1 Details of Perturbation

For each category, we illustrate its description and how we implement the perturbation. While illustrating the implementation of perturbations, we refer to the times of perturbations as "T" (i.e., the perturbation is applied T times for each prompt). T is calculated based on the product of: (1) the number of perturbable elements and (2) a pre-set frequency value (i.e., what percent of the perturbable elements will be perturbed). We set the default frequency value for each category based on our observation on the results and attempt to achieve a balance of naturalness and effectiveness. The detailed frequency of each category can be found in our replication package [1].

1.1 Categories of Addition.

The categories of Addition are related to adding an element (e.g., a character, a word, or a space) to the prompt.

A1: Extra Space outside Words. Add an extra space outside the words (i.e., before or after a word).
Implementation. NLPerturbator identifies the locations of word boundaries such as the start and end of the prompt or spaces. NLPerturbator then randomly selects T locations to add an extra space. We use spaCy [2] to extract English words from the prompt for this and the remaining categories.

A2: Extra Space inside Words. Add an extra space inside a word, resulting in the original word being split into two words.
Implementation. NLPerturbator randomly selects T words that are larger than 3 characters and adds a space inside the selected words. NLPerturbator only inserts a space between the characters of a word (i.e., does not insert space at the start or end of a word).

A3: Repeated Words. Repeat an existing word in the prompt, especially short words like *"to"*, and *"a"*.
Implementation. NLPerturbator first identifies short words in the prompt that are not larger than three characters and then randomly selects T such words and repeats each selected word once.

A4: Repeated Chars. Repeat a character in the prompt.
Implementation. NLPerturbator randomly selects T words in the prompt and repeats a random character in each selected word.

1.2 Categories of Deletion.

The categories of Deletion are related to the removal of an element in the prompt.

D1: Char Deletion. Drop a character from the prompt.
Implementation. NLPerturbator first identifies the characters that meet the following requirements: (1) reside in a word with at least three characters; (2) not at the start or the end of a word; (3) are lowercase alphabets. NLPerturbator then randomly drops T such characters from the prompt.

D2: Preposition Deletion. Drop a preposition or a subordinating conjunction (e.g., *"to"*, *"of"*) from the prompt.
Implementation. NLPerturbator leverages spaCy [2] to identify the part of speech for each word and then randomly drops T prepositions and subordinating conjunctions in the prompt.

D3: Determiner Deletion. Drop a determiner (e.g., *"a"*, *"the"*) from the prompt.
Implementation. NLPerturbator randomly drops T prepositions identified by using spaCy [2].

D4: Space Deletion. Drop a space between the words from the prompt.

Implementation. NLPerturbator identifies the whitespace characters between the words. Such space can include the space on the keyboard, a newline mark, and a tab mark. NLPerturbator then randomly drops T spaces from the prompt.

1.3 Categories of Editing.

The categories of Editing are related to editing a character or a word in the prompt.

E1: Keyboard Typo. Replace a character in the prompt with its adjacent character on the keyboard.

Implementation. NLPerturbator identifies the lowercase alphabetic characters in the prompt and randomly selects T characters. For each selected character, NLPerturbator then randomly replaces it with a character that is adjacent on the keyboard. The retrieval of adjacent characters is based on our predefined mappings. For example, if the original character is *"r"*, NLPerturbator then randomly selects from *"e"*, *"d"*, *"f"*, and *"t"* to replace it. Note that our implementation is based on the standard layout (i.e., QWERTY layout) of keyboard.

E2: Extra Capital Letter. Capitalize a lowercase character .

Implementation. NLPerturbator randomly selects T lowercase alphabetic characters and capitalizes them.

E3: Grammatical Person Variation. Convert a verb between its base form and its third person singular.

Implementation. NLPerturbator identifies the verbs in the prompt and randomly selects T of them. Then NLPerturbator interchanges these verbs between their original and third person singular forms with the help of spaCy [2] and LemmInflect [3].

E4: Active and Passive Voice Variation. Convert a verb between active voice and passive voice.

Implementation. NLPerturbator identifies the verbs in the prompt and randomly selects T of them. If the word is in the past participle form, NLPerturbator converts it to the active form; and vice versa. NLPerturbator accomplishes it by spaCy [2] and LemmInflect [3].

E5: Word Class Variation. Convert a word to its different part of speech.

Implementation. NLPerturbator randomly selects T words and then transforms their word classes. NLPerturbator converts words randomly among the forms of nouns, verbs, adjectives, and adverbs.

E6: Synonym Substitution. Substitute a word to its synonym.

Implementation. NLPerturbator identifies substantive words (i.e., prepositions, determiners, etc. will be excluded) and randomly selects T words. Then NLPerturbator replaces these words with their synonyms using WordNet [4], which is widely used by prior studies to retrieve synonyms [5–7].

1.4 Categories of Swap.

The categories of Swap are related to the swap of characters or words in the prompt.

S1: Swap Adjacent Chars. Swap two characters that are adjacent in a word.

Implementation. NLPerturbator first identifies the characters that meet the following requirements: (1) reside in a word with at least three characters; (2) not at the start or the end of a word; (3) are lowercase alphabetic characters. NLPerturbator then randomly selects T characters to swap with their next character in the word. While randomly selecting the candidate characters to swap, NLPerturbator only selects non-adjacent characters to swap with their next characters to avoid consecutive swaps in the same word.

S2: Swap Adjacent Words. Swap two words that are adjacent in the prompt.

Implementation. NLPerturbator randomly selects T words in the prompt to swap with their next words. Similar to S1, NLPerturbator only selects non-adjacent words to swap with their next words to avoid consecutive swaps.

1.5 Categories of Paraphrasing.

P1: Rephrasing Sentence. Rephrase the prompt while preserving the original semantic meaning.
Implementation. NLPerturbator splits the prompt into sentences through punctuations and randomly selects T sentences. NLPerturbator then utilizes paraphraser Parrot [8] to rephrases them.

P2: Declarative to Interrogative. Convert the request in the prompt from declarative to interrogative.
Implementation. NLPerturbator identifies the imperative sentence in the prompt (e.g., "Write a program...") and leverages the API of OpenAI LLMs [9] to convert to an interrogative sentence.

1.6 Combinations of Co-occurred Categories.

We discuss the implementation of co-occurred categories summarized from our survey. The details of the survey results can be found in RQ1 of Section **??**. The time of perturbations for each member category in the combination is computed independently. It is calculated based on the product of perturbable elements and half of the default frequency value. We refer to the perturbation times of each member category as T_1 and T_2.

C1: Extra Space outside Words & Keyboard Typo (A1 + E1). Add an extra space outside the words and replace a character in the prompt with its adjacent character on the keyboard.
Implementation. NLPerturbator first performs the perturbation of *Keyboard Typo* (as discussed in E1 of this section) T_1 times. NLPerturbator then randomly selects T_2 locations that are outside of a word to add an extra space.

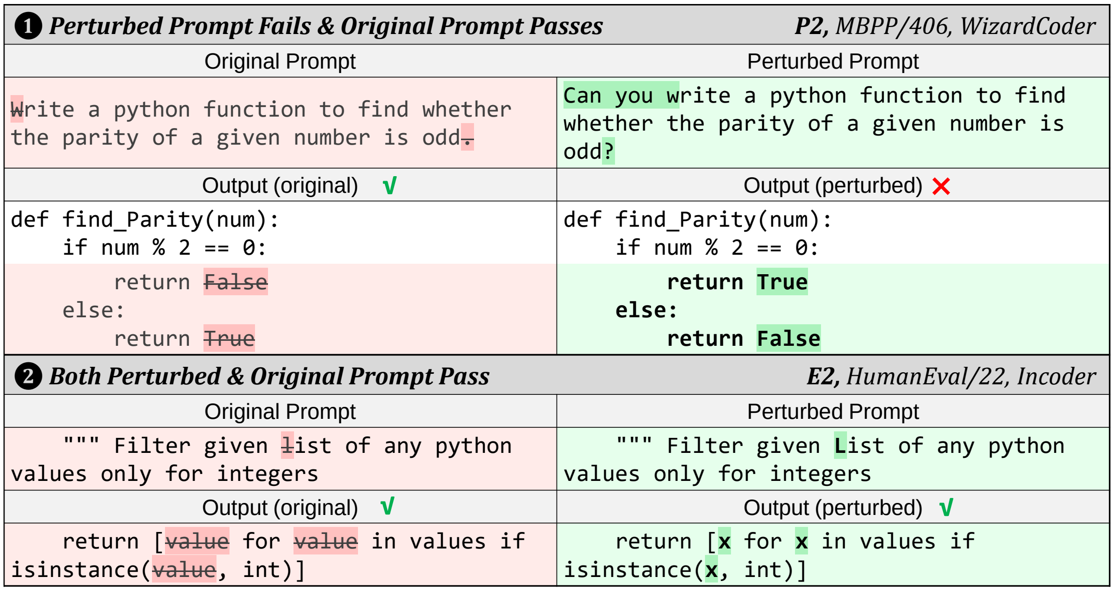
C2: Repeated Chars & Keyboard Typo (A4 + E1). Repeat a character in the prompt and also replace another character with its adjacent character on the keyboard.
Implementation. NLPerturbator randomly selects T_1 characters to apply *Keyboard Typo* and further randomly selects T_2 words to repeat a random character in each selected word.

C3: Char Deletion & Keyboard Typo (D1 + E1). Drop a character from the prompt and also replace a character with its adjacent character on the keyboard.
Implementation. After NLPerturbator performing *Keyboard Typo* T_1 times, NLPerturbator further identifies the characters that are not affected by *Keyboard Typo* and randomly drops T_2 such characters.

2 Case Study of RQ3

We discuss the cases in two aspects: the generated code using ❶ the perturbed prompt fails the tests but the original prompt passes the tests, and the generated code using ❷ perturbed and original prompts both pass the tests. Figure 1 shows the examples of generated code with and without perturbations.

❶ *Perturbed Prompt Fails & Original Prompt Passes.* As shown in Figure 1, the natural language part of the prompt is converted from a declarative sentence *"Write ...is odd."* to an interrogative sentence *"Can you ...?"* following the category of *Declarative to Interrogative (P2)*. However, with such limited modifications, the logic of the code generated by the model is completely opposite (i.e., an interchange between return values True and False), consequently leading to the failure in this code generation request (i.e., judging parity). Despite that *WizardCoder* is relatively less affected by other perturbations (i.e., as shown in the results of Table **??**), the changes of sentence structure (although the same semantic meaning is preserved) may still impact LLMs for their understanding of the prompt.



❶ ***Perturbed Prompt Fails & Original Prompt Passes***	***P2,** MBPP/406, WizardCoder*
Original Prompt	Perturbed Prompt
`~~W~~rite a python function to find whether the parity of a given number is odd~~.~~`	`Can you write a python function to find whether the parity of a given number is odd?`
Output (original) ✓	Output (perturbed) ✗

`def find_Parity(num):`
`    if num % 2 == 0:`
`        return ~~False~~`
`    else:`
`        return ~~True~~`
	`def find_Parity(num):`
	`    if num % 2 == 0:`
	`        return True`
	`    else:`
	`        return False`

❷ ***Both Perturbed & Original Prompt Pass***	***E2,** HumanEval/22, Incoder*
Original Prompt	Perturbed Prompt
`    """ Filter given ~~l~~ist of any python values only for integers`	`    """ Filter given List of any python values only for integers`
Output (original) ✓	Output (perturbed) ✓
`    return [~~value~~ for ~~value~~ in values if isinstance(~~value~~, int)]`	`    return [x for x in values if isinstance(x, int)]`

Fig. 1. Examples of code generation using our manually-verified perturbation datasets (RQ3). We only present the part with differences inside each pair of prompts and generated code for brevity.

❷ *Both Perturbed & Original Prompt Pass.* In this case, the first letter *"l"* of word *"list"* is capitalized (i.e., "***L**ist*") in the prompt following the category of *Extra Capital Letter (E2)*. By comparing the generated code before and after perturbations, although there are differences in the name of identifiers within the list (i.e., *"value"* → *"x"*), the semantics remain the same. The results in Table **??** and this example suggest that these LLMs might be relatively insensitive to the case of natural language in the prompt.

References

[1] "Link to our replication package," https://github.com/iCSawyer/NLPerturbator, 2025, last accessed April 2025.

[2] "spaCy, Explosion," https://spacy.io/, 2023, last accessed August 2023.

[3] "LemmInflect,bjascob," https://github.com/bjascob/LemmInflect, 2023, last accessed August 2023.

[4] C. Fellbaum, "Wordnet," in *Theory and applications of ontology: computer applications.* Springer, 2010, pp. 231–243.

[5] X. Chen, C. Chen, D. Zhang, and Z. Xing, "Sethesaurus: Wordnet in software engineering," *IEEE Transactions on Software Engineering*, pp. 1960–1979, 2019.

[6] Y. Liu, J. Lin, J. Cleland-Huang, M. Vierhauser, J. Guo, and S. Lohar, "Senet: a semantic web for supporting automation of software engineering tasks," in *2020 IEEE Seventh International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, 2020, pp. 23–32.

[7] K. Bhatia, S. Mishra, and A. Sharma, "Clustering glossary terms extracted from large-sized software requirements using fasttext," in *Proceedings of the 13th Innovations in Software Engineering Conference on Formerly known as India Software Engineering Conference*, 2020, pp. 1–11.

[8] P. Damodaran, "Parrot: Paraphrase generation for nlu." 2021.

[9] "OpenAI API," https://platform.openai.com/docs/guides/gpt/chat-completions-api, 2023, last accessed August 2023.