TECHNICAL PAPER · A

Automated Source Code Churn Measurement: Logical and Physical Line Differencing in Large Codebases

A technical overview of two-snapshot churn analysis, the distinction between SLOC and LLOC, and the principal sources of misclassification in automated differencing.

Draft · v1.7.0 era · 27 May 2026

ABSTRACT

Source code churn — the volume of code changed between two snapshots of a project — is conventionally expressed in lines added, deleted, changed, and unchanged. The line, however, is not a uniform unit. Two distinct definitions are in common use: **Source Lines of Code (SLOC)**, which counts newline-delimited physical lines after comment and blank-line stripping, and **Logical Lines of Code (LLOC)**, which counts statement-delimited logical units. The two diverge whenever physical and logical structure do not align — a single physical line may contain several statements, and a single statement may span several physical lines. This paper describes how automated differencing can compute both metrics, the role of the underlying longest-common-subsequence algorithm, the additional alignment passes required when token-level repetition causes ambiguous matches, and the principal classes of churn that resist clean classification.

01 Scope and Definitions

This paper concerns the measurement of differences between two textual snap
project, where each snapshot is a directory tree of files written in one or more programming

Ask CodeDelta

languages. The objective is to quantify, per file and in aggregate, the volume of change in the codebase as expressed in the categories *added*, *deleted*, *changed*, and *unchanged*.

Two units of line counting are in routine use:

- **Source Lines of Code (SLOC)** — physical lines remaining after blank lines and comment lines have been removed. The count corresponds closely to what a developer sees in a text editor with comments and whitespace ignored. SLOC is the historical default for code-size reporting [3].
- **Logical Lines of Code (LLOC)** — statement-delimited logical units. The delimiter is language-specific: a semicolon in C, C++, Java, and similar languages; a newline (subject to continuation rules) in Python; a combination of semicolons and newline-driven Automatic Semicolon Insertion in JavaScript and TypeScript. The relationship between SLOC and LLOC for a given file is determined by the interaction of language convention and coding style; neither metric strictly bounds the other (see Section 04).

A third quantity, **Physical Lines of Code** (raw line count including blanks and comments), is sometimes reported but is rarely useful for churn analysis because it inflates with trivial whitespace edits and reformatting.

The terms *SLOC change*, *LLOC change*, *SLOC added*, etc., always refer to comparison between two snapshots of the same logical project — typically successive commits, builds, releases, or development branches.

02 The Diff Operation as a Sequence-Alignment Problem

Differencing two text files is, in its standard formulation, a longest-common-subsequence problem. Given two sequences A (the old file) and B (the new file), find the longest subsequence common to both; the lines of A not in that subsequence are deletions, the lines of B not in it are additions, and everything else is unchanged. Lines that fall in deletion and addition gaps at the same position are typically reclassified as changes if their similarity exceeds a threshold; otherwise they remain as separate deletes and adds.

The algorithm of choice in practical implementations is Myers' O(ND) edit-script algorithm [1], which computes the shortest edit script in time proportional to the sum of file lengths and the number of differences. Variants and refinements were established by Hunt and McIlroy [2], and the foundational longest-common-subsequence theory by Hunt and Szymanski [5].

For churn analysis the diff is run twice over each file pair, once at the SLOC level and once at the LLOC level. The two diff passes are independent: the same source change may produce different add/delete/change patterns in the two metrics because the underlying token streams differ.

03 The SLOC and LLOC Token Streams

Before the diff algorithm runs, each input file is reduced to a sequence of tokens:

SLOC Tokenization

The file is read line by line. Each line is stripped of trailing whitespace. Blank lines and lines consisting only of a comment are discarded. The remaining lines, in order, form the SLOC token stream for that file. Comments embedded at the end of a code line are stripped, but the code preceding the comment is retained as a SLOC.

LLOC Tokenization

The file is read character by character through a lexer that understands the language's statement-delimiter and continuation rules. Single-line comments, block comments, string literals, character literals, and (where applicable) template literals and regular-expression literals are recognized and excluded from the token stream so their internal contents do not produce spurious delimiters. The lexer emits one token per logical statement boundary.

For C-family languages the principal LLOC delimiter is the semicolon (with the exception of the semicolons inside `for` headers, which form one logical statement together with the loop body opener). For Python, the delimiter is a newline at the same indentation level as the enclosing block, with continuation lines (those ending in a backslash, or where an open bracket or parenthesis has not yet been closed) merged into the same logical line. For JavaScript and TypeScript, the delimiter is either a semicolon or an implicitly-inserted statement boundary at a newline where Automatic Semicolon Insertion applies.

04 Examples: When SLOC and LLOC Diverge

The relationship between SLOC and LLOC is not a fixed ratio. Three patterns are common.

4.1 Multiple LLOC on a single SLOC line

A single physical line containing multiple statements (separated by semicolons in C-family languages) counts as one SLOC but several LLOC.

EXAMPLE 4.1.1 — C-FAMILY

```
int a = 1; int b = 2; int c = a + b; printf("%d\n", c);
```

One physical line. SLOC = 1, LLOC = 4 (four semicolon-terminated statements).

EXAMPLE 4.1.2 — PYTHON (UNCOMMON IDIOM)

```
x = 1; y = 2; z = x + y
```

One physical line. Python permits explicit semicolons between statements on a single line, although the style is rare. SLOC = 1, LLOC = 3.

4.2 One LLOC spanning multiple SLOC lines

A single statement may be broken across several physical lines for readability — a method call with many arguments, a long expression, a multi-line string concatenation, or any expression whose tokens require continuation.

EXAMPLE 4.2.1 — JAVA METHOD CALL

```
String result = formatMessage(
    template,
    customerName,
    orderTotal,
    deliveryDate,
    promotionCode
);
```

Six physical lines (assuming no trailing blank). SLOC = 6 (each non-blank, non-comment line is one SLOC). LLOC = 1 (the statement is terminated by the final semicolon).

EXAMPLE 4.2.2 — PYTHON WITH EXPLICIT CONTINUATION

```
total = (price_excluding_tax
         + sales_tax
         + delivery_charge
         - any_discount_applied)
```

Four SLOC. LLOC = 1: the unclosed parenthesis from the opening bracket means Python treats the entire expression as one logical line, even though the newlines are physically present.

EXAMPLE 4.2.3 — JAVASCRIPT TEMPLATE LITERAL

```
const message = `Hello, ${name},
your order ${orderId}
will arrive ${formatDate(deliveryDate)}.`;
```

Three SLOC. LLOC = 1: the template literal contains literal newlines but is a single expression terminated by one semicolon. A naive newline counter would yield LLOC = 3, which is incorrect.

4.3 Observed ratios in production codebases

The ratio of LLOC to NCLOC (non-comment, non-blank physical lines) in any given codebase is determined by language convention and coding style. Published measurements using the *phploc* tool on widely-used PHP frameworks report ratios in the range of 28–34% (LLOC as a percentage of NCLOC) [6]: symfony-standard 31%, zend-framework2 29%, laravel-framework 28%, and cakephp 34%. These figures reflect a single language and a single tooling implementation; they should not be extrapolated to other languages or other definitions of LLOC without independent measurement.

For languages where one statement per physical line is the dominant convention (notably Python, where explicit semicolon separators are syntactically permitted but stylistically rare), LLOC and NCLOC will tend to track more closely. For C-family code that places opening braces on separate lines or breaks long argument lists across several physical lines, LLOC is typically lower than NCLOC because the closing brackets and continuation lines contribute SLOC without contributing LLOC. The Wikipedia entry on Source Lines of Code illustrates the opposite direction with a single-physical-line C statement (`for (int i = 0; i < 100; i++) printf("hello");`) that counts as 1 SLOC and 2 LLOC [7] — demonstrating that LLOC can exceed SLOC for any line containing more than one statement-terminating delimiter.

A substantial deviation from the ratios documented for similar codebases is a useful signal that the LLOC tokenizer may be mishandling something — in particular, multi-line string or template literals whose internal newlines are being miscounted as statement boundaries.

05 Per-Snapshot Aggregates

For each snapshot, the per-file counts are summed to produce project totals:

- **Total SLOC** — sum of SLOC across all files matched by the configured language extensions.
- **Total LLOC** — sum of LLOC across the same set of files.
- **Comment lines** — reported separately by comment style (single-line, block, end-of-line) for visibility but not included in SLOC or LLOC.
- **Blank lines** — reported only if a raw physical line count is also produced.

Aggregation is straightforward; the interesting questions arise in the diff comparison between two snapshots.

06 Churn Classifications

For a pair of snapshots, every file in either snapshot is placed into one of four file-level states, and the lines within each file are placed into four line-level categories.

File states

STATE	MEANING
`UNCHANGED`	The file appears in both snapshots with identical content (after newline normalization).
`ADDED`	The file appears in the new snapshot only.
`DELETED`	The file appears in the old snapshot only.
`CHANGED`	The file appears in both snapshots with differing content. The diff algorithm runs on this file.

Line categories

CATEGORY	DEFINITION
`UNCHANGED`	Line appears in both old and new at the position chosen by the longest-common-subsequence.
`ADDED`	Line in the new snapshot has no corresponding line in the old snapshot.
`DELETED`	Line in the old snapshot has no corresponding line in the new snapshot.
`CHANGED`	A pair of lines — one deleted, one added at the same position in the edit script — whose similarity exceeds a configured threshold (typically Jaccard token similarity above 0.3 to 0.7 depending on context). Such pairs are reported as one change instead of one delete plus one add.

The aggregate counts `CHG`, `DEL`, `ADD`, and `CRN` (churn = CHG + DEL + ADD) are produced for each metric (SLOC and LLOC) separately. `UNCHANGED` is computed from the totals: unchanged-old = oldTotal - CHG - DEL, unchanged-new = newTotal - CHG - ADD.

07 Worked Example: A Small Function Change

Consider the following modification to a single C function.

OLD SNAPSHOT

```
int calculate(int x, int y) {
    int result = x * y;
    return result;
    int unused = 99;
}
```

NEW SNAPSHOT

```
int calculate(int x, int y) {
    int result = x + y;
    int doubled = result * 2;
    return doubled;
}
```

SLOC analysis

After comment-stripping (none here) and blank-line removal, both files have five physical lines. The diff algorithm produces:

```
1   int calculate(int x, int y) {
2       int result = x * y;      →    int result = x + y;
3       int doubled = result * 2;
4       return result;           →    return doubled;
5   ~~~~int unused = 99;~~~~
6   }
```

SLOC counts: CHG = 2, ADD = 1, DEL = 1, UNCHANGED = 2 (the function header and closing brace).

LLOC analysis

The LLOC tokenizer emits one token per semicolon-terminated statement. The function brace itself contributes no LLOC. The token sequences are:

- Old: `int result = x * y`, `return result`, `int unused = 99` — three LLOC.
- New: `int result = x + y`, `int doubled = result * 2`, `return doubled` — three LLOC.

LLOC counts: CHG = 2, ADD = 1, DEL = 1, UNCHANGED = 0. (The two changed pairings — the `result` assignment and the `return` — pass the similarity threshold; the `unused` and `doubled` lines do not pair with anything and are pure DEL and ADD respectively.)

In this case the SLOC and LLOC counts happen to agree numerically; in larger or more structurally diverse changes they typically do not.

08 Misclassification and the Two-Pass Refinement

The classifications produced by a single pass of Myers' algorithm are not always semantically correct. The algorithm finds *a* shortest edit script, which is not necessarily the one a human reviewer would choose. Several common patterns produce confusing outputs.

8.1 Reordered lines

If two adjacent lines are swapped, Myers' algorithm typically reports this as one delete and one add — not as a "move." From the algorithm's perspective the old line at position n no longer

exists, and a new line has appeared at position n+1. There is no built-in concept of relocation in the standard formulation.

8.2 Repeated identical tokens

Code frequently contains identical tokens — repeated closing braces, repeated blank-line-equivalent patterns, repeated boilerplate calls. When the LLOC tokenizer emits the same logical-line token at many positions, Myers' algorithm has many equally-short edit scripts to choose from. The one it picks may align unrelated braces with each other and report an entire function body as deleted plus another function body as added, when in fact only one statement was altered.

8.3 Renamed identifiers and minor edits

When a variable is renamed throughout a function, every occurrence of the old name disappears and every occurrence of the new name appears. Under a strict line-equality criterion, Myers reports a large number of deletes and adds. The similarity-threshold logic discussed in Section 06 mitigates this by pairing deletes with adds whose token sets overlap sufficiently — at which point most of these become CHGs rather than separate DELs and ADDs.

8.4 Two-pass refinement

A common refinement runs the diff twice. The first pass aligns on raw line equality and produces an initial classification. The second pass re-runs the alignment using a richer token, often constructed by concatenating each line's contents with the enclosing scope name (e.g., function or class qualifier). The scope-qualified tokens distinguish the closing brace of `foo()` from the closing brace of `bar()`, allowing the second pass to discover better-anchored matches that the first pass could not see because its tokens were ambiguous.

The second pass typically improves the CHG count at the expense of ADD and DEL — that is, what the first pass reported as a deleted line in one place and an added line elsewhere is sometimes recognized, on the second pass, as a single moved or modified line. Section 09 quantifies the typical effect.

09 Boundary Cases in Churn Classification

Several patterns sit on the boundary between CHG and ADD+DEL, and reasonable implementations will classify them differently.

9.1 Below-threshold pairs

The Jaccard threshold for promoting a delete+add pair to a CHG is itself a configuration choice. A pair with token similarity 0.31 is almost certainly a CHG; a pair at 0.05 is almost certainly an unrelated DEL and ADD. The transition between the two depends on the chosen threshold. For lines that fall close to the threshold, the same physical edit will be classified as CHG under one threshold and as DEL+ADD under a slightly different one.

9.2 Inserted blocks

When a block of new code is inserted in the middle of an existing function, Myers' algorithm reports the inserted lines as additions but may also report some preceding or following lines as "changed" if their contents shift relative to surrounding unchanged anchors. This is correct behavior from the algorithm's standpoint but can be counterintuitive: a clean insertion produces a small number of CHGs the reviewer did not consciously make.

9.3 Whitespace-only changes

Reformatting (indentation changes, tab-to-space conversions, line-wrap adjustments) can produce arbitrarily large diffs at the SLOC level while producing zero changes at the LLOC level. This is one of the principal practical reasons for reporting both metrics: LLOC is approximately invariant under reformatting, SLOC is not.

9.4 Comment-only changes

When the only difference between two versions of a line is its trailing or leading comment, comment-stripping at the tokenizer stage removes the comment before the diff runs. The line is then classified as UNCHANGED at both SLOC and LLOC levels. Separate comment-churn metrics (CHG_COM, DEL_COM, ADD_COM) can be computed by running a third diff pass over comment lines only.

9.5 File renames and moves

Standard two-snapshot diffing treats files by path. A file moved from `src/foo.c` to `lib/foo.c` appears as a DELETED file at the old path and an ADDED file at the new path, with no churn information about its internal stability. Detecting moves requires a separate similarity comparison between deleted-file and added-file content sets, typically using hash digests or content fingerprints. Most line-counting tools do not perform this step.

10 Output and Verification

For any non-trivial churn measurement to be trustworthy, the underlying classification of every file pair should be inspectable. Two outputs are conventionally produced:

- A per-file table listing, for each file in either snapshot, its state (UNCHANGED, ADDED, DELETED, CHANGED) and counts for SLOC, LLOC, CHG, DEL, ADD, UNCHANGED at both granularities.
- A per-line diff view for changed files showing the alignment chosen by the algorithm, with each line tagged by its category.

The per-line view supports manual verification: a reviewer can confirm that a particular line is correctly classified as CHG rather than DEL+ADD, or vice versa. Without this view the aggregate numbers are difficult to audit. The examples in Section 04 and Section 07 are small enough to verify by inspection; production codebases producing thousands of changed lines require the diff view to spot-check classifications.

11 Practical Considerations

11.1 Language coverage

SLOC counting is essentially language-agnostic once comment syntax is known. LLOC counting requires per-language tokenizers; supporting a new language requires implementing or borrowing a lexer that knows the language's comment, string, and statement-delimiter rules. For ill-defined "languages" (configuration files, mixed-content templates), LLOC may not be meaningful and is conventionally reported as equal to SLOC.

11.2 Binary and generated files

Files identified as binary by a content sniff (control-byte density, magic-number heuristics) are excluded from line counting. Generated files (typically identifiable by directory naming convention or build-system patterns) should also be excluded, as their churn does not reflect human authorship effort. Both exclusions are policy choices and should be documented for any given measurement.

11.3 Performance characteristics

The Myers algorithm runs in O(ND) where N is the sum of file lengths and D is the number of edit operations. For typical source-file pairs (a few thousand lines, a few hundred edits) this is fast; for pathological cases (files that have been entirely rewritten, or files of tens of thousands of lines) the constant factors matter and divide-and-conquer or linear-space variants [1] are preferred. Project-wide churn measurement processes file pairs independently and parallelises naturally across CPU cores.

11.4 Configuration sensitivity

The numerical results of a churn measurement depend on several configuration choices: the file extension list (which determines what is counted as source), the comment-stripping rules (line-only versus end-of-line versus block), the Jaccard threshold for CHG classification, and the two-pass refinement options. Measurements taken with different configurations are not directly comparable. Any reported figure should specify the configuration used.

12 Summary

Source code churn is measured by aligning two snapshots of a project with a longest-common-subsequence algorithm, separately at the physical-line (SLOC) and logical-line (LLOC) levels. The two metrics diverge whenever statements and physical lines do not coincide: multi-statement lines yield more LLOC than SLOC, multi-line statements yield more SLOC than LLOC. Classification of changed lines into CHG, DEL, ADD, and UNCHANGED is sensitive to the similarity threshold used to recognize modifications and to the alignment behavior of the underlying algorithm. Pathological alignments, particularly those caused by repeated identical tokens, are partially mitigated by a second alignment pass using scope-qualified tokens. Several boundary cases — renamed identifiers, reformatting, moved files, comment-only edits — require either auxiliary passes or explicit policy choices. Production use depends on inspectable per-line output for verification.

References

1. Myers, E. W. (1986). *An O(ND) Difference Algorithm and Its Variations*. Algorithmica, 1(2): 251–266.

2. Hunt, J. W. and McIlroy, M. D. (1976). *An Algorithm for Differential File Comparison*. Bell Laboratories Computing Science Technical Report 41.
3. Wheeler, D. A. (2001–2004). *SLOCCount — Source Lines of Code Counter*. Project documentation and user manual.
4. Park, R. (1992). *Software Size Measurement: A Framework for Counting Source Statements*. Software Engineering Institute, Carnegie Mellon University, CMU/SEI-92-TR-20.
5. Hunt, J. W. and Szymanski, T. G. (1977). *A Fast Algorithm for Computing Longest Common Subsequences*. Communications of the ACM, 20(5): 350–353.
6. de Greef, D. (2015). *Measuring Maintainability*. Presentation using *phploc* on Symfony Standard, Zend Framework 2, Laravel, and CakePHP, reporting LLOC-to-NCLOC ratios across PHP frameworks.
7. *Source Lines of Code*. Wikipedia. Encyclopaedic reference describing physical and logical SLOC and demonstrating with a single-physical-line C statement that LLOC may exceed SLOC.
8. Nguyen, V., Deeds-Rubin, S., Tan, T., Boehm, B. (2007). *A SLOC Counting Standard*. COCOMO II Forum, USC Center for Systems and Software Engineering.
9. Ukkonen, E. (1985). *Algorithms for Approximate String Matching*. Information and Control, 64(1–3): 100–118.
10. Prechelt, L. (2000). *An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl*. IEEE Computer 33(10): 23–29. Cross-language size and productivity measurements for the same set of requirements.
11. Chowdhury, S. A., Uddin, G., Holmes, R. (2022). *An Empirical Study on Maintainable Method Size in Java*. Empirical study of ~785K Java methods, providing distributional data on method size and its relationship to maintenance effort.

Appendix: Links to Academic Papers

Direct links to the cited sources, in reference order. Where a paper sits behind a publisher paywall, the link resolves to the canonical record (DOI, proceedings page, or institutional repository) from which open versions can usually be located.

1. Myers (1986), *An O(ND) Difference Algorithm and Its Variations* — doi.org/10.1007/BF01840446 (open copy: xmailserver.org/diff2.pdf)
2. Hunt & McIlroy (1976), *An Algorithm for Differential File Comparison* — cs.dartmouth.edu/~doug/diff.pdf
3. Wheeler (2001–2004), *SLOCCount* — dwheeler.com/sloccount/
4. Park (1992), *Software Size Measurement* (CMU/SEI-92-TR-020) — insights.sei.cmu.edu/library/software-size-measurement...

5. Hunt & Szymanski (1977), *A Fast Algorithm for Computing Longest Common Subsequences* — doi.org/10.1145/359581.359603
6. de Greef (2015), *Measuring Maintainability* (phploc ratios) — tool: github.com/sebastianbergmann/phploc
7. *Source Lines of Code* — en.wikipedia.org/wiki/Source_lines_of_code
8. Nguyen, Deeds-Rubin, Tan, Boehm (2007), *A SLOC Counting Standard* — semanticscholar.org/.../A-SLOC-Counting-Standard
9. Ukkonen (1985), *Algorithms for Approximate String Matching* — doi.org/10.1016/S0019-9958(85)80046-2
10. Prechelt (2000), *An Empirical Comparison of Seven Programming Languages* — doi.org/10.1109/2.876288 (open copy: page.mi.fu-berlin.de/.../jccpprt_computer2000.pdf)
11. Chowdhury, Uddin, Holmes (2022), *An Empirical Study on Maintainable Method Size in Java* — arxiv.org/abs/2205.01842