Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Is my dependency up-to-date? Exploring challenges with automated dependency updates

Miguel Botto-Tobar

miguelangel.bottotobar@alumnos.uai.edu.ar

Universidad Abierta Interamericana

Santiago A. Vidal

National University of Central Buenos Aires

Claudia Marcos

National University of Central Buenos Aires

Research Article

Keywords: dependency management, JUnit, test-code co-evolution, code generation, Maven, software maintenance

Posted Date: May 11th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9390837/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Is my dependency up-to-date? Exploring challenges with automated dependency updates

Miguel Botto-Tobar[1,2*], Santiago A. Vidal[3] and Claudia Marcos[3]

[1*]Facultad de Informática, Universidad Abierta Interamericana (UAI), Buenos Aires, Argentina.
[2]Universidad de Guayaquil, Guayaquil, Ecuador.
[3]Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN), Tandil, Argentina.

*Corresponding author(s). E-mail(s): miguelangel.bottotobar@alumnos.uai.edu.ar;
Contributing authors: svidal@exa.unicen.edu.ar; claudia.marcos@isistan.unicen.edu.ar;

Abstract

Dependency upgrades often require coordinated edits in application or test code, yet mainstream automation tools update only build manifests and leave code adaptation to developers. This paper evaluates whether these co-changes can be learned for JUnit upgrades in Maven-based Java projects. From 2,223 upgrade commits mined from GitHub, we derive a snippet-level dataset of 8,735 input-output pairs, each combining an XML dependency block, a Java test fragment, and a natural-language instruction. A four-part empirical analysis shows that most JUnit 3.x/4.x upgrades are conservative: 78.8% of method-level pairs change less than 8% of characters (median normalised Levenshtein distance 0.02), and magnitude varies little across release types (mean edit distances 0.466–0.486; Kruskal–Wallis $\boldsymbol{p = 0.026}$). Snippet decomposition, modality markers, and near-duplicate filtering yield a copy rate of 0.71 within a 512-token budget. A fine-tuned CodeT5 model attains EM-Java 38.5% and CodeBLEU 0.854 on held-out snippets, outperforming a manifest-only baseline (0% and 0.502), with the largest gains on substantial edits (+0.528 CodeBLEU). Under our build environment, model-generated patches compile and pass tests in all 37 evaluable cases (5 of 21 repositories); manifest-only bumps fail in 50% of control cases. The dataset and evaluation framework are publicly released.

Keywords: dependency management, JUnit, test-code co-evolution, code generation, Maven, software maintenance

1 Introduction

Modern software systems rely heavily on third-party libraries and frameworks that are typically called *dependencies*. In ecosystems like Maven, packages are connected by dense dependency networks in which the use of outdated versions, often described as *technical lag*, is associated with increased exposure to known vulnerabilities, incompatibilities, and maintenance overheads [1–3]. For this reason, updating dependencies is not an occasional improvement but a recurrent maintenance obligation in order to reduce security and reliability risks and to keep systems evolvable. Nevertheless, performing an update is rarely a simple version bump: maintainers often need ecosystem-specific knowledge to interpret release notes and breaking changes, resolve dependency conflicts, and adapt client code and tests accordingly; upgrades can be disruptive, time-consuming, and risky. Prior work across package managers shows that many projects lag behind the latest available releases and that this lag can persist for long periods even when updates are available [2, 3]. Consequently, there is a growing need for tools and techniques that reduce the manual effort of keeping dependencies current.

Numerous automated dependency-management tools have been developed so as to address this gap. In the Java/Maven ecosystem, the *Versions Maven Plugin* can list and apply available updates to dependencies by analyzing the `pom.xml` files. At the platform level, hosted services such as GitHub Dependabot and Renovate automatically open pull requests that bump manifest entries to newer versions, with a particular focus on security-related updates [4–6]. Evidence indicates that these tools can make it easier to notice and apply updates, thereby reducing the tendency for projects to fall behind [3, 5, 6]. However, all of these tools operate exclusively on the dependency manifest; that is to say, they propose version bumps but leave any required adaptation of application and test code entirely to the developer.

Despite the widespread use of automated dependency checkers (Dependabot, Renovate, Maven Enforcer), updates to test frameworks like JUnit are a particularly challenging case: they often still break projects because they require API-level code changes that manifest-only tools cannot anticipate. When an update fails to compile or alters observable behavior, developers must manually adjust both application and test code to match the new library version. Prior work on library migration and API evolution has produced techniques for migration mining and patch generation, although these largely target *production code* [7–12]. Migrations in *test code*, in particular framework upgrades, remain underexplored despite their high frequency and practical relevance. Test code often constitutes a substantial portion of a project's codebase and serves as the primary safety net during software evolution [13]; nonetheless, it is rarely the target of migration techniques. When dependency updates break tests, developers must choose between costly manual repairs or deferred upgrades that accumulate technical lag [2, 3], and current tools provide almost no direct support for updating tests apart from reporting failures. Since each of these upgrades leaves a co-evolution trace in version control, the accumulated history across projects constitutes, in principle, a reusable source of evolution knowledge—one that could be learned so as to assist future upgrades automatically.

In order to address this gap, this paper examines updates to the JUnit testing framework in Maven-based Java projects and how these updates affect both configuration and test code. JUnit is a de facto standard for unit testing in the Java ecosystem; in fact, it appears as a dependency in the majority of Maven-based projects, which makes it an ideal subject for studying test-code co-evolution at scale. Our primary goal is to understand how JUnit dependencies actually evolve in practice and to provide a reusable dataset that captures these co-changes. As a secondary goal, we assess whether a specialized, JUnit-aware model can leverage this dataset to generate realistic dependency-update patches. More specifically, we study (i) the extent to which generated patches resemble developer-written edits, (ii) how update effort varies across different kinds of JUnit releases, and (iii) how dataset construction choices influence the learnability of these updates. We organise our study around the following research questions:

RQ1. To what extent can the model reproduce developer-written dependency-update patches for JUnit?

RQ2. To what extent does update magnitude differ across patch, minor, and other JUnit releases?

RQ3. How do preprocessing and dataset design influence the learnability of JUnit dependency updates?

RQ4. How does the proposed approach relate to existing dependency-update tools and workflows?

To answer these questions, we combine lexical similarity metrics (ROUGE, BLEU, normalised Levenshtein distance) with structure-aware CodeBLEU estimates [14] for RQ1; empirical characterisation of update behaviour across release types for RQ2; corpus-level analysis of preprocessing effects for RQ3; and a baseline comparison against manifest-only updaters together with functional validation via a Build and Test Pass Rate (BTR) pipeline over 172 cases from 21 repositories for RQ4.

Overall, this paper makes four contributions:

1. An empirical characterisation of 2,223 JUnit update commits showing that most updates are conservative;

2. A snippet-level dataset of 8,735 input–output pairs for JUnit-oriented update generation;

3. A comparative evaluation of CodeT5-base and CodeT5+ for dependency-update patch generation, showing that context window size is a critical factor for functional correctness; and

4. A positioning of model-based generation within existing dependency-management workflows, including conditional functional validation via BTR on 5 evaluable repositories (out of 21 sampled). A replication package containing data, scripts, and configurations accompanies this paper.

By focusing on a concrete and widely used testing framework, this work contributes empirical evidence and artifacts to the emerging intersection of automated dependency management and code generation models. It illustrates how historical co-evolution of configuration and tests can be leveraged to train specialised models that assist developers with low-disruption updates, and in addition it highlights both the opportunities and boundaries of such approaches in the context of JUnit 4.x and Maven.

2 Related Work

Third–party libraries are now the backbone of modern software development; yet, keeping them up-to-date remains difficult in practice. Several empirical studies have mapped this landscape across ecosystems and document both persistent technical lag and uneven adoption of versioning discipline. Kula *et al.* analyzed 4,600 GitHub projects and 2,700 libraries and reported that 81.5% of systems retain outdated dependencies even when vulnerabilities are disclosed, in many cases due to awareness gaps and perceived effort [15]. Furthermore, Dietrich *et al.* measured versioning practices across 17 package managers and over 70 million dependencies; their results reveal limited, ecosystem-specific uptake of semantic versioning as well as substantial tension between strict and flexible version constraints [16]. Both studies point to the same conclusion: organizations depend heavily on external packages, but systematic updating is far from routine. More recent ecosystem studies in specialized domains (*e.g.*, machine learning stacks) reinforce this observation and highlight additional challenges such as frequent transitive dependencies and compatibility fragility [17, 18]. In short, the evidence identifies a dual challenge: (i) how to determine when an update is necessary and (ii) how to execute it without breaking builds or tests.

In order to reduce manual effort, platforms have adopted update bots (*e.g.*, Dependabot, Renovate) that create pull requests (PRs) to bump versions. He *et al.* conduct an exploratory study of Dependabot's impact and developer reception: after adoption, projects reduce technical lag and are often receptive to PRs; however, developers also throttle notifications and report missing compatibility signals, with a non-trivial fraction abandoning the bot [6]. Hejderup and Gousios, on the other hand, investigate the degree to which tests actually exercise dependency API calls in 521 well-tested Java projects; they find coverage of only 58% of direct and 21% of transitive calls. On the basis of these results, they argue that automation that relies solely on tests can miss subtle conflicts [19]. The implication is pragmatic: a passing CI run after a bot PR is not a reliable proof of functional compatibility, especially for transitive changes or behavior drift that tests do not capture.

Surveys highlight how third-party libraries introduce vulnerabilities and, in some cases, malicious payloads [20]. For instance, Ohm *et al.*'s "Backstabber's Knife Collection" catalogues 174 real-world malicious packages across npm, PyPI, and RubyGems and outlines common attack techniques; they show how ecosystem practices enable injection vectors throughout the dependency tree [21]. Ladisa *et al.*, in turn, systematize install-time and run-time exploit strategies available to attackers via package manager features and offer defensive recommendations for consumers of dependencies [22]. In parallel, recent work on Software Bill of Materials (SBOMs) underscores both the promise and the variability of tooling used to enumerate and scan dependencies variability that can materially change vulnerability assessments [23, 24].

Empirical works observe that updates often require synchronized edits (*e.g.*, migrating APIs or adapting tests) rather than version bumps alone [15, 16]. Apart from bots, the program-analysis community provides building blocks for automated transformations: AST differencing (*e.g.*, GumTree) supports fine-grained change analysis [25]; Spoon, moreover, offers a robust Java transformation framework suitable for

programmatic refactorings and migrations [26]. RefactoringMiner, in addition, contributes high-precision refactoring detection and commit-level AST diffs, which enables mining of consistent change patterns that link configuration edits to code edits.

Large pre-trained models for code have advanced rapidly and now support transformation tasks beyond natural-language code synthesis. CodeBERT (encoder-only) learns bimodal NL–PL representations and, as a result, improves retrieval and understanding tasks [27]. PLBART adopts a sequence-to-sequence architecture trained via denoising so as to serve both understanding and generation across Java/Python [28]. CodeT5 moves to a unified encoder–decoder with identifier-aware pretraining (that is, it distinguishes identifiers and leverages comments via dual generation), and it shows strong results for multiple generation tasks [29]. Subsequent decoder-only large language models trained on vastly larger corpora, including Code Llama [30], StarCoder2 [31], DeepSeek-Coder-V2 [32], and more recent instruction-tuned models such as Qwen2.5-Coder and the o1/o3 series, have improved their performance on code refinement, repair, and migration tasks. Recent studies have leveraged these capabilities (often via zero/few-shot prompting, agents, or lightweight fine-tuning) in order to automate code repairs for breaking dependency updates or library migrations [33–36].

Our work differs in its focus on *proactive* paired generation of both the Maven dependency declaration update and the corresponding test-code adaptations for the specific case of JUnit 3→4 migrations, and it uses supervised sequence-to-sequence learning from mined historical co-changes rather than post-failure LLM-based repair.

3 Methodology

In this study, we investigated the automation of dependency updates in Java projects using Maven and JUnit as of February 2025. We constructed a dataset from open-source GitHub repositories and fine-tuned a CodeT5 model so as to generate updates that modify both `pom.xml` dependency declarations and associated Java test code. The replication package of our study, including the dataset generation scripts, raw and processed datasets, and model training code, is publicly available in [DOUBLE BLIND REVIEW] Below, we detail data collection, preprocessing, training, and evaluation.

3.1 Data collection

3.1.1 Repository selection.

In order to create our dataset, we developed a custom Python script that employs GitHub's API to collect data from repositories that met the following selection criteria: (i) at least 20 commits, to ensure ongoing activity; (ii) a minimum of 2 active collaborators, to reflect collaborative software practices; (iii) at least 20 stars and 10 forks, as indicators of community validation; and (iv) explicit use of JUnit declared in `pom.xml`, verified by parsing Maven dependency blocks. We focused on JUnit since it is widely used in Java projects and because its upgrades can require coordinated changes in test code beyond manifest edits.

Applying these criteria yielded a set of 43 Maven-based Java repositories hosted on GitHub. Each repository was cloned locally so that we could inspect its complete version-control history.

3.1.2 Commit identification.

A Maven `pom.xml` (Project Object Model) file is an XML descriptor that declares dependencies, plugins, and project metadata. Since every Maven project must provide a `pom.xml`, it is a reliable artifact for the identification of dependency versions. We extracted commits that modified JUnit dependency versions and recorded repository metadata, JUnit versions before and after the update, update type (patch, minor, major) classified according to semantic versioning [37], POM XML snippets before and after the change, and associated Java test files with a focus on `import` statements, annotations (*e.g.*, `@Test`, `@Before`), and method bodies.

Not all commits that edit `pom.xml` are suitable for our analysis. We include commits that change the `<version>` of an existing JUnit dependency, potentially affect test code by modifying imports, annotations, or method bodies in files under `src/test/java`, and represent single-step JUnit updates that can be classified as patch, minor, or major. We exclude merge commits (which mix independent changes), commits where `pom.xml` only changes in formatting or comments, and commits that add or remove JUnit entirely rather than updating its version. Commits that update the JUnit version but do not modify any test file are kept in the commit-level corpus but excluded from the snippet-level training dataset.

3.2 Data construction

After applying the filtering criteria, we obtain a commit-level dataset of 2,223 JUnit 3.x/4.x update commits, which we use to characterise how JUnit dependency declarations evolve in practice. From the subset of commits that also modify tests, we derive a snippet-level dataset of 8,735 structured input–output pairs that link JUnit dependency updates in `pom.xml` with the corresponding test-code edits.

Each pair links a pre-update context to the corresponding developer-written update and contains:

- *Input*: an XML dependency block, a Java test-code snippet, and a task instruction, concatenated with explicit modality markers (see Section 3.3).
- *Output*: the updated XML dependency block and the corresponding modified Java code, using the same marker format.

We serialise examples in JSONL for reproducible ingestion and release both raw and processed forms in the replication package. We explicitly retained JUnit updates, which in Maven are typically scoped as `test`. Although prior work often excludes test-scoped dependencies, we included them because JUnit upgrades directly affect Java test suites and are critical to software reliability. Dependencies were categorised into patch, minor, and major updates following semantic versioning [37]. We do not observe JUnit 5 in the mined upgrade commits.

Figure 1 summarises the end-to-end pipeline from mining upgrade commits to model training and evaluation.

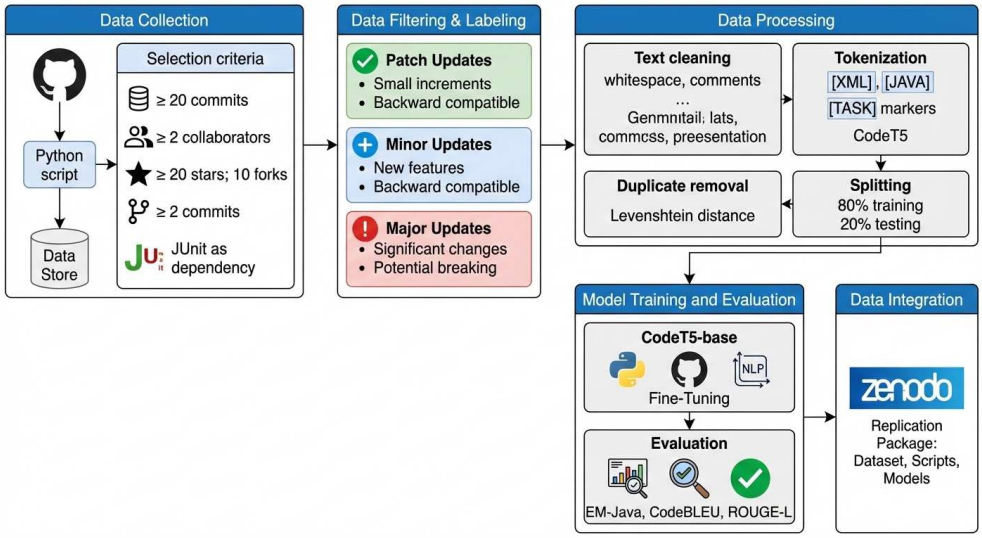


Fig. 1: Overview of the end-to-end methodology: repository selection, commit mining, snippet construction, preprocessing, CodeT5 fine-tuning, and evaluation (metrics + baselines).

3.3 Preprocessing

We apply preprocessing steps that are designed to preserve syntactic structure while reducing noise:

- **Whitespace normalisation:** we collapse consecutive blank lines and trailing spaces in both XML and Java fragments so that formatting variation does not inflate edit distances.
- **Comment stripping:** we remove inline and block comments from Java code in order to focus the model on structural and semantic content rather than natural-language annotations.
- **Modality markers:** we prepend `[XML]`, `[JAVA]`, and `[TASK]` tokens to the corresponding sections of each example. These markers provide clear boundaries between the three modalities and are registered as special tokens in the tokeniser.
- **De-duplication:** we remove near-duplicate input–output pairs using normalised Levenshtein distance [38] so as to reduce over-representation of highly repetitive edits.
- **Splitting:** we split the 8,735 examples randomly into 80% training (6,988 examples) and 20% held-out test (1,747 examples) using a fixed random seed (seed = 42) to ensure reproducibility. This split is performed at the example level, not at the repository level; we discuss this as a threat to validity in Section 6.

The snippet-level inputs have a mean length of 535 characters ($\sigma = 265$), with no input or output sequence exceeding the 1,024-token encoder limit after subword tokenisation. For CodeT5-base, however, 55% of inputs exceeded its 512-token limit,

which caused truncation; this motivated the adoption of CodeT5+, as described in Section 3.4.

3.4 Model

We initially fine-tuned CodeT5-base [29], an encoder–decoder transformer with 220M parameters and a maximum context of 512 tokens. Nevertheless, this model produced truncated outputs on a significant fraction of inputs (55% exceeded the 512-token limit), and this led to incoherent predictions and BTR = 0% when evaluated via the commit-level surgical patching pipeline (Section 4.4.2). We therefore extended the evaluation to CodeT5+ [39], which supports a context window of 1,024 tokens. As a result, this eliminated truncation entirely and enabled functional patch generation. Unless otherwise noted, results reported in RQ4 correspond to the CodeT5+ model.

3.4.1 Training configuration.

We used the Hugging Face Transformers library (v4.47.1) [40] to fine-tune both models with the following shared hyperparameters: learning rate 3×10^{-5}, 10 epochs, batch size 2 with gradient accumulation over 4 steps (effective batch size 8), AdamW optimiser with linear learning-rate schedule, and mixed-precision training (fp16). The maximum sequence length differs between models: 512 tokens for CodeT5-base and 1,024 tokens for CodeT5+. The three modality markers (`[XML]`, `[JAVA]`, `[TASK]`) were registered as special tokens in the tokenizer of both models, and the embedding layer was resized accordingly. The training seed was fixed at 42 in both cases.

3.4.2 Input and output format.

Each input sequence is structured as a concatenation of three sections separated by the special tokens:

```
[XML] <dependency>...<version>4.12</version>...</dependency>
[JAVA] public void testSomething() {...}
[TASK] Update the artifact version: junit (Actualizar la versión del
artifact: junit, in Spanish)
```

The output (label) follows the same format and contains the updated XML dependency block together with the modified Java test code. During teacher-forcing, the decoder learns to generate both sections conditioned on the input context. During inference, the model generates the full output sequence autoregressively using beam search with beam width 4 and a length penalty of 1.0.

3.4.3 Decoding and post-processing.

At inference time, the model generates output with `skip_special_tokens=True`, which strips the modality markers from the decoded sequence. We extract the Java and XML sections from both prediction and reference using boundary-based heuristics: for references with explicit `[JAVA]` markers, we extract the text between `[JAVA]` and `[TASK]`; for model outputs without markers, we identify the last XML closing

tag and treat the remaining text as the Java section. This extraction enables fair component-level metrics (EM-Java, EM-XML).

3.5 Evaluation

We evaluate the fine-tuned model on the held-out test split ($N = 200$ randomly sampled from the 1,747 test examples, due to computational constraints). For each test example, the model receives the input sequence and generates a candidate patch, which we then compare against the developer-written patch using the following metrics:

3.5.1 EM-Java (Exact Match – Java).

The fraction of predictions whose extracted Java section matches the reference exactly after whitespace normalisation:

$$\text{EM-Java} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}\left[\hat{y}_i^{\text{java}} = y_i^{\text{java}}\right] \tag{1}$$

where $\hat{y}_i^{\text{java}}$ and y_i^{java} denote the Java fragments extracted from the prediction and the reference, respectively. This metric isolates code-generation quality from the XML version bump.

3.5.2 CodeBLEU.

We adopt CodeBLEU [14], a composite metric that augments standard BLEU [41] with code-aware components:

$$\text{CodeBLEU} = \alpha \cdot \text{BLEU} \;+\; \beta \cdot \text{BLEU}_w \;+\; \gamma \cdot \text{Match}_{\text{syntax}} \;+\; \delta \cdot \text{Match}_{\text{dataflow}} \tag{2}$$

We set $\alpha = \beta = \gamma = \delta = 0.25$ following the original formulation. CodeBLEU is computed on the extracted Java sections only. Our implementation uses a regex-based approximation for $\text{Match}_{\text{syntax}}$ and $\text{Match}_{\text{dataflow}}$ rather than a full tree-sitter parser; we report this as a limitation in Section 6.

3.5.3 Additional metrics.

We also report BLEU [41], ROUGE-L [42], normalised Levenshtein distance [38], and the individual $\text{Match}_{\text{syntax}}$ and $\text{Match}_{\text{dataflow}}$ sub-scores. We also report **EM-XML** (exact match on the extracted XML section) and analyse the distribution of errors among non-exact predictions using a taxonomy based on token-level similarity: near-misses (>80% similarity), partial updates, truncated outputs, and wrong transforms.

3.5.4 Baseline.

We compare the model against a *manifest-only baseline* that simulates the behaviour of tools such as Dependabot [43] and Renovate [44]: it applies the version bump in `pom.xml` but leaves the Java test code unchanged. On the snippet-level dataset, where

every example contains a Java code change by construction, the manifest-only baseline achieves 0% EM-Java by definition.

3.5.5 Metrics scope and limitations.

Compiling and executing every patched project for every generated patch is computationally expensive at this scale. For this reason, we complement similarity metrics with a functional Build and Test Pass Rate (BTR) evaluation over 172 stratified cases from 21 open-source repositories, detailed in Section 4.4.2. That is to say, similarity metrics serve as proxies for patch correctness on the full held-out split (under an example-level partition), whereas BTR provides conditional functional evidence on the evaluable subset.

4 Results

This section addresses our four research questions by presenting the quantitative and qualitative analyses derived from the dataset, the model's outputs, and the empirical characterisation of real JUnit update behaviour.

4.1 RQ1: To what extent can the model reproduce developer-written dependency-update patches?

To answer RQ1, we evaluate how closely the fine-tuned model reproduces developer-written JUnit upgrade patches using three complementary views: (i) token-level similarity metrics (ROUGE, BLEU) computed on the held-out test split, (ii) a corpus-level analysis of how conservative real JUnit 4.x upgrades are, measured via normalised edit distance on 2,474 method-level pairs extracted from the full commit corpus, and (iii) qualitative inspection of representative model outputs in order to illustrate typical agreement patterns and failure modes.

4.1.1 Lexical fidelity and overlap

Table 1 presents descriptive statistics for ROUGE and BLEU on the held-out test split. Scores are expressed as percentages.

Table 1: Descriptive statistics for ROUGE and BLEU on the held-out test split (scores in %).

Metric	Mean	Std	Min	25%	50%	75%	Max
ROUGE-1	88.40	4.36	80.00	85.50	88.89	91.70	94.74
ROUGE-2	82.69	3.76	75.93	80.25	82.97	85.34	88.89
ROUGE-L	88.40	4.36	80.00	85.50	88.89	91.70	94.74
BLEU-4	80.15	3.05	73.73	78.25	80.72	82.41	83.67

These results indicate that model-generated patches are close to developer-written references at the token and short n-gram level. ROUGE-L around 88% suggests substantial subsequence overlap, while BLEU-4 around 80% indicates that many four-gram fragments are reproduced exactly. These metrics primarily capture lexical fidelity and should be interpreted together with the structure-aware evaluation (CodeBLEU, Section 4.4).

In the case shown in Figure 2, the model output reproduces the developer patch exactly: a single `throws Exception` clause is added while the rest of the method remains intact. This kind of localised, syntactically patterned edit is the most common transformation in the corpus and represents the model's strongest suit.

```
public void forcedShutdownVerifyingLogs() {
}
```

```
public void forcedShutdownVerifyingLogs() throws Exception {
}
// added throws clause
```

Fig. 2: Low-distance update: the model introduces a throws Exception clause, matching the developer patch exactly.

By contrast, Figure 3 shows a case where the developer simultaneously renames a method and replaces its body. Lexical overlap drops sharply despite both versions being plausible tests in isolation, which illustrates a limitation of text-based metrics for semantically equivalent but syntactically divergent patches.

```
- public void shouldDecodeTwoCommands()
    throws IOException {
    ...
    assertThat( bye.getCommandType(),
                is( BYE_ACK ) );
    ...
}
```

```
+ public void shouldThrowUnsupportedException5() {
    TestLessInputStreamBuilder builder =
        new TestLessInputStreamBuilder();
    builder.getCachableCommands()
           .provideNewTest();
}
// -- full rewrite
```

Fig. 3: High-distance update: full method rewrite where lexical overlap is low despite both versions being plausible tests.

4.1.2 Edit distances and update conservatism

In order to contextualise the similarity scores, we analyse the magnitude of changes in real JUnit 4.x upgrade commits. From the 2,223 commits in our corpus, we extract 2,474 method-level pairs where both the pre and post-update implementations are available. For these pairs, the normalised Levenshtein distance between pre and post-update code has mean 0.075, median 0.020, and standard deviation 0.142, with 78.8% of pairs below 0.08.

The vast majority of JUnit 4.x method-level updates are conservative: fewer than 8% of characters change in a typical update, and the modifications are generally limited to assertion refinements, import adjustments, or `throws`-clause additions. This structural regularity creates a favourable learning landscape for a sequence-to-sequence model; that is to say, the model can learn a "copy-and-edit" strategy that reproduces most of the input and applies targeted local modifications.

This conservatism analysis covers method bodies only. Annotation contexts (*e.g.*, `@Test` blocks with surrounding code) show substantially higher divergence (mean normalised Levenshtein 0.77 across 4,245 pairs), since the extraction heuristic captures surrounding code that often changes structurally between versions. This divergence reflects the wider context window's sensitivity to unrelated refactorings rather than drastic changes in test logic. Higher edit distances are concentrated in cases where the developer carries out a full method rewrite (*e.g.*, new test scenarios or restructured assertions), as illustrated in Figure 3. These cases represent an upper bound on model difficulty.

4.1.3 Structural behaviour and CodeBLEU

At the commit level, only 4–6% of commits involve substantive changes in test method bodies; even in those cases, changes are typically minor modifications rather than additions or deletions of entire methods. Manual inspection of model–reference pairs confirms that the model largely preserves method signatures and parameter lists, focusing its edits inside method bodies and XML dependency blocks. We defer the full CodeBLEU evaluation to Section 4.4; CodeT5-base achieves CodeBLEU = 0.854 and EM-Java = 38.5%, while CodeT5+ achieves CodeBLEU = 0.868 and EM-Java = 43.6% on the snippet-level held-out split.

4.1.4 Error analysis

In order to characterise the difficulty landscape faced by the model, we classify every test sample according to the divergence between the manifest-only baseline (*i.e.*, the pre-update Java code left unchanged, as Dependabot would leave it) and the developer-written reference patch. We define four error categories on the basis of token-level similarity. *Near-miss*: token similarity above 80%, indicating that only a minor, localised edit is required. *Partial update*: similarity between 50% and 80%, where the developer applied additional changes. *Wrong transform*: similarity below 50%, corresponding to substantial method rewrites. *Truncated*: the reference is more than twice as long as the input, indicating significant new code.

Table 2 reports the distribution across 500 held-out snippet-level test samples. Unlike the generation-based metrics (which use $N = 200$ due to inference cost; Section 3.5), this taxonomy compares only the pre-update input against the developer reference and therefore does not require model inference, which allows a larger sample.

Table 2: Divergence taxonomy between the manifest-only baseline and the developer patch on the held-out snippet-level test split ($N = 500$). Categories are defined by token-level similarity between unchanged input and reference.

Category	Count	%	Implication
Near-miss (>80% sim.)	182	36.4	Minor edit suffices
Partial update (50–80%)	210	42.0	Moderate code changes needed
Wrong transform (<50%)	77	15.4	Major rewrite required
Truncated (<50% length)	31	6.2	New code must be generated
Total (all non-exact)	500	100.0	

These results reveal a graded difficulty landscape. Over a third of the test cases (36.4%) are near-misses where the model needs to learn only small, localised edits such as a new `throws` clause or an updated assertion call. A further 42.0% require moderate changes, typically a combination of import updates, annotation changes, and assertion-style migrations. The remaining 21.6% fall into harder categories: wrong transforms (15.4%) and truncated cases (6.2%). These harder categories set a practical upper bound on what a local, snippet-level sequence-to-sequence model can be expected to handle in the absence of broader project context.

4.2 RQ2: To what extent does update magnitude differ across patch, minor, and major JUnit releases?

In order to address RQ2, we operate on the filtered snippet-level JSONL dataset used for training and evaluation ($N = 8{,}735$ pairs). For each pair, we extract the JUnit version from the `<version>` tag in the input and output XML and classify the update as patch, minor, or major according to semantic versioning [37]. We then quantify update magnitude on the *Java fragment only* using two complementary measures: a character-based normalised edit distance (computed as $1 -$ similarity after consistent whitespace/comment normalisation) and CodeBLEU [14], which adds syntax- and data-flow-aware components.

4.2.1 Distribution of change and update types

Table 3 shows the distribution of change types across the snippet-level pairs. Consistent with the conservative profile observed in RQ1, JUnit 4.x upgrades are predominantly assertion and API call-site edits (82.7%). Annotation-level changes are

rare (0.1%), which suggests that `@Test` and similar markers are rarely the primary target of an upgrade at the snippet level.

Table 3: Distribution of change types across the 8,735 snippet-level pairs. Categories are not mutually exclusive. Import-level changes are excluded as snippets do not contain class-level import statements.

Change Type	Count	%
Assertion / API call-site edits	7,220	82.7%
Annotation-level edits	9	0.1%
Other / structural	1,506	17.2%
Total records	8,735	100.0%

Table 4 summarises the frequency of patch, minor, and major updates as well as the corresponding normalised edit distances.

Table 4: Distribution and normalised edit distances (input vs. output Java fragment) by update type.

Type	#Pairs	%Pairs	Mean	Median	Q1	Q3
Patch	4,183	47.9%	0.474	0.431	0.262	0.697
Minor	2,542	29.1%	0.466	0.424	0.242	0.692
Major	2,010	23.0%	0.486	0.456	0.278	0.707

Patch updates represent nearly half of the examples (47.9%), minor updates account for 29.1%, and major updates for 23.0%. This balanced composition allows comparison of update magnitudes across types.

4.2.2 Normalised edit distances across update types

The key question of RQ2 is whether patch, minor, and major updates differ substantially in the amount of code they change. As Table 4 shows, mean distances fall in a narrow band (0.466–0.486), with medians between 0.424 and 0.456 and broadly similar interquartile ranges. A Kruskal–Wallis test yields $H = 7.28$, $p = 0.026$, indicating a statistically significant but small difference across the three groups, driven primarily by major updates. Major updates show a slightly lower proportion of near-identical changes and a slightly higher proportion of large rewrites; nevertheless, this does not constitute a qualitatively different regime.

4.2.3 Structure-aware magnitude (CodeBLEU)

To complement character-level edit distance, we compute CodeBLEU [14] between the pre and post-update Java fragments, stratified by update type. Table 5 reports the results.

Table 5: CodeBLEU between pre and post-update Java fragments by update type ($N = 8{,}735$ snippet pairs). Higher values indicate greater structural similarity.

Type	#Pairs	Mean	Median	IQR
Patch	4,183	0.489	0.513	[0.317, 0.643]
Minor	2,542	0.482	0.513	[0.286, 0.656]
Major	2,010	0.470	0.513	[0.292, 0.560]

CodeBLEU confirms the pattern observed with edit distance: the three update types occupy a narrow band of means (0.470–0.489) and share the same median (0.513). A Kruskal–Wallis test yields $H = 6.98$, $p = 0.030$, which is again statistically significant but of small magnitude. The identical medians indicate that the bulk of the distributions overlap substantially.

These results reinforce the main conclusion of RQ2: the semantic-versioning label (patch/minor/major) is a weak predictor of how much test code actually changes during a JUnit dependency upgrade. From a modelling perspective, this means that a single model trained on the combined corpus, rather than type-specific models, is a reasonable design choice, since the three categories do not present qualitatively different learning problems.

4.3 RQ3: How do preprocessing and dataset design influence the learnability of JUnit dependency updates?

In order to address RQ3, we analyse the concrete design decisions in the data pipeline and quantify their effect on the resulting training corpus. Our pipeline starts from a commit-level corpus of 2,223 JUnit update commits and derives a filtered snippet-level dataset of 8,735 input–output pairs after snippet extraction, text normalisation, and filtering of identical or near-identical Java fragments. We examine three key components: snippet-level decomposition, special-token structuring, and near-duplicate filtering.

4.3.1 Pipeline design and corpus properties

The first design decision is to decompose commit-level updates into snippet-level pairs, yielding approximately 3.93 input–output pairs per commit. This decomposition increases the number of supervision signals per commit and focuses the learning problem on small, tightly scoped edits rather than entire multi-method diffs. As a practical

benefit, snippet-level decomposition ensures that no example exceeds 512 whitespace tokens in either input or output (0% overflow), whereas the commit-level corpus has 7.0% of examples exceeding this threshold. For CodeT5-base, this truncation caused incoherent predictions; CodeT5+ resolves this by supporting a 1,024-token context window.

The second component is the explicit structuring of inputs with special tokens (`[XML]`, `[JAVA]`, `[TASK]`) and whitespace normalisation. An inspection of all 8,735 examples confirms that 100% of inputs contain all three markers. The `[TASK]` marker contains a natural-language instruction ("Actualizar la versión del artifact: junit"), which aligns the corpus with instruction-style code generation. Table 6 reports the sequence-length statistics of the filtered corpus; the large majority of examples are short to medium-length snippets well within the encoder limits of both models.

Table 6: Sequence-length statistics for the 8,735 snippet-level pairs (whitespace-tokenised).

	Mean	Median	P75	Max	>512
Input (chars)	535	507	678	2,087	–
Output (chars)	474	480	619	2,674	–
Input (WS tokens)	38.2	36	44	224	0%
Output (WS tokens)	29.8	28	34	277	0%

The third component is the removal of trivial cases where the Java code is effectively unchanged. The pipeline retains only records with similarity < 0.95 (computed via `difflib.SequenceMatcher`). Similarity values in the released dataset range from 0.04 to 0.95 (exclusive), with a mean of 0.53 and a median of 0.57. This filter ensures that the remaining examples span a range from small edits to substantial modifications, preventing evaluation metrics from being inflated by trivial identity mappings.

4.3.2 Learnability landscape

To characterise the learning problem that the pipeline creates, we analyse two properties of the resulting corpus: the *copy rate* and the *difficulty distribution*.

The mean copy rate (fraction of whitespace-tokenised output tokens that appear in the input) is 0.714 (median 0.786), indicating that a copy-and-edit strategy can produce most of the output by retaining tokens from the input and learning to modify a minority of positions. When stratified by input–output Java similarity, the corpus divides into three difficulty bands, summarised in Table 7.

The low-similarity band (42.9%), which includes method rewrites and multi-site refactorings, constitutes the largest stratum and ensures that the model is not trained predominantly on trivial edits. These examples are also the longest on average (42.3 input tokens vs. 34.0 for high-similarity), which implies that the model faces its hardest cases precisely where the input context is most complex.

Table 7: Difficulty bands by input–output Java similarity, with sequence-length statistics (whitespace tokens).

Similarity band	N	%	Mean input tokens	Mean output tokens
High (>0.80)	1,528	17.5	34.0	26.5
Moderate (0.50–0.80)	3,456	39.6	35.6	27.7
Low (<0.50)	3,751	42.9	42.3	33.0

In summary, the corpus-level analysis provides evidence for two key design choices. The snippet-level decomposition eliminates all sequences exceeding 512 tokens (vs. 7.0% overflow in the commit-level corpus) and creates a copy-rate distribution centred around 0.79. The near-duplicate filter ensures that 43% of examples require substantial edits (similarity < 0.50), preventing a trivial identity mapping. We prioritised end-to-end functional validation (BTR, Section 4.4.2) over component-level ablations due to resource constraints; controlled experiments that isolate each pipeline component (no task markers, no snippet decomposition, weaker deduplication) remain as follow-up work.

4.4 RQ4: How does the proposed approach relate to existing dependency-update tools and workflows?

In order to address RQ4, we position our model-based patch generation against common dependency-update workflows in the Java/Maven ecosystem. We present a quantitative baseline comparison against the manifest-only strategy, followed by functional validation via Build and Test Pass Rate.

4.4.1 Quantitative baseline: Manifest-only vs. Model-Generated patches

The simplest baseline for dependency updates is the *manifest-only* strategy: bump the version in `pom.xml` and leave all Java code unchanged. We operationalise this as the typical behaviour of PR-based updaters such as Dependabot [43] and Renovate [44]. We evaluate this baseline and both fine-tuned models on the snippet-level held-out test split (example-level partition; see Section 6) using CodeBLEU [14] and EM-Java.

Table 8 reports the results on the snippet-level split ($N = 200$), where every example contains a Java code change by construction.

Although CodeT5-base achieves a marginally higher CodeBLEU than CodeT5+ (0.854 vs. 0.850), CodeT5+ improves on every other metric: EM-Java (43.6 % vs. 38.5 %), BLEU (0.868 vs. 0.822), and ROUGE-L (0.940 vs. 0.910). More critically, CodeT5-base achieves BTR = 0 % at the commit level due to systematic truncation (55 % of inputs exceed its 512-token limit), whereas CodeT5+ achieves BTR = 100 % on the 5 evaluable repositories (out of 21 sampled; see Section 4.4.2). This dissociation between CodeBLEU and functional correctness is consistent with known limitations of proxy metrics [45] and reinforces our use of BTR as the primary correctness criterion.

Table 8: Baseline comparison on the snippet-level held-out split ($N = 200$). Manifest-only leaves Java unchanged. EM-Java compares only the extracted Java fragment.

Approach	EM-Java (%)	CodeBLEU	BLEU	Syntax	Dataflow
Manifest-only (version bump only)	0.0	0.502	0.486	0.293‡	0.726‡
CodeT5-base (512 tokens)	38.5	0.854	0.822	0.796‡	0.960‡
CodeT5+ (1024 tokens)	43.6	0.850	0.868	0.880†	0.869†

‡ Regex-based approx. (Manifest-only & CodeT5-base): mutually comparable, exploratory only.
† `javalang` AST-based pipeline (CodeT5+ only): different scale from ‡.
CodeBLEU and BLEU are computed identically across all rows and are directly comparable.

The manifest-only baseline achieves 0% EM-Java and CodeBLEU = 0.502. Among the 123 non-exact model predictions, 53 are near-misses (token similarity above 80%), 42 partial updates, 3 truncated, and only 2 wrong transforms—confirming that even non-exact outputs typically require only minor manual refinements.

Table 9 stratifies the snippet-level results by difficulty tier.

Table 9: CodeBLEU by difficulty tier on the snippet-level split ($N = 200$).

Tier	N	Manifest-only	CodeT5-base	Δ
Trivial (sim. > 0.90)	13	0.923	0.838	-0.085
Moderate (sim. 0.50–0.90)	97	0.629	0.876	$+0.248$
Significant (sim. < 0.50)	90	0.303	0.832	$+0.528$

The tier breakdown reveals a clear complementarity pattern. On trivial cases, the manifest-only baseline slightly outperforms the model (0.923 vs. 0.838): copying the input verbatim yields high overlap when the change is minimal. On significant cases, the model achieves 0.832 where the baseline drops to 0.303 (+0.528 CodeBLEU), confirming that the model adds the most value precisely where manifest-only tools fall short.

At the class level, both models achieve 0% EM-Java and CodeBLEU = 0.213, with 100% of predictions classified as truncated. This negative control confirms that snippet-level decomposition is necessary for the model to function; class-level test files routinely exceed the encoder limits of both models.

4.4.2 Functional validation: Build and Test Pass Rate (BTR)

Similarity metrics do not guarantee that generated patches compile or pass tests. We therefore evaluate the model via a BTR pipeline over 172 cases from 21 open-source repositories, selected via stratified sampling across update type and difficulty tier (max

30 cases per repository). The sample includes 121 evaluation cases (with Java code changes) and 51 control cases (version bump only).

For each case, the pipeline: (1) clones the repository at the pre-update commit; (2) applies the version bump in `pom.xml`; (3) surgically replaces the target method with the model's predicted snippet; (4) compiles via `mvn test-compile`; and (5) executes the test class via `mvn test`.

We distinguish build reproducibility from patch correctness. Of the 21 sampled repositories, 16 failed at the baseline build stage due to infrastructure constraints (unresolvable transitive dependencies, incompatible Maven plugin versions, unavailable Java runtimes) before any model-generated patch was applied. This is confirmed by 31 of 51 control cases also failing in those repositories. The 5 evaluable repositories were determined entirely by build-environment compatibility, not by model performance.

Table 10: BTR results on evaluable repositories ($N = 5$ repos, 47 cases). Wilson 95 % CI at case level.

Approach	Cases	BTR (%)	95 % CI (Wilson)
Manifest-only (version bump only)	10	50.0	[23.7%, 76.3%]
CodeT5+ (model)	37	100.0	[90.6%, 100.0%]

Table 11: BTR breakdown by repository. FC = compile failure; FT = test failure; IE = infrastructure error.

Repository	N	PASS	FC	FT	IE	BTR
maven-common-artifact-filters	39	34	0	5	0	87.2%
maven-gpg-plugin	3	3	0	0	0	100.0%
executor	2	2	0	0	0	100.0%
download-maven-plugin	2	2	0	0	0	100.0%
maven-help-plugin	1	1	0	0	0	100.0%
Evaluable subtotal	47	42	0	5	0	89.4%
olingo-odata4	43	0	43	0	0	—
maven-enforcer	14	0	13	1	0	—
jasmine-maven-plugin	13	0	13	0	0	—
maven-shade-plugin	13	0	0	0	13	—
sortpom	13	0	8	5	0	—
Other (11 repos)	29	0	23	5	1	—
Total	**172**	**42**	**100**	**16**	**14**	24.4%

All 37 model-generated patches compile and pass tests in the 5 evaluable repositories (Conditional BTR = 100%; Table 10), whereas manifest-only version bumps fail in 50% of control cases. The remaining 16 repositories could not be evaluated due to infrastructure constraints, as confirmed by control-group failures in those same environments (Table 11).

The case-level Wilson CI is [90.6%, 100.0%], which quantifies uncertainty within the current build environment. However, the repository-level bootstrap CI (10,000 iterations) is considerably wider at [1.8%, 55.6%], reflecting that only 5 of 21 repositories are evaluable and 81% of PASS cases originate from a single repository (`maven-common-artifact-filters`). These results should therefore be interpreted as conditional functional evidence within our build environment rather than as a general correctness guarantee.

Semantic preservation was verified on all 42 PASS cases (37 model-generated patches + 5 manifest-only control cases): 90.5% of patched methods (38/42) preserve the number of `assert*` calls, and none exhibit empty bodies, always-true conditions, or suppressed assertions. This confirms that passing tests reflect genuine behavioural verification rather than trivial acceptance.

Three factors bound scope: (i) evaluable repositories are determined by build-environment compatibility, not patch quality; (ii) the pipeline applies single-snippet surgical patching, so multi-method commits fall outside its reach; and (iii) 81% of PASS cases originate from one repository. Extending evaluation to broader infrastructure and multi-snippet commits is a priority for future work.

5 Discussion

This section interprets the findings from RQ1–RQ4, discusses practical implications, and outlines directions for future work.

5.1 Conservative JUnit Evolution and Modelling Implications

The analyses in RQ1 and RQ2 converge on a central finding: JUnit 3.x/4.x upgrades in the studied repositories are predominantly conservative, and this conservatism holds across semantic-versioning categories (Tables 4–5). From a modelling standpoint, this structural regularity creates a favourable learning landscape: a sequence-to-sequence model can adopt a copy-and-edit strategy in which most output tokens are retained from the input and only a small number of positions are modified. The corpus-level copy rate reported in RQ3 (Section 4.3) confirms this pattern.

The error taxonomy (Table 2), however, delineates the boundaries of this advantage. Near-miss and partial-update cases (78.4% combined) are well within the model's reach, whereas wrong-transform and truncated cases (21.6%) require multi-site rewrites or entirely new test logic that a snippet-level model cannot reconstruct from local context alone. These harder categories set a practical ceiling on what the current architecture can achieve without broader project-level information.

The evidence across RQ1–RQ3 suggests that the approach is strongest for *local, syntax-regular, and repetitively observed* co-changes—primarily import edits,

annotation-level edits, and assertion/API call-site edits (Table 3). It is intrinsically weaker for structural refactorings (*e.g.*, method moves, renames, extract/inline) and semantic incompatibilities, which require cross-file reasoning and project-specific design knowledge. This characterisation is not specific to JUnit; similar co-change patterns arise in other dependency upgrades where APIs shift. However, the degree of conservatism observed here is a key enabling factor: when upgrades routinely require multi-file refactors, a purely learned patch generator must be paired with stronger oracles and, in all likelihood, with static analysis or type-aware constraints.

5.2 Positioning Relative to Existing Tools

RQ4 compared the fine-tuned models against the manifest-only strategy and situated them within the broader tooling landscape. The quantitative results (Tables 8 and 9) show that the model adds the most value precisely where manifest-only tools fall short—namely in updates that require substantial test-code adaptation (significant-change tier). Conversely, for trivial cases, simply copying the input already yields high overlap, and the model's minor rewrites can even reduce surface similarity.

Manifest-only updaters such as Dependabot and Renovate are well suited to propose version bumps and integrate into PR-based workflows; however, they stop at the point where compilation or tests fail [43, 44, 46]. Rule-based codemods can handle well-scoped, repeatable transformations, but JUnit upgrades exhibit a mix of recurring local edits and project-specific adaptations that codemods cannot fully cover. The fine-tuned model occupies a middle ground: more specialised than general-purpose LLM prompting, broader in scope than manifest-only updaters, and less semantically constrained than IDE refactoring engines—hence the need for build/test validation as a gating mechanism.

The class-level evaluation (Section 4.4) serves as a negative control: both models achieve 0% EM-Java and CodeBLEU = 0.213, with 100% of predictions truncated, confirming that snippet-level decomposition is not merely convenient but necessary. A production deployment would therefore require either a reassembly step or a model with longer context.

5.3 Practical Implications

The complementarity pattern observed in RQ4 suggests a natural integration scenario: a bot proposes a JUnit version bump in `pom.xml` (*e.g.*, via a pull request), and the model is then invoked to generate candidate edits for affected test files in the same PR. A difficulty estimate could determine which files warrant model-generated patches versus those where the manifest-only bump suffices.

For teams that maintain large test suites across many modules, even partial automation of these adjustments can shorten the feedback cycle between dependency updates and stable builds. The model produces the exact developer patch in a substantial fraction of snippet-level cases (Table 8) and a near-miss in a further proportion; in other words, the suggested edit is typically either correct or requires only minor manual refinement.

An additional implication concerns multilingual development contexts. Because the task descriptions in our dataset are expressed in Spanish, the work illustrates how code-focused models can be aligned with non-English natural-language instructions without sacrificing code-generation quality.

5.4 Future Work

The most immediate priority is a *multi-snippet BTR stress test* that groups all model-predicted snippets belonging to the same commit, applies them atomically, and runs the full test suite. This would surface interaction failures that single-snippet patching cannot detect and provide a substantially stronger functional correctness claim.

Second, extending the dataset and model to cover JUnit 5 and other testing frameworks (*e.g.*, TestNG) would test whether the approach scales beyond the conservative regime of JUnit 4.x. Third, broadening BTR coverage beyond the current 5/21 evaluable repositories on dedicated infrastructure would provide tighter confidence intervals. Fourth, a head-to-head workflow benchmark—running a manifest-only bot alongside the model under the same CI configuration—would make the improvement claim measurable at the workflow level. Fifth, replacing the regex-based CodeBLEU fallback with a full tree-sitter parser would remove the "approximate" caveat from the current evaluation. Finally, integrating this approach into real-world CI pipelines and running a practitioner study would provide insights into usefulness and acceptance.

6 Threats to Validity

6.1 Internal Validity

Commit-level granularity. Some Java code changes in commits that modified JUnit dependencies may not be causally related to the dependency update, which introduces noise. We mitigated this by focusing on imports, annotations, and test methods most likely tied to JUnit; nonetheless, unrelated modifications may persist.

Data split. The 8,735 snippet-level examples are split randomly (80/20) at the example level using a fixed seed, rather than at the repository level. Although the deduplication step (similarity threshold < 0.95) reduces the risk of near-identical examples spanning partitions, information leakage through shared coding styles or project conventions cannot be ruled out. Consequently, held-out results should be interpreted as within-distribution evidence under an example-level partition, not as evidence of generalisation to unseen repositories. A repository-level split would provide a stricter guarantee but would reduce the effective test-set size; we leave this as future work.

Sequence length. All snippet-level examples fit within the encoder limit of both models (512 tokens for CodeT5-base, 1,024 for CodeT5+), so no truncation occurs at this granularity. At the class level, sequences routinely exceed this limit, and the resulting truncation is a primary cause of model failure (CodeBLEU $= 0.213$, 100% truncated errors). The class-level evaluation is therefore presented as a negative control rather than a practical operating point.

Evaluation sample size. Due to computational constraints, the model evaluation uses $N = 200$ from the 1,747 held-out test examples. While sufficient to observe clear

trends, a larger evaluation ($N = 500+$) would provide tighter confidence intervals. We prioritised end-to-end functional validation (BTR) over the ablation protocol defined in RQ3; component-level ablation runs remain as follow-up work.

6.2 External Validity

Scope of dependencies. Our dataset focuses exclusively on JUnit 3.x and 4.x; no JUnit 5 versions are present. The model has not been exposed to cross-framework migrations or to the richer set of breaking changes that accompany them. Although many of the ideas are transferable, applying the same approach to other languages, testing frameworks (*e.g.*, TestNG), or dependency managers would require new data collection, preprocessing, and evaluation.

Repository sample. The 43 repositories were selected using popularity-based criteria (stars, forks, collaborators). Projects with different governance models, sizes, or testing practices may exhibit different co-evolution patterns. The findings should be interpreted as representative of moderately popular, collaborative Maven-based Java projects rather than of the Java ecosystem as a whole.

6.3 Construct Validity

Evaluation metrics. BLEU and ROUGE capture surface similarity but do not guarantee semantic correctness or code executability. CodeBLEU adds syntax-match and data-flow-match components; however, our implementation uses a regex-based fallback rather than a full tree-sitter parser, which may over- or under-estimate structural similarity in edge cases [14, 45]. These sub-scores should be interpreted as approximate and exploratory. We partially address this through the BTR evaluation (Section 4.4.2), which provides conditional functional evidence on the 5 evaluable repositories; extending BTR coverage beyond the current 5/21 repositories remains as future work.

Similarity filtering. Deduplication uses character-level Levenshtein distance, a lexical measure that does not capture structural redundancy. AST-based or Jaccard similarity could provide stronger diversity guarantees.

EM-Java extraction. The EM-Java metric relies on a boundary-based heuristic to extract the Java section from model outputs (which lack explicit `[JAVA]` markers due to `skip_special_tokens=True` during decoding). Although verified on manual inspection, edge cases may introduce noise in a small number of comparisons.

7 Conclusions

This paper investigated how JUnit dependency upgrades affect test code in Maven-based Java projects and whether these co-changes can be learned from historical data. Starting from a corpus of 2,223 real upgrade commits, we constructed a filtered snippet-level dataset of 8,735 input–output pairs—publicly released to support reproducibility and future research—and analysed it through four research questions that cover corpus fidelity (RQ1), update magnitude across semantic-versioning categories

(RQ2), the influence of preprocessing and dataset design on learnability (RQ3), and positioning relative to existing dependency-update tools (RQ4).

JUnit 3.x/4.x upgrades are predominantly conservative. At the method level, 78.8% of pre-/post-update pairs differ by less than 8% of their character content, with a median normalised Levenshtein distance of 0.02. Moreover, this conservatism is stable across patch, minor, and major releases: mean edit distances fall within a narrow band (0.466–0.486) and CodeBLEU means are equally close (0.470–0.489). This confirms that the semantic-versioning label is a weak predictor of how much test code actually changes.

Snippet-level decomposition keeps all sequences within the encoder limits of both models (512 tokens for CodeT5-base, 1,024 for CodeT5+), explicit modality markers (`[XML]`, `[JAVA]`, `[TASK]`) provide clear input structure, and near-duplicate filtering ensures that 43% of examples require substantial edits. The resulting corpus has a mean copy rate of 0.71. This indicates that most of the output can be produced by copying from the input and applying targeted modifications, that is, a pattern well suited to encoder–decoder architectures.

Both fine-tuned models substantially outperform the manifest-only baseline on the snippet-level held-out split (example-level partition; see Section 6). In particular, CodeT5-base achieves EM-Java = 38.5% and CodeBLEU = 0.854, while CodeT5+ reaches EM-Java = 43.6% and CodeBLEU = 0.868, compared to 0% and 0.502 for the baseline. The improvement is largest on the significant-change tier (+0.528 CodeBLEU). Furthermore, conditional functional validation via BTR shows that model-generated patches compile and pass tests in all 37 evaluable cases across 5 of 21 sampled repositories, whereas manifest-only version bumps fail in 50% of control cases. These results provide functional evidence beyond lexical similarity, although they remain conditional on build-environment reproducibility and should not be interpreted as a general correctness guarantee.

Nevertheless, several limitations temper these findings. The corpus covers only JUnit 3.x/4.x, the CodeBLEU implementation uses a regex-based fallback rather than a full tree-sitter parser, and the held-out evaluation uses $N = 200$ samples. Results on held-out snippets should be interpreted as within-distribution evidence under an example-level partition; repository-level generalisation remains untested. The most immediate directions for future work are, therefore, to extend the dataset to JUnit 5 and other frameworks, to broaden BTR coverage beyond the current 5/21 repositories on dedicated infrastructure, to replace the regex fallback with tree-sitter, and to conduct a head-to-head workflow benchmark against manifest-only bots.

Independent of these extensions, the empirical characterisation of JUnit co-evolution patterns and the publicly released snippet-level dataset constitute reusable resources for the research community, applicable to future models, tools, and empirical studies on dependency management beyond the specific architecture evaluated here.

Declarations

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Data availability

The dataset, scripts, model training code, and configurations used in this study are publicly available in the replication package at [DOUBLE BLIND REVIEW]. Raw and processed datasets in JSONL format, as well as fine-tuned model weights, are included.

Author contributions

Miguel Botto-Tobar: Conceptualisation, Methodology, Software, Data curation, Investigation, Writing – original draft, Visualisation. Santiago A. Vidal: Supervision, Methodology, Validation, Writing – review & editing. Claudia Marcos: Supervision, Methodology, Validation, Writing – review & editing. All authors read and approved the final manuscript.

Conflict of interest

The authors declare that they have no conflict of interest.

References

[1] Cox, J., Bouwers, E., Van Eekelen, M., Visser, J.: Measuring dependency freshness in software systems. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 109–118 (2015). IEEE

[2] Decan, A., Mens, T., Constantinou, E.: On the evolution of technical lag in the npm package dependency network. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 404–414 (2018). IEEE

[3] Stringer, J., Tahir, A., Blincoe, K., Dietrich, J.: Technical lag of dependencies in major package managers. In: 2020 27th Asia-Pacific Software Engineering Conference (APSEC), pp. 228–237 (2020). IEEE

[4] Mohayeji, H., Agaronian, A., Constantinou, E., Zannone, N., Serebrenik, A.: Securing dependencies: A comprehensive study of dependabot's impact on vulnerability mitigation. Empirical Software Engineering **30**(3), 89 (2025)

[5] Bifolco, D., Romano, S., Nocera, S., Francese, R., Scanniello, G., Di Penta, M.: An empirical study on the accuracy of github's dependency graph and the nature of its inaccuracy. Information and Software Technology, 107854 (2025)

[6] He, R., He, H., Zhang, Y., Zhou, M.: Automating dependency updates in practice: An exploratory study on github dependabot. IEEE Transactions on Software Engineering **49**(8), 4004–4022 (2023) https://doi.org/10.1109/TSE.2023.3278129

[7] Jezek, K., Dietrich, J., Brada, P.: How java apis break–an empirical study. Information and Software Technology **65**, 129–146 (2015)

[8] Nguyen, H.A., Nguyen, T.T., Wilson Jr, G., Nguyen, A.T., Kim, M., Nguyen, T.N.: A graph-based approach to api usage adaptation. ACM Sigplan Notices **45**(10), 302–321 (2010)

[9] Alrubaye, H., Mkaouer, M.W., Khokhlov, I., Reznik, L., Ouni, A., Mcgoff, J.: Learning to recommend third-party library migration opportunities at the api level. Applied Soft Computing **90**, 106140 (2020)

[10] Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. ACM Transactions on Software Engineering and Methodology (TOSEM) **20**(4), 1–35 (2011)

[11] Meng, N., Kim, M., McKinley, K.S.: Lase: locating and applying systematic edits by learning from examples. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 502–511 (2013). IEEE

[12] Xing, Z., Stroulia, E.: Api-evolution support with diff-catchup. IEEE Transactions on Software Engineering **33**(12), 818–836 (2007)

[13] Zaidman, A., Van Rompaey, B., Van Deursen, A., Demeyer, S.: Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Software Engineering **16**(3), 325–364 (2011)

[14] Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S.: Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297 (2020)

[15] Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K.: Do developers update their library dependencies? Empirical Software Engineering **23**(1), 384–417 (2019) https://doi.org/10.1007/s10664-017-9521-5

[16] Dietrich, J., Pearce, D., Stringer, J., Tahir, A., Blincoe, K.: Dependency versioning in the wild. In: Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 349–359 (2019). https://doi.org/10.1109/MSR.2019.00061

[17] Wang, Y., Chen, B., Huang, K., Shi, B., Xu, C., Peng, X., Wu, Y., Liu, Y.: An empirical study of usages, updates and risks of third-party libraries in java projects. In: 2020 IEEE International Conference on Software Maintenance and

Evolution (ICSME), pp. 35–45 (2020). IEEE

[18] Han, J., Deng, S., Lo, D., Zhi, C., Yin, J., Xia, X.: An empirical study of the dependency networks of deep learning libraries. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 868–878 (2020). IEEE

[19] Hejderup, J., Gousios, G.: Can we trust tests to automate dependency updates? a case study of java projects. Journal of Systems and Software **183**, 111097 (2022) https://doi.org/10.1016/j.jss.2021.111097

[20] Zeng, J., Han, D., Zhu, Y., Wang, Y., Weng, F.: A survey of third-party library security research in application software. arXiv preprint arXiv:2404.17955 (2024)

[21] Ohm, M., Plate, H., Sykosch, A., Meier, M.: Backstabber's knife collection: A review of open source software supply chain attacks. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 23–43 (2020). Springer

[22] Ladisa, P., Sahin, M., Ponta, S.E., Rosa, M., Martinez, M., Barais, O.: The hitchhiker's guide to malicious third-party dependencies. In: Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, pp. 65–74 (2023)

[23] Xia, B., Bi, T., Xing, Z., Lu, Q., Zhu, L.: An empirical study on software bill of materials: Where we stand and the road ahead. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 2630–2642 (2023). IEEE

[24] O'Donoghue, E., Boles, B., Izurieta, C., Reinhold, A.M.: Impacts of Software Bill of Materials (SBOM) Generation on Vulnerability Detection. In: Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, pp. 67–76. ACM, Salt Lake City UT USA (2023). https://doi.org/10.1145/3689944.3696164

[25] Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 313–324 (2014)

[26] Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon: A library for implementing analyses and transformations of java source code. Software: Practice and Experience **46**(9), 1155–1179 (2016)

[27] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al.: Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)

[28] Ahmad, W., Chakraborty, S., Ray, B., Chang, K.-W.: Unified pre-training for program understanding and generation. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 2655–2668 (2021)

[29] Wang, Y., *et al.*: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pp. 567–580 (2021)

[30] Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al.: Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023)

[31] Lozhkov, A., Li, R., Allal, L.B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., et al.: Starcoder 2 and the stack v2: The next generation. arXiv preprint arXiv:2402.19173 (2024)

[32] Dearing, M.T., Tao, Y., Wu, X., Lan, Z., Taylor, V.: Leveraging llms to automate energy-aware refactoring of parallel scientific codes. arXiv preprint arXiv:2505.02184 (2025)

[33] Tawosi, V., Alamir, S., Liu, X., Veloso, M.: Llm agents for automated dependency upgrades. arXiv preprint arXiv:2510.03480 (2025)

[34] Ziftci, C., Nikolov, S., Sjövall, A., Kim, B., Codecasa, D., Kim, M.: Migrating code at scale with llms at google. In: Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, pp. 162–173 (2025)

[35] Reyes, F., Mahmoud, M., Bono, F., Nadi, S., Baudry, B., Monperrus, M.: Byam: Fixing breaking dependency updates with large language models. arXiv preprint arXiv:2505.07522 (2025)

[36] Almeida, A., Xavier, L., Valente, M.T.: Automatic library migration using large language models: First results. In: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 427–433 (2024)

[37] Preston-Werner, T.: Semantic Versioning 2.0.0. https://semver.org/. Accessed: 2025-02-01 (2013)

[38] Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet physics doklady **10**(8), 707–710 (1966)

[39] Wang, Y., Le, H., Gotmare, A., Bui, N., Li, J., Hoi, S.: Codet5+: Open code large language models for code understanding and generation. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp.

1069–1088 (2023)

[40] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al.: Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:1910.03771 (2019)

[41] Papineni, K., Roukos, S., Ward, T., Zhu, W.-J.: Bleu: A method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, pp. 311–318 (2002). https://doi.org/10.3115/1073083.1073135

[42] Lin, C.-Y.: Rouge: A package for automatic evaluation of summaries. In: Text Summarization Branches Out: Proceedings of the ACL-04 Workshop, pp. 74–81 (2004)

[43] GitHub: About Dependabot version updates. https://docs.github.com/en/code-security/dependabot/dependabot-version-updates/about-dependabot-version-updates. Accessed: 2025-02-01 (2024)

[44] Mend: Renovate: Automated Dependency Updates. https://docs.renovatebot.com/. Accessed: 2025-02-01 (2024)

[45] Evtikhiev, M., Bogomolov, E., Kobyzev, I., Bryksin, T.: Out of the BLEU: How should we assess quality of the code generation models? Journal of Systems and Software **203**, 111741 (2023) https://doi.org/10.1016/j.jss.2023.111741

[46] MojoHaus: Versions Maven Plugin. https://www.mojohaus.org/versions/versions-maven-plugin/. Accessed: 2025-02-01 (2024)