OmniDiff: A Fast, Robust, Syntax-Aware Code Diffing Tool

Marko Ivanković
Universität Passau
Passau, Germany
marko@ivankovic.me

Goran Petrović
Universität Passau
Passau, Germany
stgpetrovic@gmail.com

René Just
University of Washington
Seattle, USA
rjust@cs.washington.edu

Gordon Fraser
Universität Passau
Passau, Germany
gordon.fraser@uni-passau.de

Abstract

Code diffing, the process of comparing two pieces of source code and computing the set of operations that transforms one into the other, underlies modern version control and, especially, modern code review. During a review, developers read the diff to see what changed and what stayed the same, so the quality of the diff directly shapes the ease and quality of the review. A syntax-aware diff maps one abstract syntax tree (AST) onto another. AST diffing algorithms assume the following: that the best diff can be expressed with the four standard operations (insert, delete, update, move) and is the cheapest one under a unit cost model with a free move, that computing it is possible within an interactive wall-clock budget, and that the AST node mapping can be visualised without losing quality. We test each assumption on a corpus of real code changes, to measure the viability of AST diffing.

We built a corpus of 1687 real-world changes in 24 languages, each carrying a human-authored AST mapping and 1420 of them a human-painted text rendering. On 13.7% of changes the best possible correspondence requires a mapping that the common operations (insert, delete, update and move) cannot express; on at least 2.7% the human-preferred mapping is strictly costlier than the cheapest AST diff; and 62.0% of painted changes admit at least two equally correct text renderings of one settled node mapping. 32.4% of AST nodes carry no text of their own and never reach the screen, so a node-level metric gives a third of its weight where no user can see. A whole-tree tree-edit-distance computation completes within one second for only 46.8% of 19,798 code file pairs stratified-sampled by size. Against this corpus, no configuration of eight established tools renders more than 79% of changes perfectly, and the one that reaches it, srcDiff, parses five of the corpus's languages; of the tools that parse most of it, none renders more than 64%, and `diff`, the most popular tool, renders only 59%.

As a tool contribution, we present OmniDiff, which combines existing state-of-the-art algorithms into one robust tool. It maps 99.96% of AST nodes correctly and renders 90% of changes perfectly, at a median of 4.6 ms per diff including its parse. We tested it on 442,530 diffs in 7,444 repositories, and it completes 99.95% of them successfully.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'26, June 03–05, 2026, Ludbreg, Croatia
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/26/06
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts

• **Software and its engineering → Software maintenance tools**; *Empirical software validation.*

Keywords

source code differencing, syntax-aware diffing, empirical software engineering, source code tree edit distance

ACM Reference Format:
Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2026. OmniDiff: A Fast, Robust, Syntax-Aware Code Diffing Tool. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference'26).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Code diffing, the process of comparing two pieces of source code and computing the set of operations that transforms one into the other, underlies modern version control and, especially, modern code review. During a review, developers read the diff to see what changed and what stayed the same, so the quality of the diff directly shapes the ease and quality of the review. Whether that reading succeeds depends on how the change is shown, not only on the change. Reviewers name understanding the change as the main challenge of code review [4], and ask for tools that show a change's structure rather than its text delta [31]. Controlled experiments show the presentation alone affecting the outcome. Files shown later in a review receive fewer comments, and a seeded defect in the last file has 64% lower odds of being found than the same defect in the first [15, 29]. The unified view of a diff costs less visual effort and finds more bugs than the split view of the same diff [1]. A change decomposed into its separate concerns is reviewed with fewer wrongly reported issues than the same change tangled [5, 10, 17]. A reviewer's working-memory capacity predicts whether they find a *delocalised* defect, one whose parts the diff shows far apart [6, 32].

The cost of a diff that is hard to read is paid in time and in quality. At Google, useful comments fall and review latency rises with the size of a change [30], and Mozilla's contributors name size-related factors as the largest driver of review latency [20]. Components

DRAFT

Before	After

`def save(rec):`
`  db.begin()`
`  db.put(rec)`
`  db.commit()`
	`def save(rec):`
	`  if rec.dirty:`
	`  db.begin()`
	`  db.put(rec)`
	`  db.commit()`

Figure 1: Wrapping a block in a condition. Only whitespace (indentation) changed, but a line-based diff reports all of it as deleted and reinserted. An AST diff can report one insertion and a move.

with low review participation ship more post-release defects [24]. Practitioners rate the reaction time of review tooling as very important [21], and at Meta, developer satisfaction with code review falls with the time the slowest quarter of a developer's changes wait for review [7]. Where a diff shows a refactoring as a refactoring rather than as deleted and inserted lines, reviewers report lower cognitive effort and read fewer lines [8, 16], and syntactic deltas modelled on how developers edit are judged more understandable than the cheapest tree edit [9].

The standard diff used by almost all developers and platforms today is line-based: the file is treated as a sequence of text lines, and the tool computes an edit script between the two sequences using three operations, insert, delete, and update. This approach is computationally cheap and fast even on modest hardware, but most implementations offer no move operation, and the tools that do support move use it only in limited ways. A whole class of common changes—extracting code into a reusable function, wrapping a block in a condition, reordering declarations—is a move of existing code, and without a move operation every such change must be rendered as a full deletion plus a full insertion. Reconstructing the correspondence between the deleted and inserted halves is then left to the reviewer, by eye, which is the kind of mechanical, attention-taxing work humans make mistakes when doing. Figure 1 shows a small example of the problem: wrapping three unchanged lines in a conditional, which a line-based tool reports as six lines deleted and seven inserted, and which a reader has to diff again in their head to see that only the wrapper is new.

One way to compute diffs with a move operation is to compare the code's abstract syntax tree (AST) instead of its raw text. An AST diff maps nodes of one tree onto nodes of the other; a move is a mapping whose nodes match but whose ancestry changed. Tree diffing is a well-studied field, and Section 2 surveys the work this paper builds on. In practice, however, tree diffing has seen little adoption, and two obstacles stand between the algorithms and a tool developers use. The first is runtime: pure exact tree-diffing algorithms are $O(n^3)$ in the number of nodes. The second is that an AST diff is not directly usable by a developer. It has to be visualised, usually overlaid over the textual representation of the source code, and for any given AST diff there are several possible visualisations—so a tool must choose one, and the quality of that choice determines adoption as much as the mapping underneath it does.

We examined four groups of research questions. Each group is answered in its own section, in the order listed here.

- **Research Question Group 1 (Viability):** What percentage of real-world code changes have an optimal AST node mapping that cannot be expressed using common AST diff operations (Insert, Delete, Update, Move)? What percentage of real-world code changes have solutions that humans prefer whose cost is higher than the lowest cost AST diff, using the commonly accepted costs of AST diff operations?
- **Research Question Group 2 (Speed):** What percentage of real-world source-code changes can a single, whole-tree tree-edit-distance computation complete within one second?[1]
- **Research Question Group 3 (Uniqueness):** What percentage of real-world code changes have multiple optimal AST node mappings? What percentage of real-world code changes have multiple equally correct textual visualisations of the diff? What percentage of the AST influences the textual visualisation?
- **Research Question Group 4 (State of the Art):** How accurately do current state-of-the-art code diffing tools find optimal visualisations? How fast are real-world state-of-the-art tools?

This paper makes the following contributions:

- **Dataset:** A set of 1687 real-world code changes. The changes cover 24 languages and were selected using 3 different selection algorithms from 7,444 open source repositories. The repositories themselves are a combination of a curated list and every repository used by the Gentoo Linux distribution. Every change was solved by hand and the optimal map between the two ASTs was found, or if the optimal map is impossible using the common AST diff operations, a best-possible solution with a comment explaining the missing part. Additionally, 1420 of those solutions also have the optimal visualisations: the spans of text a developer should see marked, independently of which AST nodes carry them. Often, two visualisations are provided: the minimal one and the maximal one, covering human preferences between equally valid solutions. The dataset, the tool used to create the dataset and the entire history of the dataset is available freely for future research. To allow comparison with existing research, the dataset has a fourth part: the Defects4J [19] changes of the AST node-mapping oracle of Alikhanifard and Tsantalis [2], solved by us by hand for comparison with prior work. The dataset is described in detail in Section 3.
- **Evaluation:** An analysis of the real-world code change data shape, irrespective of any concrete implementation, and a quantified comparison of several state-of-the-art code diffing tools on the human annotated dataset.
- **Tool:** OmniDiff, a new state-of-the-art tool. Combining all current research with strict engineering principles and carefully implemented to maximise performance, based on the collected data.

2 Background

This section lists two types of existing work: pure algorithms and implemented tools.

[1] One second is where Nielsen's response-time limits put the boundary between an interaction that keeps a user's train of thought and one that visibly interrupts it [26].

PREPRINT

2.1 Algorithms

Zhang-Shasha [33] gives the classic dynamic-programming algorithm for ordered tree edit distance. Given two rooted, ordered trees, it computes the minimum-cost sequence of node insertions, deletions, and relabelings that transforms one tree into the other. It is exact, but its keyroot decomposition does repeated work across overlapping subproblems.

APTED [28] is the state-of-the-art exact tree-edit-distance algorithm. It improves on Zhang-Shasha by choosing, for each subproblem, the cheapest of three decomposition strategies (left path, right path, or heaviest inner path), rather than a single fixed strategy. This choice provably minimises the total work across the whole computation. APTED still costs $O(n^3)$ in the worst case for general trees, the same asymptotic bound as Zhang-Shasha, but its real-world runtime is markedly lower.

Myers' O(ND) algorithm [25] solves a different, more restricted problem: the shortest edit script between two flat sequences, not two trees. It underlies most line-based diff tools, including Unix `diff`. Its cost scales with the size of the edit script, not the size of the inputs, so it is fast when two sequences are similar.

2.2 Tools

Unix `diff` is the golden standard. It treats each file as a sequence of lines, computes an edit script between the two sequences with Myers' algorithm, and reports it as hunks of deleted and inserted whole lines. GNU `diff` caps the algorithm's cost with a heuristic by default, so on large inputs with many differences the script it reports can be longer than the shortest one. It has no move operation and no sub-line detail.

`git diff` defaults to the `xdiff` engine, which offers four algorithms: Myers, the default; *minimal*, which spends more time to find a shortest script; *patience*, which anchors on lines unique to both sides before recursing; and *histogram*, a faster generalisation of patience. All four emit the same output form as Unix `diff`.

Neovim's diff mode (`nvim -d`) runs the same `xdiff` line pass as `git` and then a second, character-level pass that highlights what changed inside each paired line.

BDiff [23] is a text-based tool that reasons about blocks of lines rather than single lines and reports character ranges within changed lines.

GumTree [14] is a source code AST diff. Instead of one global tree-edit-distance computation, it runs a top-down phase that greedily matches large isomorphic subtrees, then a bottom-up phase that matches remaining container nodes by a Dice coefficient of their already-matched descendants, and a recovery pass that pairs up deleted and inserted identical subtrees the top-down and bottom-up phases missed. For performance, it also introduces InsertTree and DeleteTree operations that are equivalent to repeatedly calling Insert or Delete operations on all children in the given subtree.

difftastic [18] and **diffsitter** [12] are a newer generation of tree-sitter-based diffing tools. Both prioritise a diff a human reads comfortably over an exact node-to-node mapping: difftastic reduces the parse tree to an s-expression and aligns it with its own structural algorithm, and diffsitter applies a related tree-sitter-based alignment over a smaller, hand-picked set of compiled-in grammars.

srcDiff [9] diffs the srcML markup of C, C++, C#, Java and Python source. It aligns the two files with a sequence difference and then applies syntactic rules, modelled on how developers edit code, to decide which differing elements are changes to one construct and which are a construct replaced by another. It detects moves, and it writes one merged srcML document in which the text of each side is marked as deleted, inserted or common.

3 Empirical Dataset

To understand the real-world environment code diffing tools are used in, we built a dataset of 7,045,754 files in 30 languages. Of these, we sampled 1687 real open source code changes and examined them in depth.

To ensure good coverage, we built two sets of open-source repositories:

- The Curated list - 100 open source repositories, selected manually from several lists of most popular repositories, enhanced further by repositories we were already aware of. This biased the list towards active, widely used codebases, with good coverage in many programming languages. The Curated list was also used as a smaller list to repeatedly test the paper's own codebase, including in automated test infrastructure.
- The Full List - 7,444 repositories, created by expanding the Curated list with all repositories present in the Gentoo Linux distribution's package list. This includes a wide array of codebases of different activity, code quality, userbase and programming language mixes.

We cloned each repository to a depth of 50 commits from every main branch. While we cannot guarantee that all repositories remain accessible, the entire list of repositories, the cloning script and all analysis is available from the paper's repository and can be replicated by readers, if desired.

We fetched, parsed and measured the corpus on a single machine: an Intel Xeon E3-1275 v5 with 64 GB of RAM, running Linux 6.8. The checkouts and the measurement database sit on 4 16 TB WDC Ultrastar DC HC550 drives in RAID 5. We record this because some results are hardware-bound, notably the wall-clock budget we hold whole-tree APTED to in Section 5 and the per-tool wall-clock timing of Section 7.

This section describes the overall shape of the resulting dataset.

3.1 Shape of real-world files

We used a simple extension- and content-based method to categorise the files. The first thing to note about the dataset is that only two thirds of all files are code (Figure 2). The rest is data, configuration, documentation, and a residual of unknown type, which limited manual inspection suggests is mostly data. Note however that any "code" diffing tool still must support visualising changes to non-code files as well.

Restricted to code files, the top row of Figure 3 shows the distribution of bytes, lines of code and AST nodes per file. The median code file has 43 lines, the 99th percentile 2,511 and the largest 13,025,371. The node counts are over the 76.5% of code files that can be parsed (3,586,682 of 4,688,419); the rest are in a language without a grammar, flagged as generated and not parsed, or empty. Of those that

PREPRINT

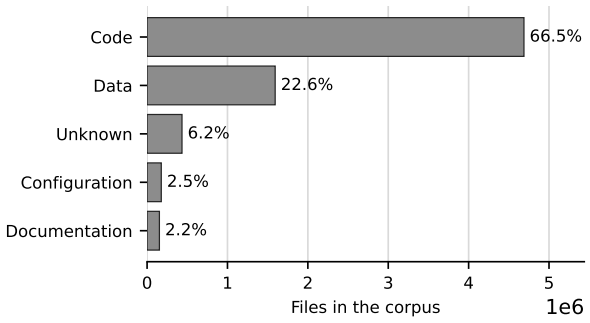


Figure 2: Distribution of file types across the corpus.

parse, the median has 407 AST nodes, the 99th percentile 27,546 and the largest 23,584,040.

Byte size and AST node count are strongly correlated for files within the 99th percentile of both, Pearson $r = 0.9156$, which simplifies to $|\text{AST}| \approx \text{bytes}/5$ for quick estimates.

3.2 Shape of real-world file edits

Across both repository lists we analysed 114,817 commits. To stop large repositories, notably the Linux kernel, from skewing the results, each repository was analysed at most 50 commits deep. 31.5% of the files those commits touched were code files, giving 578,783 code-file edits, of which 75.2% were modifications; the remainder created or deleted a file and so present no pair of versions to map between.

To allow for comparison with existing research, `git diff` was used to compute the number of changed lines.

The bottom row of Figure 3 shows the sizes of real-world edits. The median file edit changes 5 lines and the median commit 11; 64.8% of file edits and 49.1% of commits change ten lines or fewer, while the 99th percentiles are 989 and 2,712 lines. Our results are broadly aligned with existing research, among them Arafat and Riehle, who report that 47.5% of all commits touch 10 lines or fewer [3], with the long tail stretching to tens of thousands of lines. The median edit rewrites only 3.2% of the file it touches, and 59.9% of edits rewrite under 5%. In the 99th percentile, an edit rewrites the file entirely (100.0%). A diffing tool therefore has to be good at two unrelated jobs—finding a handful of changed lines in an otherwise untouched file, which is the overwhelming majority of edits, and staying usable when nothing is left to anchor on.

3.3 Shape of optimal file diffs

From both Curated and Full list of repositories, we fully randomly sampled 10 diffs per language. That gave 220 diffs from the Curated list across 22 languages, and 255 from the Full list across 24 languages. The two language counts differ because the Curated list's repositories hold almost no R or Scala code, so those two languages are drawn from the Full list only. The Full list's count exceeds 10 per language because some languages were drawn a second time after their first rounds were largely rejected. Every sampled diff was manually inspected for sampling errors. At this stage, 25 diffs were rejected, mostly because programming language detection detected the wrong language. In particular TypeScript (7 rejected) and R (11) file extensions were used for other languages or data. This resulted in a dataset of 219 diffs from the Curated and 231 diffs from the Full list. As shown in Section 3.2, a pure random sample is heavily slanted towards small changes. To account for this, the sample was further augmented by a stratified sample based on the lines of code of the files being changed. The strata were: 1 to 10 lines, 11 to 30, 31 to 100, 101 to 300, 301 to 1000, 1001 to 3000 and more than 3000 lines of code, and the larger of the two sizes was used. In each stratum, 10 diffs were randomly sampled per language, and inspected as above.[2]

A fourth list further extends the corpus. The AST node-mapping oracle of Alikhanifard and Tsantalis [2] contains the 996 AST mappings for 800 bug-fixing changes of Defects4J [19]. The file contents its offsets index into come from Falleri and Martinez' replication package [13]. We solve those same 996 diffs, so that our mappings can be compared against an independently authored reference on a corpus that other AST-differencing evaluations already use.[2]

Each diff was solved by hand. Solving a diff means writing down, for every node of the AST, which node of the other AST it corresponds to, or that it has none. For 1420 changes the annotator additionally painted the rendering: the spans of text a reader should see marked, chosen independently of which nodes carry them. The annotators are the authors, who each have at least 15 years of experience in software development and have reviewed tens of thousands of real-world code changes in industrial environments.

Please note, diffing multiple files and detecting cross-file code moves is out of scope of this paper.

3.4 Cross-validation with existing datasets

The AST node-mapping oracle of Alikhanifard and Tsantalis [2] records, for 800 bug-fixing changes of Defects4J over 996 Java compilation units, which before-side node each after-side node corresponds to: 3,031,434 node correspondences in all, built by running six tools, inspecting every diff by hand against six stated criteria, and resolving disagreements by vote. It was authored by different people, for a different tool, on a different AST, before OmniDiff existed. The file contents its character offsets index into come from Falleri and Martinez' replication package [13].

Their prior art uses Eclipse JDT nodes and ours tree-sitter nodes, so the two can be compared only where a record's character span is one some tree-sitter node also has. 78.0% of records resolve; the remainder are JDT constructs tree-sitter does not model separately, chiefly the elements inside a Javadoc comment, which tree-sitter reads as a single token.

The 444 compilation units we have solved so far are reported as a fourth dataset, Defects4J, beside the Curated, Full and Stratified samples throughout, and are included in every pooled total. Over the 440 changes we have mapped by hand so far, our mapping and the oracle's agree to 99.62%: 624 disagreeing node pairs out of 84,887, and 67 of 14,059 at statement level.

[2]The corpus and this paper are being written in parallel. Drawing and solving the stratified sample, and hand-annotating the Defects4J list below, are both still in progress, and the ambiguity pass over those two has not been made at all. Every figure in this paper is over the 1687 changes solved so far, except the tool comparison of Section 7 and the line-level, speed and robustness figures of Section 8, which were measured at an earlier stage of the corpus, over the 1217 changes solved then. Every figure that depends on the ambiguity pass is a lower bound for those two datasets.

PREPRINT

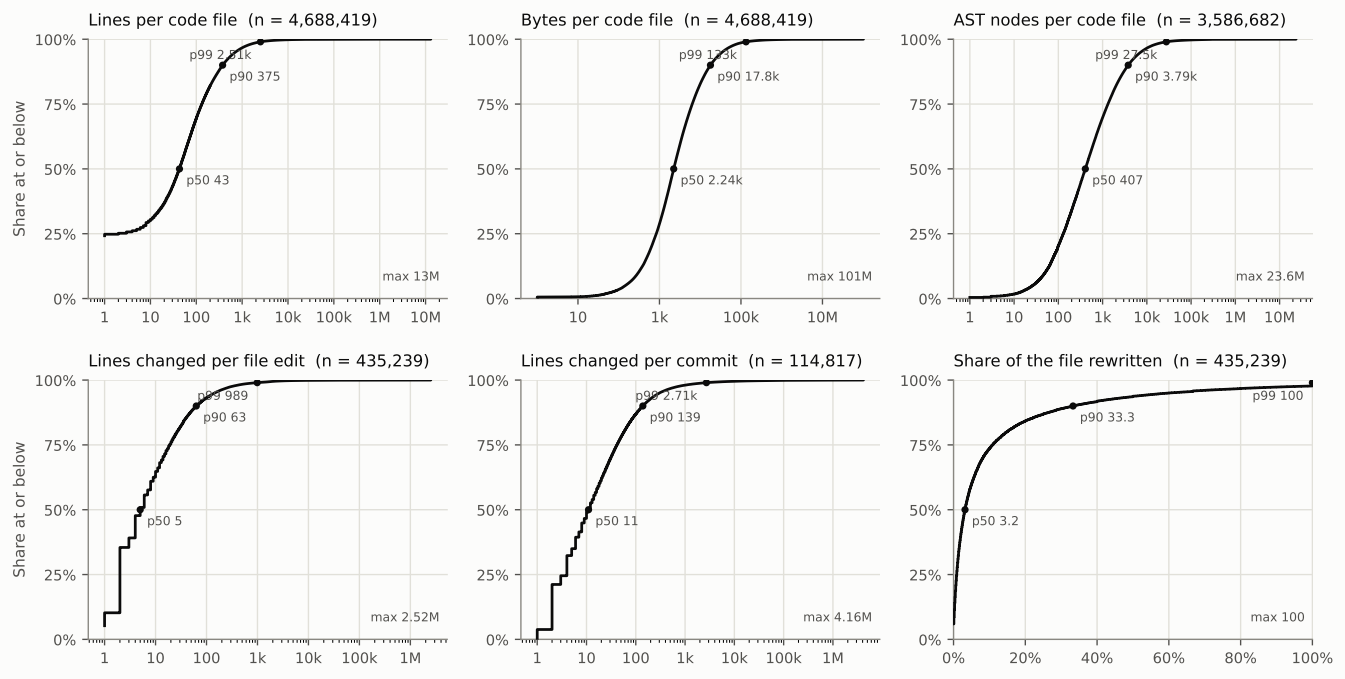


Figure 3: The shape of the corpus, every distribution drawn the same way: the share of the population at or below the value on the axis, with the median, 90th and 99th percentiles marked on each curve and the maximum in the corner. Top: size of every code file in the corpus, in lines and bytes, and in AST nodes for the code files that parse, which is why the third panel's n is smaller. Bottom: size of every code-file modification in the corpus's recent history, in lines changed per file, lines changed per commit, and the share of the file an edit rewrites. The horizontal axes are logarithmic except the last.

4 Viability

In Section 3.3 we note that human solvers were not able to solve all diffs using the four standard AST diff operations. We used that information to answer the following research question:

RQ1.1. What percentage of real-world code changes have an optimal AST node mapping that cannot be expressed using common AST diff operations (Insert, Delete, Update, Move)?

62 of 452 solved diffs (13.7%) require an N:M mapping: one or more nodes from one side map to one or more nodes from the other.[3] That rate is over the 452 changes of the Curated and Full lists. The two lists agree within sampling noise, at 15.0% of 220 changes on the Curated list and 12.5% of 232 on the Full list (Table 1). Those two are the datasets whose ambiguity pass is finished, and the pooled rate RA1.1 states is over them. The table also carries the Stratified sample (5.9%) and the Defects4J changes (14.2%), and a rate over all 1687 changes, but all three are lower bounds: those two passes are still running, and a change nobody has reviewed for ambiguity is recorded as having none. Read the gap between 13.7% and 10.2% as the distance annotation still has to travel, not as a difference between the datasets.

[3]The most common cause of an N:M mapping is a refactoring, where two or more similar blocks of code are moved into a common function. The correct solution is the one that maps all of them as moved.

Table 1: Changes whose optimal mapping needs an N:M site, per dataset of Section 3 and pooled.

Dataset	n	N:M changes	Share
Curated	220	33	15.0%
Full	232	29	12.5%
Stratified*	791	47	5.9%
Defects4J*	444	63	14.2%
Curated + Full	452	62	13.7%
*All**	1687	172	10.2%

*The ambiguity pass over the Stratified and Defects4J changes is not finished, so these three figures are lower bounds: a change nobody has reviewed for ambiguity is recorded as having none. *Curated + Full* is the rate over the two datasets whose pass is complete, and is the figure RA1.1 states.

RA1.1 (Expressibility)

13.7% of changes (62 of 452) have optimal solutions that cannot be expressed at all using the common AST diff operations (Insert, Delete, Update, Move).

This result bears on how tools are scored, because no existing tool can produce these mappings. OmniDiff's node-level accuracy (Section 8) counts a group as correct whenever its mapping is any of the pairings the group allows. The line-level comparison of Section 7

PREPRINT

needs one answer per node, so it projects each group through a single representative pairing, members paired in source order; a tool that chose a different, equally correct pairing can be charged for the lines on which the two differ.

Tree diffing algorithms find the lowest cost diff, i.e. smallest number of operations needed to transform one tree into another. However, this does not necessarily correspond to the optimal visualisation of the diff overlayed over the text of the source code.

RQ1.2. What percentage of real-world code changes have solutions that humans prefer whose cost is higher than the lowest cost AST diff, using the commonly accepted costs of AST diff operations?

The lowest edit distance was computed using a state of the art implementation. Of the 1217 fixtures with a human mapping, the two costs are equal 1049 times. In 33 (2.7%) the human solution is strictly costlier. The most common reason for this is that humans treat groups of tokens as one unit, not based on their syntax or semantical meaning, but rather based on higher-level logic. A common example is the `return` keyword; if one function was added and another deleted, matching the two `returns` leads to an optimal tree edit distance, but a human would find the match confusing because they typically treat the keyword as "belonging" to the logic of the function containing it. Existing tools do take notice of this. GumTree [14] avoids matching subtrees smaller than a threshold size to avoid such spurious mappings.

RA1.2 (Cost)

On at least 2.7% of the changes in the corpus (33 of 1217), the human-authored mapping costs strictly more, under unit costs for insert, delete and update and a free move, than a mapping a tree edit distance algorithm found for the same change.

5 Speed

This section explores the speed of the exact algorithm. The asymptotic complexity of APTED and Zhang-Shasha both is $O(n^3)$ for general trees, where n is the number of nodes in the tree. However, APTED performs considerably better in real trees because it is less sensitive to the tree shape. Given this existing direction in the literature, we wanted to further focus on source-code trees specifically and ask the following question:

RQ2. What percentage of real-world source-code changes can a single, whole-tree tree-edit-distance computation complete within one second?

One second was chosen as a budget because it is a practical limit that can be computed, and because it is where interaction stops feeling immediate. Nielsen's response-time limits [26] put one second at the boundary at which a user's flow of thought stays uninterrupted but the delay becomes noticeable; below about 100 ms a response reads as instantaneous. From an industry perspective, practitioners rate the reaction time of review tooling as very important [21], and developer satisfaction with code review falls as changes wait longer [7]; one second is the high end of acceptable. Developers have had decades to get used to instant diffs from line-based tools, so any AST diffing tool that expects to be adopted must meet a similar bar.

We sampled 19,798 real (before, after) file pairs from single-parent commits in the dataset, stratified per language and per size bucket, with the buckets defined in Section 3.3: up to 142 pairs in each of the 168 cells of 24 languages and 7 buckets. 127 cells reached that target; the rest hold every pair the corpus had. On each pair, we ran our highly optimised, state of the art APTED implementation, with a one second runtime budget enforced using common operating system runtime limits.

The left panel of Figure 4 shows, for each artifact category, the share of pairs completed as a function of the time allowed. Size decides the outcome: within one second the computation finishes for 97.7% of code file pairs at 31 to 100 lines and for 46.1% at 101 to 300. The shape of the input matters as well: scripting languages complete more often than code, and data files less often.

RA2 (Speed)

Within one second, pure tree edit distance algorithms find the diff for only 46.8% of code pairs.

6 Uniqueness

This section measures how often a change has more than one correct answer, first in the mapping and then in its rendering. Existing tools all provide a single diff. Line based tools mark lines as inserted, deleted or match lines 1:1. Tree based tools and algorithms have 4 basic operations: insert node, delete node, update node and match nodes 1:1. Some algorithms extend the basic operations with tree equivalents, e.g. insert tree or delete tree. No tool allows for multiple solutions.

Manually examining real-world inputs, we found that some diffs have multiple equally correct solutions: multiple equally correct 1:1 mappings within a given many-to-many node mapping must cover one side of the mapping completely, and the remaining nodes on the other side should be marked as inserted or deleted, as appropriate.

Figure 5 is a motivating example. The AST of the before snippet contains one `boolean_operator` node. The after snippet contains two, one a direct child of the other. There is no objective way to determine which of the two is added and which one should be matched. The edit distance cost of both mappings is exactly equal, and since `boolean_operator` is an internal node with no rendering of its own, both also produce the same visualisation—so human judgement cannot break the tie either.

RQ3.1. What percentage of real-world code changes have multiple optimal AST node mappings?

We count a change as admitting several optimal mappings when its ground truth records at least one *multi-map group*: a set of nodes on one side and a set on the other that the annotator judged interchangeable, so that any consistent pairing within the group is equally correct. Across the corpus the annotators recorded 983 such groups.

Even without such groups, multiple solutions can exist. Where the annotator's mapping and the matcher's mapping for one change cost exactly the same under the unit cost model and still disagree about which nodes pair with which, the cost function cannot separate two different answers. That happens on 27 changes (2.2%).

PREPRINT

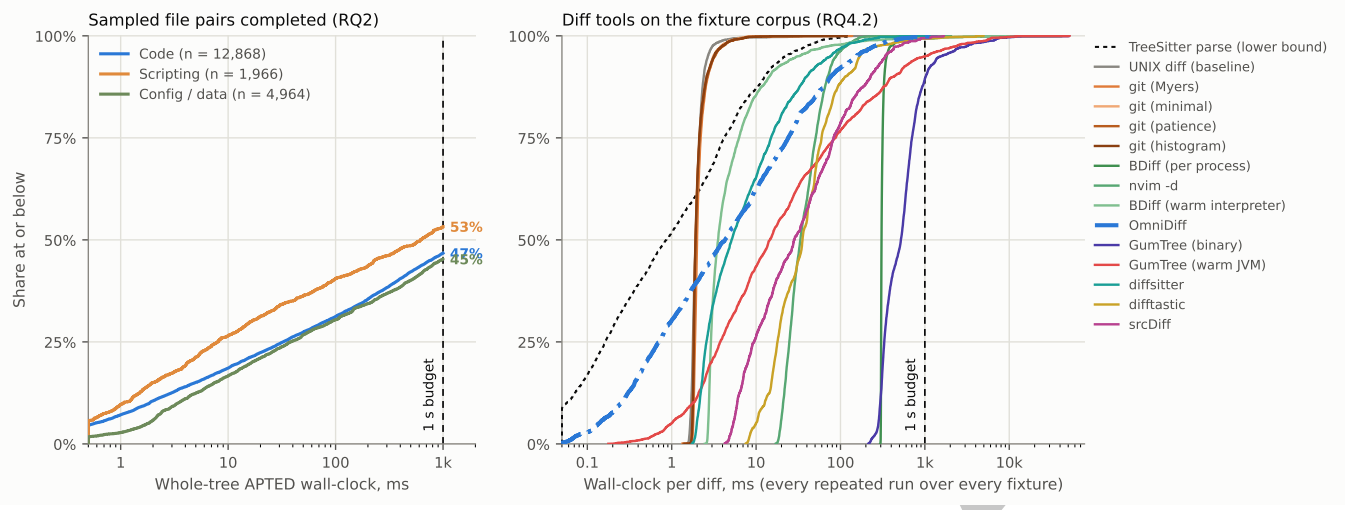


Figure 4: The one-second budget, read the same way twice. Left: the share of sampled file pairs whose whole-tree APTED computation completes within the time on the axis, per artifact category; the height of each curve at the dashed line is RA2, and the pairs the budget killed are the gap to 100%. Right: wall-clock per diff for every tool configuration of Section 7, every repeated run over every fixture, with OmniDiff charged for its parse and drawn dash-dotted; the dotted curve is the tree-sitter parse alone, the floor every AST-aware tool pays first. The dashed line is the same one second in both panels.

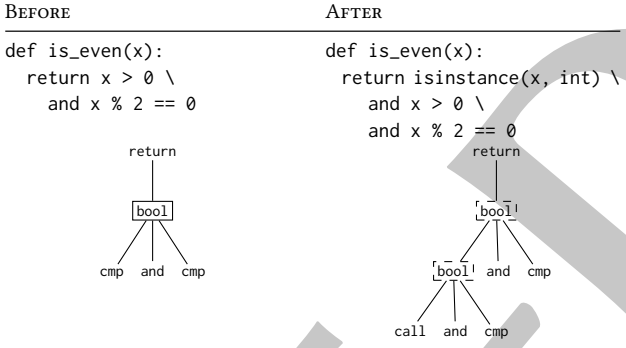


Figure 5: A change with no unique correct mapping. The before AST holds one `boolean_operator` node (`bool`, boxed) and the after AST two, one nested in the other (dashed); matching the before node to either costs the same and renders the same. `cmp` is a `comparison_operator`.

RA3.1 (Mapping uniqueness)

In at least 13.7% of changes (62 of 452), humans found that some set of nodes admits several equally correct pairings. Independently of that annotation, on 2.2% of changes the human and algorithmic mappings tie on cost while disagreeing on the mapping, so the cost function alone cannot pick between them.

RQ3.2. What percentage of real-world code changes have multiple equally correct textual visualisations of the diff?

Settling the mapping does not settle what the reader sees. Of the 1420 painted changes, 880—62.0%—carry at least two paintings, the annotator's finding that there are multiple correct renderings of the same change.

RA3.2 (Visualisation uniqueness)

Of the 1420 painted changes taken from real commits, 880 (62.0%) admit at least two equally correct renderings.

6.1 How much of the mapping reaches the reader

RQ3.3. What percentage of the AST influences the textual visualisation?

A syntax tree contains two kinds of nodes. Some carry text of their own—an identifier, an operator, a literal, a comment. Others are pure structure: a `block` whose extent is exactly its children's, an `expression_statement` wrapping one expression, the chain of single-child nodes a grammar interposes between a statement and the name inside it. A rendered diff highlights regions of source text, so a structural node does not appear in the output on its own. A tool that maps every displayed token correctly and every structural wrapper wrongly produces a diff that is correct; such a tool scores badly on a metric that counts all nodes equally. A node is visible if any byte inside its extent lies outside every one of its children—that is, if it contributes text of its own to the rendering.

RA3.3 (Visible share of the AST)

32.4% of the AST nodes in 1687 fixtures carry no text of their own and cannot appear in a rendered diff at all: 67.6% are visible. Roughly one AST node in three is scaffolding.

PREPRINT

The consequence is not that node-level metrics should be abandoned. It is that a rate over all nodes and a rate over visible nodes answer different questions, and any paper reporting one should state which. This paper reports both.

7 State of the Art

We run every established diffing tool we could obtain over the corpus of Section 3, score each against the same human mapping, and time each on the same machine.

RQ4.1. How accurately do current state-of-the-art code diffing tools find optimal visualisations?

RQ4.2. How fast are real-world state-of-the-art tools?

Comparison tools. We measure eight established tools, all introduced in Section 2, on 1217 of the fixtures described in Section 3.[2] `git diff` is measured under each of its four algorithms, Myers, minimal, patience and histogram, giving eleven configurations in all. Five configurations report at line granularity only: Unix `diff` and the four `git` algorithms. Six report sub-line detail: BDiff [23], Neovim's diff mode, and the four AST-aware tools, GumTree (v4.0.0-beta8) [14], difftastic [18], diffsitter [12] and srcDiff [9]. Our own tool's figures are reported in Section 8, on the same basis and against the same ground truth.

Line-level agreement. None of the eleven configurations reports an AST-node mapping in a form that can be compared directly, so we compare them on a common ground: which source lines each marks as touched, against the lines the human mapping touches. Coverage varies widely: of the 1217 fixtures this comparison was run over, the text-based tools cover all of them, while diffsitter's compiled-in grammar set can process 891, GumTree's generator set 1005, difftastic's 1182, and srcDiff, which parses five languages, 519. Because those subsets differ, percentages are used.

We report agreement per fixture rather than pooled over lines in Table 2, and Figure 6 splits the same reading by the dataset each fixture was sampled from. *Perfect* means zero mismatched lines. ≥99% is "a developer has to look twice"; 95–99% is "a developer has to check". Below 95% the tool is no longer usable.

RA4.1 (Tool accuracy)

srcDiff renders 79% of the changes it parses perfectly, but it parses five languages; no tool that parses most of the corpus renders much more than three fifths of it perfectly, the best being difftastic at 64%.

The split by dataset in Figure 6 orders the datasets the same way for every configuration: each scores its own best on either the Stratified sample or the Defects4J changes, and all five line-granularity configurations do best on Defects4J. Bug fixes are short, localised edits, which is the shape a line-based diff recovers most reliably, so the ordering is a property of the datasets rather than of any tool.

Cost. Table 3 reports wall-clock percentiles, sorted fastest to slowest by median. Each tool is measured over the fixtures it supports rather than over one common subset, so the populations behind these rows differ—the text-based tools draw on every fixture, while an AST tool draws only on the languages its grammars cover. Two tools are reported twice: GumTree once invoked as a fresh process per fixture, the realistic cost of a single CLI invocation, and once run inside one persistent JVM across all fixtures, isolating JVM startup from GumTree's own matching cost; and BDiff likewise per process and inside one persistent Python interpreter, for the same reason.

RA4.2 (Tool cost)

The eleven configurations span more than two orders of magnitude at the median, from Unix `diff`'s 1.9 ms to GumTree's 534.9 ms per invocation. At the 99th percentile the same range runs from 4.1 ms to 6570.9 ms, three orders of magnitude apart.

We report both a cold and a warm figure for GumTree and BDiff because they answer different questions: the cold number is what a developer running a diff from the command line waits for, and the warm number is the matching cost once process startup is amortised away.

Every timing number in this paper is the result of repeated runs rather than a single measurement. Each tool is run 3 times on every fixture, and the percentiles are taken over every individual run pooled across fixtures rather than over per-fixture means: the repeat count is uniform, so no fixture is over-weighted.

8 OmniDiff

This section is the paper's tool contribution. Table 4 lists what each empirical result decided in OmniDiff's design; the phases that follow are the same list read top to bottom, and the tool is then evaluated against the same ground truth and on the same basis as Section 7's eleven configurations.

OmniDiff has two targets:

- **Robust:** OmniDiff must handle every file up to the measured maximum, 23,584,040 AST nodes, with headroom above that number.
- **Fast:** OmniDiff must compute a diff in under 1000 ms for 99% of real-world changes, RQ2's one second applied to the population of real changes rather than to a corpus of hard cases.

OmniDiff matches two ASTs in five phases. Each phase either resolves part of the mapping directly or narrows the residual left for the next phase. Later phases only ever see nodes earlier phases left unmatched. A sixth step closes the mapping, recording the deletion or insertion implied by whatever every phase above left undecided.

Phase 1, hash-based descent. OmniDiff matches subtrees by two hash variants, in order: byte-identical subtrees, then same-shape subtrees with any leaf value. This phase is motivated directly by the commit-size data of Section 3: since most commits touch a small fraction of a file's lines, most of a typical file's AST is byte-identical between before and after. Resolving that identical majority with cheap hashing, before any tree-edit-distance computation runs at all, avoids paying $O(n^3)$ on content that never changed.

Phase 2, contextual exact matching. Comments and modifiers (attributes, decorators) that immediately precede an already-matched node, and diagnostic statements (logging calls, assertions, panics) that are byte-identical on both sides, get matched directly. Neither is reachable by phase 1's structural hashing: comments and modifiers are not part of the hashed AST shape, and diagnostic

PREPRINT

Table 2: Per-fixture line-level agreement with the human mapping, bucketed, for all eleven configurations. The upper block holds the tools that report whole lines only; the lower block holds the tools whose output carries sub-line detail, scored on the same line-level basis so that the eleven are comparable. "Perfect" means zero mismatched lines, not a rounded 100%. Each tool is scored on its own applicable subset (n), so percentages, not counts, are comparable across rows.

Tool	n	Perfect	≥99%	95–99%	<95%
Line granularity: line-only tools					
Unix `diff`	1217	713 (59%)	249 (20%)	160 (13%)	95 (8%)
`git` (Myers)	1217	718 (59%)	248 (20%)	159 (13%)	92 (8%)
`git` (minimal)	1217	718 (59%)	248 (20%)	159 (13%)	92 (8%)
`git` (patience)	1217	717 (59%)	247 (20%)	160 (13%)	93 (8%)
`git` (histogram)	1217	717 (59%)	247 (20%)	161 (13%)	92 (8%)
Line granularity: tools reporting sub-line detail					
BDiff (cold, per-invocation)	1217	688 (57%)	254 (21%)	179 (15%)	96 (8%)
`nvim -d`	1217	728 (60%)	247 (20%)	156 (13%)	86 (7%)
diffsitter	891	482 (54%)	204 (23%)	115 (13%)	90 (10%)
difftastic	1182	755 (64%)	197 (17%)	135 (11%)	95 (8%)
GumTree (cold, per-invocation)	1005	621 (62%)	158 (16%)	128 (13%)	98 (10%)
srcDiff	519	409 (79%)	68 (13%)	32 (6%)	10 (2%)



Figure 6: The line-level agreement of Table 2, split by the dataset of Section 3 each fixture was drawn from, with the pooled reading on the right. Each bar is one configuration over the fixtures it supports (n beside the bar); the segments are the four agreement buckets, and the Perfect segment carries its share.

statements are held back deliberately, so that matching one does not fragment a larger match around it. These are the preferences RA1.2 measures: each of them can cost more than the cheapest mapping.

Phase 3, syntax-aware subtree matching. This phase generalises three matching strategies: matching by fully-resolved name (handling overloads and duplicate names across scopes), a greedy cost-estimate matcher for containers with no name structure, such as an `if` block or a loop body, and an overlap matcher pairing import statements that share a base path. Before any of these run, large flat subtrees, such as a macro's token list or a big flat argument list, are pre-matched directly with Myers' algorithm [25], since a flat sequence has no tree structure worth exploiting. Each accepted pair is then handed to APTED [28], scoped to that pair alone, to compute its exact internal edit script. This is RA2 applied: the exact computation is affordable inside one matched pair, and only there.

Phase 4, residual alignment. Whatever remains unmatched after phases 1 through 3 is resolved by a Myers-LCS alignment [25] over the residual forest, bracketed by two passes of strict bottom-up propagation: a node whose children all match the children of exactly one node on the other side is matched to it, which shrinks the residual, and is run again afterward so that pockets the alignment just resolved can still bubble up to their now-complete ancestors.

Phase 5, move-detection recovery. The final matching pass. Ordered tree edit distance cannot express a subtree that relocated across a matched boundary as anything but a delete plus an insert. This phase, adapted directly from GumTree's recovery-mappings phase [14], pairs up whole subtrees left fully deleted on one side

PREPRINT

Table 3: Wall-clock time percentiles for the eleven tool configurations, milliseconds, sorted fastest to slowest by median, with OmniDiff on the last row. *Max* is the slowest single one of the 3 runs over any fixture. Every configuration is timed as a whole process except OmniDiff, which is timed in-process and so pays no process startup; see Section 9.

Tool	p50	p90	p99	Max
`git` (histogram)	1.9	2.6	5.3	118.8
`git` (minimal)	1.9	2.6	5.2	38.0
Unix `diff`	1.9	2.4	4.1	246.1
`git` (patience)	1.9	2.7	5.5	35.9
`git` (Myers)	2.0	2.7	5.4	137.6
BDiff (warm interpreter)	3.6	14.0	277.1	41747.6
diffsitter	5.4	38.4	188.9	511.9
GumTree (warm JVM)	14.0	434.5	5389.2	12200.9
srcDiff	31.0	230.0	818.2	1682.1
`nvim -d`	33.6	70.9	150.4	2039.3
difftastic	35.0	113.8	637.4	4068.6
BDiff (cold, per-invocation)	306.5	330.8	605.1	42611.0
GumTree (cold, per-invocation)	534.9	1027.3	6570.9	13740.2
OmniDiff	4.6	77.4	420.1	1337.4

Table 4: What each empirical result decided in OmniDiff's design.

Finding	Consequence
The median edit rewrites 3.2% of its file (Section 3.2)	Phase 1 settles the unchanged majority by hashing before any edit-distance computation runs
Whole-tree APTED finishes within one second for only 46.8% of code pairs (RA2)	APTED runs only inside one already-matched pair (Phase 3), never over the whole tree
On 2.7% of changes the human mapping costs more than the cheapest one (RA1.2)	Phases 2 and 3 encode reader preferences that cost more than the optimum: comments and modifiers follow their node, names beat cost, diagnostics are held back
13.7% of changes have no expressible optimum and 2.2% tie at different mappings (RA1.1, RA3.1)	The phase order is the tie-break: the earliest phase to claim a node wins, and the closing step marks what no phase claimed
32.4% of nodes never reach the screen (RA3.3)	Accuracy is reported on visible nodes and on rendered lines, not on the node count alone
No established tool that parses most of the corpus renders more than 64% of changes perfectly, and srcDiff 79% of the five languages it parses (RA4.1)	The bar the evaluation has to clear, on the same basis
Files reach 23,584,040 nodes (Section 3)	The Robust target below

Table 5: OmniDiff against the human mapping, in the four agreement buckets of Table 2 and on the same line-level basis, so that a row here can be read directly against a row there. *Perfect* means literally zero mismatched lines.

Dataset	*n*	Perfect	≥99%	95–99%	<95%
Curated	220	190 (86%)	20 (9%)	5 (2%)	5 (2%)
Full	232	194 (84%)	23 (10%)	9 (4%)	6 (3%)
Stratified	491	464 (95%)	8 (2%)	7 (1%)	12 (2%)
Defects4J	274	246 (90%)	15 (5%)	12 (4%)	1 (0%)
All	1217	1094 (90%)	66 (5%)	33 (3%)	24 (2%)

and fully inserted on the other, when they are byte-identical and large enough to rule out coincidence.

Result. OmniDiff renders 90% of the corpus perfectly, against 64% for the best established tool that parses most of it, at a median of 4.6 ms including its parse, and it completes every fixture. On the 519 changes srcDiff parses, it renders 89% perfectly against srcDiff's 79%.

Accuracy. Against the human mapping, OmniDiff maps 99.96% of AST nodes correctly (7,779 mismatches in 18,623,801), and 99.96% of the 67.6% of nodes that carry text and can reach the screen (5,204 in 12,587,060). On the line-level basis of Section 7 it marks 3,025 lines differently from the human mapping over 1217 fixtures, a rate of 0.221%, the lowest of any tool measured. Table 5 splits the result by dataset.

Robustness. Run over the 1,217 fixtures with a 120-second budget per fixture and no cap on tree size, OmniDiff completes every one. The largest fixture holds 198,406 AST nodes per side, the slowest takes 1236.9 ms, and peak heap use is 56.6 MB. The same measurement over the whole Full corpus reaches the files the fixture corpus does not. Every modified code file in the corpus's recent history (Section 3.2), 442,530 pairs across 25 languages, was diffed with no node cap under a 120-second budget and a 6 GB memory cap per process. OmniDiff completes 99.95% of them with 0 panics, at a median of 12.2 ms and a 99th percentile of 2,095 ms; the largest completed pair holds 7,092,498 AST nodes on each side, against the 23,584,040 of the largest file in the corpus that the Robust target names. The remainder is 96 pairs past the budget and 112 past the memory cap. The killed pairs are commits of 20 distinct files, all but one of them generated parser tables, minified bundles or embedded binary data; the exception is one commit of a 40,000-line single-header C++ library. Given 24 GB and a five-minute budget, one commit of each memory-killed file was tried again: 11 completed, one needed more memory still, and the rest ran past even that five-minute budget.

Speed. Over the 1217 fixtures, OmniDiff's median wall-clock time is 4.6 ms, its 90th percentile 77.4 ms and its 99th percentile 420.1 ms, parse included. The right panel of Figure 4 places that curve among the eleven configurations of Table 3: at the median the line-based tools sit to its left and the AST-aware tools to its right, and besides diffsitter, over the 891 fixtures its grammars cover, it is the only AST-aware curve that reaches 100% before the budget line.

PREDRAFT

9 Threats to Validity

Construct validity. Every rate this paper reports about ambiguity is a floor rather than an estimate. A multi-map group exists where an annotator noticed that two pairings were equally correct and recorded it; a change whose alternative mapping nobody saw is filed as unique. The same holds for the paintings. We report these as existence results and lower bounds throughout, and RA3.1's second reading—cost ties at differing mappings, which needs no annotation to detect—exists precisely because it does not depend on annotator attention. For the same reason we score RA1.1 and RA3.1 over the 452 changes whose ambiguity pass is complete rather than over all 1687. That pass covers the Curated and Full lists in full, so for those two lists the bound is one on annotator attention rather than one on coverage. The Stratified sample (47 of 791 changes) runs far under the rate the two finished lists show, and the Defects4J units (63 of 444) under it, which is an annotation gap rather than a finding about those changes; Table 1 prints both rows and its *All* row under that caveat.

Because no whole-file exact computation is affordable at these sizes (RA2), RA1.2 compares the human mapping's cost to a heuristics-augmented computation's rather than to a proven optimum. Its rate is therefore a lower bound.

Section 7 scores tools at the line level because that is the only granularity all eleven configurations share. A line-level projection of a node mapping discards detail that the AST-aware tools compute and the line-based ones never had, which if anything favours the line-based tools. It also means the paintings—the only place the corpus records a second correct rendering—never enter that comparison, so RA3.2's error term sits in those tables unremoved.

Internal validity. OmniDiff is timed in-process while every external tool is timed as a whole process, so it pays no process-startup cost that the others pay in full. We state this rather than correct for it, and report GumTree and BDiff both cold and warm so that a reader can see how large the effect is for the two tools where it dominates. Per-tool mean runtimes are additionally distorted by run order, whichever tool runs first per fixture absorbing cold-cache cost; to minimise this effect we quote percentiles rather than means throughout, and every timing number is the result of repeated runs.

The ground truth is authored by the same people who built OmniDiff, which is a real risk of bias in its favour. Three things limit it: the mappings are authored against the change itself in a dedicated annotation tool with no tool output shown, the full annotation history is published with the corpus so that any individual mapping can be inspected and disputed, and Section 3.4 compares our annotation against ground truth authored by other people, for a different tool, before this one existed. That last check is the only one of the three that does not depend on our own discipline, and it is the reason the corpus was extended with Defects4J at all.

External validity. The fixture corpus is deliberately not a fully random sample of real commits. It concentrates hard, structurally interesting changes, because easy ones exercise nothing worth measuring—so every tail figure in this paper is a stress reading rather than an estimate of what a developer waits for on a typical commit. Every accuracy result is also reported per dataset (Figure 6, Table 5), so a reader can separate the two random samples, the stratified one, and the Defects4J bug fixes from each other. The Defects4J rows carry the selection caveat of Section 3.3: the units solved so far are how far annotation has reached, not a draw, and they run shorter than the ones still unsolved.

The painted subset is smaller still and is not randomly drawn: painting is manual, and which changes have been painted is an artifact of the order the annotators worked in. It runs to shorter changes than the corpus as a whole, so RA3.2's rate may move in either direction as painting reaches larger changes.

Finally, the fixture corpus's largest file holds 198,406 AST nodes, well short of the Robust target's 23,584,040. The target is exercised only by the whole-corpus run of Section 8, whose budget and memory cap were applied with eight processes sharing the machine, so a pair near either limit may have crossed it for want of a core rather than of an algorithm. Re-running the timeouts on a quiet machine moved two of them into the completed set and no more, and the memory kills recur across every commit of the same files, so neither limit looks like an artefact of the sharing. The empirical-study numbers of Section 3 do cover the full 7,444-repository list, but the 1687 solved changes are drawn from it by the sampling of Section 3.3 rather than being representative of it by construction.

10 Related Work

Tree-diffing tools. Section 2 describes what each of the tools compared here does; what matters for positioning this work is where each one's goal differs from ours. GumTree [14] remains the reference point, and OmniDiff's move-recovery phase adapts its recovery-mappings heuristic directly: the difference is that GumTree targets research use, with broad language and backend flexibility that stays valuable well beyond what RA4.1's single corpus measures, while OmniDiff narrows to a tree-sitter-only frontend and spends the saved generality on production targets. difftastic [18] and diffsitter [12] share that parsing layer but optimise for a diff a human reads comfortably rather than for fidelity of the underlying node mapping, so RA4.1 measures them against an objective they did not choose. Against all of them, Unix `diff` and `git diff` [25] remain the baseline: on our corpus they are the fastest measured and, at the line level, more accurate than every pre-existing AST-aware tool compared. BDiff's [23] block awareness does not move that metric, and neither does the choice among `git`'s four algorithms [27].

Ground truth and its ambiguity. GumTree's own evaluation, and most that followed it, score a tool against a single reference mapping per change. RA1.1, RA3.1 and RA3.2 are an argument that this default understates what a corpus can be asked: on the changes where the correspondence is not one-to-one, where several mappings are equally correct, or where several renderings are, a fixed reference records a correct answer as wrong. The corpus here departs from the default in two ways—a multi-map group, which records an $N{:}M$ site and admits any consistent one-to-one pairing of it, and a second, independently authored painting where the rendering is not unique—and both were added because annotating real changes forced them, not by design. We are not aware of a code-diffing ground truth that records either. RA1.2 adds a third departure from the usual assumptions: the unit cost model that ranks candidate mappings disagrees with the annotators on a measurable minority of changes.

PREDRAFT

Empirical studies. Dyer et al. [11] and Lämmel et al. [22] mine AST nodes at scale to study API usage; Arafat and Riehle [3] measure the commit-size distribution of open-source software, the distribution OmniDiff's Fast target is stated against; and Defects4J [19] collects real, reproducible bug fixes from open-source Java projects, each a before/after file pair, and has since served as the file-pair corpus behind most evaluations of AST differencing on Java. This paper's empirical study differs from these in purpose rather than method: it measures file- and AST-size distributions specifically to set, and then evaluate, the engineering targets of a diffing tool.

11 Conclusions

Line-based and AST-based diffing tools can meet developers' speed requirements: the line-based tools answer within tens of milliseconds at every percentile measured, and the fastest AST-aware tools stay inside the one-second bound across the corpus (RA4.2). Current algorithms and implementations fall short of meeting the full quality requirements that real changes impose. On 13.7% of changes the best possible correspondence cannot be expressed with the four standard operations at all (RA1.1), on 2.7% the cheapest mapping is not the one humans judge correct (RA1.2), the mapping and its rendering are not unique (RA3.1, RA3.2), and no established tool that parses most of the corpus maps much more than three fifths of it perfectly (RA4.1). Closing that gap needs a ground truth that records the correspondence itself, a cost model that agrees with readers, and tools that are measured against both; the corpus published with this paper and the OmniDiff tool are a first step towards all three.

References

[1] Juan Pablo Sandoval Alcocer, Alejandra Cossio-Chavalier, Tiara Rojas-Stambuk, and Leonel Merino. 2023. An Eye-Tracking Study on the Use of Split/Unified Code Change Views for Bug Detection. *IEEE Access* 11 (2023), 136195–136205. https://doi.org/10.1109/ACCESS.2023.3336859

[2] Pouria Alikhanifard and Nikolaos Tsantalis. 2025. A Novel Refactoring and Semantic Aware Abstract Syntax Tree Differencing Tool and a Benchmark for Evaluating the Accuracy of Diff Tools. *ACM Transactions on Software Engineering and Methodology* (2025). https://doi.org/10.1145/3696002

[3] Osama Arafat and Dirk Riehle. 2009. The Commit Size Distribution of Open Source Software. In *2009 42nd Hawaii International Conference on System Sciences*. 1–8. https://doi.org/10.1109/HICSS.2009.421

[4] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*. IEEE, 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[5] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*. IEEE, 134–144. https://doi.org/10.1109/ICSE.2015.35

[6] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2019. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering* 24, 4 (2019), 1762–1798. https://doi.org/10.1007/s10664-018-9676-8

[7] Louise Braunstein, Brendan Fallon, Rebecca Lewis, and Kristin Pandit. 2022. Move faster, wait less: Improving code review time at Meta. Meta Engineering blog, https://engineering.fb.com/2022/11/16/culture/meta-code-review-time-improving/. Accessed 2026-09-24.

[8] Rodrigo Brito and Marco Tulio Valente. 2021. RAID: Tool Support for Refactoring-Aware Code Reviews. In *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC 2021)*. IEEE, 265–275. https://doi.org/10.1109/ICPC52881.2021.00033

[9] Michael John Decker, Michael L. Collard, L. Gwenn Volkert, and Jonathan I. Maletic. 2020. srcDiff: A syntactic differencing approach to improve the understandability of deltas. *Journal of Software: Evolution and Process* 32, 4, Article e2226 (2020). https://doi.org/10.1002/smr.2226

[10] Marco di Biase, Magiel Bruntink, Arie van Deursen, and Alberto Bacchelli. 2019. The effects of change decomposition on code review—a controlled experiment. *PeerJ Computer Science* 5 (2019), e193. https://doi.org/10.7717/peerj-cs.193

[11] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining billions of AST nodes to study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 779–790. https://doi.org/10.1145/2568225.2568295

[12] Afnan Enayet. 2020. diffsitter: A Tree-Sitter-Based AST Difftool for Meaningful Diffs. https://github.com/afnanenayet/diffsitter. Accessed 2026-07-31.

[13] Jean-Rémy Falleri and Matias Martinez. 2024. Replication package of "Fine-grained, accurate and scalable source differencing". Zenodo. https://doi.org/10.5281/zenodo.10474674

[14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. Västerås, Sweden, 313–324. https://doi.org/10.1145/2642937.2642982

[15] Enrico Fregnan, Larissa Braz, Marco D'Ambros, Gül Çalıklı, and Alberto Bacchelli. 2022. First come first served: the impact of file position on code review. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. Association for Computing Machinery, 483–494. https://doi.org/10.1145/3540250.3549177

[16] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. 2017. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2017)*. IEEE, 71–79. https://doi.org/10.1109/VLHCC.2017.8103453

[17] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013)*. IEEE, 121–130. https://doi.org/10.1109/MSR.2013.6624018

[18] Wilfred Hughes. 2018. Difftastic: A Structural Diff That Understands Syntax. https://github.com/Wilfred/difftastic. Accessed 2026-07-31.

[19] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[20] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. 2016. Code review quality: how developers see it. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, 1028–1038. https://doi.org/10.1145/2884781.2884840

[21] Gunnar Kudrjavets and Ayushi Rastogi. 2023. Does code review speed matter for practitioners? *Empirical Software Engineering* 29, 1, Article 21 (2023). https://doi.org/10.1007/s10664-023-10401-z

[22] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. 2011. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. 1317–1324.

[23] Yao Lu, Wanwei Liu, Tanghaoran Zhang, Kang Yang, Yang Zhang, Wenyu Xu, Longfei Sun, Xinjun Mao, Shuzheng Gao, and Michael R. Lyu. 2025. BDiff: Block-aware and Accurate Text-based Code Differencing. arXiv preprint arXiv:2510.21094, https://github.com/BDiff/BDiff. Accessed 2026-08-23.

[24] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189. https://doi.org/10.1007/s10664-015-9381-9

[25] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1, 2 (1986), 251–266. https://doi.org/10.1007/BF01840446

[26] Jakob Nielsen. 1993. *Usability Engineering*. Morgan Kaufmann, San Francisco, CA.

[27] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. How different are different diff algorithms in Git? Use `-histogram` for code changes. *Empirical Software Engineering* 25, 1 (2020), 790–823. https://doi.org/10.1007/s10664-019-09772-z

[28] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient Computation of the Tree Edit Distance. *ACM Transactions on Database Systems* 40, 1, Article 3 (2015), 3:1–3:40 pages. https://doi.org/10.1145/2699485

[29] Md Shamimur Rahman, Zadia Codabux, and Chanchal K. Roy. 2026. Breaking the Alphabet: Rethinking File Ordering in Code Review. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE '26)*. Association for Computing Machinery, 2492–2504. https://doi.org/10.1145/3744916.3787838

[30] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18)*. Association for Computing Machinery, 181–190. https://doi.org/10.1145/3183519.3183525

[31] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes? An exploratory study in

PREPRINT

industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. Association for Computing Machinery, Article 51, 11 pages. https://doi.org/10.1145/2393596.2393656

[32] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2022. Do explicit review strategies improve code review performance? Towards understanding the role of cognitive load. *Empirical Software Engineering* 27, 4, Article 92 (2022). https://doi.org/10.1007/s10664-022-10123-8

[33] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262. https://doi.org/10.1137/0218082

PRE-DRAFT