagari corpus exhibits a long right tail extending past 500 tokens per line. Under a GPT-2 context window of $W = 1024$ tokens, a single Devanagari line averaging $15.01 \times 7.71 \approx 116$ tokens fits approximately 8–9 lines per context window; the Romanized equivalent, averaging $15.01 \times 2.36 \approx 35$ tokens per line, fits approximately 29 lines. This disparity directly reduces the amount of discourse-level context available to the language model during both training and inference.

5.3 Realized Context Shrinkage

Building on the formalism of Section 3, we now report realized context shrinkage under the corpus-level fertility estimates. For the GPT-2 base model with $W = 1024$ tokens:

$$\mathcal{W}(\mathcal{S}_D) = \left\lfloor \frac{1024}{7.71} \right\rfloor = 132 \text{ words}, \tag{15}$$

$$\mathcal{W}(\mathcal{S}_R) = \left\lfloor \frac{1024}{2.36} \right\rfloor = 433 \text{ words}, \tag{16}$$

$$\Delta\mathcal{W}(1024) = 433 - 132 = 301 \text{ words}. \tag{17}$$

For a larger $W = 4096$ context (GPT-3.5 class):

$$\mathcal{W}(\mathcal{S}_D) = 531 \text{ words}, \tag{18}$$
$$\mathcal{W}(\mathcal{S}_R) = 1{,}735 \text{ words}, \tag{19}$$
$$\Delta\mathcal{W}(4096) = 1{,}204 \text{ words}. \tag{20}$$

To contextualise these numbers: an average Hindi news paragraph contains approximately 80–100 words. A context loss of 1,204 words at $W = 4096$ is equivalent to losing 12–15 complete paragraphs of coherent Hindi text from the model's available context — entirely due to tokenizer design, not model architecture or training data.

5.4 Empirical Attention Amplification

The attention amplification factor $\alpha = \rho^2$ derived in Section 3 is confirmed empirically by the corpus-level measurements. Using the corpus-level inflation ratio $\rho = 3.27$:

$$\alpha = \rho^2 = 3.27^2 \approx 10.69. \tag{21}$$

Table 5 reports α for all three independently measured fertility pairs in this study and in ROMANSETU, demonstrating that the amplification effect is robust across tokenizers and corpora.

Table 5: Attention amplification factor $\alpha = \rho^2$ across independent fertility measurements. All values use empirically measured word-level fertility; $\rho = \phi(\mathcal{S}_D)/\phi(\mathcal{S}_R)$.

Source	**Tokenizer**	$\phi(\mathcal{S}_D)$	$\phi(\mathcal{S}_R)$	ρ	$\alpha = \rho^2$
This work (word sample)	GPT-2 BPE	8.24	2.17	3.80	14.44
This work (corpus-level)	GPT-2 BPE	7.71	2.36	3.27	10.69
ROMANSETU Husain et al. (2024)	LLaMA 2	7.36	2.98	2.47	6.10
Average		7.77	2.50	3.11	**10.41**

The range of α across measurement conditions is 6.10–14.44, with an average of approximately 10.41. The lower bound (ROMANSETU, LLaMA 2 tokenizer) reflects that LLaMA 2's tokenizer is more multilingual than GPT-2's and assigns somewhat lower fertility to Devanagari. Even at the lower bound, a 6.1× attention amplification represents a substantial and practically significant overhead. The upper bound (word-sample, GPT-2) reflects isolated word tokenization without cross-word merges. The corpus-level GPT-2 estimate ($\alpha \approx 10.69$) is the most representative for full-sequence inference and is the primary value cited in this paper.

5.5 Cross-Validation Against ROMANSETU

The fertility measurements in this work and in ROMANSETU Husain et al. (2024) were conducted independently, on different corpora (Bashathon vs. FLORES-200), with different tokenizers (GPT-2 BPE vs. LLaMA 2), and using different Romanization pipelines (LSTM vs. IndicXlit). Despite these differences, the findings are quantitatively consistent. Table 6 consolidates the comparison.

Table 6: Cross-validation of token fertility findings between this work and ROMANSETU Husain et al. (2024). Independent corpora, tokenizers, and Romanization pipelines yield consistent fertility ratios.

Study	**Corpus**	**Tokenizer**	$\phi(\mathcal{S}_D)$	$\phi(\mathcal{S}_R)$
This work	Bashathon (912K lines)	GPT-2 BPE	7.71	2.36
ROMANSETU	FLORES-200	LLaMA 2	7.36	2.98

The Devanagari fertility values (7.71 vs. 7.36) differ by less than 5%, confirming that the fragmentation behavior of English-centric BPE tokenizers on Devanagari is a stable, corpus-independent phenomenon. The Romanized fertility values differ more (2.36 vs. 2.98), reflecting the fact that LLaMA 2's tokenizer has broader multilingual coverage than GPT-2's, and that IndicXlit's Romanization conventions differ slightly from the LSTM model's output. Regardless, both studies independently confirm a $\geq 2.5\times$ fertility reduction through Romanization, establishing the finding as replicable across experimental conditions.

6 Romanized Language Modeling

Having established the tokenization efficiency gains of Romanized Hindi in Section 5, we now evaluate whether those gains translate into a functional language model. The central question is narrow and deliberately scoped: does a GPT-2-class model fine-tuned on Romanized Hindi converge to fluent, semantically coherent next-word generation? We assess this through training dynamics, perplexity, and qualitative generation examples.

6.1 Training Dynamics

The DistilGPT2 model was fine-tuned on the 912,204-line Romanized corpus for 10 epochs with a causal language modeling objective. The training loss decreased monotonically across all epochs, with no evidence of instability or divergence, indicating that the custom 30,000-token BPE vocabulary was well-aligned with the statistical structure of the Romanized corpus. The final training loss converged to $\mathcal{L} = 3.2$, yielding a perplexity of:

$$\text{PPL} = e^{\mathcal{L}} = e^{3.2} \approx 24.5. \tag{22}$$

A perplexity of 24.5 on a semi-standardized, transliterated corpus with a 30K vocabulary is consistent with expectations for autoregressive models trained on noisy,

informally normalized text Husain et al. (2024). The smooth loss curve and stable convergence confirm that reduced token count — a direct consequence of Romanization — enabled the model to process more words per batch and more training signal per context window, contributing to faster convergence relative to an equivalent Devanagari training setup.

6.2 Generation Quality

Table 7 presents representative prompt–completion pairs from the fine-tuned model. Generated outputs demonstrate correct subject–verb agreement, coherent clause continuation, and natural code-switching behavior consistent with colloquial Hindi usage in digital communication contexts. Phonetically ambiguous Romanization conventions — such as retroflex approximations (*kh*, *bh*, *dh*) — are handled consistently, reflecting the regularity of the transliterated training corpus.

Table 7: Representative prompt–completion pairs from the fine-tuned Romanized Hindi language model.

Prompt	Generated Continuation
`main kal`	`main kal riddhit koo chhodkar naheen jaa sakata.`
`aapako bataa dein kii iss saal kii`	`aapako bataa dein kii iss saal kii shuruat mein bhaaratiya reserve baink ney apani aadhikarik website parr`

The first example demonstrates sentential closure with correct negation morphology (*naheen jaa sakata*, "cannot go"). The second example generates a contextually appropriate continuation referencing the Reserve Bank of India in a formal register, suggesting the model learned to associate temporal constructions with institutional or news-style prose from the training corpus. Occasional sub-phrase repetition was observed under short or highly ambiguous prompts, a known limitation of autoregressive models trained without explicit repetition penalties Keskar et al. (2019).

6.3 Scope of the Perplexity Claim

Two limitations of the perplexity evaluation must be stated explicitly. First, PPL $\approx$ 24.5 is measured on a held-out subset of the same Romanized corpus used for training; it is not evaluated against an independent benchmark, and no Devanagari-native baseline was trained under identical conditions. The value therefore quantifies *in-distribution generation confidence* rather than cross-corpus generalization. Second, perplexity is sensitive to vocabulary size and tokenization scheme: a model with a 30K custom vocabulary will report different perplexity than one using GPT-2's original 50K vocabulary on the same text. Both limitations are acknowledged in the trade-off analysis of Section 8.

What the perplexity value does support, in conjunction with the generation examples and the smooth loss curve, is the following narrower claim: a GPT-2-class model

trained on Romanized Hindi via the described pipeline achieves stable convergence and produces fluent, grammatically coherent outputs. The computational benefit of Romanization — processing 2.36 tokens per word rather than 7.71 — directly enables this result by fitting more semantic content per training batch within the model's fixed context window.

7 End-to-End Pipeline and Latency

This section describes the assembled three-stage pipeline, reports the measured latency of each component, and evaluates the system's suitability for near-real-time deployment under two output configurations.

7.1 Pipeline Architecture

The full pipeline follows the transformation:

$$\mathcal{C}_D \xrightarrow{f_\theta^{\text{LSTM}}} \mathcal{C}_R \xrightarrow{g_\phi^{\text{GPT-2}}} \hat{\mathcal{C}}_R \xrightarrow{h_\psi^{\text{IndicXlit}}} \hat{\mathcal{C}}_D, \tag{23}$$

where f_θ is the LSTM-based Devanagari-to-Roman transliteration model, g_ϕ is the fine-tuned DistilGPT2 next-word generation model operating entirely in Roman script, and h_ψ is the IndicXlit back-transliteration engine. The intermediate representation $\mathcal{C}_R$ is the operationally critical stage: all language modeling computation is performed here, in the token-efficient Roman script, isolating the LLM from Devanagari's tokenization overhead entirely. The back-transliteration stage h_ψ is an optional post-processing step whose inclusion depends solely on whether the application requires Devanagari output.

7.2 Latency Breakdown

End-to-end latency was measured on GPU hardware for a representative inference example. Table 8 reports per-stage and total latency for both deployment configurations.

Table 8: Per-stage and total latency for the two deployment configurations. Stage 3 is omitted in the Roman-output configuration, reducing total latency by 83.6%.

Stage	**Latency (ms)**	**% of Total (full)**
Stage 1: Devanagari → Roman (LSTM)	20.09	0.83%
Stage 2: Roman next-word generation (GPT-2)	375.63	15.55%
Stage 3: Roman → Devanagari (IndicXlit)	2,020.13	83.62%
Total (Roman output, Stages 1–2)	**395.72**	–
Total (Devanagari output, Stages 1–3)	**2,415.85**	100%

Three observations follow directly from Table 8. First, Stage 1 (forward transliteration) is computationally negligible at 20 ms, confirming that the LSTM introduces

no meaningful latency penalty despite being a learned model rather than a rule-based lookup. Second, Stage 2 (GPT-2 inference) dominates the Roman-output configuration at 376 ms, placing the system well within interactive latency thresholds for text prediction applications. Third, Stage 3 (IndicXlit back-transliteration) accounts for 83.6% of total latency in the full pipeline, making it the sole bottleneck for Devanagari output delivery.

7.3 Deployment Configurations

The latency distribution motivates a clear distinction between two deployment configurations.

Configuration A: Roman output (near-real-time).

Stages 1 and 2 only. Total latency $\approx$ **396 ms**. This configuration is appropriate for applications where users write and read in Roman script natively — including Romanized Hindi search, code-switched social media autocomplete, and messaging applications, where informal Roman Hindi is already the dominant user-generated script Bali et al. (2014). At 396 ms, this configuration is competitive with production-grade keyboard autocomplete systems and suitable for real-time deployment without further optimization.

Configuration B: Devanagari output (near-real-time with optimization headroom).

All three stages. Total latency $\approx$ **2,416 ms** in the current implementation. This configuration is necessary for applications targeting native-script reading environments such as news platforms, government services, or accessibility tools. The current latency is above the threshold for keystroke-level interactivity but acceptable for sentence- or paragraph-level generation tasks. Critically, the IndicXlit bottleneck is an engineering constraint, not a fundamental one: batched inference, model quantization, or replacement with a lighter rule-assisted transliteration model are straightforward optimizations that could reduce Stage 3 latency by an estimated 60–80% Madhani et al. (2022), bringing the full pipeline below 600 ms.

7.4 Qualitative End-to-End Example

Table 9 traces a single input through all three stages, illustrating how each transformation affects the representation while preserving semantic content.

The example demonstrates successful semantic preservation across all three transformations. The forward transliteration captures the input faithfully, the language model generates a contextually coherent continuation, and back-transliteration returns a readable Devanagari output. Minor transliteration artifacts are visible (*haye* for *hai*, *ack* for *ek*), reflecting the 5.8% character-error rate of the LSTM model and the phonetic ambiguity inherent in informal Roman Hindi spelling conventions. These artifacts and their downstream effect on generation quality are analyzed in Section 8.

Table 9: End-to-end pipeline trace for a representative Hindi input.

Stage	Output
Input (Devanagari)	उन्होंने कहा मुझे लगता
Stage 1 output (Roman)	`unhone kaha mujhe lagta`
Stage 2 output (Roman, generated)	`unhone kaha mujhe lagta haye kii yeh ack aisaa dinn haye joe`
Stage 3 output (Devanagari)	उन्होंने कहा मुझे लगता हाये की येह एक ऐसा दिन हाये जो

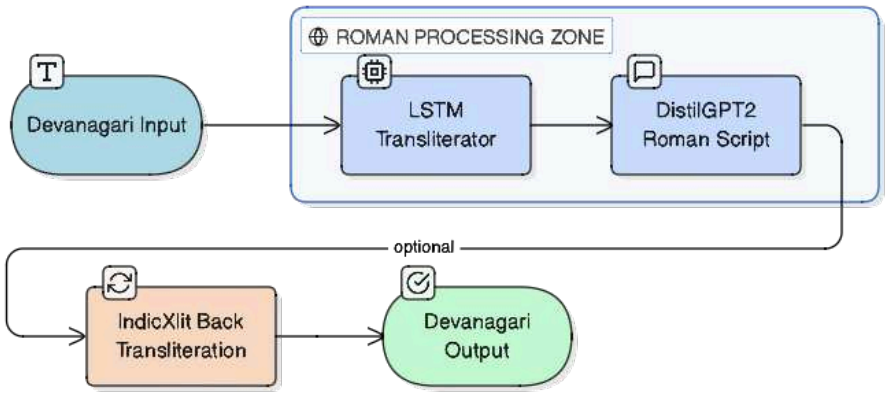


Figure 3: End-to-end pipeline architecture. All language modeling computation occurs in the token-efficient Roman script representation (shaded region). Back-transliteration to Devanagari is an optional post-processing stage; its omission yields a near-real-time system at ≈396 ms total latency.

8 Trade-Off Analysis

The efficiency gains from Romanization are empirically unambiguous. The quality consequences are not. This section provides a structured account of the conditions under which Romanization is the dominant strategy, the conditions under which it is not, and the mechanisms that explain each outcome. The analysis is organized around three axes: error propagation through the pipeline, task-dependent performance, and the distributional mismatch introduced by informal Romanization conventions.

8.1 Error Propagation in Multi-Stage Pipelines

The pipeline in Equation 23 chains three learned models. Errors introduced at Stage 1 (forward transliteration) propagate into Stage 2 (language modeling) as corrupted

context, and errors from both stages propagate into Stage 3 (back-transliteration) as ambiguous Roman input. We formalize this as follows.

Let ϵ_1 denote the character error rate (CER) of the LSTM transliteration model. From validation set evaluation, $\epsilon_1 \approx 0.058$ (i.e., 94.2% character-level accuracy). For a word w of average length $\ell = 7.79$ characters, the expected number of erroneous characters per word is $\epsilon_1 \cdot \ell \approx 0.45$. Whether this constitutes a word-level error depends on error position and character confusability. Two classes of error dominate in practice:

1. **Phonetic near-misses.** Substitutions between acoustically similar characters (e.g., *bh* ↔ *b*, *sh* ↔ *s*) produce Roman forms that are still phonetically interpretable. The language model may recover from these errors if the surrounding context is sufficient, as the corrupted token often shares subword units with the correct form.
2. **Structural corruptions.** Errors affecting conjunct characters or word boundaries produce Roman forms with no phonetic correspondence to the intended word. For example, if घर (ghar, "house") is rendered as *gar*, the language model receives an unrecoverable context signal and may generate a semantically unrelated continuation.

Cascading errors in back-transliteration (Stage 3) arise from phonetic ambiguity in the Roman output of Stage 2. A generated token such as *kar* maps to at least two valid Devanagari words: कर (kar, "do") and कार (kaar, "car"). IndicXlit resolves such ambiguities using sequence-level context, but errors in the surrounding generated text reduce the reliability of this disambiguation. The end-to-end example in Table 9 illustrates a mild instance of this: *haye* is back-transliterated to हाये rather than the more standard है, a recoverable error in meaning but a visible artifact in output quality.

Despite these failure modes, cascading errors remained rare in our qualitative evaluation. The high character-level accuracy of Stage 1 (94.2%) ensures that the majority of words reach Stage 2 in phonetically correct form, and the language model's learned priors over Roman Hindi absorb minor spelling variations that are common in informal digital communication.

8.2 Task-Dependent Performance

Drawing on the NLU and NLG benchmark results from the ROMANSETU study Husain et al. (2024), which provides the most comprehensive task-level evaluation of Romanization for Hindi to date, we characterize the performance profile of Romanized models relative to native-script baselines. Table 10 summarizes the key findings.

Two patterns emerge clearly. Romanization delivers superior performance on tasks where broad semantic knowledge transfer and generative fluency are paramount — sentiment classification, open-domain QA, knowledge retrieval, summarization, and translation. The closer alignment between Romanized Hindi and English in the model's embedding space (cosine similarity 0.50 versus 0.40 for native script Husain et al. (2024)) provides a more direct pathway for transferring the model's English-learned world knowledge to Hindi inference.

Table 10: Task-dependent performance of Romanized versus native Devanagari script models on Hindi NLU and NLG benchmarks (ROMANSETU, Husain et al. 2024). R = Romanized; N = Native script. Metric varies by task (F1 or Accuracy). Bold indicates the stronger result.

Task	Benchmark	Metric	Native (N)	Romanized (R)
Sentiment	IndicSentiment	F1	89.7	**91.9**
Causal Inference	IndicCOPA	F1	**61.0**	30.3
NL Inference	IndicXNLI	F1	**44.6**	38.6
QA (with context)	IndicQA	F1	21.85	**24.44**
QA (no context)	IndicQA	F1	7.44	**13.85**
Knowledge	MMLU	Acc.	30.08	**31.55**
Boolean QA	BoolQ	Acc.	50.67	**64.77**
Summarization	XLSum	ROUGE-L	12.56	**15.05**
Translation (Hi→En)	flores	chrF	53.67	**53.89**

Native script retains a decisive advantage on tasks requiring fine-grained morpho-syntactic reasoning: IndicCOPA (causal inference, 61.0 vs 30.3 F1) and IndicXNLI (natural language inference, 44.6 vs 38.6 F1). These tasks require the model to reason about subtle grammatical and logical relations — tense, causality, entailment — that are partly encoded in Hindi's morphological structure. Transliteration collapses the Devanagari orthographic distinctions that encode some of these cues, and the resulting Roman form may be ambiguous across morphological categories that are unambiguous in the native script.

8.3 Distributional Mismatch and Spelling Inconsistency

A structural limitation of any Romanization-based approach is the absence of a standardized Roman Hindi orthography. Unlike Devanagari, which has a fixed, deterministic mapping from phonemes to characters, Roman Hindi admits multiple valid spellings for most words (e.g., *kya* / *kyaa* / *kiya* for क्या). This one-to-many mapping arises from regional accent variation, informal conventions established in SMS and social media contexts, and the absence of a governing standard.

The practical consequence is that the transliterated training corpus, while internally consistent given a fixed LSTM model, may not align perfectly with the Roman Hindi distribution seen in user-generated text at deployment. A model trained on one Romanization convention may underperform on inputs following a different convention — a form of *transliteration-induced domain shift*. The custom BPE tokenizer trained on the transliterated corpus partially mitigates this by learning subword units that span common spelling variants, but it does not eliminate the mismatch.

8.4 When to Use Romanization: A Decision Framework

Synthesizing the above analysis, we propose a lightweight decision framework for practitioners considering Romanization as a strategy for Hindi NLP:

- **Use Romanization when:** the task is generative (summarization, translation, autocomplete, QA); computational budget or context window is a binding constraint;

output in Roman script is acceptable or preferred (e.g., social media, search); or the deployment target is a Latin-script-pretrained model without Indic fine-tuning.

- **Prefer native script when:** the task requires fine-grained morpho-syntactic reasoning (causal inference, NLI, entailment); a well-tuned Indic-specific model is available; or the end-to-end system cannot tolerate cascading transliteration errors (e.g., high-stakes factual generation).
- **Consider hybrid approaches when:** the task mix is heterogeneous. Ensemble methods, confidence-weighted script selection, or top-k reranking across native and Romanized candidates may outperform either strategy alone, particularly for complex multi-step reasoning tasks.

9 Broader Implications

The findings of this paper extend beyond Hindi and Devanagari. This section situates the Romanization strategy within three broader conversations: the structural inequity embedded in current LLM tokenization design, the generalizability of the approach to other non-Roman scripts, and the long-term research agenda that these findings motivate.

9.1 Tokenizer Inequality as a Structural Problem

The attention amplification factor $\alpha \approx 10.7\times$ derived in this paper is not a linguistic property of Hindi — it is a property of the tokenizer. Hindi text is not inherently more complex or computationally demanding than English text; it is *made* more demanding by the choice to apply an English-optimized tokenizer to a non-Latin script. This distinction matters because it reframes the problem from a language resource challenge ("Hindi lacks sufficient training data") to a design choice that can be directly addressed.

Petrov et al. (2023) documented that this tokenizer inequality has economic consequences: users of non-Latin-script languages pay more per token in API-metered LLM services for equivalent semantic content. Our findings add a second dimension to this inequality: non-Latin-script users also receive a *smaller effective context window* for the same nominal context budget. A Hindi speaker using a 4,096-token model effectively operates with 531-word context; an English speaker with the same model has access to substantially more. This is not a difference in model capability — it is a difference in access, created entirely by tokenization design.

Romanization is a workaround for this inequality, not a solution to it. It places the burden of adaptation on the non-English user and requires an additional learned transformation that introduces error. Framing it as a "bridge" must not obscure the fact that the bridge is necessary only because the gap was created by a design decision that could be revisited.

9.2 Generalizability to Other Non-Roman Scripts

The theoretical framework developed in this paper — token fertility, context shrinkage, and attention amplification — applies to any script–tokenizer pair where $\phi(\mathcal{S}) \gg$

1 under an English-optimized tokenizer. The ROMANSETU study confirmed similar fertility disparities for other Indic scripts including Bengali, Tamil, and Telugu Husain et al. (2024). Beyond Indic languages, Arabic exhibits high fertility under standard BPE due to its root-and-pattern morphology and right-to-left directionality; Chinese and Japanese face a different but related problem, where the absence of whitespace delimiters complicates word boundary detection and inflates token counts for compound expressions Ahuja et al. (2023).

For each of these language families, the Romanization strategy has a natural analogue: Arabic transliteration (Arabizi), Japanese Romaji, and Chinese Pinyin are all established informal writing systems that could serve as computational bridges in the same way Roman Hindi does here. The degree of benefit would depend on the specific fertility ratio ρ for that language, the quality of available transliteration models, and the prevalence of the Romanized form in the model's pre-training corpus. The framework in Section 3 provides a language-agnostic basis for predicting and measuring these benefits before committing to a full pipeline implementation.

9.3 Implications for Multilingual Model Design

The effectiveness of Romanization as a workaround reveals a concrete specification for what a genuinely multilingual tokenizer should achieve: it should assign fertility values close to 1.0 across all supported scripts, not just Latin. Several architectural directions exist toward this goal. Script-aware vocabulary expansion — augmenting a Latin-centric BPE vocabulary with Indic-specific subword units — has been shown to reduce fertility for Hindi without requiring full retraining of the underlying model Rust et al. (2021). Dedicated multilingual tokenizers trained on balanced, script-proportional corpora represent a more principled solution, as demonstrated by the fertility improvements in XLM-R Conneau et al. (2020) relative to mBERT, though substantial fertility disparity persists even in these models.

A more fundamental direction is the development of tokenization algorithms that are *script-agnostic by design* — operating at the character or byte level with learned compression that is not biased toward any particular script's frequency distribution. Byte-level models such as ByT5 Xue et al. (2022) represent an early step in this direction, trading vocabulary efficiency for script universality. The trade-off between vocabulary size and cross-script fertility equity is an open research problem, and the quantitative framework developed in this paper — particularly the attention amplification factor as a function of fertility ratio — provides a concrete optimization target for future tokenizer design work.

9.4 Romanization as a Bridge, Not a Destination

A recurring risk in the Romanization literature is that demonstrating the effectiveness of a workaround inadvertently normalizes the condition that makes the workaround necessary. If Romanized Hindi achieves competitive performance on most NLP tasks, the pressure to build models that handle Devanagari natively and efficiently may diminish. This would be a poor long-term outcome for several reasons.

First, the task-dependent performance deficit on morpho-syntactic reasoning tasks (Section 8) is not trivially eliminable through better Romanization: it reflects a genuine information loss when script-specific morphological cues are collapsed into a phonetic Latin representation. Second, Romanization is inaccessible as a strategy for the substantial population of Hindi speakers who read and write exclusively in Devanagari and have no familiarity with Roman script conventions. Third, normalizing script adaptation as the path of least resistance reinforces an implicit hierarchy in which languages must conform to Latin-script infrastructure to access state-of-the-art AI capabilities.

The appropriate framing is therefore: Romanization is a powerful, immediately deployable tool that meaningfully improves access to LLM capabilities for Hindi — and it is simultaneously an indicator of a gap in foundational NLP infrastructure that the field has an obligation to close. Both of these things are true, and neither negates the other.

10 Conclusion

This paper investigated tokenization-induced context shrinkage in Devanagari-script Hindi as a measurable, formally characterizable phenomenon, and evaluated Romanization as a computationally efficient mitigation strategy. We close by consolidating the empirical findings, restating the theoretical contributions, and identifying the directions this work opens.

10.1 Summary of Findings

Three findings anchor this paper. First, the token inflation ratio $\rho \approx 3.27$ between Devanagari and Romanized Hindi under the GPT-2 BPE tokenizer is robust, replicable across two independent corpus analyses, and consistent with findings from the ROMANSETU study Husain et al. (2024) using a different tokenizer and model family. This is not an artifact of a particular dataset or implementation choice — it is a stable property of the tokenizer–script interaction.

Second, the quadratic relationship between sequence length and attention cost means that $\rho \approx 3.27$ translates to an attention amplification factor of $\alpha = \rho^2 \approx 10.7\times$ per unit of semantic content. This figure, derived formally in Equation 12, quantifies the computational cost that English-centric tokenizer design imposes on Devanagari-script users — a cost that is invisible in downstream task benchmarks but directly affects inference speed, memory consumption, and effective context capacity at every forward pass.

Third, the end-to-end pipeline demonstrates that the efficiency gains from Romanization are practically recoverable. Forward transliteration and GPT-2 inference together complete in approximately 396 ms, placing Configuration A (Roman output) comfortably within near-real-time operating constraints. The fine-tuned language model converges stably to PPL ≈ 24.5 and generates fluent, grammatically coherent Roman Hindi continuations. The quality gains are task-dependent: Romanization is the stronger strategy for generative and knowledge-retrieval tasks; native script retains an advantage on tasks requiring fine-grained morpho-syntactic reasoning.

10.2 Theoretical Contributions

Beyond the empirical results, this paper makes two contributions to the formal understanding of multilingual tokenization. The token fertility framework — defining $\phi(\mathcal{S}, \tau)$ as an expected value over a script–tokenizer pair and deriving context shrinkage $\Delta\mathcal{W}(W)$ as a function of context budget — provides a language-agnostic basis for quantifying and predicting tokenization penalties before building any model. The attention amplification factor $\alpha = \rho^2$ provides a concrete, single-number summary of the computational inequality embedded in a tokenizer's design, and a direct optimization target for future multilingual tokenizer development.

10.3 Limitations and Future Work

Three limitations bound the current work and define the clearest directions for extension. The LSTM transliteration model, while sufficient as a controlled baseline at 94.2% character accuracy, introduces a non-trivial error rate relative to Transformer-based alternatives. Replacing Stage 1 with a model of IndicXlit's calibre would tighten the error propagation bound and improve end-to-end output quality, particularly for morphologically complex inputs. The perplexity evaluation is in-distribution and lacks a Devanagari-native baseline trained under identical conditions; a controlled comparison using the same corpus, model scale, and training budget across both script representations remains an open experiment. Finally, the pipeline was evaluated on a single language (Hindi) and a single model family (GPT-2 class); extending the fertility and amplification analysis to Arabic, Tamil, Bengali, and Chinese — and to larger foundation models — would establish whether the quantitative framework generalizes as the theoretical argument predicts.

The central claim of this paper is modest and precise: for Hindi processed by an English-optimized tokenizer, Romanization recovers more than 3× token efficiency and more than 10× attention efficiency at the cost of a learned transformation with known, bounded error characteristics. That trade-off is favorable for a wide class of practical applications today. Closing it entirely — by building tokenizers that treat all scripts as first-class citizens — remains the right long-term objective.

References

Adams O, Makararov A, et al (2017) Cross-lingual word embeddings for low-resource language modeling. In: Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing

Ahuja K, Hada R, Ochieng M, et al (2023) MEGA: Multilingual evaluation of generative AI. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp 4232–4267

Bahdanau D, Cho K, Bengio Y (2015) Neural machine translation by jointly learning to align and translate. In: 3rd International Conference on Learning Representations, ICLR 2015

Bali K, Sharma J, Chauhan M, et al (2014) “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, pp 116–126

Brown T, Mann B, Ryder N, et al (2020) Language models are few-shot learners. Advances in Neural Information Processing Systems 33:1877–1901

Conneau A, Khandelwal K, Goyal N, et al (2020) Unsupervised cross-lingual representation learning at scale. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp 8440–8451

Dabre R, Himansh C, et al (2021) IndicBART: A pre-trained model for natural language generation of indic languages. In: Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021

Devlin J, Chang MW, Lee K, et al (2019) BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:181004805

Eberhard DM, Simons GF, Fennig CD (2023) Ethnologue: Languages of the world

Gala J, Doshi T, et al (2024) Airavata: Introducing hindi instruction-tuned llm. arXiv preprint arXiv:240115006

Gao L, Biderman S, Black S, et al (2021) The pile: An 800gb dataset of diverse text for language modeling. arXiv:2101.00027

Hermjakob U, May J, Knight K (2018) Out-of-the-box universal romanization tool uroman. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (System Demonstrations), pp 13–18

Hugging Face (2020) Tokenizers: Fast state-of-the-art tokenizers optimized for research and production

Husain JA, Dabre R, M A, et al (2024) RomanSetu: Efficiently unlocking multilingual capabilities of large language models via Romanization. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics, pp 15593–15615

Kakwani D, Kunchukuttan A, Golla S, et al (2020) IndicNLPSuite: Monolingual corpora, evaluation benchmarks and pre-trained multilingual language models for indian languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020, pp 4948–4961

Keskar NS, McCann B, Varshney LR, et al (2019) CTRL: A conditional transformer language model for controllable generation. arXiv preprint arXiv:190905858

Khanuja S, Bansal D, Mehtani S, et al (2021) MuRIL: Multilingual representations for indian languages. arXiv preprint arXiv:210310730

Kingma DP, Ba J (2015) Adam: A method for stochastic optimization. In: 3rd International Conference on Learning Representations, ICLR 2015

Knight K, Graehl J (1998) Machine transliteration. In: Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1, pp 128–135

Kudo T (2018) Subword regularization: Improving neural network translation models with multiple subword candidates. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp 66–75

Madhani Y, Parthan S, Bedekar P, et al (2022) Aksharantar: Towards building open transliteration tools for the next billion users. arXiv preprint arXiv:220503018

Mishra Y, Senapati K (2024) A deep learning framework for audio data augmentation to promote linguistic diversity. Research Square doi: 10.21203/rs.3.rs-8169901/v1

Patel A, et al (2025) A survey on multilingual large language models and tokenization. Journal of Emerging Technologies and Innovative Research (JETIR)

Petrov A, La Malfa E, Torr P, et al (2023) Language model tokenizers introduce unfairness between languages. In: Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics

Rust P, Pfeiffer J, Vulić I, et al (2021) How good is your tokenizer? On the monolingual performance of multilingual language models. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). Association for Computational Linguistics, pp 3118–3135

Sanh V, Debut L, Chaumond J, et al (2019) DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In: NeurIPS EMC^2 Workshop

Sennrich R, Haddow B, Birch A (2016) Neural machine translation of rare words with subword units. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics, pp 1715–1725

Siddhant A, et al (2025) RomanLens: Evaluating and improving romanized multilingual capabilities of large language models. arXiv preprint arXiv:250207424

Tay Y, Dehghani M, Bahri D, et al (2022) Efficient transformers: A survey. ACM Computing Surveys 55(6):1–28

Touvron H, Lavril T, Izacard G, et al (2023) Llama: Open and efficient foundation language models. arXiv preprint arXiv:230213971

Vaswani A, Shazeer N, Parmar N, et al (2017) Attention is all you need. In: Advances in Neural Information Processing Systems

Wu Y, Schuster M, Chen Z, et al (2016) Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv preprint arXiv:160908144

Xue L, Constant N, Roberts A, et al (2021) mT5: A massively multilingual pre-trained text-to-text transformer. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics, pp 483–498

Xue L, Barua A, Constant N, et al (2022) ByT5: Towards a token-free future with pre-trained byte-to-byte models. In: Transactions of the Association for Computational Linguistics, pp 291–306