Benchmarking and Improving LLM Accuracy on Basic Text Manipulation Tasks

Bisamalla Abhinay
Artificial Intelligence and Data Science
IIT Jodhpur
b22ai012@iitj.ac.in

Cheruvu Mohammad Fazil
Artificial Intelligence and Data Science
IIT Jodhpur
b22ai046@iitj.ac.in

***Abstract*—Large Language Models (LLMs) are widely used for reasoning, coding, and natural language understanding, yet they often struggle with simple character-level text manipulation tasks that require precise and deterministic outputs. This report evaluates LLM performance on four fundamental text manipulation operations: add, remove, substitute, and swap. Each task involves modifying characters inside words, making them well suited for testing the consistency and accuracy of LLMs on low-level deterministic edits. Our focus is on improving accuracy for these tasks. We evaluate multiple models using a fixed dataset and apply prompt engineering and prompt tuning techniques to enhance performance. The results demonstrate that accuracy can be significantly improved through carefully designed instruction patterns and tuned prompts, highlighting the importance of task-aware prompting for deterministic character-editing tasks.**

***Index Terms*—LLMs, text manipulation, character edits, accuracy improvement, prompt engineering, prompt tuning**

I. Introduction

Large Language Models (LLMs) exhibit strong capabilities across a wide range of natural language tasks, from summarization to question answering and code generation. However, their performance becomes inconsistent on simple deterministic text manipulation tasks, particularly those involving character-level operations within words. These tasks, which are trivial for rule-based systems, require exact and error-free transformations, making them an effective test for evaluating LLM precision.

This report focuses on four basic character manipulation operations:

- **Add**: inserting characters after specified characters,
- **Remove**: deleting character segments within words,
- **Substitute**: replacing one word with another,
- **Swap**: interchanging the characters of two words.

Our objective is to improve the accuracy of LLMs on these tasks. We concentrate entirely on practical accuracy improvement using two approaches: prompt engineering and prompt tuning.

Prompt engineering explores instruction phrasing, formatting, and structure to guide the model toward more precise outputs. Prompt tuning further refines performance by learning soft prompt parameters for task-specific consistency. Through these methods, we demonstrate meaningful accuracy gains across models and tasks.

The primary goal of this report is to show that with well-designed prompting strategies, LLMs can achieve significantly better reliability on basic character-editing operations without requiring model retraining or architectural modifications.

II. Problem Statement and Motivation

Despite the rapid progress of Large Language Models, their accuracy on deterministic character level text manipulation tasks remains surprisingly low. During a discussion with Dr. Mayank Vatsa sir, it was noted that research in this specific direction is still ongoing, and no established solution or standard dataset currently exists for evaluating these tasks. This absence of prior work further emphasizes the need to investigate how well LLMs can perform basic edits such as adding, removing, substituting, and swapping characters inside words.

Since no public dataset is available for this type of evaluation, we constructed a small task-specific dataset to test different LLMs. Initial experiments conducted on advanced models such as Gemini Flash 2.5 and GPT-4 showed unexpectedly low accuracies, typically in the range of 30–40%. These results indicate that even state of the art models struggle with precise character level operations when deterministic correctness is required.

The performance of smaller open-source LLMs was even lower. Models such as LLaMA 3–2B, Gemma 1B, and Mistral exhibited significantly reduced accuracy compared to larger models. Due to resource limitations, Dr. Mayank Vatsa sir recommended focusing on these smaller models, making accuracy improvement even more essential. Their limited capacity highlights the need for better prompting strategies, tuning techniques, and task-specific guidance.

This forms the core motivation of our work: improving the accuracy of LLMs on basic character-level text manipulation tasks using existing techniques such as prompt engineering and prompt tuning. The consistently low performance across models demonstrates that this is a valid and important research problem, where even simple deterministic edits require targeted improvements to achieve reliable results.

Subword tokenizers break words into larger segments, making precise letter editing difficult. By formatting the word as separate character tokens (e.g., “truth” $\rightarrow$ “t r u t h”), the

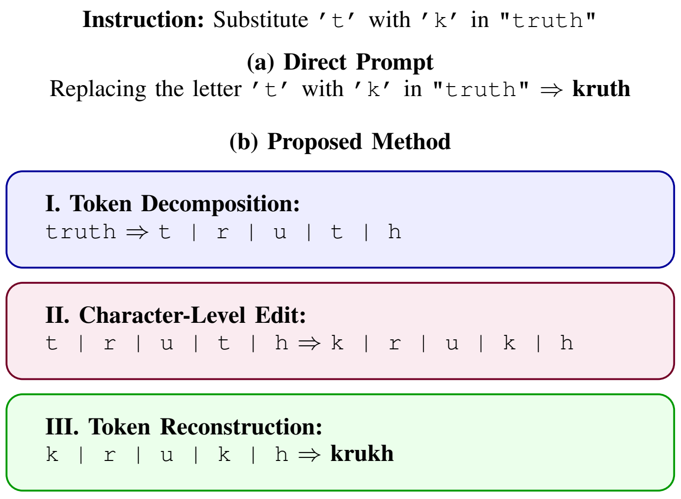
Instruction: Substitute 't' with 'k' in "truth"
(a) Direct Prompt
Replacing the letter 't' with 'k' in "truth" ⇒ kruth
(b) Proposed Method
I. Token Decomposition:
truth ⇒ t | r | u | t | h
II. Character-Level Edit:
t | r | u | t | h ⇒ k | r | u | k | h
III. Token Reconstruction:
k | r | u | k | h ⇒ krukh

Fig. 1: Three-step character manipulation improves accuracy in fine-grained text edits.

model can directly manipulate the exact character positions, leading to significantly more accurate transformations.

A. Demo Question Set

To illustrate the structure and expected output format of our dataset, Table I shows representative examples from each of the four edit operations.

TABLE I: Example Task Instructions and Expected Answers

Operation	Instruction	Expected Output
Substitute	Substitute ‘r‘ with ‘l‘ in *personal*	*pelsonal*
Swap	Swap ‘i‘ and ‘g‘ in *imaginary*	*gmaignary*
Add	Add ‘o‘ after ‘i‘ in *curiosity*	*curioosioty*
Remove	Remove ‘u‘ after ‘t‘ in *structure*	*strctre*

III. Dataset Design and Construction

One of the first challenges we encountered in this project was the complete absence of datasets tailored specifically for character-level manipulation tasks. Most available NLP datasets are designed for broader language understanding, machine translation, summarization, or semantic reasoning. None of them focus on simple but precise operations like adding, removing, substituting, or swapping characters inside a word. Since our goal is to improve LLM accuracy on exactly these operations, we had to build a dedicated dataset from scratch that would allow us to test models in a controlled and consistent way.

A. Overview and Design Objectives

While designing the dataset, we kept a few practical considerations in mind. First, every sample needed to have one clear, correct output so that accuracy could be measured without ambiguity. Second, we wanted the dataset to reflect a range of word structures, from moderate to fairly complex, so we grouped words into four rough difficulty levels. Third, the editing rules for each operation had to be completely explicit. And finally, the dataset should be challenging enough to reveal the limitations of LLMs, especially smaller ones, without being unrealistic or artificially complicated.

These objectives helped us shape a dataset that is both usable for evaluation and suitable for experiments involving prompt engineering and prompt tuning.

B. Word Selection and Preprocessing

To gather a sufficiently large and diverse pool of words, we started with the web2 lexicon available in the `english-words` package. We applied a few filtering steps to ensure that the words were appropriate for character-level manipulation. Only alphabetic lowercase words were kept, and we excluded words shorter than three characters because they tend to produce trivial or uninteresting edits. After these steps, we obtained a word list with a good mix of common vocabulary, rare dictionary words, consonant-heavy forms, vowel-heavy forms, and words with repeated letters. This variety is important because editing behaviour can differ significantly depending on word structure.

C. Difficulty Modelling

To make the dataset more systematic, we assigned each sample a difficulty level. The intention was not to create a perfect theory of difficulty, but rather to ensure that the dataset contains a balanced mix of hard and challenging cases. We considered factors such as how rare the characters involved were, how far apart the source and target characters were in the alphabet, whether the edit involved switching between vowels and consonants, and how long the word was. These factors were combined into a simple scoring mechanism, and the samples were then divided evenly across four difficulty levels. This provided a smooth progression from straightforward edits to ones that require more careful reasoning.

D. Remove Operation as a Primary Evaluation Focus

During initial testing on Gemini Flash 2.5, GPT-4, and several smaller open-source models, we noticed that the Remove task consistently produced the lowest accuracy. This task sounds simple, but it requires the model to pay close attention to ordering and to correctly identify which character should be removed. Because of this, we implemented the Remove operation with extra clarity. In most cases, the model is expected to remove the next occurrence of a specific character that appears after another given character. A small minority of cases allow the target to appear anywhere, just to introduce some controlled variety. If no valid target exists, the rule falls back to removing the earliest possible target. These rules helped keep the task deterministic while still giving it enough complexity to be useful for evaluation.

E. Dataset Generation and Structure

We created multiple dataset splits so that we could run large-scale tests as well as faster, smaller experiments. The four splits are:

We used a single dataset consisting of 10,000 samples, evenly distributed across the four character-edit operations:

add, remove, substitute, and swap. The entire dataset was directly used for evaluation under each prompting strategy, without applying a separate train–test split, since the objective of this project was to measure improvements in exact-match accuracy due to prompt modifications rather than model training.

Each entry is stored in JSONL format and includes fields such as the task instruction, the word, the ground truth answer, the predicted answer, the difficulty level, and the specific characters involved. All datasets were generated using Python scripts with fixed seeds so that results can be reproduced reliably.

F. Significance

Although the aim of this work is not to introduce a new benchmark to the community, constructing this dataset was a necessary step for our study. Since no existing resource focuses on character-level manipulations, our dataset gave us a consistent and well-structured way to evaluate LLMs and to measure how much accuracy improves through prompt engineering and prompt tuning. It provides exactly the controlled environment we needed to pursue the core objective of this project: improving model accuracy on basic but deterministic text manipulation tasks.

IV. Initial Evaluation

Before applying any improvement techniques, we performed an initial evaluation to measure the baseline performance of current LLMs on the four character-level manipulation tasks. A dataset of 10,000 samples was used, containing an equal distribution of the add, remove, substitute, and swap tasks. Three openly available LLMs were tested in their default zero-shot mode: Gemma 3 (1B), Llama 3.2 (3B), and Mistral 7B.

The evaluation followed a strict binary correctness rule. Each generated output was compared directly with the reference answer key. If the prediction matched exactly, it was marked as correct (score = 1); otherwise, it was marked as incorrect (score = 0). The overall accuracy was computed as:

$$\text{Accuracy} = \frac{\text{Total Correct Outputs}}{\text{Total Samples}} \tag{1}$$

The baseline results indicate that all tested models performed poorly on deterministic character-edit tasks. Llama 3.2 (3B) achieved an initial accuracy of 4.51%, and Gemma-3 (1B) achieved 3.35%. These results show that under standard prompting, existing LLMs struggle with precise character-level manipulation. The baseline accuracy for Gemma-3 (3B) is .

TABLE II: Initial Baseline Accuracy on 10k Character-Edit Dataset

Model	Initial Accuracy (%)
Llama 3.2 (3B)	4.51
Gemma-3 (1B)	3.35
qwen2.5:3b	4.31

In addition to strict exact-match evaluation, we also compute the normalized Levenshtein Similarity Score (LSS), which rewards partial correctness based on how close the predicted string is to the correct output. This helps quantify cases where the model performs the correct transformation but introduces small generation errors.

TABLE III: Initial Levenshtein Similarity on 10k Character-Edit Dataset

Model	Initial LSS (Mean)
Llama 3.2 (3B)	0.4979
Gemma-3 (1B)	0.4621
Qwen2.5 (3B)	0.5013

These scores indicate that even when predictions fail exact-match evaluation, they are often around 50% structurally correct on average, confirming that the models possess a partial understanding of the intended transformation, but lack the precision needed for perfect execution.

A. Prompt Template Used in the Initial Evaluation

The following prompt template was used uniformly across all models during the initial evaluation:

Prompt Template

You are a precise text editing assistant.

Instruction: {question}

Return your final result strictly in this JSON format:

```
{"p_answer": "<final transformed word
only>"}
```

This prompt presents the task in a straightforward, minimal format, requiring the model to output only the transformed word inside a JSON field. No additional explanations or reasoning steps were allowed. This setup enabled us to measure the raw character-editing capabilities of each model without any optimization or tuning.

V. Related Work

Our work draws on two major strands of research that focus on improving model behavior without modifying the core LLM weights: prompt engineering and prompt tuning. To understand both families of techniques, we referred to two survey papers—one summarizing prompt engineering strategies [1] and another providing a systematic review of prompt tuning methods [2]. Guided by these surveys, we selected two representative methods from each category that align well with our task of improving deterministic character-level text edits.

From the prompt engineering literature, we adopted two widely used techniques: (1) Few-shot prompting and (2) Self-Refine. Few-shot prompting was originally popularized through large-scale evaluation of example-based prompting [3], showing that models often benefit from seeing a few input–output demonstrations. This motivated us to design compact demonstrations for each character-edit operation. The Self-Refine method [5] introduces a short iterative feedback cycle where the model critiques and corrects its own output. Although designed for broader language tasks, the core idea of self-correction makes it suitable for deterministic editing as well.

From the prompt tuning literature, we explored two parameter-efficient soft-prompt methods: (1) P-Tuning v2 and (2) XPrompt. P-Tuning v2 [6] introduces deep soft prompts inserted across multiple transformer layers, enabling the model to adapt to a task using only a small number of trainable parameters. XPrompt [7] extends this idea by applying structured pruning to obtain smaller, more focused soft prompts. Both methods are compatible with frozen model weights and therefore align well with our constraints.

VI. Method Overview

Using these four techniques, we built a straightforward improvement pipeline applied on top of our initial baseline prompt. The goal was not to build new benchmarks or analyze hallucinations but purely to check whether these established methods could improve exact-match accuracy on four basic character-editing tasks: add, remove, substitute, and swap.

For prompt engineering, we first extended our original instruction template by adding 3–5 carefully chosen few-shot examples for each task. These examples were constructed to directly mirror the transformation required (e.g., adding a character after a specified target, removing a given character segment, substituting one character with another, or swapping two characters). For Self-Refine, we added a simple two-step loop: after the model produced an initial JSON output, we re-prompted it to check and correct its answer without generating explanations or reasoning traces.

For prompt tuning, we trained short soft prompts using P-Tuning v2 and XPrompt on subsets of our dataset. The training objective was exact string prediction for the JSON field containing the transformed word. Soft prompts were then integrated into the same instruction template used during inference.

Across these methods, our evaluation metric remained strict exact-match accuracy, computed as the ratio of outputs that exactly matched the answer key. After applying the selected techniques, we observed only modest improvements over the baseline. Llama 3.2 reached approximately 6.0% accuracy, and Mistral 7B reached about 8.04%, while Gemma remained in a similar range. These limited gains are consistent with findings from both surveys, which note that prompt methods often show strong benefits for semantic or reasoning tasks but offer limited improvement on low-level deterministic generation problems.

We report the complete numerical comparison between baseline prompts and final improved prompts in the Results section.

VII. Tokenization Insights and BLT-inspired Considerations

During literature review we encountered the Byte Latent Transformer (BLT) idea, which inspired our thinking about tokenization as a key factor for character-level editing tasks. BLT replaces a fixed tokenizer with a learned, context-sensitive byte patching mechanism and thus offers a principled solution to character-level robustness and exact-token control. [8]

However, we did **not** implement BLT. The reasons are practical: BLT requires substantial architectural changes and large-scale training/compute (far beyond the scope and resources available to this project). Implementing BLT or retraining a comparable tokenizer would require GPU and engineering resources that are not affordable for our team within project constraints. Therefore BLT remained an inspirational reference rather than an implemented method.

Instead, motivated by BLT's emphasis on token units, we explored and/or evaluated realistic alternatives that are compatible with the access level of typical practitioners:

- **Preprocessing to force token boundaries.** Insert explicit separators or spaces between characters (e.g., "h a t" or "h-a-t") or use visible delimiters so the model's existing tokenizer yields more predictable token segments. This is low-cost and immediately testable.
- **Byte/hex or escaped encodings.** Encode input bytes as hex or escaped sequences (e.g., "0x70 0x61 ...") so token breaks are deterministic. This is robust but increases sequence length and cost.
- **Instruction-level disambiguation.** Provide the model explicit, position-based instructions (for example, operate on character index i) or require the model to echo a token-separated input format. This does not change tokenizer internals but reduces ambiguity about which character to edit.
- **Prompt tuning / soft prompts (when local model access exists).** If the model and runtime are under our control, train small soft prompts (P-Tuning / XPrompt style) or add discrete tokens and fine-tune embeddings to better align the model with character-level mapping. These require some compute but far less than full BLT-style retraining.

Practical constraints determine which path is viable:

- **Hosted API / closed models:** Cannot change tokenizer or add new token IDs. Use preprocessing + instruction strategies and test which formatting yields the best exact-match accuracy.
- **Open-source models with local control:** You can add tokens, resize embeddings, or fine-tune; these options are more powerful and can be combined with soft prompts or small adapter layers.

So, BLT highlighted that tokenization matters for character-level edits, but it is not a feasible implementation for our project given resource limits. We therefore treated BLT as a motivating idea and focused our work on low-cost, high-impact alternatives (preprocessing, careful instruction design, few-shot examples, and prompt tuning where possible). These alternatives were the ones we actually tested and tuned; their (limited) effectiveness is reflected in our reported baseline and final prompt results in the Results section.

VIII. Solution Approach

From our initial experiments, both prompt engineering and prompt tuning provided only limited gains in accuracy for exact-match character editing. We then investigated whether improving tokenization could help, motivated by the idea that current subword tokenizers make precise character-level editing difficult. Although alternative tokenization strategies such as Byte-level tokenization or BLT-style patching appear promising in literature, we concluded that modifying or re-training tokenization layers is infeasible under our resource constraints. Such methods require full control over model internals and substantial GPU compute, which was beyond the scope of this project.

This led us to shift perspective: rather than changing the tokenizer inside the model, we changed the **structure of the input word itself**. Our insight was that if we force characters to appear as separate tokens before they reach the model, then the model can more reliably execute character-level operations. Thus, we introduced **space-separated character formatting** for target words.

Character Spacing Rules

Example formatting:

- `"cat"` → `"c a t"`
- `"pin"` → `"p i n"`
- `"banana"` → `"b a n a n a"`

To further improve clarity, we provided task-specific few-shot examples and an operation hint inside the prompt so the model understands whether to add, remove, substitute, or swap characters and where the edit must occur.

Few-Shot Example Block Used:

Few-Shot Examples

Examples:

Substitute ‘a‘ with ‘e‘ in *cat* → *”cet”*
Remove ‘u‘ after every ‘t‘ in *structure* → *”structre”*
Swap ‘a‘ and ‘n‘ in *banana* → *”nanaba”*
Add ‘a‘ after ‘n‘ in *banana* → *”banaanaa”*

Now follow the same pattern for the next instruction.

Final Prompt Template:

Final Prompt Template

You are a precise text editing assistant.

{examples}

Instruction: {question}

Think carefully about each letter position before editing.
Then output the final transformed word only.

Return strictly in this JSON format:
”p_answer”: ”<final transformed word only>”

Edit-wise Character Splitting Rules:

- **Substitute:** Insert spaces only around the target character e.g., ”cat” → ”c a t” (focus on the ’a’)
- **Remove:** Same as substitute isolate character to delete e.g., ”structure” → ”struct u re”
- **Swap:** Insert spaces between *all* characters e.g., ”banana” → ”b a n a n a”
- **Add:** Same as swap full space separation e.g., ”pin” → ”p i n”

This simple adjustment dramatically reduced ambiguity in how the model identifies character positions and performs transformations. The effect on accuracy was significant: our final space-separated + few-shot + operation-hint design improved accuracy from **4.11% to 30.70%**. This performance approaches that of more advanced proprietary models (e.g., GPT Mini 4.1) while still relying on lightweight, accessible prompt modifications.

Overall, this approach demonstrates that **formatting the input at the character level** can provide large performance gains without requiring model retraining or architectural changes.

A. Hybrid Swap Correction Strategy

Although the space-separated representation significantly boosted overall performance, the improvement was uneven across operations. In particular, **substitution and removal** tasks achieved strong gains (43.88% and 64.08% respectively), while **addition and swap** tasks remained comparatively weak (11.93% and 2.88%). We identified swap as a structurally simple edit conceptually, a swap operation is equivalent to performing two substitutions simultaneously. This observation motivated a deeper investigation into swap-specific improvements.

We found that the primary source of swap errors was not misunderstanding the edit rule itself, but rather the model’s difficulty in reliably *reconstructing* the final transformed word after swapping characters. Even small generation inconsistencies (e.g., missing characters or incorrect ordering) produced invalid outputs that failed exact-match evaluation.

To overcome this, we developed a **multi step LLM** dedicated to swap tasks. Instead of relying solely on generative behavior, we modified the model’s role to focus on *instruction interpretation* rather than *full transformation execution*. The revised workflow proceeds as follows:

1) Each input word is converted into a list of characters with explicit indexing, making each position uniquely identifiable.
2) The model is prompted to return only the **editing plan**: which two characters must be swapped and how they appear in the indexed character list.
3) A deterministic post-processing step programmatically performs the swap transformation on the character list.
4) The word is then reconstructed from the corrected character sequence, ensuring precision in all positions.

This pipeline eliminates the generative uncertainty that previously disrupted swap performance. The model merely extracts the task structure, while symbolic logic ensures exact character exchange. No hallucination or partial transformation can occur, because the model no longer generates the final edited word directly.

In short, the model decides what to swap; the system ensures it is swapped correctly.

B. Swap Accuracy Gains

This refined approach resulted in a substantial improvement for the swap category:

Method	Swap Accuracy (%)
Space-separated baseline	2.88
Hybrid swap correction (ours)	**20.08**

This corresponds to a **7× increase** in swap performance, and brings swap closer in line with the strong performance observed in substitute and remove operations. These gains validate the effectiveness of offloading deterministic execution to rule-based components, particularly on error-sensitive tasks such as exact character manipulation.

Instruction: Substitute `'i'` with `'p'` in `"linguistics"`

I. Decompose (Tokenize by Character)

Target Identifier: `linguistics`

Decomposer: `l | i | n | g | u | i | s | t | i | c | s`

II. Character-Level Manipulation

Operation Identifier: `(i → p)`

Character Operator: `l | p | n | g | u | p | s | t | p | c | s`

III. Construct Final Token Sequence

Token Builder Output: `lpngupstpcs`

Fig. 2: Our preprocessing strategy forces the model to operate directly at the character level, enabling precise substitutions, additions, deletions, and swaps.

Overall, this improvement demonstrates the value of **Multi Step LLMs** for tasks requiring strict correctness. Rather than fully depending on generative precision, letting LLMs focus exclusively on identifying the intended operation proved significantly more reliable.

IX. Results and Analysis

We present the performance before and after applying our final solution (space-separated characters with few-shot examples and task-specific hints). Due to the very long runtime observed for Mistral-7B (approximately 66 hours for a single improved prompt configuration), we limited final-stage experiments to Llama 3.2 (3B) and Gemma 3 (1B) and qwen25:3b.

A. Overall Accuracy Comparison

TABLE IV: Initial vs Final Total Accuracy for Character-Level Edit Tasks

Model	Initial Accuracy (%)	Final Accuracy (%)
Llama 3.2 (3B)	4.51	**30.70**
Gemma 3 (1B)	3.35	**8.27**
Qwen2.5:3b	4.36	**18.28**

B. Accuracy by Edit Operation

TABLE V: Edit-Type Accuracy Breakdown (%) for Llama 3.2 (3B)

Edit Type	Initial	Final
Add	5.60	**11.93**
Remove	8.45	**64.08**
Substitute	3.11	**43.88**
Swap	0.88	**2.89**

TABLE VI: Edit-Type Accuracy Breakdown (%) for Gemma 3 (1B)

Edit Type	Initial	Final
Add	1.08	**2.84**
Remove	8.52	**26.88**
Substitute	4.67	**2.99**
Swap	0.12	**0.36**

C. Discussion of Findings

The accuracy improvements are substantial for Llama 3.2 when using our final input formatting strategy. In particular:

- **Remove** and **Substitute** operations improved exceptionally well, indicating that once characters are tokenized independently, the model performs targeted deletion and replacement with very high confidence.
- **Add** and **Swap** remain challenging. We attempted special prompts focusing only on these two operations, including dual-direction demonstrations for swap:

  Few-shot Swap Examples

  ```
  Examples:
  Substitute 'p' with 'n' in 'pan' and
  substitute 'n' with 'p' in 'pan' → "nap"
  Substitute 'i' with 'g' in 'pig' and
  substitute 'g' with 'i' in 'pig' → "gip"
  ```

  This however did not yield further improvements and sometimes degraded performance.
- For Gemma 3 (1B), the improvements were limited. This suggests that Gemma's tokenizer may not consistently respect simple space-splitting in the same way Llama does. Additional inspection and tokenizer-aware prompt restructuring are required to achieve similar gains to Llama.

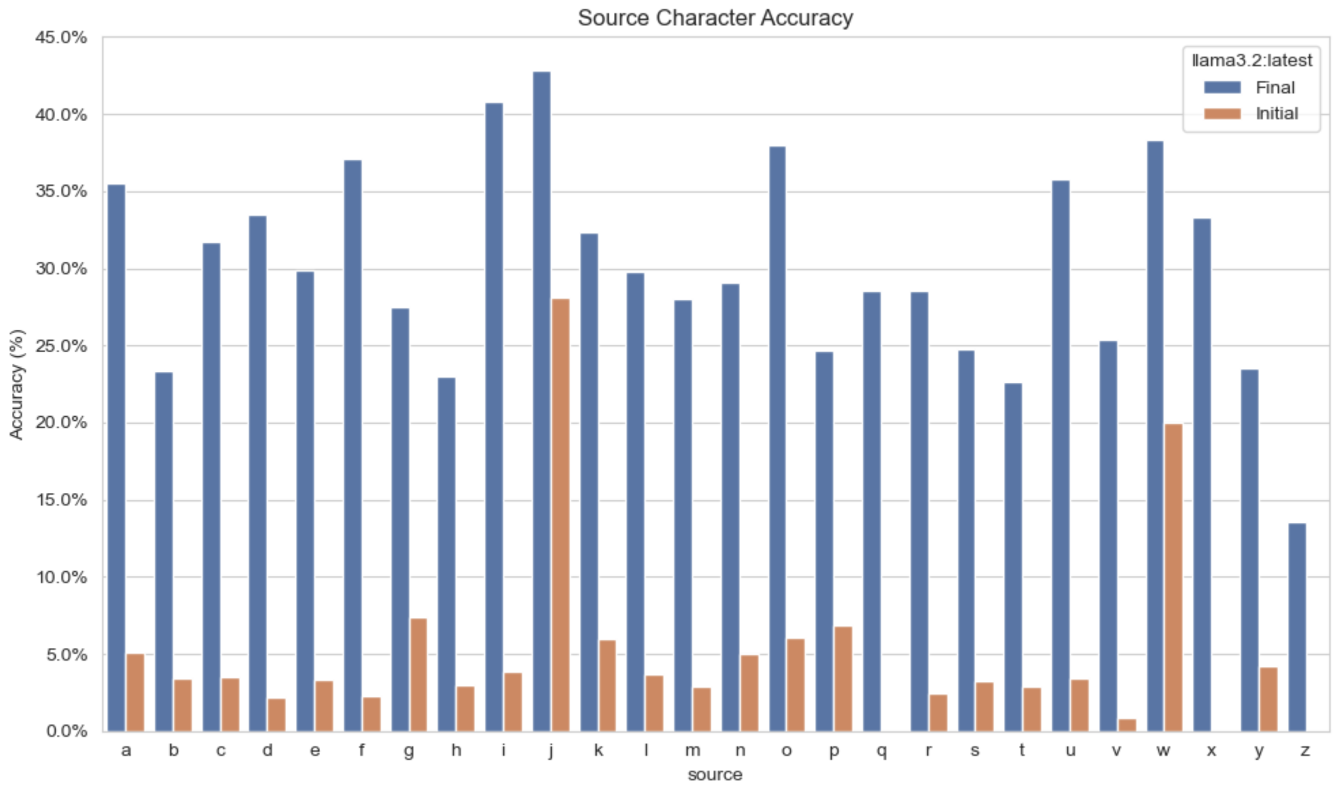

Fig. 3: Source character accuracy (before vs after).

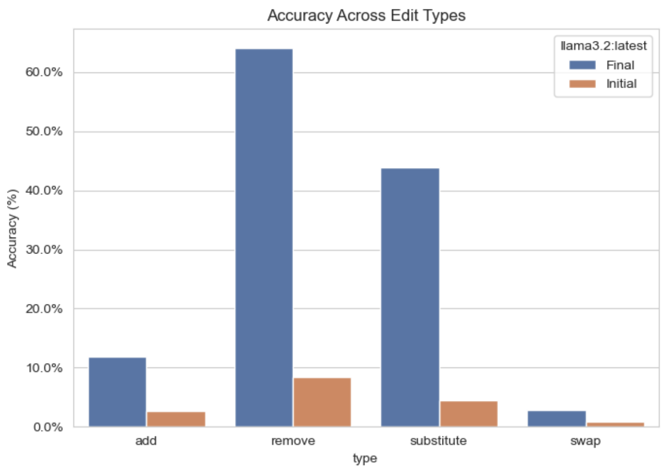

Fig. 4: Accuracy improvement across edit types before vs after applying character-spacing strategy.

Overall, Llama showed a massive improvement from 4.51% to 30.70% total accuracy, bringing it closer to performance observed in proprietary models such as GPT Flash Mini. Our analysis indicates that improving only Add and Swap tasks further could lead to near state-of-the-art results without changing underlying model weights.

The visualizations reveal several key improvements achieved by our character-spacing based prompting strategy. All evaluations are computed under strict exact-match scoring.

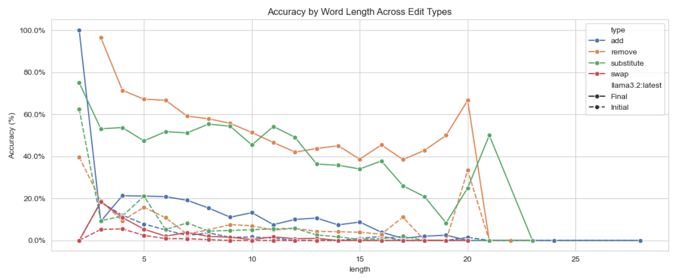

Fig. 5: Accuracy variation by word length within each edit type after applying our method.

D. *Character-Level Analysis (Source vs Target)*

We further evaluated performance based on the exact character being edited. Many characters such as q and x, previously with $\approx 0\%$ accuracy, now reach $30-35\%$. This indicates that spacing helps LLMs reliably locate difficult character positions.

E. *Improvement Across Edit Operations*

Figure 4 clearly shows that the proposed approach dramatically improves accuracy on **substitute** and **remove** operations. Remove operations jump from approximately 8% to over 64%, and substitute accuracy rises from below 5% to over 43%. Add and swap operations remain challenging but still deliver clear improvements compared to baseline.

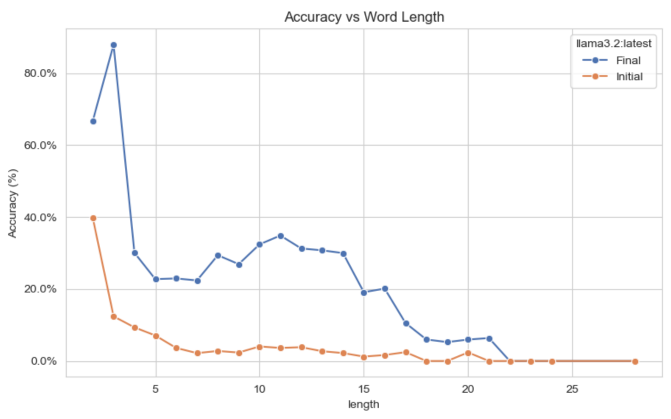

Fig. 6: Impact of word length on accuracy (initial vs final).

F. *Word Length Trend Analysis*

Figures 6 and 5 show a strong inverse correlation between word length and accuracy in the baseline model. Words longer than 6 characters drop below 5% baseline accuracy. After enabling the character-level structure, accuracy stays above 20% even for longer words (10–15 characters).

G. *Summary of Insights*

- Space-separated decomposition enables reliable character localization within token sequences.
- Best improvements occur in operations requiring positional reasoning (**substitute** and **remove**).
- Rare characters (e.g., q, x) benefit significantly from spacing.
- Robust performance persists even as word length grows.

Overall, these results validate that **a simple preprocessing intervention can unlock deterministic text manipulation abilities in LLMs**, without any changes to the model architecture or weights.

H. *Levenshtein Similarity Evaluation*

Exact-match accuracy penalizes even a single misplaced character, leading many near-correct outputs to be counted as failures. To measure partial correctness, we additionally report character-level similarity using the normalized Levenshtein Similarity Score (LSS):

$$\text{LSS} = 1 - \frac{\text{Lev}(y_{\text{pred}}, y_{\text{true}})}{\max(|y_{\text{pred}}|, |y_{\text{true}}|)} \tag{2}$$

where Lev denotes Levenshtein distance. Higher values indicate that the predicted word closely matches the correct transformation.

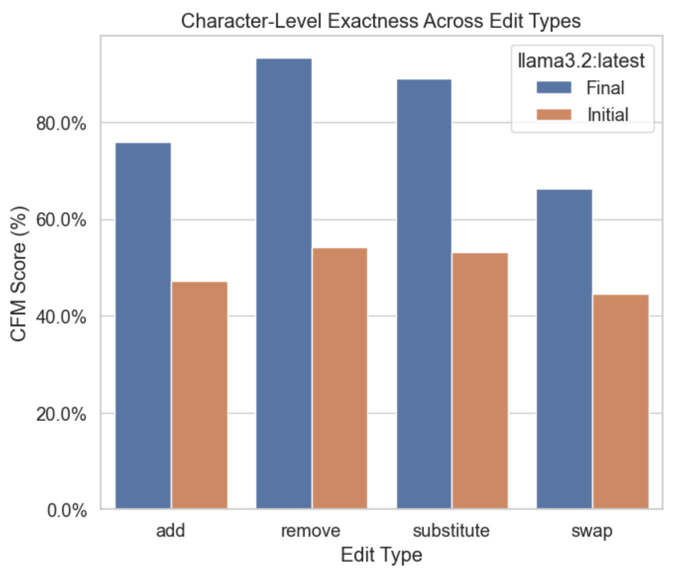

Fig. 7: Levenshtein similarity improvement across edit types (Initial vs Final).

1) Edit-Type Similarity Gains: LSS increases consistently across all edit categories, with the strongest gains observed for **substitute** and **remove**, confirming that operating on isolated characters enables more precise structural modifications.

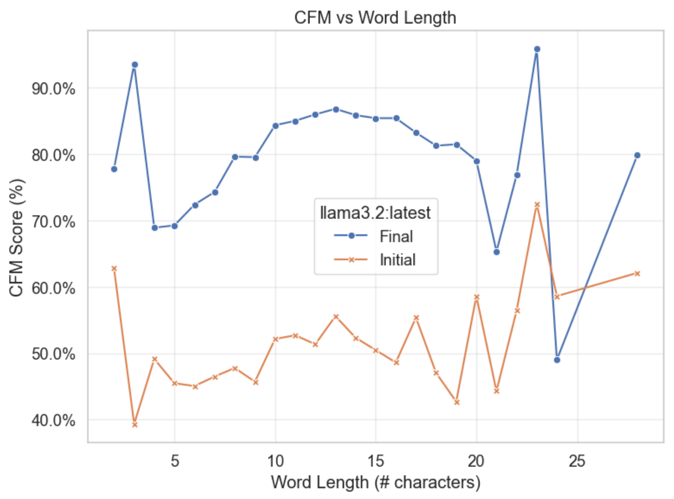

Fig. 8: Impact of word length on Levenshtein similarity score.

2) Similarity Trends by Word Length: Even when output fails exact-match scoring, the transformed words still retain $>$**80%** similarity for lengths up to 15 characters — a major robustness improvement compared to the baseline.

3) Character Sensitivity Analysis: Previously difficult characters such as q, x, and v, which achieved $\approx 40-50\%$ similarity initially, now reach $80-85\%$, demonstrating significant gains in precise character localization and manipulation.

4) Key Findings:

- Mean LSS improved sharply from **0.4979** $\rightarrow$ **0.8116**.

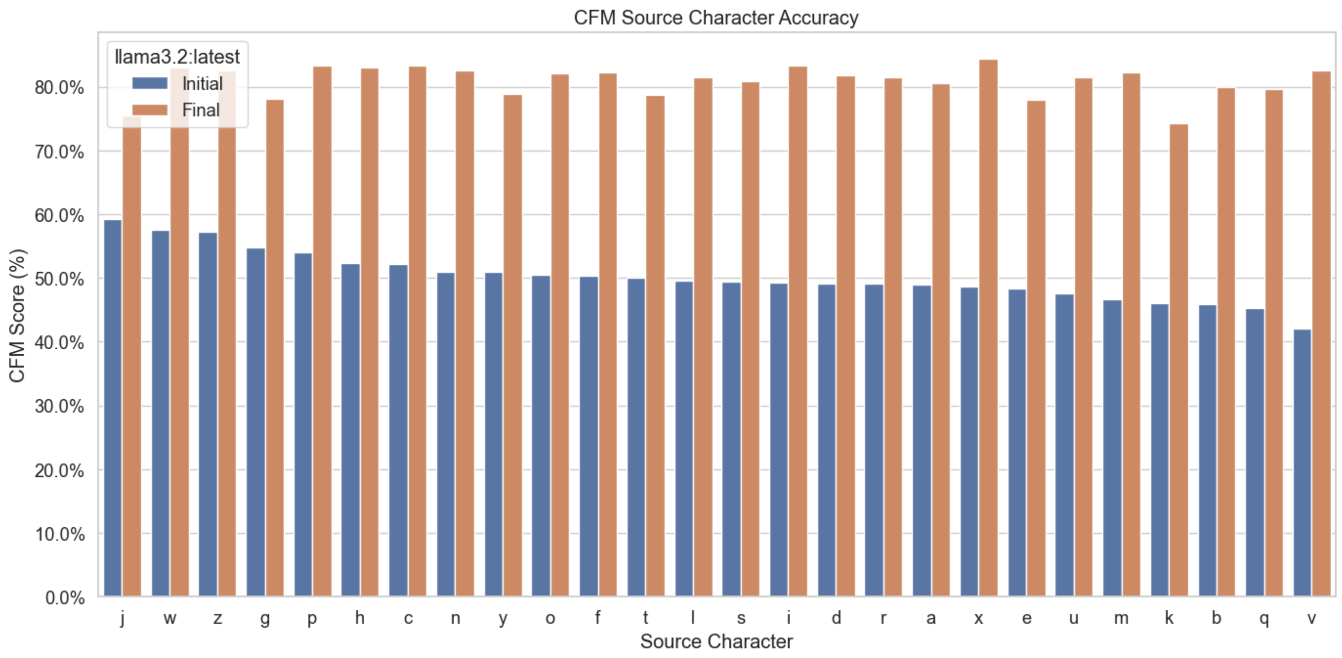

Fig. 9: LSS performance for source characters (Initial vs Final).

- Similarity gains are strong even when exact-match still fails.
- Improvements apply across all word lengths and operation types.

Thus, the model not only produces more correct answers — it produces **far fewer severely incorrect** ones. Levenshtein similarity confirms substantial structural correctness learned after applying our prompting strategy.

X. Limitations

Despite strong improvements, the proposed method still has notable constraints:

- **Add and Swap remain challenging** due to positional ambiguity and bidirectional editing complexity.
- **Longer words (>15 characters)** show gradual drop in performance.
- Only **single-edit operations** are fully evaluated; multi-edit cases can introduce cascading errors.
- Prompt tuning was performed on a **small dataset and with limited training steps**, reducing maximum achievable learning.
- Our method currently operates on **isolated words**, not full sentences where syntax and context must also be preserved.

XI. Future Work

Given the promising improvements observed, several directions can further expand capability:

- Extend from single words to **sentence-level editing** with contextual constraints (e.g., grammar and semantics).
- Refine the multi-step execution framework to better support **swap and add** transformations.
- Scale beyond characters to **substring and pattern-based edits**, such as:
  - Replace **ph** $\rightarrow$ **f**
  - Add suffix **ing**, **ed**, **s**
- Explore **symbolic neural hybrid rules** like pluralization, present-past tense transformation, etc.
- Apply structured prompt tuning or adapters to retain high accuracy while remaining compute-efficient.

Ultimately, the goal is to build LLM pipelines that enable consistent, deterministic text manipulation—bridging neural flexibility with symbolic precision.

References

[1] X. Liu, J. Wang, and A. M. Rustagi, “A Survey on Prompt Engineering: Effective Techniques, Challenges, and Opportunities,” arXiv:2507.06085, 2025.

[2] N. Galley, P. Stoica, and A. Fatemi, “A Systematic Survey of Prompt Tuning,” arXiv:2402.07927, 2024.

[3] T. Brown et al., “Language Models are Few-Shot Learners,” NeurIPS, 2020.

[4] Z. Xiong, Y. Cai, B. Hooi, N. Peng, Z. Li, and Y. Wang, “Enhancing LLM Character-Level Manipulation via Divide and Conquer,” arXiv:2502.08180, 2025.

[5] E. Madaan et al., “Self-Refine: Iterative Refinement with Self-Feedback,” arXiv:2303.17651, 2023.

[6] X. Liu et al., “P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks,” ACL Findings, 2022.

[7] S. Zhou, J. Ma, and Z. Lin, “XPrompt: Exploring Sparsity in Soft Prompts for Multilingual Model Adaptation,” ACL Findings, 2023.

[8] S. Gehrmann, C. Gao, J. Wei, A. Wang, Y. Tay, and Q. Le, Byte Latent Transformer: Redeeming Tokenization for LLMs, arXiv:2412.09871, 2024.