CANDYMERGE: CI/CD as Oracle for Candidate-Driven Merge Conflict Resolution

***Abstract*—Automated merge conflict resolution approaches are typically evaluated against a single reference: the resolution a developer committed. In line with prior empirical findings, we argue that a merge resolution should be judged not by textual similarity to that reference, but by whether it builds and passes the project's tests. We build our approach on this principle: rather than guess a single answer, use the project's CI/CD pipeline as an oracle and let compilation and testing pick the best resolution from a set of likely candidates. We present CANDYMERGE, a candidate-driven merge conflict resolver that treats resolution as a guided search over whole-merge *variants*—candidate resolutions of the entire merge—built by combining conflict resolution patterns from recent empirical studies. Instead of resolving each chunk independently, it generates coherent variants that resolve all conflicts jointly, ranks them so the most promising are validated first, and returns the best one the oracle identifies.**

We evaluate CANDYMERGE along three axes: search guidance, search efficiency, and resolution quality against the developers' committed resolutions and competing approaches. We summarize our findings as follows: First, a learned predictor efficiently guides the search toward promising variants, locating the developer's committed resolution within one or two attempts in the median. Second, contrary to our initial expectation, parallelization and caching do not help a good search order reach the best resolution sooner. Third, on 1,000 real-world merges from 268 Java/Maven projects, CANDYMERGE's best validated variant is comparable to or better than the developer's committed resolution on 90.6% of merges under our build-and-test oracle. CANDYMERGE trades time for quality: with a single build it matches the best merge tools, and with its full time budget it clearly outperforms every tool we could obtain and apply. Notably, the search sometimes identifies resolutions that build and test better than the developer's committed resolution (5.6% of merges), highlighting CI/CD-validated, candidate-driven merge conflict resolution as a practical alternative to merely imitating human behavior in automated conflict resolution.

***Index Terms*—Software Merging, Merge Conflict Resolution, Candidate-Driven Search, CI/CD Oracle, Machine Learning**

I. INTRODUCTION

Parallel software development in teams makes merging unavoidable. During a merge, the concurrent changes of multiple working copies are reconciled into a shared development artifact [2]. In about 5–20% of all merges, *merge conflicts* occur [3], [4], [5], [6]. These conflicts are highly disruptive [7], as work on the shared artifact cannot continue until the conflict is *resolved*. Decades of research and increasingly sophisticated merge tools have still not led to their adoption in practice [8], [9]. We hypothesize that this stems from three major gaps in how prior research frames conflict resolution.

1. While nearly all merge conflict resolution approaches target source-code artifacts, typically for one language at a time [10], [11], [12], [13], [14], [15], the resolution of conflicts in non-source artifacts (e.g., build and configuration files) can by itself decide whether the merged project compiles and passes its tests [16], [3], [17], [18].

2. Only a few conflict resolution approaches treat a merge's *chunks* (i.e., its conflicted regions) jointly, even though their resolutions often depend on each other [8]. We found only three approaches capable of this crucial ability [19], [20], [21].

3. Existing approaches almost all avoid what is arguably the most valuable evidence that a merge is correct—whether it builds and passes its tests—with only two exceptions [22], [23]. They have good reasons: only a fully resolved merge builds at all [14], [24], old merges often no longer build as their dependencies vanish [23], [13], builds and tests do not always reproduce [25], [16], and validating every candidate is costly [26]. Lacking this signal, the field's benchmarks [27], [28], [29] treat the developer's committed resolution as the 'ground truth' and score a merge by whether it reproduces that resolution verbatim. However, the developer's committed resolution is not necessarily correct, as prior work shows it to be "biased" [12], [30], "unreliable" [31], and "faulty" [23]. Even when correct, it is only one of many valid resolutions, as the literature notes: developer resolutions are "a lower bound, as multiple resolutions may be correct" [15], exact-match metrics reject "semantically equivalent code" [29], and developers accept non-verbatim alternatives [32].

Fig. 1 shows a real-world broken 'ground truth': the developers broke the committed merge, confirming it themselves in their next commit, titled "*fix compile error*" 17 minutes later. A naive verbatim match would reward this broken resolution.

Following prior work, we treat a merge that builds and passes the project's tests as correct [16], [33], [34], a necessary but not sufficient proxy, since tests can miss errors [16], [3]. We use this not merely to evaluate a finished merge, as these works do, but as an *oracle* during resolution, building and testing variants to measure their quality. This oracle is the project's own build and the tests its build file declares, run locally: a subset of the project's full CI/CD pipeline. Because the 'ground truth' can be broken, an automated resolution can even improve on its build and test results—e.g., by keeping both sides (OURS's `moduleName` and THEIRS's `applicationName`) in the conflict of Fig. 1.

We present CANDYMERGE, a candidate-driven merge conflict resolver that addresses these three gaps. CANDYMERGE resolves all conflicting chunks of a merge together, across source and non-source files alike. Merge candidates are *whole-merge variants* formed by combining chunk resolutions drawn from a small set of textual, language-agnostic patterns distilled

```
public abstract class TransactionMessage {
    protected String groupId;
<<<<<<< OURS
    protected String moduleName;
||||||| BASE
=======
    protected String applicationName;
>>>>>>> THEIRS
    // ... shared handle(...) unchanged ...
```

(a) The conflict as `git merge --diff3` reports it: Starting from the common ancestor (BASE), both branches (OURS and THEIRS) add a field to the same place in class `TransactionMessage`.

```
public abstract class TransactionMessage {
    protected String groupId;
    protected String moduleName;
    // ... shared handle(...) unchanged ...
```

```
class TransactionNotifyEvent
    extends TransactionMessage {
  void init(String app) {
    this.applicationName = app;
    // ERROR: cannot find symbol
```

(b) The consequence: The developer decided to resolve Fig. 1a by only keeping the field `moduleName` (OURS), but the subclass `TransactionNotifyEvent` merged in from the other branch (THEIRS) still references `applicationName`.

Fig. 1: Excerpt of a real merge (codingapi/tx-lcn, merge 0de724a4) where the developer's committed resolution is broken. When running the CI/CD pipeline, their resolution (OURS) builds only 4/13 modules and passes no tests. In this case, keeping both fields (OURSTHEIRS) restores `applicationName`, every reference resolves, and the build passes (13/13 modules, tests pass).

from prior empirical studies [8], [7] (gap 1). CANDYMERGE generates variants whose chunk resolutions are *jointly plausible*, learning which pattern combinations commonly occur together and ranking candidates accordingly (gap 2). Instead of language-dependent conflict analyses, CANDYMERGE delegates syntactic, semantic, and behavioral validation to a *generic oracle*, the project's CI/CD pipeline, selecting the variant that best satisfies compilation and test execution (gap 3).

The motivating example in Fig. 1 already hints at CANDYMERGE's potential: a correct resolution is quickly found in the OURSTHEIRS pattern, one of the chunk-level patterns from prior empirical work [8]. In general, however, merges comprise many conflicting chunks, so CANDYMERGE must combine their patterns into coherent whole-merge variants and identify the best one through CI/CD validation. This raises two empirical questions: *does it scale?* and *is it worth it?* We break them into three research questions: two on search guidance and efficiency, one on the quality of the resulting merges against both developer resolutions and competing merge tools.

RQ1: *Can we learn to guide search-space traversal?* Since a fixed time budget covers only a fraction of the search space, the exploration order becomes crucial. We therefore compare ranking predictor strategies by how well they prioritize high-quality variants. Our evaluation on 64,113 real-world Java merges found that every informed predictor reaches the developer's committed resolution within one or two attempts in the median.

RQ2: *Beyond ordering, does higher throughput find better resolutions or reach them sooner?* Ordering determines which variants are explored first. Within a fixed time budget, however, we can also increase *throughput* by building and testing more variants per unit time through optimizations such as caching and parallelism. We compare five configurations, a naive baseline and four combinations of caching and parallelism, measuring both the quality of the best variant at the end of the budget and the time taken to reach it. In an evaluation on 100 real-world merges, we found, against our expectation, that higher throughput does not reach the best variant sooner: a good ordering concentrates the most promising ones at the front of the queue, which every configuration reaches early.

RQ3: *How does the quality of our best validated variants compare to the human baseline and competing merge tools?* RQ1 and RQ2 establish how to search effectively and efficiently; RQ3 asks whether it pays off in practice. We evaluate on 1,000 real-world conflicting merges from 268 Java/Maven projects, comparing CANDYMERGE's best variant against the developer's committed one and every merge tool we could obtain and apply. We found that under our build-and-test oracle, CANDYMERGE's resolutions are mostly comparable to the developer's, and occasionally better or worse. Against other tools, the outcome depends on the time budget: with a single build CANDYMERGE matches the best of them, and with its full budget it clearly outperforms them.

We summarize our main contributions as follows:

- **A whole-merge candidate-driven conflict resolver.** CANDYMERGE resolves all chunks of a merge jointly, across source and non-source files alike. Its predictor reaches the developer's committed resolution within one or two attempts in the median (RQ1). Its throughput optimizations make repeated whole-project build-and-test practical within a realistic budget (RQ2).
- **Evidence at scale.** On 1,000 real-world conflicting merges from Java/Maven projects, and against every merge tool we could obtain and apply, CANDYMERGE is competitive with the developer's committed resolution; it matches the best alternative tools with a single build and outperforms them with its full time budget (RQ3).
- **CI/CD as generic oracle.** Overall, we demonstrate the feasibility of candidate-driven merging, in which multiple variants are validated by a CI/CD oracle. The oracle is tool-agnostic: we run a wide range of competing tools through it and rank their variants on equal footing.
- **Replication package.** All results, code, data, and supplementary material are online in our replication package [1].

II. CANDYMERGE ARCHITECTURE

Given a conflicting merge, CANDYMERGE returns a resolution of all its conflicts. It works as a search over *variants*, i.e., candidate resolutions of the whole merge (Sec. II-A). A

generator emits variants one at a time, the most promising first (Sec. II-B); a *build engine* builds and tests each, using the build-and-test steps of the project's CI/CD pipeline as an oracle (Sec. II-C). Within a given time budget, CANDYMERGE returns the variant the oracle scores best (Sec. II-D). RQ1 evaluates the predictor, RQ2 the build engine, and RQ3 compares CANDYMERGE against the human baseline and other merge tools.

A. Variants and Resolution Patterns

A merge reconciles two versions of a project, OURS and THEIRS, against their common ancestor, BASE. Where both versions edit the same region, Git emits a conflicting *chunk* that holds all three fragments (Fig. 1a).

For a single chunk, CANDYMERGE draws from a fixed set of *conflict resolution patterns* empirically shown to cover most committed resolutions [8], [7], [35]. The four atomic patterns are OURS, THEIRS, BASE, and EMPTY: the first three keep that version's fragment, and the last removes all of them. Concatenating two or three fragments in any order yields the remaining patterns (e.g., OURS followed by THEIRS in Fig. 1), for 16 in total. Any other resolution is treated as MANUAL.

A *variant* chooses a resolution pattern for *every* conflicting chunk in the merge. With $p = 16$ patterns available per chunk, a merge with n conflicting chunks admits p^n distinct variants. These variants form CANDYMERGE's search space. We call the resolution the developer committed the *human baseline* (e.g., the OURS choice that erroneously drops `applicationName`). It lies outside the p^n variants if it contains MANUAL chunk resolutions, or if the developer additionally edited non-conflicting regions during the merge [8].

B. Variant Generation with ML-AR

CANDYMERGE's generator decides in which order variants reach the build engine. At its core is a *predictor*, a model that predicts the pattern of each chunk; the generator turns its predictions into a ranked stream of variants. Our predictor, ML-AR, is an autoregressive transformer. It builds a variant one chunk at a time, basing each chunk's pattern on the chunk's features and on the patterns chosen for preceding chunks. Each variant thus represents a coherent resolution across all chunks rather than independent per-chunk predictions.

ML-AR is not tied to a particular feature set: it takes any numeric features describing the merge as a whole and each chunk. The model is a causal transformer with two layers, four attention heads, and hidden size 128. It reads one token for the merge, then one per chunk combining the chunk's features with the previous chunk's pattern. It is trained with a cross-entropy loss on the patterns developers chose in past merges.

ML-AR emits up to 1,000 distinct variants per merge. For the first variant, the generator takes the most likely pattern for each chunk. Later variants are sampled instead: duplicates are discarded, and the sampling temperature is increased when the duplicate rate becomes too high.

C. Build Engine

Our implementation of CANDYMERGE targets Java projects built with Maven, the dominant Java build tool [36], whose standardized build-and-test support enables automated experimentation and comparison with existing merge tools, most of which target Java [16], [29]. In Maven, a build is a reactor of *modules*: each module compiles and runs its own test suite. We assume a continuous-integration gate checking that all modules compile and tests pass, and adopt it as our oracle: a variant is evaluated by building it and recording how many modules compile and how many tests pass. A resolution is *perfect* when all modules build *and* all tests pass; the human baseline in Fig. 1 is imperfect, as only four of thirteen modules compile.

The search space is far too large to build every variant, so the build engine is optimized to evaluate as many variants as possible within the time budget. Several optimizations reduce the work per variant. *Copy-on-write workspaces* give each variant a thin overlay on one checkout of the project instead of a fresh copy. *In-memory builds* keep these overlays off the disk. A *warm build daemon* is reused across variants, and build steps the oracle never reads are skipped. *Cross-variant caching* lets later variants reuse the compiled output of the first variant that builds. *Parallel builds* run several variants at once. Finally, the engine *stops on success*, i.e., once a variant is perfect.

D. CI/CD Oracle: Selecting the Best Variant

Most variants are imperfect, and a binary accept/reject oracle cannot distinguish between two imperfect variants. To rank variants and pick the best one even when none is perfect, we grade each variant by how close it comes to passing: how many of its modules compile and how many of its tests pass. We call this its *build score*. Modules count first, since a module that fails to compile runs none of its tests. We pick the variant that compiles the most modules and, on a tie, passes the most tests. A variant v_1 is *better* than a variant v_2 ($v_1 > v_2$) if it compiles more modules, or the same number and passes more tests by over 1% of the tests run and by more than one test. This margin absorbs flaky tests [37], [38]. In the mirrored case v_1 is *worse* ($v_1 < v_2$), and otherwise the two are *comparable*. Switching from v_1 to v_2 has an *impact* when $v_1 > v_2$ or $v_1 < v_2$. Throughout the paper, we use *better*, *comparable*, and *worse* in this sense only, i.e., as judged by this oracle.

III. RELATED WORK

Approaches for automatic merge conflict resolution differ not only in *how* they construct a resolution but also in the *evidence they trust* to settle on one. We organize prior work from the weakest evidence to the strongest: none, heuristic, unclear, learned historical priors, surrogate similarity, static program structure, synthetic tests, and the project's own build and test results. Tab. I places representative tools on this ladder; CANDYMERGE occupies a cell no prior tool reaches.

A. Unclear Evidence

What a general-purpose LLM relies on to resolve a conflict is hard to pin down. The conflict-marked tentative merge, the very input a resolver consumes, is never committed, so a model trained on crawled repositories sees the parents and final

TABLE I: Representative merge tools placed by the strongest evidence they trust to settle on a resolution.

Approach	Search space	Evidence trusted
DIFF3 [39]	1, deterministic	none
IVN [16] / AUTOMERGE [40]	1 / $p \cdot n$	heuristic
Generative / LLM [15], [29], [41]	—	learned prior/unclear
Classifier [42], [35], [32], [43]	$p \cdot n$, independent	learned prior
Surrogate search [26]	$p \cdot n$, independent	surrogate similarity
Structured merge [44], [13], [12]	1, deterministic	static structure
Test-based [23]	$p \cdot n$, independent	(synthetic) tests
CANDYMERGE	$\mathbf{p^n = 16^n}$, **joint**	**build+tests**

merge but never their conflict [32], [27]. Genuine conflict-to-resolution supervision exists only in purpose-built corpora that replay `git merge` and localize the committed resolution [15], [29], themselves never checked to build or pass tests [45].

Yet several resolvers let a general-purpose LLM settle the merge outright, with no build or test gate. Some invest in elaborate context for the model—DEEPGRAPHMERGE [19], the RAG method of Zhang et al. [45], ROVER [46], and GMERGE [47]—while a study prompts CHATGPT zero-shot [41]; either way the raw generation is the resolution.

B. Learned Priors over Historical Resolutions

Data-driven resolvers infer a resolution from historical examples and judge it by the knowledge encoded in their models. Pattern-based classifiers predict a resolution *strategy* per chunk—RPREDICTOR [42], MESTRE [35], MERGEBERT [32], and CHATMERGE [48] (with LLM generation for complex chunks)—and EDITFUSION [49] accepts or rejects each line-level edit, while generative models synthesize resolutions directly: DEEPMERGE [15], MERGEGEN [50], and LLMERGEJ [29]. All resolve each chunk in isolation and pick a resolution by model probability or by resemblance to past resolutions. CANDYMERGE's predictor also learns from historical priors as a proxy, but CANDYMERGE generates variants with conjoined chunk patterns.

C. Search-Based Ranking

Search-based approaches build an explicit search space and rank it by a cheap proxy for a good resolution, never validated by execution. AUTOMERGE [40] enumerates structured resolutions and ranks them by three-way-merge priority rules; SBCR [26] recombines parent lines under a similarity surrogate. Their per-chunk spaces are in fact larger than ours—AUTOMERGE averages fifty candidates per syntactic hole—but each hole or chunk is resolved *independently*, so the space never grows jointly across the merge, and it is ranked by a structural or textual proxy, not tested. Neither compiles nor tests its candidates, which SBCR deems "computationally expensive ... when dealing with a large number of candidates" [26].

D. Static Program Structure

A fourth line exploits program structure to suppress spurious conflicts, aligning program graphs into a single deterministic merge: semi-structured and structured merge—FSTMERGE [10], JDIME [51], SPORK [13], MASTERY [30], and refactoring-aware INTELLIMERGE [12]—at the cost of language-specific parsers [51]. IVN [16] instead resolves Java imports and version numbers with narrow heuristics, also in build-deciding files such as Maven `pom.xml`, yielding a single deterministic output.

E. Positioning CANDYMERGE

CANDYMERGE ranks variants by the project's own build and test results, over the whole assembled merge and including the non-source files that decide it. The closest prior art is test-based: Xing and Maruyama [22] repair the merged program against prepared test cases, but on a single hand-built example and with no released tool, while SMERGE [23] supplements the project's own tests with EvoSuite- and GPT-4-generated tests, validating one conflict at a time on 153 conflicts. Neither ranks a variant set by build score, and neither scales the project's own suite over a whole variant. Across variant multiplicity, ranking by real tests, whole-merge scope, language-agnostic operation, and non-source coverage, no prior tool occupies CANDYMERGE's position (Tab. I). Lighter tools resolve many merges instantly. CANDYMERGE accepts a longer wait in return for the evidence of a build and test run. This evidence has stayed unused because the oracle is costly; making per-merge build-and-test affordable is the throughput contribution we evaluate in RQ2. For our comparative evaluation, we extend the only prior cost model of conflict resolution, that of Schesch et al. [16], to outcomes that beat the human baseline.

IV. RQ1: GUIDED SEARCH-SPACE TRAVERSAL

Our first research question is one of the two pillars of CANDYMERGE's feasibility. A CI/CD run per variant often takes minutes to hours, so only a few variants can enter the pipeline. To be effective, CANDYMERGE must efficiently predict an ordering that prioritizes the most promising variants.

A. Method

We learn this ordering with variant-ranking predictors. Ideally, training and evaluation would use the full CI/CD pipeline and build score of Sec. II-D, but compiling and testing every variant is prohibitively expensive at scale. We instead use the developer's committed resolution as a tractable proxy. It is a practical surrogate under the assumption that comparable or better variants, if they exist, are at least similar; a predictor that quickly prioritizes the developer's resolution should thus surface those variants early too. The developer's resolution is thus only the target our predictors learn and are ranked by here. The judge of merge quality remains the CI/CD oracle: RQ3 evaluates all four orderings on it (Sec. VI-B).

We evaluate four predictors on the conflicts of our corpus of Java projects, mined in the style of MEGA [8]: the same Java GitHub projects, re-mined with Git's own merge and filtered for complete, reproducible merges, with duplicates across forks and mirrors removed. It aggregates conflicts from 860 Java GitHub projects (483 Maven, 377 other build systems). It contains 64,113 conflicted merges and 341,799 conflicted chunks spread over 220,987 files, spanning 20 years of merge

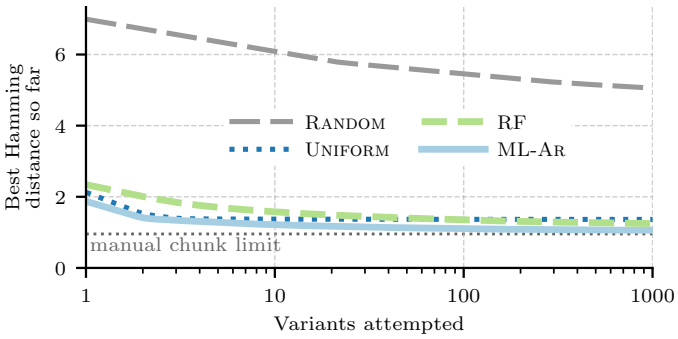


Fig. 2: Expected best Hamming distance within k attempts.

history until 2025-08-01. Crucially for this work, it records the human-chosen resolution of each conflicting chunk. It also provides the features we train our predictors on, largely those of Elias et al. [35]. Each merge has 30: authors and commits of each branch, commit-message keywords, numbers of changed and conflicting files, and branch duration. Each chunk has 17: lengths of the chunk, its fragments, and its context, its position in the file, chunks per file, and the file's cyclomatic complexity.

We evaluate the models using *forward chaining*. Merges stay atomic [45] and are split into ten chronological folds. Fold k is evaluated by a model trained on folds 0 to $k-1$ only. No predictor thus sees a merge from its evaluation period or later. The same holds for every learned ordering in RQ2 and RQ3 (checks: S1 in [1]).

B. Baseline Predictors

In prior literature, we found no predictor approach for our purpose, i.e., conditionally predicting the patterns of a merge's chunks and producing a ranked list of merge variants. We thus compare ML-AR (Sec. II-B) against three baselines of our own. As a naive baseline, RANDOM draws each chunk's pattern independently and uniformly from the 16 patterns. Requiring neither training nor domain knowledge, it sets the floor no informed predictor should fall below. The two informed baselines that follow grow in sophistication.

The simplest, UNIFORM, is count-based and exploits that a merge tends to reuse one pattern for all of its chunks [8]: it learns how often each pattern occurs across all chunks, then tries the most frequent in every chunk (all-OURS), the next (all-THEIRS), and so on. Each *uniform* variant uses one pattern throughout, so with 16 patterns it is exhausted after 16 variants.

RF instead predicts from per-chunk features, adapting Elias et al.'s state-of-the-art classification-based approach [35], a Random Forest that predicts each chunk independently. We follow their training protocol and reuse their feature set, except for the Java-keyword features, which our corpus does not provide (and which Elias et al. found weakly predictive). A variant combines the forest's per-chunk predictions. Since the per-chunk `argmax` would always compose the same variant, we instead sample multiple variants from its predicted distribution.

TABLE II: Target resolution hit rate and mean attempts (att.) within 1,000 variants (count-weighted); newest fold (fold 9), forward chaining.

#chunks	RANDOM		UNIFORM		RF		ML-AR		*Limit*
	%	*att*	%	*att*	%	*att*	%	*att*	%
1	*75.9*	*9*	**75.9**	2	75.8	2	**75.9**	**1**	*75.9*
2	*61.2*	*126*	42.9	**1**	**61.0**	3	**61.0**	3	*61.2*
3	*10.6*	*505*	35.2	**1**	**52.4**	9	**52.4**	7	*52.6*
4	*.6*	*708*	33.7	**1**	50.8	23	**51.1**	11	*51.1*
5–9	*0.0*	–	22.3	**1**	31.9	79	**35.8**	32	*38.1*
10–19	*0.0*	–	23.4	**1**	18.1	103	**29.4**	34	*35.5*
20–49	*0.0*	–	76.3	**1**	73.8	13	**77.9**	5	*79.5*
50+	*0.0*	–	57.0	**1**	42.1	50	**59.5**	4	*62.8*
mean	*44.8*	*45*	57.1	**1**	62.5	10	**64.2**	5	*64.9*
median	*0.0*	*11*	**100.0**	**1**	**100.0**	2	**100.0**	**1**	*100.0*

C. Results

Tab. II shows how often and in how many attempts the predictors exactly predicted human-chosen merge resolutions, counting the first 1,000 attempts. Take row 4: of 309 four-chunk merges, RF predicted all chunks correctly for 50.8% after 23 attempts on average. The limit for a pattern-based predictor is 51.1%, because one or more of the four chunks is often resolved with an unattainable MANUAL pattern.

We observe several properties of the predictors: beyond single-chunk merges only the informed predictors can compete, as RANDOM cannot guess the correct variant within 1,000 attempts. All three informed predictors clear this floor substantially for multi-chunk merges. For larger merges, RF collapses as it predicts chunks independently, while ML-AR's conditional prediction helps the many patterns of a variant 'fit together'. UNIFORM typically gets its prediction spot on within two attempts or not at all (all-OURS, all-THEIRS). ML-AR leads in hit rate for all merge sizes. Very large merges are resolved with all-OURS or all-THEIRS more often, as the rising limit and UNIFORM's performance beyond 20 chunks show. Notably, the median merge can be correctly predicted in one or two attempts by each informed predictor (final row), showing the feasibility of guided search-space traversal.

In Fig. 2 we complement hit rate with Hamming distance: the number of chunks whose pattern differs between a predicted variant and the human-chosen resolution. As argued above, this closeness is a rough proxy for build quality. For each merge, we report the minimum Hamming distance among all variants predicted so far, showing how closely each predictor approaches the human-chosen resolution as the budget grows. The best attainable distance for pattern-based predictors is 1.0, since the mean merge has that many MANUAL patterns.

One can see that ML-AR performs best: its first variant predicts ca. 1.9 patterns of the merge incorrectly, but within 1,000 variants its best variant approaches the theoretical limit. UNIFORM starts almost as close but plateaus once it is exhausted; RF, predicting each chunk independently, starts behind and overtakes UNIFORM only after about 80 variants.

RQ1: Guided Search-Space Traversal

All three informed predictors substantially outperform random sampling. In the median, they find the human-chosen resolution, our proxy, within one or two attempts. ML-AR performs best in both exact hits (mean: 64.2% within 5 attempts) and closeness to it. The more chunks a merge has, the higher the payoff from ranking variants.

V. RQ2: Throughput Beyond Ordering

RQ1 demonstrated that heuristic ordering concentrates the search budget on promising variants. However, for merges with many conflicting chunks, the search space remains large, and each variant evaluation can take minutes to hours. The time budget therefore binds. In RQ2, we investigate throughput optimizations that evaluate more variants within that budget.

A. Method

We sample conflicting Java/Maven merges across the projects of our corpus (Sec. IV-A): the projects in random order, one merge per project in turn, each project's most recent merge first. We retain the first 100 whose human baseline builds at least in part and, built three times, yields pairwise comparable results (Sec. II-D). The median build is the human baseline.

We build every project as its own build file prescribes and change only what the oracle needs: the build continues after a failure, so that every module and test is recorded. Every build runs on the oldest of our JDKs (8, 11, 17, 21, and 25) that supports the Java version its project declares. The exact commands, versions, and flags are in the replication package.[1]

The time budget of a merge derives from the build-and-test time of its human baseline. Because a human baseline that fails some modules or tests runs artificially short, we extrapolate the measured times to a full build:

$$\hat{T} = t_{\text{compile}} \cdot \frac{M_{\text{total}}}{M_{\text{built}}} + t_{\text{test}} \cdot \frac{N_{\text{total}}}{N_{\text{passed}}}, \tag{1}$$

with compilation time t_{compile}, number of modules M, testing time t_{test}, and number of tests N. If no modules built or no tests passed, the formula is undefined and we set $\hat{T} = 600\,\text{s}$, our timeout for a baseline build. The budget of $10 \times \hat{T}$, at least 300 s, covers generating, building, and scoring the variants.

We compare five configurations of the build engine (Sec. II-C). A *naive* configuration disables every optimization. The other four share the per-variant optimizations and cross *parallel* builds with cross-variant *caching*: *sequential*, *sequential + caching*, *parallel*, and *parallel + caching*. All rank variants with ML-AR, the best predictor from RQ1. For a fair throughput comparison, stop on success is disabled, so each processes as many variants as its budget allows.

B. Results

Fig. 3a shows the main throughput results. The naive configuration, without any of the optimizations of Sec. II-C, is on the left and has statistically significantly lower throughput than the optimized configurations (Wilcoxon signed-rank, Bonferroni-corrected, $p < .05$; Cliff's δ of .11–.41, negligible to medium). Even atop this optimized base, parallelism and caching remain unconvincing. Parallelization gains only slightly over the optimized sequential configurations ($\delta = .10$–$.32$, negligible to small). Caching adds little: most variants needed a full rebuild, and cache maintenance (SHA computation, index bookkeeping) largely outweighed its benefits.

Interestingly, higher throughput did not translate into finding the best variant sooner: Fig. 3b shows no configuration reaching the best variant meaningfully faster than the optimized sequential one (negligible effects, $\delta = .04$–$.08$). Most of the throughput gain comes from quickly discarding degenerate variants—those too broken to rank highly anyway, while the highest-ranked variants come first in every configuration. Tab. III shows the quality each configuration reaches at fixed shares of the budget: the parallel configurations lead early on, but from a fifth of the budget on, all configurations resolve nearly the same share of merges comparably or better. Parallelization thus pays off only where a resolution is needed within a small fraction of the budget, e.g., interactively.

In terms of consistency, we found that the parallel configurations, in a few cases, fail tests that the sequential configuration passes. This is the result of the concurrent builds competing for resources (e.g., for network ports; S2 in [1]).

TABLE III: Share of merges (%) whose best variant is comparable to or better than the human baseline.

	Share of the budget elapsed				
Configuration	5%	10%	20%	50%	100%
Sequential	29	52	90	93	93
Sequential + caching	30	52	91	92	93
Parallel	37	70	93	93	94
Parallel + caching	34	67	92	92	94

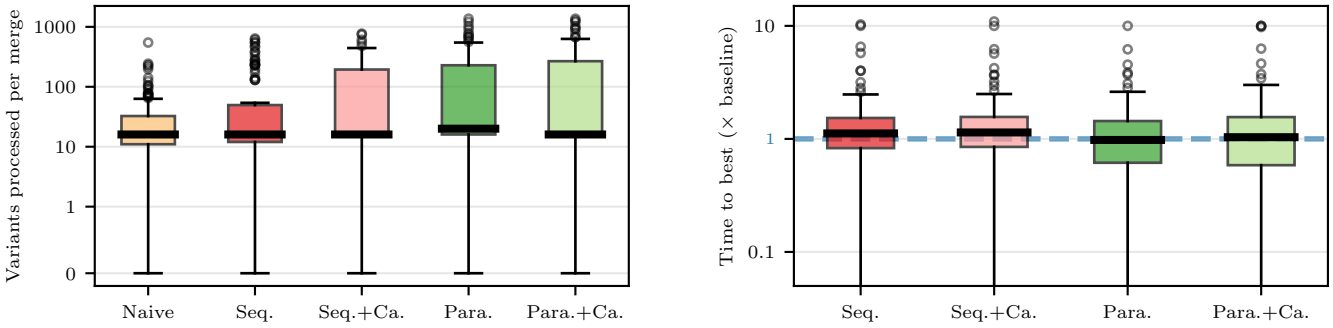
RQ2: Throughput Beyond Ordering

A strong ranking predictor makes heavy throughput optimization unnecessary: because ML-AR concentrates winning variants at the front of the queue, the optimized sequential configuration reaches the same quality as parallelization and caching. The optimizations that reduce the work per variant (copy-on-write workspaces, in-memory builds, a warm build daemon) do pay off. We therefore adopt the optimized sequential configuration as CandyMerge's default: it is by far the least complex, with the fewest parts that can break and the most consistent results.

VI. RQ3: CandyMerge Comparison vs. Baselines

The two prior research questions addressed ranking promising merge variants early (RQ1) and processing them faster (RQ2). This research question asks whether we meet our overall

[1] RQ2 ran on three virtual machines (32 vCPUs, 90 GB RAM each) and RQ3 on 16 virtual machines (16 vCPUs, 16 to 64 GB RAM), one merge at a time per machine; GPU-bound variant generation ran on a separate GPU server.



(a) Throughput: Variants processed per merge within budget.

(b) Time to reach the best variant.

Fig. 3: Variant exploration and time-to-best across configurations. Both plots are log-scaled.

goal in practice: how does CANDYMERGE compare to the human baseline and to other merge tools?

A. Method

We reuse the setup of RQ2 (Sec. V-A): sampling order, triple build of the human baseline, build configuration, and time budget. In this order, we attempt 2,451 merges. For 1,315 of them, the human baseline cannot be built and tested on our machines, leaving 1,136. We drop 18 whose baseline builds flakily: its three builds are not pairwise comparable. We also drop 118 whose baseline builds nothing and that no tool improves on. This leaves 1,000 evaluated merges from 268 projects (S3 in [1]). On these, we compare CANDYMERGE in its optimized sequential configuration against a range of merge tools. Every tool receives the same time budget for resolving a conflicting merge (wall-clock seconds: S6 in [1]). Unlike in RQ2, the search of every tool ends at its first perfect variant. As a budget ablation, we also report CANDYMERGE when it may build only its first variant (CANDYMERGE@1) or at most its first three (CANDYMERGE@3), next to the full budget (CANDYMERGE@budget). As an ordering ablation, we also run CANDYMERGE on the same merges and with the same budget under each of the other orderings of RQ1 (RANDOM, UNIFORM, RF) in place of ML-AR (S7 in [1]).

Comparative evaluations of automated merge resolvers are typically narrow, benchmarking against cherry-picked competitors while older, niche, or non-source-aware tools are excluded a priori. We reverse this stance: if the human 'ground truth' can itself be imperfect, a tool that prior work deemed uncompetitive under verbatim matching may nevertheless produce merges that compile and pass their tests. We therefore admit every merge tool available to us that operates on Java projects—from Java-aware (structured) to language-agnostic—and let our CI/CD oracle decide; Fig. 4 names them in order of publication year. We excluded SOMANYCONFLICTS [20], as it cannot be applied without leaking the 'ground truth' to the tool. We considered LLMERGEJ [29] but could not include it: it shipped without model weights and was too costly to retrain.

Several measures keep the comparison fair. Every tool runs through the identical harness—same build engine, flags, and the optimizations CANDYMERGE uses, so our speedups are no home-field advantage: every tool gains from in-memory builds, and multi-variant tools also from copy-on-write workspaces and the warm build daemon. The four tools driven by a general-purpose LLM (GMERGE, CHATMERGE, SMERGE, and ROVER) share a common model, QWEN3-8B, that fits our local GPU. Model inference is charged to the budget as generation time. We otherwise run each tool as true to its authors' released code, configuration, and weights as possible, resolving every remaining ambiguity in its favor; when a tool emits several variants, we rank them with our own CI/CD oracle and credit its best. Reused build scores of already-seen variants stabilize the comparison under flaky tests. MESTRE, RPREDICTOR, and EDITFUSION were retrained with forward chaining, like CANDYMERGE (Sec. IV-A).

From a tool's *build score* (#built modules, #passed tests), we classify its result as *worse* than, *comparable* to, or *better* than the human baseline (Sec. II-D). When a tool times out on a merge, declines to answer, or leaves a conflicted chunk unresolved, we record the merge as *unresolved*. We score every tool at the granularity of the whole merge. A merge tool's deliverable is a usable merge, so it counts as resolved only when *all* of its chunks are: a single leftover conflict—in a build script, a resource, or any other non-source file—leaves the project unmergeable. Under independent per-chunk resolution, the chance of a deliverable merge decays geometrically in the chunk count, so the per-chunk accuracy most prior resolvers report [28], [27] cannot be read as merge delivery. We report outcomes per merge throughout (Fig. 4). The *Cost* measure introduced below keeps this fair: a tool that masters `.java` but ignores other files simply accrues *unresolved* outcomes on the merges it leaves incomplete, exactly as a developer would still face them, and pays nothing where the non-source conflicts do not affect the outcome. Resolving every merge and abstaining when unsure both have merit, but precision is widely held to beat recall in merge conflict resolution—that is, resolving a merge incorrectly is more costly than abstaining, which itself is more costly than resolving it correctly [12], [23], [29], [16]. We therefore measure each tool's outcomes by its mean *Cost* per merge. We start from the cost formula of Schesch et al. [16],

who define, for $k \geq 1$,

$$Cost = 0 \cdot \#\text{comparable} + \#\text{unresolved} + k \cdot \#\text{worse}. \quad (2)$$

We extend their scoring framework by allowing negative cost: when a tool wins, the baseline needs a repair, so we credit a *better* outcome by the same amount we charge for a *worse* one,

$$Cost = -k \cdot \#\text{better} + \#\text{unresolved} + k \cdot \#\text{worse}. \quad (3)$$

A larger k penalizes overeager tools that resolve many conflicts, but often *worse*, while rewarding cautious or specialized tools that resolve few but, when they do, mostly *comparable* or *better*. To compare tools on *Cost* and time, we compute the Pareto front [52] of mean *Cost* vs. mean resolution time, where the resolution time of a merge is the time a tool spends generating and building variants, as a fraction of that merge's budget. A tool is on the front if no other tool is both cheaper and faster.

B. Results

Fig. 4 shows the main results of the tools' performance on our 1,000 merges. Because the oracle is tool-agnostic, the figure measures each generator through one shared build-and-test lens—same budget, same optimizations. The left panel reports each tool's outcomes. Most tools resolve few merges completely—usually because they handle conflicts only in `.java` files, but also from timeouts or parser errors (Java-only merges: S8 in [1]). Because our RQ1 predictor ranks whole *merge resolutions*—not independent *chunk patterns*—by predicted closeness to the human baseline, which we posited proxies build quality (Sec. IV-A), CANDYMERGE@1 already ranks among the best outcomes; only CHATMERGE, EDITFUSION, and SUMMER come close to its profile. Moving from CANDYMERGE@1 to the best of its first three variants (CANDYMERGE@3) and then its full budget improves the outcome sharply, then plateaus, pulling CANDYMERGE clear of them. Notably, CANDYMERGE@3 is also the variant closest to the human baseline (Tab. IV). Yet closeness is only a weak proxy: the further gains at the full budget come from variants that diverge from the human to beat it. The predictor is not the only ordering that achieves this: in our ordering ablation, the far simpler UNIFORM reaches similar outcomes, while RANDOM falls well behind (S7 in [1]). Tools can beat the human because the human baseline can be broken as in Fig. 1: in at least 50 of the 1,000 merges (5.0%), the developer's merge commit does not build while at least one tool's resolution does. The developers repaired 47 of these merges in a later commit, with a median delay of 13 minutes.

The center panel reports each tool's mean cost per merge; highlighted tools lie on the Pareto front at their respective k. CANDYMERGE@budget stays on the front at every k up to 8, CANDYMERGE@3 up to 8, and CANDYMERGE@1 up to 4 (S9 in [1]). As k grows, the penalty for a *worse* resolution dwarfs that for an *unresolved* merge. Abstaining then pays off more than risking a wrong merge. At $k = 8$, cautious tools such as MERGIRAF, which leave most merges untouched, stay on the front. Of the tools that resolve nearly every merge, only CANDYMERGE@3 and CANDYMERGE@budget stay ahead of not using any resolution tool (represented by DIFF3).

The right panel shows where each tool spends its time: generating variants (left of center) versus building and testing them (right). The number next to the curves is the mean share of its budget a tool spends per merge, i.e., the resolution time from which we compute the Pareto front. Little time on the building side (right of center) is not a strength: a tool that builds for less time than the human baseline's one build (top row) leaves merges unresolved or produces variants whose builds fail early. Most tools generate variants quickly; ALMOST RERERE and SMERGE are the exceptions, often exhausting their budget before producing a variant. On the build side, CANDYMERGE@budget is the outlier, spending most of its time after the first build starts. CANDYMERGE@1 spends a share similar to most other tools, CANDYMERGE@3 about twice as much.

A structural factor shapes these outcomes: most tools operate only on `.java` files, whereas non-source files often decide build success. Fig. 5 shows that varying only the `.xml` resolution has an impact in 53% of the merges with an `.xml` conflict, against 33% for `.java`, whereas the remaining file types rarely decide the outcome on their own. Tools restricted to `.java` files ignore artifacts that materially influence build quality, an effect captured by our whole-project scoring and the *Cost* measure. Fig. 6 shows a representative example: a JSON library migration split across a dependency update in `pom.xml` and the matching import in a `.java` file, where any `.java`-only tool can resolve at most half of the conflicting changes.

A second factor is the number of conflict chunks in a merge. Merges with many chunks are harder for almost every tool, but CANDYMERGE's lead grows with their number: a tool that resolves each chunk on its own has to get every one of them right, whereas CANDYMERGE builds combinations of all chunks and lets the oracle judge the merge as a whole. CANDYMERGE is thus most effective on merges whose resolution its patterns can express, especially merges with several chunks whose conflicting code the tests meaningfully cover. It is least effective where the developer's resolution needs a hand-written chunk, which its patterns cannot produce, or edits outside the conflicts, which no evaluated tool makes (S10, S11 in [1]).

RQ3: CANDYMERGE Comparison vs. Baselines

CANDYMERGE stays Pareto-optimal on merge cost. Its build engine lets the oracle check many whole-merge variants, yet even its first variant ranks among the best in the field. We further show the high impact of non-source files on the build score, `.xml` files in particular, yet most merge tools ignore them.

VII. DISCUSSION

A. Learning to Imitate vs. Learning to Build

Learned resolvers imitate the human baseline—their best case is the human, and by construction they cannot exceed it. CANDYMERGE's ceiling is instead the test suite, which can

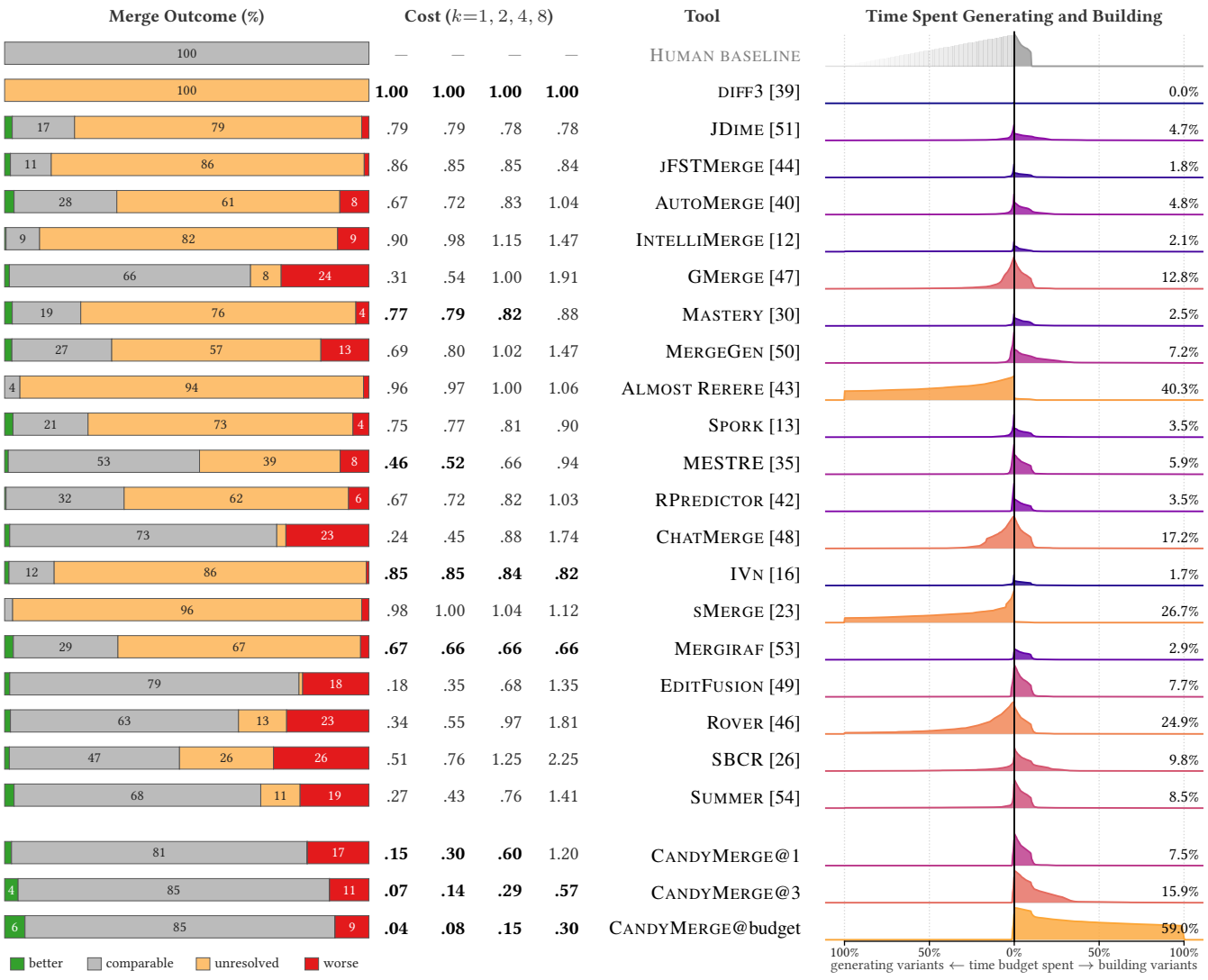


Fig. 4: Per-tool outcome, cost, and time. *Left:* outcome of each tool's best variant against the human baseline. *Center:* mean cost per merge at $k = 1, 2, 4, 8$; bold marks the Pareto-optimal tools (cost vs. mean resolution time). *Right:* share of merges on which a tool is generating variants (left of center) or building them (right of center), with the mean share a tool spends per merge.

sit above the human (cf. Figs. 1 and 4), whereas matching the developer's resolution is a by-product of selecting the build-best variant. The cases that build and test better than the human baseline are thus not luck but structural: an imitator's optimum is the reference, our oracle-driven search's optimum is build correctness. This shift is only possible because CANDYMERGE reasons over whole variants—complete, assembled merge states that can be handed to a build-and-test oracle—rather than imitating individual chunk resolutions; chunk-level imitation has no buildable artifact to certify. The same asymmetry explains why even the best verbatim matching tools (Tab. IV) miss the developer's resolution on most merges: a conflict often admits several equally correct resolutions, so the developer's committed resolution is one valid point among many [15], [32], and a CI/CD oracle credits the alternatives that a verbatim match scores as misses.

B. *Simplicity is Hard to Beat*

A single theme runs through our three research questions: the simplest option is consistently competitive, and the sophistication meant to improve on it often adds fragility or overhead instead. In RQ1, the UNIFORM predictor tries all-OURS, then all-THEIRS, and so on. It needs no features and no training beyond counting how often each pattern occurs. Yet its first guesses come almost as close as those of the autoregressive ML-AR (Fig. 2). In RQ2, neither parallelism nor caching reaches the best variant meaningfully sooner than the optimized sequential configuration (Fig. 3b): once a good ordering front-loads the winners, the extra throughput of the more elaborate configurations mostly serves variants that would

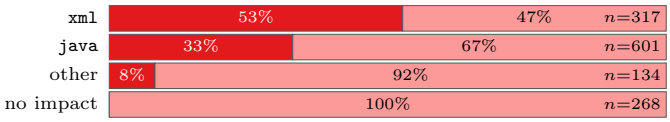


Fig. 5: Per-file-type impact on the build score: n denotes the number of merges with a conflict in such a file; the dark fraction is the lower bound of the share where resolving only that type has an impact.

```
<!-- core/pom.xml -->
<dependency>
<<<<<<< OURS
  <artifactId>javax.json-api</artifactId>
=======
  <artifactId>jackson-databind</artifactId>
>>>>>>> THEIRS
</dependency>
```

(a) Non-source half: a `<dependency>` conflict in `core/pom.xml`.

```
// PropertiesFileGenerator.java
<<<<<<< OURS
import ...core.util.JsonManager;
=======
import ...jackson.databind.ObjectMapper;
>>>>>>> THEIRS
```

(b) Source half: the matching `import` conflict in the same merge.

Fig. 6: One logical change—swapping the project's JSON library—split across a non-source and a source file (`git-commit-id-maven-plugin@2e86d4b6`). The conflicts are *coupled*: each OURS half builds only if the other is chosen too, and any mixed resolution drops the required library. A `.java`-only tool sees only Fig. 6b and ignores the POM; the developer resolved both to `javax.json`.

not win anyway, while adding flakiness and parts that can break. In RQ3, the resolution mechanism at CANDYMERGE's core is deliberately plain. It generates whole-merge variants by recombining each chunk's own textual fragments, builds and tests each, and keeps the best. Yet it is among the field's most robust. The ranking (RQ1) and throughput (RQ2) machinery serves only to make this recipe affordable at scale. At a high cost k, not resolving at all (DIFF3) beats every tool that resolves nearly every merge, except CANDYMERGE (Fig. 4). We read the pattern not as an argument against sophistication but as the bar it must clear: a good search order and a generic oracle let a plain mechanism carry most of the load, so added machinery must perform better without adding fragility.

C. Applicability of Approach

CANDYMERGE assumes a CI/CD pipeline as its oracle, but that is a realistic premise for professional software engineering. Beyond it, CANDYMERGE adds few assumptions of its own. It operates on Git's tentative merge, resolves conflicts with purely textual patterns, and applies wherever a project can be built. From this, the robustness observed in Sec. VII-B follows directly.

TABLE IV: Verbatim (merge-wise) agreement with the developer's committed resolution on the RQ3 merges: CANDYMERGE and three merge tools with the best verbatim F1.

Tool	Prec. %	Rec. %	F1	Cov. %
MESTRE	**40.5**	24.9	30.8	61.5
EDITFUSION	29.1	28.9	29.0	99.2
SUMMER	23.7	21.2	22.4	89.3
CANDYMERGE@1	30.7	30.7	30.7	**100.0**
CANDYMERGE@3	32.3	**32.3**	**32.3**	**100.0**
CANDYMERGE@budget	32.1	32.1	32.1	**100.0**

The open question is whether developers will accept the latency such an oracle adds—the trade-off between wait time and resolution quality. In our RQ3 experiment, every tool received a time budget of ten times the human baseline build and test time to generate and build its variants. ALMOST RERERE, SMERGE, and CANDYMERGE were the only tools to often use up the budget. Depending on the merge, a developer may not be able to afford this much time, which is why we also show CANDYMERGE@1 and CANDYMERGE@3 in Fig. 4. CANDYMERGE@1 has a footprint similar to most other merge tools, CANDYMERGE@3 one similar to the LLM-based tools. The full budget's higher latency, in turn, suits CANDYMERGE's natural home—an asynchronous CI bot rather than an interactive resolver: merge queues and speculative CI already spend compute validating hypothetical merge states, and CANDYMERGE reuses that existing build to validate hypothetical resolutions, paying only for the extra variant builds off the developer's critical path.

Good test suites are advantageous but not strictly required. Where a suite is weak, SMERGE [23] synthesizes extra tests (EvoSuite and LLM). We instead re-ran our RQ3 comparison under a *test-blind* oracle that ignores the test signal entirely—the worst case for a project whose suite is sparse or absent. Even test-blind, CANDYMERGE is comparable to or better than the human baseline on 90.9% of merges (85.7% comparable, 5.2% better), against 90.6% under the full CI/CD oracle, and it stays Pareto-optimal on cost for $k \in [1, 8]$, as under the full oracle (Fig. 4). The test signal only sharpens our approach by surfacing both more improvements (5.6% vs. 5.2% better) and more regressions (9.3% vs. 9.0% worse) that compilation alone cannot detect. CANDYMERGE thus degrades gracefully to whatever oracle a project affords.

D. Beyond Pattern-Based Variant Generation

CANDYMERGE is pattern-based: it resolves conflicts only by recombining existing fragments of a chunk and never authors MANUAL code (see Sec. II-A), giving it a natural limit on MANUAL resolutions, which the variant ranking predictors approach (Fig. 2). In RQ3, 30.1% of the merges contain a chunk the developer resolved by hand. On most of them, CANDYMERGE still finds a recombination that builds and tests as well as the developer's edit. Most chunks are resolved by

simple, predictable patterns [7], [8], [35], [42], which RQ1 reconfirms, so recombination suffices for most merges. Where it does bind, a two-stage design could break through this limit: first recognize the chunks that need a hand edit, which MESTRE [35] shows can be predicted, then hand only those chunks to a resolver that can write new code. We leave such hybrid generation to future work.

E. Threats to Validity

a) Construct validity: Our central construct concern is whether scoring every tool on a CI/CD oracle is a fair criterion: most competitors target a verbatim match to the developer's committed resolution and were neither designed nor trained to build and test. Yet CANDYMERGE scores best at the verbatim metric (Tab. IV)—its predictor is trained towards the developer's committed resolution; some tools reach higher *precision* only by abstaining on merges (e.g., MESTRE 40.5% over just 615 merges). Build score and verbatim match in fact diverge: going from CANDYMERGE@3 to CANDYMERGE's full budget improves the build score (Fig. 4) while slightly lowering the verbatim match (Tab. IV). We accept this threat deliberately—it is the paradigm shift the paper sets out to evaluate, from imitating the developer's text to certifying a build-correct resolution. Beyond fairness, a perfect merge certifies only compilation and the existing suite, which approximates rather than guarantees the intended or optimal resolution. We run only the self-contained tests declared in each `pom.xml`—a subset of the full CI pipeline—so absolute outcomes can shift, generally toward more failures, while relative comparisons stay valid because every tool and the human baseline share this identical setup. Moreover, building and testing are the only steps that practically every project's pipeline has, and the ones by which prior work judges merges [16], [33], [34]. Conflicts inside test files may be resolved by dropping tests or assertions, leaving the threat of 'reward hacking' the test suite until it fits [29]. Among the 333 cases across all tools in RQ3 where a resolution scored *better* than the developer's, we found no such case. Yet in 25 of them we judge that the better score does not reflect a better resolution, mostly because of version numbers in build files.

b) Internal validity: All experiments were conducted under identical conditions using a uniform evaluation pipeline. Flaky tests introduce non-determinism. We therefore keep only merges whose human baseline yields pairwise comparable build scores in three builds (Secs. V-A and VI-A). On every evaluated merge, the flakiness we observed thus stays below the margin of 1% of the tests run within which we treat build scores as comparable. Because CANDYMERGE@budget keeps the best of many builds, it has more chances than any competitor to win through flakiness alone. We checked each of its 56 *better* verdicts against three identical human-baseline builds of the same merge: 52 of these wins add whole modules, and the 4 test-only wins exceed the observed test noise 10-fold or more. Suboptimal parameter choices may have affected the performance of competing tools. Whenever possible, we used the configurations recommended by the authors or the documented defaults, resolving any remaining ambiguities in favor of the competing tool.

c) External validity: Our corpus is limited to popular, mature Java/Maven open-source projects that build, leaving generalization to other languages, build systems, and less mature or non-building projects to future work, though the approach itself is largely independent of the concrete build tool. Running third-party CI/CD pipelines at scale is itself demanding: despite accommodating many setups (JDK auto-selection, filtering deprecated Maven plugins), we could include merges from only 268 of 468 reachable projects. Moreover, our evaluation includes only merges from projects whose committed resolutions build at least partially: for 1,315 of the 2,451 merges we attempted in RQ3, the committed resolutions could not be built and tested on our machines. This introduces a survivorship bias. We accept this because our intended domain is professional software projects, where buildable revisions are generally expected before changes enter version control. By design, our sampling order also favors recent merges and merges of projects with few merges. Reweighting the evaluated merges to the mix of all merges that can be sampled from the corpus, by conflict size and file types, by the number of conflicted merges of the project, or by year, changes the share of merges on which CANDYMERGE is comparable to or better than the human baseline by at most 1.4 percentage points and keeps CANDYMERGE on the Pareto front (S4 and S5 in [1]).

VIII. CONCLUSION AND FUTURE WORK

We built CANDYMERGE as an automated merge conflict resolver producing merge results that build and pass their tests. CANDYMERGE generates whole-merge variants—resolving every chunk jointly, across source and non-source files alike—validates them against the project's own CI/CD pipeline, and returns the best one the oracle confirms. On real-world conflicting merges from Java/Maven projects and under our build-and-test oracle, it is competitive with the developer's committed resolution and sometimes resolves a merge better than the developer; with a single build it matches the best merge tools we could obtain and apply, and with its full time budget it clearly outperforms them. CI/CD-validated, candidate-driven merging is a practical path to automated merge conflict resolution beyond the predominant match-the-reference approach.

The same CI/CD signal that ranks variants could also *improve* them: compiler errors and failing tests localize what to repair, and each patched variant returns to the oracle, turning candidate-driven into candidate-feedback-driven merging.

The oracle does not depend on how a variant was produced, so it can rank other resolvers' outputs as well as CANDYMERGE's own. Any resolver that emits one or more variants can submit them to the CI/CD oracle, which crowns the best among them. Our evaluation already runs a wide range of competing tools through this oracle (RQ3); turning that setup into a single portfolio resolver—one that pools the tools' variants and returns the overall winner—is a small step, and such a portfolio can never do worse than its best member.

References

[1] "Supplementary material and replication package for 'CandyMerge: CI/CD as Oracle for Candidate-Driven Merge Conflict Resolution'," Zenodo, 2026. [Online]. Available: https://doi.org/10.5281/zenodo.21063211

[2] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, May 2002. [Online]. Available: https://doi.org/10.1109/TSE.2002.1000449

[3] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, Oct. 2013. [Online]. Available: https://doi.org/10.1109/TSE.2013.28

[4] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2013, pp. 732–741. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606619

[5] G. Vale, E. Fernandes, E. Figueiredo, and S. Apel, "Behind developer contributions on conflicting merge scenarios," in *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Oct. 2023, pp. 25–36. [Online]. Available: https://doi.org/10.1109/SCAM59687.2023.00014

[6] B. Shen, M. A. Gulzar, F. He, and N. Meng, "A characterization study of merge conflicts in Java projects," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 40:1–40:28, Mar. 2023. [Online]. Available: https://doi.org/10.1145/3546944

[7] G. Ghiotto, L. Murta, M. Barros, and A. van der Hoek, "On the nature of merge conflicts: A study of 2,731 open source Java projects hosted by GitHub," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 892–915, Aug. 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2871083

[8] A. Boll, Y. Van Dok, M. Ohrndorf, A. Schultheiß, and T. Kehrer, "Towards semi-automated merge conflict resolution: Is it easier than we expected?" in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, Jun. 2024, pp. 282–292. [Online]. Available: https://doi.org/10.1145/3661167.3661197

[9] G. Cavalcanti, P. Borba, and P. Accioly, "Should we replace our merge tools?" in *Proceedings of the IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*, May 2017, pp. 325–327. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.103

[10] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: Rethinking merge in revision control systems," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Sep. 2011, pp. 190–200. [Online]. Available: https://doi.org/10.1145/2025113.2025141

[11] O. Leßenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, Sep. 2015. [Online]. Available: https://doi.org/10.1007/s10515-014-0151-5

[12] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "IntelliMerge: A refactoring-aware software merging technique," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 170:1–170:28, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360596

[13] S. Larsén, J.-R. Falleri, B. Baudry, and M. Monperrus, "Spork: Structured merge for Java with formatting preservation," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 64–83, Jan. 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3143766

[14] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, "The impact of structure on software merging: Semistructured versus structured merge," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2019, pp. 1002–1013. [Online]. Available: https://doi.org/10.1109/ASE.2019.00097

[15] E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, and S. Lahiri, "DeepMerge: Learning to merge programs," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1599–1614, Apr. 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3183955

[16] B. Schesch, R. Featherman, K. J. Yang, B. R. Roberts, and M. D. Ernst, "Evaluation of version control merge tools," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2024, pp. 831–843. [Online]. Available: https://doi.org/10.1145/3691620.3695075

[17] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, Nov. 2012, pp. 1–11. [Online]. Available: https://doi.org/10.1145/2393596.2393648

[18] N. Nelson, "Supporting merge comprehension when resolving merge conflicts," Ph.D. dissertation, Oregon State University, Corvallis, OR, Jan. 2024. [Online]. Available: https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/3x816v95p

[19] M. Fan, W. Zhang, H. Zhao, and Z. Jin, "Reliable version merging based on deep semantic and logical understanding of critical context," in *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2025, pp. 430–439. [Online]. Available: https://doi.org/10.1109/ISSRE66568.2025.00050

[20] B. Shen, W. Zhang, A. Yu, Y. Shi, H. Zhao, and Z. Jin, "SoManyConflicts: Resolve many merge conflicts interactively and systematically," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2021, pp. 1291–1295. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678937

[21] Q. Zhang, J. Li, J. Lin, J. Ding, L. Lin, and C. Qian, "WizardMerge—save us from merging without any clues," *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 1, pp. 22:1–22:28, Dec. 2025. [Online]. Available: https://doi.org/10.1145/3731751

[22] X. Xing and K. Maruyama, "Automatic software merging using automated program repair," in *Proceedings of the IEEE International Workshop on Intelligent Bug Fixing (IBF)*, Feb. 2019, pp. 11–16. [Online]. Available: https://doi.org/10.1109/IBF.2019.8665493

[23] J. Dong, J. Sun, Y. Lin, Y. Zhang, M. Ma, J. S. Dong, and D. Hao, "Revisiting the conflict-resolving problem from a semantic perspective," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2024, pp. 141–152. [Online]. Available: https://doi.org/10.1145/3691620.3694993

[24] S. S. Towqir, B. Shen, M. A. Gulzar, and N. Meng, "Detecting build conflicts in software merge for Java programs via static analysis," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2022, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3551349.3556950

[25] R. Bajaj, E. Fernandes, B. Adams, and A. E. Hassan, "Unreproducible builds: Time to fix, causes, and correlation with external ecosystem factors," *Empirical Software Engineering*, vol. 29, no. 1, 2024, Art. no. 11. [Online]. Available: https://doi.org/10.1007/s10664-023-10399-4

[26] H. de Souza Campos, Jr., G. Ghiotto, M. Barros, A. van der Hoek, and L. Murta, "Towards a feasible evaluation function for search-based merge conflict resolution," *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 5, pp. 1–35, May 2026. [Online]. Available: https://doi.org/10.1145/3748256

[27] B. Shen and N. Meng, "ConflictBench: A benchmark to evaluate software merge tools," *Journal of Systems and Software*, vol. 214, p. 112084, Aug. 2024. [Online]. Available: https://doi.org/10.1016/j.jss.2024.112084

[28] Q. Zhang, L. Su, K. Ye, and C. Qian, "CONGRA: Benchmarking automatic conflict resolution," arXiv preprint arXiv:2409.14121, Sep. 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2409.14121

[29] B. Schesch and M. D. Ernst, "Merge-Bench: Resolve merge conflicts with large language models," in *Proceedings of the International Conference on Pattern Recognition (ICPR)*, ser. Lecture Notes in Computer Science, Aug. 2026, pp. 621–635. [Online]. Available: https://doi.org/10.1007/978-3-032-31397-3_41

[30] F. Zhu, X. Xie, D. Feng, N. Meng, and F. He, "Mastery: Shifted-code-aware structured merging," in *Proceedings of the International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA)*, ser. Lecture Notes in Computer Science, vol. 13649, Dec. 2022, pp. 70–87. [Online]. Available: https://doi.org/10.1007/978-3-031-21213-0_5

[31] M. Ellis, S. Nadi, and D. Dig, "Operation-based refactoring-aware merging: An empirical evaluation," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2698–2721, Apr. 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3228851

[32] A. Svyatkovskiy, S. Fakhoury, N. Ghorbani, T. Mytkowicz, E. Dinella, C. Bird, J. Jang, N. Sundaresan, and S. K. Lahiri, "Program merge conflict resolution via neural transformers," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Nov. 2022, pp. 822–833. [Online]. Available: https://doi.org/10.1145/3540250.3549163

[33] G. Seibt, F. Heck, G. Cavalcanti, P. Borba, and S. Apel, "Leveraging structure in software merge: An empirical study," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4590–4610, Nov. 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3123143

[34] J. P. Duarte, P. Borba, and G. Cavalcanti, "LastMerge: A language-agnostic structured tool for code integration," arXiv preprint arXiv:2507.19687, Jul. 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2507.19687

[35] P. Elias, H. de Souza Campos, Jr., E. Ogasawara, and L. Murta, "Towards accurate recommendations of merge conflicts resolution strategies," *Information and Software Technology*, vol. 164, p. 107332, Dec. 2023. [Online]. Available: https://doi.org/10.1016/j.infsof.2023.107332

[36] M. Keshani, A. Rahmati, M. H. Aref, and A. Heydarnoori, "AROMA+: A study of factors affecting reproducible builds in the Maven ecosystem," arXiv preprint arXiv:2608.27125, Aug. 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2608.27125

[37] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, Nov. 2014, pp. 643–653. [Online]. Available: https://doi.org/10.1145/2635868.2635920

[38] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–74, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3476105

[39] S. Khanna, K. Kunal, and B. C. Pierce, "A formal investigation of Diff3," in *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, ser. Lecture Notes in Computer Science, V. Arvind and S. Prasad, Eds., vol. 4855, Dec. 2007, pp. 485–496. [Online]. Available: https://doi.org/10.1007/978-3-540-77050-3_40

[40] F. Zhu and F. He, "Conflict resolution for structured merge via version space algebra," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 166:1–166:25, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276536

[41] G. Sridhara, R. H. G., and S. Mazumdar, "ChatGPT: A study on its utility for ubiquitous software engineering tasks," arXiv preprint arXiv:2305.16837, May 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.16837

[42] W. Aldndni, N. Meng, and F. Servant, "Automatic prediction of developers' resolutions for software merge conflicts," *Journal of Systems and Software*, vol. 206, p. 111836, Dec. 2023. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111836

[43] S. L. H. Gonzalez and P. Fraternali, "Almost Rerere: Learning to resolve conflicts in distributed projects," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2255–2271, Apr. 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3215289

[44] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 59:1–59:27, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133883

[45] W. Zhang, Z. Dong, H. Zhao, Y. Gu, and L. Xu, "A repository-level merge conflict resolution method based on large language models," SSRN preprint 6927019, Rochester, NY, Jun. 2026. [Online]. Available: https://doi.org/10.2139/ssrn.6927019

[46] Q. Zhang, J. Li, J. Lin, C. Luo, and C. Qian, "Rover: Context-aware conflict resolution with LLM," arXiv preprint arXiv:2605.17279, May 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2605.17279

[47] J. Zhang, T. Mytkowicz, M. Kaufman, R. Piskac, and S. K. Lahiri, "Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper)," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Jul. 2022, pp. 77–88. [Online]. Available: https://doi.org/10.1145/3533767.3534396

[48] C. Shen, W. Yang, M. Pan, and Y. Zhou, "Git merge conflict resolution leveraging strategy classification and LLM," in *Proceedings of the IEEE International Conference on Software Quality, Reliability, and Security (QRS)*, Oct. 2023, pp. 228–239. [Online]. Available: https://doi.org/10.1109/QRS60937.2023.00031

[49] C. Wang, L. Xu, R. Wang, Y. Ma, and W. Zhang, "EditFusion: Resolving code merge conflicts via edit selection," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2025, pp. 584–596. [Online]. Available: https://doi.org/10.1109/ASE63991.2025.00055

[50] J. Dong, Q. Zhu, Z. Sun, Y. Lou, and D. Hao, "Merge conflict resolution: Classification or generation?" in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2023, pp. 1652–1663. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00155

[51] S. Apel, O. Leßenich, and C. Lengauer, "Structured merge with auto-tuning: Balancing precision and performance," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2012, pp. 120–129. [Online]. Available: https://doi.org/10.1145/2351676.2351694

[52] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*, ser. Wiley-Interscience Series in Systems and Optimization. John Wiley & Sons, 2001.

[53] A. Delpeuch, "Mergiraf: A syntax-aware merge driver for Git," https://mergiraf.org, 2024, software, version 0.17.0.

[54] Q. J. Gu and M. Janota, "A universal textual merge strategy based on tokens for version control systems," arXiv preprint arXiv:2604.13813, Apr. 2026, to appear at EASE 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2604.13813