Attention Amplification in Multilingual Large Language Models

Why Script Representation Matters

Yash Mishra
Founder RAISEACAD (Solutions for Everyone)

Developer and Partner @ CodeEasy Innovation Labs Pvt Ltd.

Autonomous Software Engineering

The Governed Autonomy Pipeline

1. **Generate**
   LLM agents draft code from spec, tests and prior context
2. **Validate**
   Automated checks confirm correctness, coverage and intent
3. **Govern**
   A policy engine enforces standards before anything merges
4. **Deploy**
   Change-managed release into production, fully auditable

Open Research Questions

- **Verifiability**: Proving AI-authored code meets intent, not just passes tests
- **Explainability**: Audit trails that regulators and reviewers can actually trust

How India is Contributing to the AI Revolution

India's AI ecosystem, in numbers — and where this work fits in

3rd	6,200+	$2.57B	35,000+
Largest startup ecosystem in the world	AI startups building across India	Raised by Indian startups, Jan–Feb 2026	Delegates, 100+ nations at India AI Impact Summit 2026

In February 2026, India hosted the first AI Impact Summit held in the Global South, anchoring the government's *Viksit Bharat@2047* vision for AI-led growth.

GMMCode *by CodeEasy Innovation Labs Pvt Ltd*

GMMCode is an Indian-built, air-gapped AI code-governance platform for regulated BFSI enterprises, enforcing 302 standards across 16 domains through 37 governance models before code reaches production.

MADE IN INDIA

Motivation and Core Problem

- Most Large Language Models (LLMs) are trained mainly on English text
- English uses the Latin script, so tokenizers are optimized for it
- Indian languages like Hindi use non-Roman scripts
- This causes poor tokenization for scripts like Devanagari
- Same meaning consumes more tokens in Hindi than English
- Context window is counted in tokens, not in words
- Hindi users lose useful context unfairly
- This affects hundreds of millions of users

The Script Barrier

- LLM design is English-centric by default
- Training data like Common Crawl is mostly English
- Tokenization vocabularies favor Latin characters
- Indic scripts lack frequent subword units
- Complex characters get broken into many small tokens
- Results in higher cost and slower inference
- This is a design issue, not a language weakness

Why Tokenization Fails for Devanagari

- BPE tokenization learns patterns from English text
- Devanagari works very differently from English
- It uses vowel signs (matras) and conjunct letters
- Tokenizer treats rare Unicode parts separately
- A single Hindi word splits into 8–13 tokens
- This wastes memory and compute power
- Sequence length becomes unpredictable

Token Fertility: Key Metric

- Token fertility = average number of tokens per word
- Defined for a script under a fixed tokenizer
- Shows how well tokenizer matches a script
- Lower fertility means better efficiency
- Measured using GPT-2 BPE tokenizer
- Same Hindi text checked in two formats:
  - Native Devanagari
  - Romanized Hindi

$$\phi(S) = \mathbb{E}_{w \sim S}[T(w)]$$

Measured Token Fertility Results

- Devanagari Hindi: 7.71 tokens per word
- Romanized Hindi: 2.36 tokens per word
- Meaning is identical, representation changes
- Token inflation ratio:

$$\rho = \frac{7.71}{2.36} \approx 3.27$$

- This inflation is caused by the tokenizer
- It is not due to grammar or vocabulary
- Results are stable across datasets

Context Shrinkage Problem

- LLMs have a fixed token limit W
- More tokens per word means fewer words fit
- Higher fertility shrinks usable context
- Long-context models suffer more
- Token variance increases overflow risk
- Average values are optimistic

$$W(S) = \left\lfloor \frac{W}{\phi(S)} \right\rfloor$$

Effective Context Comparison

Tokens	Dev. Words	Roman Words	Loss
1024	132	433	301
2048	265	867	602
4096	531	1735	1204
8192	1062	3471	2409

- Hindi loses multiple paragraphs of context
- Loss increases as model size increases
- This issue is invisible in benchmarks

Attention Amplification Effect

- Transformer attention cost grows as $O(n^2)$
- Token inflation increases sequence length n
- Cost grows quadratically
- Independent of model size
- Pure tokenizer issue

$$\alpha = \rho^2 \approx (3.27)^2 \approx 10.7$$

- Devanagari needs over $10\times$ attention work
- Same meaning, higher compute cost

Romanization as a Practical Solution

- Romanization converts Hindi to Latin script
- Matches English-centered tokenizers
- Strongly reduces token fragmentation
- Restores lost context capacity
- Reduces attention computation
- Does not replace native script
- Works as a processing layer

Three-Stage Pipeline

- Stage 1: Devanagari $\rightarrow$ Roman (Transliteration)
- Stage 2: Romanized LLM processing
- Stage 3: Roman $\rightarrow$ Devanagari (Optional)
- LLM never sees Devanagari directly
- Token inefficiency fully avoided
- Modular and replaceable design

Experimental Setup Summary

Stage	Model	Data	Notes
1	LSTM seq2seq	1.3M words	Char-level
2	DistilGPT-2	912K lines	30K BPE
3	IndicXlit	Aksharantar	Transformer

- Controlled and reproducible setup
- External dependency only in Stage 3
- Baselines kept simple intentionally

Model Training Details

- Transliteration:
  - Character-level LSTM
  - 256 hidden units
  - 94.2% character accuracy
- Language Model:
  - DistilGPT-2 fine-tuned
  - 10 epochs training
  - Perplexity 24.5

Latency Analysis

Stage	Time (ms)
Dev $\rightarrow$ Roman	20
Roman GPT	376
Roman $\rightarrow$ Dev	2020

- Roman-only output 396 ms
- Full native output 2.4 seconds
- Back-transliteration is main bottleneck

Task-Level Trade-Offs

- Romanization works best for:
  - Text generation
  - Summarization
  - Question answering
- Native script works better for:
  - Logical reasoning
  - Causal inference
  - Grammar-sensitive tasks

Broader Impact

- Tokenizers treat scripts unfairly
- Non-Latin users pay higher cost
- Context windows become unequal
- Romanization is only a workaround
- Highlights infrastructure gap
- Problem affects many world scripts
- Calls for script-agnostic design

Conclusion

- Devanagari causes 3× token inflation
- Attention cost increases by 10×
- Romanization restores efficiency
- Near real-time Hindi LLMs are possible
- Quality depends on the task type
- Results are robust and reproducible
- Long-term solution is better tokenizers

Thank You