CoUpJava: A Dataset of Code Upgrade Histories in Open-Source Java Repositories

Kaihang Jiang
University of Waterloo
Waterloo, Canada
k52jiang@uwaterloo.ca

Bihui Jin
University of Waterloo
Waterloo, Canada
bihui.jin@uwaterloo.ca

Pengyu Nie
University of Waterloo
Waterloo, Canada
pynie@uwaterloo.ca

***Abstract*—Modern programming languages are constantly evolving, introducing new language features and APIs to enhance software development practices. Software developers often face the tedious task of upgrading their codebase to new programming language versions. Recently, large language models (LLMs) have demonstrated potential in automating various code generation and editing tasks, suggesting their applicability in automating code upgrade. However, there exists no benchmark for evaluating the code upgrade ability of LLMs, as distilling code changes related to programming language evolution from real-world software repositories' commit histories is a complex challenge.**

In this work, we introduce CoUpJava, the first large-scale dataset for code upgrade, focusing on the code changes related to the evolution of Java. CoUpJava comprises 10,697 code upgrade samples, distilled from the commit histories of 1,379 open-source Java repositories and covering Java versions 7–23. The dataset is divided into two subsets: CoUpJava-Fine, which captures fine-grained method-level refactorings towards new language features; and CoUpJava-Coarse, which includes coarse-grained repository-level changes encompassing new language features, standard library APIs, and build configurations. Our proposed dataset provides high-quality samples by filtering irrelevant and noisy changes and verifying the compilability of upgraded code. Moreover, CoUpJava reveals diversity in code upgrade scenarios, ranging from small, fine-grained refactorings to large-scale repository modifications.

***Index Terms*—Code upgrade, software evolution, Java, dataset, benchmark**

I. Introduction

Modern programming languages, such as Java, frequently evolve to introduce new language features and application programming interfaces (APIs). Code upgrade, i.e., adapting the codebase to a new programming language version, is an important task that developers frequently need to perform for functionality and security purposes [13], [23], [24]. Code upgrade can be enforced by the end of support of older programming language versions; as an example, Java receives 5 years of support for LTS versions and only half a year of support for non-LTS versions [18]. Replacing deprecated features and APIs to newer ones can be a tedious and time-consuming process.

Large Language Models (LLMs) trained on programming languages and natural languages [7], [16], [22] have demonstrated prosperous promise in reshaping software development across various realms of applications, such as code editing [19], [31], [32], API migration [9], [11], [15], refactoring [25], and testing [14], [30]. By 2024, Google's AI-generated code surpasses 25% of new development [6].

LLMs can be a powerful tool for automating code upgrade. Prior work using LLMs for API migration [9], [11], [15] and automated code refactoring [25] focuses on small code edits caused by replacing a deprecated API (which may or may not be related to a new programming language version). However, code upgrade in real-world software repositories can involve replacing a large amount of older (but not necessarilly deprecated) APIs, language features, and build configurations to newer ones. There has been no benchmark for understanding LLMs' code upgrade capability in practice.

To bridge this gap, we propose CoUpJava, a high-quality dataset of code upgrade samples mined and cleaned from open-source software repositories. CoUpJava consists of 10,697 samples in two granularities:

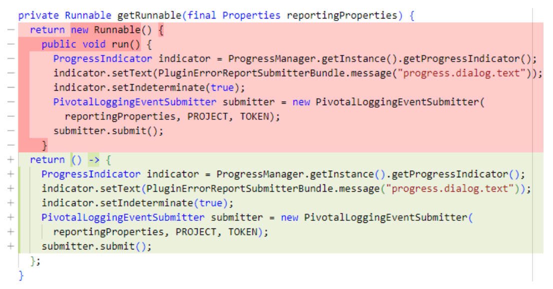
1) **CoUpJava-Fine**: 9,784 fine-grained (method-level) refactorings towards new language features, representing the scenario of upgrading individual methods;
2) **CoUpJava-Coarse**: 913 coarse-grained (repository-level) changes from an old language version to a new language version, representing the scenario of upgrading the repository as a whole.

Analysis of the data collected in CoUpJava reveals several interesting findings. In CoUpJava-Fine, we found that developers are more likely to adopt new language features that involve simple code edits. In CoUpJava-Coarse, we found that developers frequently need to modify non-code files during repository-level code upgrade, underscoring the importance of context-aware LLMs on this task.

Our contributions of this work include:

- We present CoUpJava, a dataset of 10,697 code upgrade samples, meticulously curated from 1,379 repositories covering Java versions 7–23.
- We document the steps and challenges faced during the construction of CoUpJava.
- We analyze the distributions and characteristics of different types of code upgrade in CoUpJava, as well as envision the application of LLMs on automating code upgrade.

CoUpJava and its collection scripts are available at https://github.com/uw-swag/CoUpJava



```
  private Runnable getRunnable(final Properties reportingProperties) {
-   return new Runnable() {
-     public void run() {
-       ProgressIndicator indicator = ProgressManager.getInstance().getProgressIndicator();
-       indicator.setText(PluginErrorReportSubmitterBundle.message("progress.dialog.text"));
-       indicator.setIndeterminate(true);
-       PivotalLoggingEventSubmitter submitter = new PivotalLoggingEventSubmitter(
-         reportingProperties, PROJECT, TOKEN);
-       submitter.submit();
-     }
+   return () -> {
+     ProgressIndicator indicator = ProgressManager.getInstance().getProgressIndicator();
+     indicator.setText(PluginErrorReportSubmitterBundle.message("progress.dialog.text"));
+     indicator.setIndeterminate(true);
+     PivotalLoggingEventSubmitter submitter = new PivotalLoggingEventSubmitter(
+       reportingProperties, PROJECT, TOKEN);
+     submitter.submit();
    };
  }
```

Fig. 1: Example of replace anonymous with lambda code upgrade, from `intino/magritte` commit `8c707bb2`.

II. Dataset Construction

Collecting real-world code upgrade samples from open-source repositories is challenging due to their diverse development workflows. Developers frequently mix multiple concerns (e.g., bug fixes and documentation improvements) in the same changeset [12], and accurately extracting code upgrade code changes from the noisy changesets can be cubersome. Moreover, some historical revisions of open-source repositories may fail to build any more due to their dependencies being removed from online (especially for repositories with customized build configurations or dependency repositories).

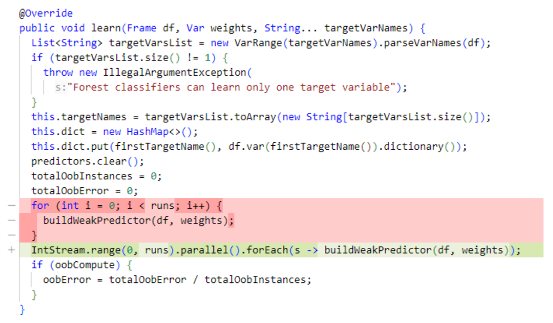
To address these challenges, we design a robust data construction workflow to collect CoUpJava from open-source Java repositories. Specifically, CoUpJava consists of two subsets, covering two different data granularities and usage scenarios. CoUpJava-Fine contains fine-grained, method-level code changes, where each sample is a minimal change that upgrades an usage of an old language feature to the equivelant new language feature; this subset simulates the usage scenario where the developer intends to incrementally upgrade the codebase with new language features. CoUpJava-Coarse contains coarse-grained, repository-level code changes, where each sample records a verified upgrade of the repository from an old language version to a new one; this subset represents the usage scenario where the developer aims to upgrade the entire codebase in a single effort.

A. *Selection of Repositories and Java Versions*

We aim to collect data from as many open-source Java repositories as possible and covering a wide range of Java programming language versions. We utilize the GitHub search API [5] to search for repositories. GitHub search API has a limitation where each query can return at most 1,000 results, thus we perform multiple queries for each Java version. We exclude duplicate repositories across multiple queries as well as fork repositories to ensure the quality of our data sources.

In this work, we focus on Java versions 7–23. Java 7, released in July 2011, is the oldest version that still allows practically compiling code on mordern operating systems[1].

[1]The official development toolkits for Java 4 and before cannot be installed on 64bit Linux systems. Although the development toolkits for Java 5 and 6 are avilable, we encounter a security protocol issue when fetching the dependencies of the open-source repositories, as the online dependency hosting services require TLS 1.3 that is not supported in Java 5 and 6.



```
  @Override
  public void learn(Frame df, Var weights, String... targetVarNames) {
    List<String> targetVarsList = new VarRange(targetVarNames).parseVarNames(df);
    if (targetVarsList.size() != 1) {
      throw new IllegalArgumentException(
        s:"Forest classifiers can learn only one target variable");
    }
    this.targetNames = targetVarsList.toArray(new String[targetVarsList.size()]);
    this.dict = new HashMap<>();
    this.dict.put(firstTargetName(), df.var(firstTargetName()).dictionary());
    predictors.clear();
    totalOobInstances = 0;
    totalOobError = 0;
-   for (int i = 0; i < runs; i++) {
-     buildWeakPredictor(df, weights);
-   }
+   IntStream.range(0, runs).parallel().forEach(s -> buildWeakPredictor(df, weights));
    if (oobCompute) {
      oobError = totalOobError / totalOobInstances;
    }
  }
```

Fig. 2: Example of replace loop with pipeline code upgrade, from `padreati/rapaio` commit `6d086a1d`.

Java 23, released in September 2024, is the latest released version at the time of dataset construction.

To identify the Java version used by a repository, we parse the build configuration file (`pom.xml`) of Maven [26], a popular build system in the Java ecosystem. In `pom.xml`, the Java version used can be specified as "$\langle$`maven.compiler.source`$\rangle$ `version` $\langle$/`maven.compiler.source`$\rangle$" , where `version` is an integer from [7, 23]; however, due to Java's historical versioning schema, the version number for Java 7–10 can also be written in the format of 1.7–1.10. We run a GitHub search API query for each version with the following parameters (where `filename` : `pom` `extension` : `xml` limits the search in `pom.xml`, and `path` : / limits the search in the file only at the repository's root directory):

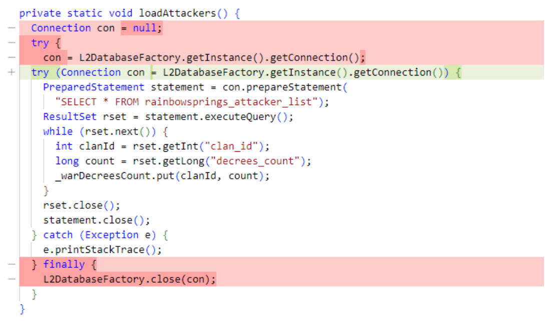
$$\langle\texttt{maven.compiler.source}\rangle\texttt{version}\langle/\texttt{maven.compiler.source}\rangle$$
$$\texttt{filename}:\texttt{pom}\ \ \texttt{extension}:\texttt{xml}\ \ \texttt{path}:/$$
$$for\ \ \texttt{version} \in [7, 23] \cup \{1.7, 1.8, 1.9, 1.10\}$$

B. CoUpJava-Fine

New programming language features, such as lambda expressions introduced in Java 8, allow implementing the same functionalities with elegant code. The upgrade from the old code snippet using the old language features (e.g., anonymous classes) to the new code snippet using the new language features (e.g., lambda expressions) can be considered as a refactoring, as the code semantics do not change.

We use a robust refactoring mining tool for Java, RefactoringMiner [1], [27], [28], to collect such code-upgrade-related refactorings. RefactoringMiner detects refactorings in the commit histories of a Java repository by analyzing abstract syntax tree (AST) differences. Out of the 102 refactoring types implemented in RefactoringMiner (version 3.0 as the time of writing), five of them are relevant to code upgrade:

- **replace anonymous with lambda** (Java 8; Figure 1): changing anonymous class with a single method (that supports functional interface) to a lambda expression.
- **replace loop with pipeline** (Java 8; Figure 2): leveraging lambda expressions, replacing `for`/`while`/`do` loops with `forEach(...)` expressions.
- **merge catch** (Java 7): merging multiple `catch` blocks into one, when they have exactly the same body, using the overloaded "|" operator to match multiple exception types.



```
  private static void loadAttackers() {
-   Connection con = null;
-   try {
-     con = L2DatabaseFactory.getInstance().getConnection();
+   try (Connection con = L2DatabaseFactory.getInstance().getConnection()) {
      PreparedStatement statement = con.prepareStatement(
        "SELECT * FROM rainbowsprings_attacker_list");
      ResultSet rset = statement.executeQuery();
      while (rset.next()) {
        int clanId = rset.getInt("clan_id");
        long count = rset.getLong("decrees_count");
        _warDecreesCount.put(clanId, count);
      }
      rset.close();
      statement.close();
    } catch (Exception e) {
      e.printStackTrace();
-   } finally {
-     L2DatabaseFactory.close(con);
    }
  }
```

Fig. 3: Example of try with resources code upgrade, from `valanths1990/l2j-server-datapack` commit `f249ce92`.

TABLE I: Statistics of CoUpJava: number of repositories, samples, and average number of lines edited in each sample.

Dataset	#Repo	#Data	#Line$_\Delta$
CoUpJava	1,379	10,697	401.2
CoUpJava-Fine	742	9,784	10.7
CoUpJava-Coarse	742	913	4,585.6

- **replace generic with diamond** (Java 7): ommiting the generic type parameter when it can be inferred from the context (e.g., `List⟨Integer⟩ l = new ArrayList⟨Integer⟩()` where the second `Integer` can be inferred from the first one).
- **try with resources** (Java 7; Figure 3): simplifying the `try-catch-finally` block with the newly introduced try-with-resources feature, so that the resource variables can be declared in the `try(...)` construct.

We instruct RefactoringMiner to mine these five types of refactorings from the open-source repositories selected in §II-A. Each output from RefactoringMiner includes the refactoring type and description, the revision where the refactoring is found, the code element locations (file path, AST type, line and column numbers) before and after refactoring. From the code element locations, we locate the methods containing the code elements before and after the refactoring.

However, we find that a significant portion of these refactorings are entangled with other code edits irrelevant to code upgrade, such as formatting code and changing logging content. We decide to filter out such cases and collect CoUpJava-Fine as a purified dataset with only code upgrade edits. Specifically, we compare the changed lines reported by RefactoringMiner (that only contains the refactored part) against the git diff for the method. If git diff contains more changed lines than RefactoringMiner's diff, we discard such data. Note that we do not attempt to craft a diff based on RefactoringMiner's output (e.g., from old revision of code, modifying the changed lines reported by RefactoringMiner and copying the other lines) because doing so frequently leads to uncompilable code.

Each sample in CoUpJava-Fine includes: repository name, the revision's hash and timestamp, the refactoring type and description, the method on the old revision, and the method on the new revision.

C. CoUpJava-Coarse

During the collection of CoUpJava-Fine, we noticed that code upgrade frequently requires changes beyond the method level, e.g., deleting methods/classes that can be better replaced by new APIs introduced in the new Java version. Sometimes, developers also change the non-code portion of the repository, e.g., the build configuration file for upgrading dependencies' versions. Disentangling such code edits into atomic ones corresponding to specific new language features or new/deprecated APIs can be extremely difficult. Instead, we decide to use compilability check[2] to determine if a code upgrade is successful or not, considering that newer LLMs with larger context windows have the potential to perform large and complicated code edits. We collect edits at the repository level, perversing edits in all code and non-code files, as a realistic benchmark on the challenging code upgrade task.

For each repository, we search for the revisions that upgrades the Java version specified in its build configuration file (`pom.xml`), with the help of git's regex search functionality: `git log -G '⟨maven.compiler.source⟩'`. Then, we ensure that the repository is compilable (i.e., the command `mvn clean test-compile` succeeds) on the revisions before and after the Java version change. If the revision right before/after the Java version change is not compilable, we move to try at most 3 revisions prior/subsequent to that revision, and discard the data if all 3 trails fail. This approach excludes the intermediate revisions where the code upgrade is in progress.

We ensure that the revisions collected in our dataset are compilable for two reasons: (1) compilable code is more likely to be high-quality code and less likely to contain unfinished edits; (2) compilation can be used as an oracle to decide if code upgrade is successful, when building an automated code upgrade technique on our dataset. We use the official Java development toolkits version 7–23 to compile the repositories, downloaded from Oracle's Java Archive [17]. We use Maven 3.9.9 as the build system, except for Java 7 where an older version of Maven 3.8.8 is required. The runtime environments for all Java versions and the script to switch among them are provided together with our dataset for replicability.

Each sample in CoUpJava-Coarse includes: repository name, the old/new revisions' hash and timestamp, the old/new Java versions, and the changeset between the two revisions.

III. Dataset Statistics

Table I shows the statistics of CoUpJava and its two subsets, CoUpJava-Fine and CoUpJava-Coarse. CoUpJava contains 10,697 code upgrade samples (9,784 fine-grained and 913 coarse-grained) from 1,379 repositories. On average, each fine-grained code upgrade involves 10.7 lines of code edits, and each coarse-grained code upgrade involves 4,585.6 lines of code and non-code edits. Although 15,391 repositories are selected following the process in §II-A, the final collected data

[2]Although tests are available in many of the open-source Java repositories, directly performing test execution check is not practical due to the difficulty in setting up correct runtime environments and excluding falky tests. We leave integrating tests into our benchmark as future work.

TABLE II: Code upgrade types in CoUpJava-Fine.

Code Upgrade Type	Java Ver.	#Repo	#Commit	#Data	#Line$_{old}$	#Line$_{new}$	#Line$_{\triangle}$
all	N/A	742	2,577	9,784	31.8	29.5	10.7
replace anonymous with lambda	8	204	592	1,884	40.4	35.7	23.6
replace loop with pipeline	8	9	9	10	10.7	9.0	4.5
merge catch	7	103	164	320	32.3	28.6	8.1
replace generic with diamond	7	384	1,199	5,144	26.4	26.2	2.9
try with resources	7	293	732	2,426	36.8	32.0	17.6

TABLE III: Edits by file type in CoUpJava-Coarse.

File Type	#Data	#File$_{mod}$	#File$_{add}$	#File$_{del}$
all	913	17.2	7.9	9.4
Java source code	546	17.9	7.9	12.7
Java test code	249	3.7	1.6	0.3
build scripts	913	1.6	0.1	0.0
others	576	6.2	4.2	2.7

comes from a smaller set of repositories, because we apply strict filters to collect only high-quality samples.

Table II presents the distributions of data samples in CoUpJava-Fine by code upgrade refactoring types. We found that replace generic with diamond is the most popular code upgrade type, with 5,144 samples spanning 1,199 revisions of 384 repositories, followed by try with resources (2,426 samples) and replace anonymous with lambda (1,884 samples). The replace anonymous with lambda code upgrade involves the largest diff (23.6 diff lines on average), while replace generic with diamond are among the simplest types of code upgrade (2.9 diff lines on average).

To better understand how developers perform code upgrade in practice, we analyze the types of files being changed in CoUpJava-Coarse. Specifically, we study four types of files: Java source code (the Java files under `src/main/java`), Java test code (the Java files under `src/test/java`), build configuration files (`pom.xml`), and all other files (resources, documentations, etc.). Table III shows the analysis results. We find that 546 samples changed source code and 249 changed test code, indicating that changing source code or test code files is sometimes not required for upgrading Java versions, thanks to the backward-compatibility design philosophy of the Java programming language. At least one build configuration file needs to be changed to specify the new Java version, but we find that changes may be required in multiple build configuration files in a repository (on average: 1.6 files per code upgrade). We also observe a significant amount of other files being involved in code upgrade revisions.

IV. Sample Applications and Future Work

Automating Code Upgrade with LLMs. Our dataset is suitable for exploring the ability of LLMs in automating code upgrade in two usage scenarios:

1) On CoUpJava-Fine: upgrading of individual method with the goal of adopting a specific new language feature. Evaluation metrics can be syntactical similarity between the predicted code edits against ground truth, e.g., exact match accuracy and CodeBLEU [21], plus checking whether the edited code uses the new language feature.
2) On CoUpJava-Coarse: upgrading the entire repository from an old programming language version to a new one. Evaluation metrics include compilation check (whether the codebase can compile after integrating predicted edits) and syntactical similarity between each file's predicted edits against ground truth.

Syntactical similarity metrics have the limitation of not accounting for functionally correct code with alternative naming conventions or implementations. Compilation check on CoUpJava-Coarse can partially detect functional correctness. We plan to extend CoUpJava with executable tests, to be extracted and cleaned to remove failing and flaky tests, that can more accurately measure functional correctness and enable evaluation metrics like Pass@K [2].

There are existing datasets [3], [4], [8]–[10], [29] that explores API migrations at method-level, which are similar to CoUpJava-Fine. However, there exists no repository-level code upgrade benchmark before CoUpJava-Coarse.

Amazon Q [15], an LLM-based code assistant by Amazon, offers a feature called transformation [20] that supports repository-level code upgrades from Java 8/11 to 17. It requires uploading the entire repository to a remote server for a code upgrade and verification process that typically takes 10 to 30 minutes. CoUpJava will be an ideal benchmark for evaluating Amazon Q and other LLMs.

Future Work. In addition to collecting executable tests, we also plan to expand CoUpJava with data from more open-source repositories. We will improve the construction of CoUpJava in several aspects: (1) supporting more refactoring types related to code upgrade, especially those introduced in recent Java versions; (2) detecting Java version from multiple sources to increase coverage and robustness, including configuration files of other build systems (e.g., Gradle) and CI (e.g., GitHub Actions); (3) regularly expanding the dataset upon the release of new Java versions.

Acknowledgments

We thank Yu Liu and the anonymous reviewers for their comments and feedback. This work is enabled in part by support provided by Compute Ontario (computeontario.ca) and the Digital Research Alliance of Canada (alliancecan.ca). This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under funding reference number RGPIN-2024-04909 and the University of Waterloo start-up grant.

References

[1] P. Alikhanifard and N. Tsantalis, “A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools,” *Transactions on Software Engineering and Methodology*, vol. 34, no. 2, 2025. [Online]. Available: https://doi.org/10.1145/3696002

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.

[3] M. Fazzini, Q. Xin, and A. Orso, “Automated API-usage update for Android apps,” in *International Symposium on Software Testing and Analysis*, 2019, pp. 204–215. [Online]. Available: https://doi.org/10.1145/3293882.3330571

[4] ——, “APIMigrator: an API-usage migration tool for Android apps,” in *International Conference on Mobile Software Engineering and Systems*, 2020, pp. 77–80. [Online]. Available: https://doi.org/10.1145/3387905.3388608

[5] GitHub Inc. REST API endpoints for search. [Online]. Available: https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28

[6] Google, “Q3 earnings call: CEO’s remarks.” [Online]. Available: https://blog.google/inside-google/message-ceo/alphabet-earnings-q3-2024/#search

[7] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, F. Luo, Y. Xiong, and W. Liang, “DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence,” *arXiv preprint arXiv:2401.14196*, 2024.

[8] S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, “Automatic Android deprecated-API usage update by learning from single updated example,” in *International Conference on Program Comprehension*, 2020, pp. 401–405. [Online]. Available: https://doi.org/10.1145/3387904.3389285

[9] M. Islam, A. K. Jha, S. Nadi, and I. Akhmetov, “PyMigBench: A benchmark for Python library migration,” 2023, pp. 511–515. [Online]. Available: https://doi.org/10.1109/MSR59073.2023.00075

[10] M. Lamothe, W. Shang, and T.-H. P. Chen, “A3: Assisting Android API migrations using code examples,” *Transactions on Software Engineering*, vol. 48, no. 2, pp. 417–431, 2022.

[11] J. Liang, W. Zou, J. Zhang, Z. Huang, and C. Sun, “A deep method renaming prediction and refinement approach for Java projects,” 2021, pp. 404–413.

[12] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” *Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[13] R. Nascimento, A. Hora, and E. Figueiredo, “Exploring API deprecation evolution in JavaScript,” in *International Conference on Software Analysis, Evolution and Reengineering*, 2022, pp. 169–173.

[14] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, “Learning deep semantics for test completion,” in *International Conference on Software Engineering*, 2023, pp. 2111–2123.

[15] B. Omidvar Tehrani, I. M, and A. Anubhai, “Evaluating human-AI partnership for LLM-based code migration,” in *CHI Conference on Human Factors in Computing Systems*, 2024. [Online]. Available: https://doi.org/10.1145/3613905.3650896

[16] OpenAI *et al.*, “GPT-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.

[17] Oracle. Java archive. [Online]. Available: https://www.oracle.com/java/technologies/downloads/archive/

[18] Oracle. Oracle Java SE support roadmap. [Online]. Available: https://www.oracle.com/java/technologies/java-se-support-roadmap.html

[19] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. J. Mooney, “Learning to update natural language comments based on code changes,” in *Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 1853–1868.

[20] D. Poccia. Upgrade your Java applications with Amazon Q code transformation (preview). [Online]. Available: https://aws.amazon.com/blogs/aws/upgrade-your-java-applications-with-amazon-q-code-transformation-preview/

[21] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “CodeBLEU: A method for automatic evaluation of code synthesis,” *arXiv preprint arXiv:2009.10297*, 2020.

[22] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, “Code Llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2024.

[23] A. A. Sawant, R. Robbes, and A. Bacchelli, “On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK,” *Empirical Software Engineering*, vol. 23, pp. 2158–2197, 2018.

[24] ——, “To react, or not to react: Patterns of reaction to API deprecation,” *Empirical Software Engineering*, vol. 24, pp. 3824–3870, 2019.

[25] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, “Refactoring programs using large language models with few-shot examples,” in *Asia-Pacific Software Engineering Conference*, 2023, pp. 151–160.

[26] The Apache Software Foundation. Welcome to Apache Maven. [Online]. Available: https://maven.apache.org/

[27] N. Tsantalis, A. Ketkar, and D. Dig, “RefactoringMiner 2.0,” *Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[28] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, “Accurate and efficient refactoring detection in commit history,” in *International Conference on Software Engineering*, 2018, pp. 483–494. [Online]. Available: http://doi.org/10.1145/3180155.3180206

[29] S. Xu, Z. Dong, and N. Meng, “Meditor: Inference and application of API migration edits,” in *International Conference on Program Comprehension*, 2019, pp. 335–346. [Online]. Available: https://doi.org/10.1109/ICPC.2019.00052

[30] J. Zhang, Y. Liu, P. Nie, J. J. Li, and M. Gligoric, “exLong: Generating exceptional behavior tests with large language models,” in *International Conference on Software Engineering*, 2025.

[31] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, “Multilingual code co-evolution using large language models,” 2023, pp. 695–707.

[32] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, “CoditT5: Pretraining for source code and natural language editing,” in *Automated Software Engineering*, 2022, pp. 22:1–12.