2 DiffParser: A Tool for Tracking Code Changes

To conduct our experiment and evaluate the code comparisons based on the metrics (described in **??**), we have developed a set of Python scripts tailored to our study, including a custom *DiffParser* designed to meet our specific requirements. We used the following key libraries:

- `javalang (v0.13.0)`[3] - to leverage the parsers capability to work with Java code
- `difflib (3.13)`[4] - to produce the unified diffs and analyze the code changes with `difflib.SequenceMatcher`
- `code_diff (0.1.3)`[5] - for AST-based code differencing
- `NetworkX (3.4.2)`[6] - to build a graph that captures code changes over multiple iterations
- `openai`[7] - to access the ChatGPT API and streamline the generation of responses (i.e., the refactored code)

The Underlying Unified Diff Format. To compare code sequences between versions, we utilized the *unified diff* format. In this format, lines prefixed with `--` and `+++` indicate the compared files (the original and the modified version, respectively). Each subsequent `@@ -l,n +l,m @@` header specifies the location of the change: it marks the affected line ranges in the old (`-l`) and new (`+l`) file, together with the number of lines involved (`n, m`). Within a *diff block* (i.e. scope between two headers), removed lines from the original file are prefixed with a minus sign (`-`), while added lines in the modified version are prefixed with a plus sign (`+`).

For example, in the excerpt below, the method name was changed from `approach1` to `areAnagramsBySorting`, and variable names were updated accordingly, while the underlying logic of the algorithm remained identical.

[inline]probably do not need this, otherwise fix it

Parsing Procedure Overview. To classify line-level changes between two code versions, we implemented a two-stage parsing procedure. In the first stage, the algorithm iterates over all diff headers (lines beginning with `@@`) and processes each diff block separately. Within each block, removed and added lines are compared in order to identify modifications, while unmatched lines are provisionally labeled as insertions or deletions. These provisional results are stored in a candidate set for later refinement.

In the second stage, all remaining unmatched lines are collected across diff blocks and subjected to a dedicated *crossmatching* procedure. This step allows the parser to correctly align lines that were moved between different locations or otherwise shifted in ways that prevent them from being matched within a single diff block. The results of the crossmatching stage are then integrated with the initial block-level classifications to produce the final set of modifications, insertions, and deletions.

By design, the algorithm ensures consistency through internal validation checks: the total number of lines before and after crossmatching must remain constant. This guarantees that each line in the diff is accounted for exactly once, preventing both undercounting and duplication.

The final output of the method consists of three components:

- (1) a mapping of modifications (removed-added line pairs),
- (2) a list of deletions (removed lines without a match), and
- (3) a list of insertions (added lines without a match).

Together, these results provide a complete classification of all line-level changes between the two code versions.

Line Matching and Similarity Score Calculation. To quantify modifications between code snippets, we process each diff block by comparing removed lines $r \in R$ (lines prefixed with "-") against added lines $a \in A$ (lines prefixed with "+"). Each line is first normalized by removing diff prefixes, isolating comments, and tokenizing code and

[3] https://github.com/c2nes/javalang
[4] https://docs.python.org/3/library/difflib.html
[5] https://github.com/cedricrupb/code_diff
[6] https://networkx.org
[7] https://openai.com/index/openai-api/

comment segments separately. This results in four representations per line: raw code (r_{code}, a_{code}), tokenized code ($\text{tok}(r_{\text{code}})$, $\text{tok}(a_{\text{code}})$), raw comments ($r_{\text{comm}}$, a_{comm}), and tokenized comments ($\text{tok}(r_{\text{comm}})$, $\text{tok}(a_{\text{comm}})$).

The use of both sequence-based and token-based similarity measures is motivated by their complementary strengths. From our observations, sequence similarity tends to be rather strict: even minor structural or positional changes in a line (e.g., reordering of terms) can cause a disproportionately low similarity score, although the semantic meaning of the line remains almost unchanged. Token-based similarity, in contrast, is more tolerant to such reordering or formatting changes, as it abstracts away from the exact sequence of characters and instead focuses on the multiset of tokens. However, this tolerance may lead to overly high similarity scores in cases where semantically important structural differences are introduced. By employing both measures in parallel, we balance these two perspectives: the sequence ratio ensures sensitivity to structural order, while the token ratio provides robustness against superficial reordering. This combination yields a more reliable overall similarity estimation for code line comparisons.

For each candidate pair (r, a), we compute multiple similarity scores:

$$s_{\text{seq,code}} = \text{SeqSim}(r_{\text{code}}, a_{\text{code}}), \quad s_{\text{tok,code}} = \text{TokSim}(\text{tok}(r_{\text{code}}), \text{tok}(a_{\text{code}})),$$

$$s_{\text{seq,comm}} = \text{SeqSim}(r_{\text{comm}}, a_{\text{comm}}), \quad s_{\text{tok,comm}} = \text{TokSim}(\text{tok}(r_{\text{comm}}), \text{tok}(a_{\text{comm}})),$$

where SeqSim denotes character-level sequence similarity, and TokSim denotes token-level similarity (both using `difflib.SequenceMatcher`).

In addition, an abstract-syntax-tree (AST) similarity $s_{\text{AST}} \in [0, 1]$ is computed when both lines can be successfully parsed. Since the input consists only of code fragments extracted from the diff, preprocessing is required to make the fragments parsable by `javalang`. This preprocessing includes basic syntax completion, such as closing unmatched brackets, and wrapping fragments in a minimal Java context (e.g., embedding method or field declarations inside a dummy class, enclosing incomplete control structures in a dummy method and block, or wrapping isolated statements). From the resulting full parse tree, a simplified AST representation was derived by extracting only structurally dominant node types (e.g., declarations, control flow constructs, expressions). The purpose of this reduction was to obtain a compact node-type sequence, which can be stored as a list and compared across versions using a greedy, string-based matching process, again via `difflib.SequenceMatcher`. If parsing fails, s_{AST} is set to -1, and only sequence and token-based scores are used.

The weighted similarity score $s(r, a)$ for each candidate pair is computed as follows:

$$s(r,a) = \begin{cases} 0.5 \cdot s_{\text{seq,comm}} + 0.5 \cdot s_{\text{tok,comm}}, & \text{if } r_{\text{code}} = \emptyset, \\ 0.2 \cdot s_{\text{seq,code}} + 0.25 \cdot s_{\text{tok,code}} + 0.5 \cdot s_{\text{AST}} + 0.05 \cdot s_{\text{seq,comm}}, & \text{if } s_{\text{AST}} \neq -1, \\ 0.4 \cdot s_{\text{seq,code}} + 0.5 \cdot s_{\text{tok,code}} + 0.1 \cdot s_{\text{seq,comm}}, & \text{otherwise.} \end{cases}$$

A pair (r, a) is accepted as a modification if $s(r, a) \geq \tau$, where τ is a predefined similarity threshold of 0.6. To resolve multiple candidates, the best match is chosen by maximizing $s(r, a)$ and, in case of ties, minimizing the index distance $|r - a|$. Each line participates in at most one match, ensuring a one-to-one mapping between removed and added lines.

The weights in the similarity function as well as the similarity threshold were determined empirically through a systematic evaluation on a custom validation set. This validation set consisted of code snippet variants for which the ground truth of line correspondences was fully known (since we create variants "manually"), enabling us to assess the accuracy of different weighting schemes. To this end, we conducted an iterative optimization procedure: starting from equal weights for all components, we gradually adjusted the relative contributions of sequence-based, token-based, and AST-based similarities. At each step, the alignment results were compared against the known correspondences. The final weights were selected as those that maximized the proportion of correctly identified matches across the diverse cases in the validation set.

Conceptually, the chosen distribution reflects the intended role of each component:

- For purely comment lines ($r_{\text{code}} = \emptyset$), only comment-level similarities are considered, with equal weights assigned to sequence and token similarity. This ensures that both structural and lexical similarity in comments are taken into account without bias.
- When AST information is available, it is given the largest weight (0.5), as it provides the most reliable signal of syntactic and semantic correspondence between code fragments. Sequence and token similarities of the code part remain important (0.2 and 0.25, respectively), but serve as complementary signals. A small weight (0.05) is reserved for comment similarity to account for aligned inline comments.
- In cases where no AST could be obtained, the sequence and token similarities of the code dominate (0.4 and 0.5, respectively), while a smaller weight (0.1) is attributed to comments. This balances strict structural sensitivity with robustness to reordering, while avoiding overreliance on comment similarity.

In sum, the weighting scheme combines theoretical considerations about the relative reliability of each signal with empirical fine-tuning against a controlled test set, ensuring that the resulting similarity score $s(r, a)$ is both principled and effective in practice.

All accepted matches form the *modifications* set, while unmatched removed lines are classified as *deletions* and unmatched added lines as *insertions*. The Average Similarity Score is then computed as the mean of $s(r, a)$ over all modifications, providing a quantitative measure of how similar the changed lines are to their previous versions.

References

[1] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–28. arXiv:2308.02828 [cs] http://arxiv.org/abs/2308.02828