J2KMigBench: A Dataset for Benchmarking Java-to-Kotlin Code Migration

Moshood A. Fakorede, Marcoantoni Rubio-Gonzalez, Umar Farooq
Louisiana State University
Baton Rouge, LA, USA
{mfakor1, mrubio9, ufarooq}@lsu.edu

***Abstract*—Kotlin has been Android's officially preferred language since 2019, and legacy Java codebases have been migrating to it ever since, file by file, in thousands of public repositories. Despite this sustained, observable activity, no existing benchmark provides real, historical, file-level ground truth for it. We present J2KMigBench, a dataset of confirmed Java-to-Kotlin migrations, an open-source tool that mines, verifies, and packages them, and a public web explorer for browsing the result, covering 1,961 repositories, 5,379 pull requests, 23,951 commits, and 80,723 touched files as of September 2026. Each task instance links a pre-migration Java file to its human-authored post-migration Kotlin counterpart with full commit- and pull-request-level provenance. J2KMigBench is the first dataset in this space to combine human-authored, non-synthetic ground truth at this scale with the open-source tool used to produce it, both publicly available at https://github.com/j2kmigbench/dataset. [TODO: Insert the permanent, DOI-bearing archive link (Zenodo or Figshare) here before submission, and append the demonstration/dataset video URL, as required by the submission instructions.]**

***Index Terms*—dataset, benchmark, code migration, Java, Kotlin, mining software repositories, reproducibility**

I. Introduction

Since Google named Kotlin the preferred language for Android development in 2019, production teams have migrated their Java codebases directly, as Duolingo's Android app [1], [2] and Meta's internal Java-to-Kotlin translation effort [3] both document, and the same shift plays out incrementally, file by file, as sustained, observable history across thousands of smaller public repositories. Yet existing code-migration benchmarks are either synthetic, small, or scoped to same-language upgrades and library migrations [4]–[8], none targeting Java-to-Kotlin, and the one prior study of this specific migration trained a file-ranking model on a custom corpus of developer migrations rather than releasing real, human-authored, file-level ground truth [9].

We built J2KMigBench to close that gap. It is mined directly from confirmed Java-to-Kotlin migration commits in real, actively maintained open-source projects, retains full commit- and pull-request-level provenance, and is released as a standalone, reusable artifact, independent of any single consuming system.

Reusable, real-history benchmark datasets are an established software-engineering-research tradition, from library migration with PyMigBench [10] to more recent backporting datasets [11]. J2KMigBench follows that tradition for the specific case of a cross-language, same-runtime migration.

Before (`.java`)

```java
package com.futsch1.medtimer;

public interface OnFragmentReselectedListener {
  void onFragmentReselected();
}
```

After (`.kt`)

```kotlin
package com.futsch1.medtimer

interface OnFragmentReselectedListener {
  fun onFragmentReselected()
}
```

Fig. 1. A real migrated instance (medTimer, PR #1279, commit `2d1f1d2`), the smallest single-file migration in the dataset.

We make the following contributions:

- **J2KMigBench**, a dataset of 23,951 confirmed Java-to-Kotlin migration commits across 1,961 real, actively maintained repositories, each linking a pre-migration Java file to its human-authored post-migration Kotlin counterpart.
- An open-source tool that mines, verifies, and packages these task instances from GitHub.
- A public, interactive web explorer over the full, versioned index, offering per-repository and per-pull-request views.

This paper shows sample migrations from the dataset (Section II), describes its construction methodology (Section III), outlines the research questions and use cases it enables (Section IV), positions it against the closest existing migration benchmarks (Section V), discusses threats to validity (Section VI), and concludes (Section VII).

II. Sample Migrations

Two real migrated instances illustrate the dataset's range, from a purely syntactic change to one requiring semantic judgment. Figure 1 shows the smallest kind: a five-line interface from medTimer's PR #1279, dropping the redundant `public` modifier and translating `void onFragmentReselected();` to `fun onFragmentReselected()`.

Figure 2 shows a harder case: `MwParseResult`, an 18-line file from a real pull request in *commons-app/apps-android-commons*. It compresses several of the recurring, genuinely hard translation decisions this dataset is meant to exercise into one file: nullability has to be inferred, since `private MwParseText text;` becomes the nullable `private val text: MwParseText? = null`, which forces the caller at `return text.text;` to become the safe call `return text?.text`; and the nested `public class MwParseText` has to become an explicit `inner class`, a distinction Java does not require and Kotlin does not infer.

Before (`.java`)

```java
package fr.free.nrw.commons.media;

import ...annotations.SerializedName;

public class MwParseResult {
  @SuppressWarnings private int pageid;
  @SuppressWarnings private int index;
  private MwParseText text;

  public String text() {
    return text.text;
  }


  public class MwParseText{
    @SerializedName("*") private String text;
  }
}
```

After (`.kt`)

```kotlin
package fr.free.nrw.commons.media

import ...annotations.SerializedName

class MwParseResult {
  private val pageid = 0
  private val index = 0
  private val text: MwParseText? = null

  fun text(): String? {
    return text?.text
  }

  inner class MwParseText {
    @SerializedName("*")
    internal val text: String? = null
  }
}
```

Fig. 2. A real migrated instance (`commons-app/apps-android-commons`, PR #6369, commit `8fc7e10`). Highlighted lines carry the migration's nullability inference, safe-call insertion, and `inner class` correction; unhighlighted lines are unchanged context.

TABLE I
CORPUS STATISTICS (JAVA-TO-KOTLIN, ALL TIME).

Statistic	Value
Date range	2013-09-04 to 2026-08-23
Files touched (total)	80,723
Lines changed (total, +/-)	17,880,366
Files edited per commit (mean / median)	3.4 / 1
Lines edited per commit (mean / median)	746.5 / 252

III. Dataset Construction

A. Data source and mining methodology

We start with a GitHub search for repositories with both Java and Kotlin source and at least 10 stars, a recognized open-source license, and no fork flag, which returns 6,224 candidates. Ground truth is mined from these repositories by scanning commit history for the signature of a file-level Java-to-Kotlin migration: a `.java` file deleted, or emptied, in the same commit that introduces a `.kt` file preserving a recognizable share of the original's structure, class name, and package. Candidate commits are checked against a set of migration-pattern heuristics (consistent renaming, import-preserving moves, matching test references) to reject unrelated file churn. PR resolution then consolidates each confirmed commit into a task instance at pull-request granularity, so every migration in the dataset traces back to a real, citable point in a real project's history rather than a synthesized example. The pipeline narrows 6,224 candidate repositories to 1,961 confirmed repositories and 23,951 confirmed commits, consolidated into 5,379 pull-request-level task instances.

B. Scale

As of September 2026, the dataset covers 1,961 repositories, 5,379 pull requests, 23,951 confirmed migration commits, and 80,723 touched files. Table I summarizes the corpus beyond these headline counts: most migrations are small and incremental, consistent with the file-by-file pattern Section I describes, but the long tail of larger commits is what pulls the mean well above the median.

Figure 3 plots monthly migration commits by the detection pattern our tool assigns them (Section III-D). Migration activity accelerates sharply around 2017, tracking Kotlin's 2019 designation as Android's preferred language, and a rename-then-modify pattern becomes prominent from 2021 onward as more teams migrate incrementally rather than rewriting a file in one commit.

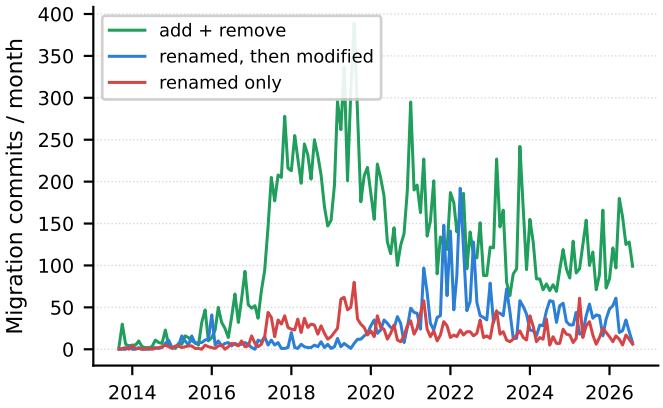

Fig. 3. Monthly migration commits by detected pattern: `add_remove` (a `.kt` file added and the matching `.java` file removed in the same commit), `renamed` (a pure rename with no content rewrite), and `renamed_then_modified` (a rename followed by a separate rewriting commit).

TABLE II
TASK INSTANCE SCHEMA.

Field	Description
`instance_id`	Unique id, `{org}__{repo}-pr-{number}`
`repo`	`owner/name` on GitHub
`pr`	Pull request number
`base_commit_sha`	Full SHA of the PR's pre-migration base commit
`head_commit_sha`	Full SHA of the PR's post-migration head commit
`migrated_files`	List of before/after file path pairs touched by the migration
`pre_migration_code`	Full source of each file before migration
`migration_patch`	Unified diff between pre- and post-migration source
`post_migration_code`	Full source of each file after migration (ground truth)
`commit_date`	Date the migration was committed
`num_files`	Number of files touched in this instance

Figure 4(a) shows cumulative growth: repositories and commits track a similar S-shaped curve, repositories leading slightly as they join before accumulating further migrations. Panel (b) confirms the file-by-file framing directly: 15,872 of 23,951 commits (66.3%) touch exactly one file.

C. Schema

PR resolution produces one task instance per pull request, a flat JSON object with the fields in Table II; the instance shown earlier in Figure 2 is one real example.

D. The J2KMigBench tool

The dataset is produced and kept current by an open-source command-line tool, not a one-off script: it re-runs the mining pipeline against new commit activity, resumes interrupted mining runs from persisted state, and assigns each confirmed commit one or more of the three detection patterns in Figure 3. Every field in Table II is produced by the tool, not curated by

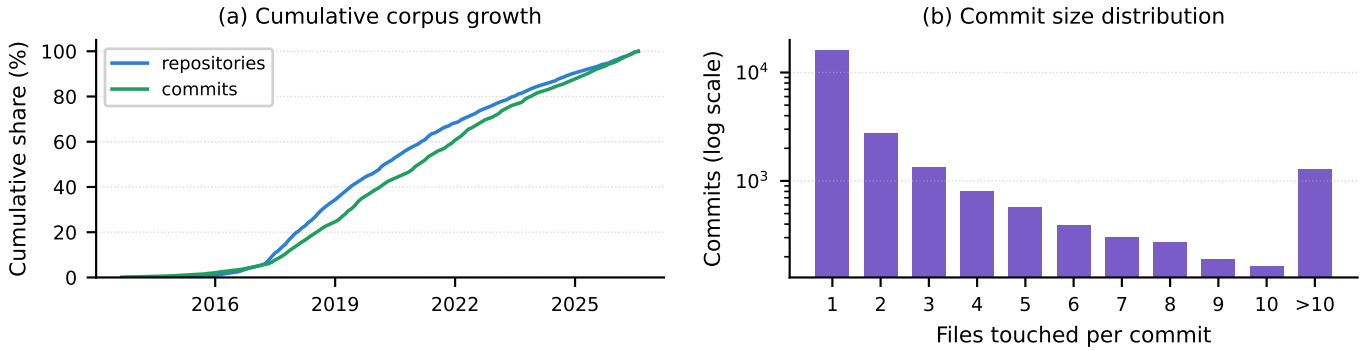

Fig. 4. (a) Cumulative share of repositories and commits over time, each normalized to its September 2026 total. (b) Distribution of files touched per commit (log scale), bucketed at each count from 1 to 10 with a single `>10` tail bucket.

hand, so a later run reproduces or extends the dataset without manual intervention.

E. Access

The full task-instance index is published as versioned JSON, browsable through a public explorer[1]. Records are filterable by repository, detected pattern, commit-year range, pull-request association, files touched, and lines changed; any combination of filters produces a shareable link to that exact view, and the filtered results export directly as JSONL for offline use. A dashboard view aggregates the corpus by detection pattern, by repository, and by year, matching Figures 3 and 4. Each task instance also has its own page showing the migrated files side by side, Java and Kotlin source syntax-highlighted, alongside its commit, branch, and pull-request metadata. The underlying data and tool are hosted in a public repository[2]. No account or credential is required to read either the index or the source.

IV. Use Cases

Evaluating migration agents. Each task instance's `base_commit_sha` identifies a real pre-migration repository state that a migration agent can be checked out and scored against, comparing its output to the real human-authored `post_migration_code` rather than a synthesized task.

Mining idiomatic-translation patterns. The full, 23,951-commit corpus supports corpus-level studies of how developers translate Java constructs and library usages into Kotlin idioms.

Training and fine-tuning data. Each task instance's `pre_migration_code` and `post_migration_code` are a directly usable (Java, human-Kotlin) exemplar pair for supervised fine-tuning or retrieval-augmented-generation approaches to migration-specific models.

Ecosystem-level migration studies. Prior work on why developers migrate [12] and the resulting cross-dependency defects [13] used a small, custom corpus each. J2KMigBench's 1,961-repository corpus lets the same questions be re-asked at a scale spanning single-contributor repositories to production migrations at companies like Duolingo and Meta [1], [3].

[1]https://j2kmigbench.github.io/

[2]https://github.com/j2kmigbench/dataset

V. Related Work

Several recent benchmarks target code migration, but none combines Java-to-Kotlin's cross-language scope with the same ground-truth provenance as J2KMigBench (Table III). MigrationBench [4] and JMigBench [5] both target Java version upgrades (8 to 17/21 and 8 to 11 respectively), not a cross-language migration, so neither carries the interface- and idiom-translation ground truth a Java-to-Kotlin migration requires. ScarfBench [6] targets cross-framework migration within enterprise Java, and MiG.4 [7] curates library migrations across Java and Python. CodeMenv [8] adapts Python or Java code to newer library versions rather than translating between languages. Outside migration, prior work has targeted real, curated ground truth for other software engineering problems: Defects4J [14] and BugsInPy [15] curate reproducible real bugs for controlled testing studies in Java and Python respectively. SWE-bench [16] extended that same real-issue grounding to full GitHub issue resolution with LLM agents, and MobileDev-Bench [17] applies the same principle to mobile issue resolution specifically. LintBench [18] holds real, developer-written test suites out as an oracle for LLM-generated Android Lint checks, and AutoComply [19] detects platform-specific violations in Android Auto apps. None targets code migration. J2KMigBench's originality is this specific combination: a cross-language pair with a shared runtime, human-authored (not LLM-synthesized) ground truth mined from real project history, and the open-source tool that produced it, so the dataset can be reproduced or extended rather than treated as a static, one-off release.

Beyond benchmarks, several empirical studies motivate this dataset without proposing one themselves. Martinez and Gois Mateus [12] study why developers migrate Android applications from Java to Kotlin. Feng et al. [13] characterize cross-dependency defects at the Kotlin-Java boundary, the same interface-level boundary J2KMigBench's ground truth targets. Mateus et al. [9] learn migration models from historical migrations on a custom, non-reusable corpus, the same limitation J2KMigBench's scale is meant to address.

TABLE III
J2KMIGBENCH RELATIVE TO DIRECTLY RELATED BENCHMARKS.

Dataset	Migration type	Size
MigrationBench [4]	Java 8 to 17/21 (same language)	300 repos
JMigBench [5]	Java 8 to 11 (same language)	150 functions
ScarfBench [6]	Cross-framework, same language (Java)	204 tasks
MiG.4 [7]	Library migration (Java, Python)	800 instances
PyMigBench [10]	Library migration (Python)	75 migrations
CodeMenv [8]	Library-version migration (Python, Java)	922 examples
J2KMigBench (ours)	**Java-to-Kotlin (cross-language)**	**5,379 instances**

VI. THREATS TO VALIDITY

Construct validity. The commit-mining heuristics favor precision over recall: they may miss migrations split across unrelated commits or heavily refactored beyond straightforward renaming, so the dataset undercounts true migration activity rather than overcounting it.

External validity. GitHub-hosted history mining inherits GitHub's own survivorship bias: repositories that migrated and then deleted their history, or migrations performed outside a public repository entirely, are structurally invisible to this methodology.

VII. CONCLUSION AND FUTURE WORK

J2KMigBench packages real, historical Java-to-Kotlin migrations, and the open-source tool that produces them, as a benchmark: 1,961 repositories and 5,379 pull-request-level task instances. Generalizing the tool to other cross-language, same-runtime migrations, and supporting the ecosystem-level and idiom-mining studies outlined above, are future work building directly on the pipeline this paper documents.

ACKNOWLEDGMENT

This material is based upon the work supported in part by the National Science Foundation (NSF) under Grant No. 2449694 and the Louisiana Board of Regents.

REFERENCES

[1] A. Chaidarun, "Migrating duolingo's android app to 100% kotlin," Duolingo Engineering Blog, 2020. [Online]. Available: https://blog.duolingo.com/migrating-duolingos-android-app-to-100-kotlin/

[2] Android Developers, "Duolingo completes migration to kotlin and reduces its line count by an average of 30%," Android Developer Stories, 2020. [Online]. Available: https://developer.android.com/stories/apps/duolingo-kotlin

[3] J. Luizzi, J. Yang, and E. Matthaey, "Translating java to kotlin at scale," Meta Engineering Blog, 2024. [Online]. Available: https://engineering.fb.com/2024/12/18/android/translating-java-to-kotlin-at-scale/

[4] L. Liu, X. Liu, Q. Zhou, L. Chen, Y. Liu, H. Nguyen, B. O. Tehrani, X. Shen, J. Huan, O. Tripp, and A. Deoras, "Migrationbench: Repository-level code migration benchmark from java 8," in *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, ser. KDD '26. New York, NY, USA: Association for Computing Machinery, 2026, pp. 9427–9438.

[5] N. Amin, Z. Fei, X. Li, J. Petke, and H. Ye, "Jmigbench: A benchmark for evaluating llms on source code migration (java 8 to java 11)," 1st International Workshop on Code Translation, Transformation, and Modernization, 2026, arXiv:2602.09930.

[6] A. Pavuluri, B. McGinn, A. Saxena, G. Safta, S. Tamilselvam, R. Pavuluri, M. Merler, B. Ray, and R. Krishna, "Scarfbench: A benchmark for cross-framework application migration in enterprise java," 2026, arXiv:2605.06754.

[7] M. Barbosa, P. Baptista, and J. E. Montandon, "Mig.4: A curated dataset of library migrations in java and python," in *Proceedings of the 2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering*, ser. FORGE '26. New York, NY, USA: Association for Computing Machinery, 2026, pp. 218–222.

[8] K. Cheng, X. Shen, Y. Yang, T. Wang, Y. Cao, M. A. Ali, H. Wang, L. Hu, and D. Wang, "Codemenv: Benchmarking large language models on code migration," in *Findings of the Association for Computational Linguistics: ACL 2025*, 2025, arXiv:2506.00894.

[9] B. G. Mateus, M. Martinez, and C. Kolski, "Learning migration models for supporting incremental language migrations of software applications," *Information and Software Technology*, vol. 153, p. 107082, 2023.

[10] M. Islam, A. K. Jha, S. Nadi, and I. Akhmetov, "Pymigbench: A benchmark for python library migration," in *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories (MSR 2023)*, 2023, pp. 511–515.

[11] K. Kahapola, S. Galappaththi, D. Ranasinghe, R. Shariffdeen, N. de Silva, S. Perera, and S. Wickramanayake, "Javabackports: A dataset for benchmarking automated backporting in java," in *Proceedings of the 23rd International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2026, pp. 712–716.

[12] M. Martinez and B. Gois Mateus, "Why did developers migrate android applications from java to kotlin?" *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4521–4534, 2022.

[13] Q. Feng, X. Ma, J. Sheng, H. Ji, and P. Liang, "An empirical study of kotlin-java cross-dependency issues and their detection," *IEEE Transactions on Software Engineering*, pp. 1–14, 2026, arXiv:2405.04602.

[14] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*, 2014, pp. 437–440.

[15] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020, pp. 1556–1560.

[16] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" 2024. [Online]. Available: https://arxiv.org/abs/2310.06770

[17] M. A. Fakorede, K. Upadhyay, A. B. Siddique, and U. Farooq, "Mobiledev-bench: A benchmark for issue resolution in mobile application development," 2026. [Online]. Available: https://arxiv.org/abs/2603.24946

[18] M. Fakorede, A. B. Siddique, M. Sridharan, and U. Farooq, "How far can llms go in generating android lint checks from natural language?" 2026, Add the arXiv eprint id/URL once uploaded.

[19] M. A. Fakorede and U. Farooq, "Understanding and detecting platform-specific violations in android auto apps," in *Proceedings of the 7th ACM/IEEE International Conference on Automation of Software Test*, ser. AST '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 123–133. [Online]. Available: https://doi.org/10.1145/3793654.3793745