POSITION PAPER · MEASUREMENT · LINES, STATEMENTS AND THE REAL CODE

# Why LLOC is what really counts

*A 35-line source file holds five statements of working code. An edit session that churns twelve lines turns out, measured properly, to be one real change. This paper dissects a file through every counting tier — LOC, SLOC, LLOC, data elements — with real tool output at each step, and explains why the statement, not the line, has been the churn unit demanding customers ask for since the beginning.*

First release 27 July 2026

**ABSTRACT**

Every codebase can be counted four ways: physical lines (LOC), source lines (SLOC), logical statements (LLOC), and — inside the statements — data elements. Each tier discards something the tier below counted, and what it discards is exactly what corrupts change measurement: blank lines, comments, formatting, and generated data. We dissect a single 35-line C file through all four tiers (35 lines → 25 source lines → 6 statements → 5 working statements), then edit it the way real programmers do — a reformat, a table refresh, one behavioural change — and show that a line counter reports **12 lines of churn** where the truth is **one changed statement of working code**. Logical lines are not a refinement of line counting; they are the point of it. The measurement lineage bears this out: statement-level churn was not invented by a tool vendor, it was demanded by an avionics organisation whose line counter could not tell them what had changed.

## 1 Three ways to count the same file

**LOC** is every physical line — identical to `wc -l`, blanks and comments included. It is the rawest measure and the only one with zero interpretation in it: any two tools that count physical

lines honestly will agree exactly. (CodeDelta and the free counter cloc agree on LOC to the line on identical file sets — we verify this routinely, and you can too.)

**SLOC** strips blank lines and comment-only lines: the lines that carry source. This already requires language knowledge — you cannot strip comments without knowing the comment syntax.

**LLOC** counts *statements* — for C-family languages, roughly the semicolons outside strings and comments. A statement spread over five lines is five SLOC but one LLOC; three statements packed on one line are one SLOC but three LLOC. LLOC is invariant to formatting, which is precisely what a change metric needs.

**Data elements** go one tier further: some statements are not working code at all but serialised data — lookup tables, calibration arrays, generated instruction databases — and inside them the meaningful unit is the element, not the statement.

## 2 A 35-line dissection

Here is a complete, compilable C file — small enough to count by hand:

```c
/* ------------------------------------------------------------
 * thermostat.c — temperature control loop
 * Comment lines: counted in LOC, excluded from SLOC.
 * ------------------------------------------------------------ */

#include <stdio.h>
#include "sensor.h"

/* Calibration table — generated from the 2024 bench run.
   Data, not working code. */
static const int calib[16] = {
     3,  5,  9, 12,
    17, 23, 31, 40,
    52, 66, 83, 99,
   120, 144, 171, 200
};

static int clamp(int v,
                 int lo,
                 int hi)
{
    if (v < lo)
        return lo;
    if (v > hi)
        return hi;
```

```
    return v;
}

int read_temperature(int raw)
{
    int idx = clamp(raw / 16,
                    0,
                    15);
    return calib[idx];
}
```

CodeDelta's engine measures it as follows (verbatim CSV output, snapshot mode):

| TIER | COUNT | WHAT FELL AWAY |
|---|---|---|
| **LOC** | 35 | — the whole file, exactly `wc -l` |
| **SLOC** | 25 | 10 lines of blanks and comments |
| **LLOC** | 6 | 19 lines of formatting — statements spread across lines |
| of which data | 1 | the calibration table: one statement, **16 elements** |
| **Working statements** | **5** | **the real code** |

Thirty-five lines; five statements that actually count. (Two of the 35 are preprocessor lines — tracked separately as PLOC.) The ratios vary by codebase, but the shape never does: on a 3,153-file tree we measure the same funnel at 1,118,557 LOC → 947,233 SLOC → 792,903 LLOC.

## 3 The churn test: twelve lines, one change

Now we edit the file the way real work happens — three edits in one commit:

- **Reformat** `clamp()` : parameters onto one line, single-line returns. Logic untouched.
- **Refresh the table**: four new calibration points (16 → 20 elements).
- **One behavioural change**: the index statement's divisor and bound ( `/16, 15` → `/20, 19` ).

Measured churn, old → new (verbatim engine output):

| VIEW | CHURN | READING |
|---|---|---|
| Line view (SLOC) | `CHG 7 · DEL 4 · ADD 1 —` **`CRN 12`** | “a third of the file changed” |
| Statement view (LLOC) | `CHG 2 · DEL 0 · ADD 0 —` **`CRN 2`** | two statements changed, in place |
| Data partition | `DATA_CHG 1 ·` **`ELEM_ADD 4`** | one of the two is the table: “4 calibration points added” |
| **Working code** | **`1 statement`** | **the divisor — the only edit that can alter behaviour** |

The line counter cried twelve. The truth was one — and the measurement names it. This is not an artificial worst case: reformatting, table refreshes and small logic edits travel together in ordinary commits, and line-level churn adds them into a single number. Statement-level churn with a data partition takes them back apart. The shape metrics built on LLOC — REWORK, REP_CHURN — inherit this immunity; built on lines, they would measure your formatter.

## 4 The fourth tier at scale

In a 35-line example the data tier is one small table. At scale it dominates: in NVIDIA’s open GPU kernel modules we measured single C statements up to 3.0MB, and **28.4% of that tree’s source lines are data, not code**. In Erlang/OTP’s 2026 repair regime, the sceptic’s question — *is the striking REWORK figure just table refresh?* — is answerable only because the tiers are separated: working-code REWORK 22.6%, unchanged by the data filter. The full method and findings are in the data-metrics report.

## 5 Where LLOC came from

Statement-level churn was not invented in a whitepaper. Twenty-odd years ago, a major avionics organisation — people who certify flight software — measured their code with a line counter of the same class as today’s free tools. At their scale it broke, and the rebuild they commissioned added a real database and language-specific parsing for the languages they actually flew: Ada and Fortran among them. Then came the request that defined the product line CodeDelta descends from: *report churn in logical lines, because lines are not telling us*

*what changed*. A global semiconductor manufacturer later adopted the same tool for the same reason. The demand for LLOC came from the users with the least tolerance for measurement error — that provenance is the strongest argument this paper has.

## 6 Count it yourself

Physical lines are the honest baseline, and any good line counter agrees with us on them exactly — which is why we host one. The free GPL utility cloc, on our download page, will validate the size of your codebase and cross-check CodeDelta's LOC to the line. Where it stops — statements, data elements, churn shape — is where this paper began. The worked example above ships with CodeDelta's documentation; run it and check every number by hand.

## References


1. Paper A (this series), *Automated Source Code Churn Measurement* — paper-churn.html
2. Paper C (this series), *REP_CHURN: Replacement-Dominant Churn* — paper-repchurn.html
3. Measurement report, *Data wearing code's syntax* — paper-data-metrics.html
4. Companion report, *What a line counter misses: six and a half years of Erlang/OTP* — paper-cloc-erlang.html
5. cloc, by Al Danial and contributors (GPL-2.0) — hosted with its full licence text on our download page