MobileDev-Bench: A Benchmark for Issue Resolution in Mobile Application Development

Anonymous Author(s)
Affiliation
Address
email

Abstract

Large language models (LLMs) have shown strong performance on automated software engineering tasks, yet existing benchmarks focus primarily on library-style repositories, leaving mobile application development largely unexplored despite its framework-specific build systems, heterogeneous artifact types, and coordinated multi-file fix requirements. We introduce MobileDev-Bench, a benchmark comprising 415 real-world issue-resolution tasks collected from 19 production mobile applications spanning Android Native (Java/Kotlin), React Native (TypeScript), and Flutter (Dart). Each task pairs a verified developer-reported issue with executable test patches, enabling fully automated validation of model-generated fixes within mobile build environments. The benchmark exhibits substantially greater patch complexity than prior benchmarks: fixes modify 13.5 files and 344.2 lines on average, and 42.2% of instances require coordinated changes across multiple artifact types, such as source, build configuration, and resource files. Evaluation of four frontier LLMs (Claude Sonnet 4.5, Qwen3-Coder, GPT-5.2, and Gemini 2.5 Flash) yields end-to-end resolution rates of only 3.90%–4.28% under automated fault localization and at most 6.07% under oracle localization, well below resolution rates reported on existing benchmarks. We release MobileDev-Bench with task instances, an evaluation harness, and containerized environments to support reproducible research on AI-assisted mobile application development.

1 Introduction

Large language models (LLMs) such as GPT-4 [OpenAI, 2023], Claude [Anthropic, 2023], and Code Llama [Roziere et al., 2023] have demonstrated strong progress in automated software engineering, particularly in repository-level issue resolution. Benchmarks such as SWE-bench [Jimenez et al., 2024] establish a realistic evaluation paradigm by requiring models to resolve real-world GitHub issues through patch generation, where correctness is validated by executing the relevant test suite. Subsequent extensions [Yang et al., 2024b, Zhang et al., 2024a, Kabir, 2025, Mhatre et al., 2025, Zan et al., 2024, 2025] broaden coverage across programming languages and evaluation settings, revealing persistent performance gaps even for frontier code-capable LLMs.

Despite these advances, existing benchmarks primarily target issue resolution in general-purpose libraries or web applications. However, mobile application (app) development remains largely underexplored in benchmark design, despite introducing distinct engineering constraints. Mobile apps are built within platform-defined architectures with lifecycle-managed execution [Android, 2020]. They frequently employ declarative and event-driven programming [Flutter, 2026] and rely on asynchronous state management across user interface and service components. Compilation and testing are mediated by framework-specific build systems [Android, 2025], and correctness often depends on coordinated changes across heterogeneous artifacts, including source files, manifests, resource definitions, and configuration metadata [Android, 2024, Flutter, 2025]. These characteristics

Submitted to 40th Conference on Neural Information Processing Systems (NeurIPS 2026). Do not distribute.

Benchmarks	Domain	#Tasks	#Repos	Verified	Difficulty level	Multi-type artifact fixes	Patch # Files	Complex. # Lines
SWE-bench	Python libraries	2294	12	✗	✗	✗	1.7	32.8
SWE-bench Verified	Python libraries	500	12	✓	✗	✗	1.2	14.3
SWE-bench Multimodal	JavaScript libraries	617	17	✓	✗	✓	3.9	131.6
SWE-PolyBench	Multi-lang. libraries	2110	21	✗	✗	✗	2.6	51.2
SWE-bench Multilingual	Multi-lang. libraries	300	42	✗	✗	✗	1.7	47.8
OmniGIRL	Multi-lang. repos	959	15	✗	✗	✗	1.2	46.3
SWE-bench-java	Java libraries	91	6	✓	✗	✗	2.3	38.7
SWE-Sharp-bench	C# libraries	150	17	✓	✗	✗	4.9	131.2
Rust-SWE-bench	Rust repositories	500	34	✓	✗	✗	9.8	139.9
Multi-SWE-bench	Multi-lang. repos	1632	39	✓	✓	✗	4.9	163.3
MobileDev-Bench	**Mobile Apps**	**415**	**19**	✓	✓	✓	**13.5**	**344.2**

Table 1: Comparison of issue-resolution benchmarks. Unlike prior work that primarily evaluates source-code fixes in library-style repositories, MobileDev-Bench targets mobile apps, incorporates human verification and difficulty stratification, evaluates multi-artifact fixes (e.g., source and resources artifacts such as manifest files), and restores pre-fix buildability to enable execution-based evaluation.

make mobile app repositories structurally different from the library-centric settings that dominate existing issue-resolution benchmarks.

We introduce *MobileDev-Bench*, a benchmark of 415 manually verified issue-resolution tasks collected from 19 production mobile app repositories spanning Android Native (Java/Kotlin), React Native (TypeScript), and Flutter (Dart). Each instance pairs an authentic developer-reported issue with its merged pull request and executable test patches, enabling automated validation of model-generated fixes. To support reliable evaluation, MobileDev-Bench executes tests under three configurations: the base commit, the test patch only, and the full fix, capturing transitions such as Fail-to-Pass, None-to-Pass, Pass-to-Pass, and Pass-to-Fail. Unlike prior benchmarks that exclude instances with compilation failures, we explicitly model transitions from non-existent or non-executable test states and restore projects so that they compile when necessary. This compilation-aware design is particularly important for *statically typed languages* common in mobile app development, such as Java and Kotlin, where successful compilation is required before tests can execute.

Table 1 summarizes how MobileDev-Bench differs from prior issue-resolution benchmarks along four key dimensions. First, it targets production mobile apps drawn from mature and actively maintained open-source repositories, each with at least 400 GitHub stars. Second, it evaluates fixes that span multiple artifact types, including source code, resources, and configuration metadata. Third, it incorporates *manual verification* and difficulty stratification to ensure benchmark quality, with tasks distributed across Easy (139), Medium (140), and Hard (136) tiers. Finally, it exhibits substantially higher patch complexity: fixes modify 13.5 files and 344.2 lines on average, and 42.2% of instances require coordinated changes across multiple artifact types. These properties capture the multi-file and multi-artifact dependencies common in mobile app development that are largely absent from existing library-centric benchmarks. A detailed comparison of MobileDev-Bench against existing issue-resolution benchmarks, including patch complexity analysis and per-repository statistics, is provided in Appendix C.

We extend Agentless [Xia et al., 2024] to support Java, Kotlin, Dart, and TypeScript and evaluate four state-of-the-art LLMs on MobileDev-Bench. Despite strong performance on prior issue-resolution benchmarks, models achieve low resolution rates of 3.90%–4.28% (Agentless) and 1.94%–6.07% (Oracle) on MobileDev-Bench. Our analysis reveals recurring failure modes specific to mobile development, including misaligned resource updates, improper dependency configuration, and incomplete cross-artifact coordination. Across models, fault localization emerges as the dominant bottleneck, with file-level recall ranging from 13.5% to 18.3%.

This work makes the following contributions:

- We introduce MobileDev-Bench, a benchmark of 415 verified mobile app issue-resolution tasks drawn from 19 production repositories across Android, React Native, and Flutter.
- We develop a compilation-aware, execution-based evaluation framework for issue resolution across Java, Kotlin, Dart, and TypeScript mobile app projects and release MobileDev-Bench with task instances, an evaluation harness, and containerized environments at `https://huggingface.co/datasets/MobileDev-Bench/mobiledev-bench`.

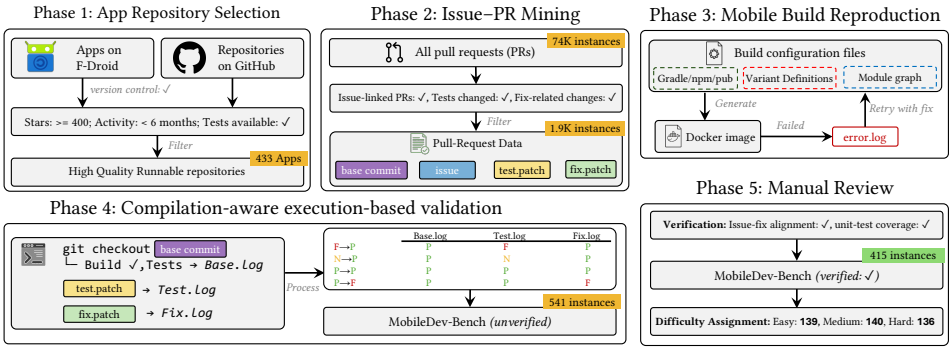

Figure 1: Overview of MobileDev-Bench construction pipeline.

- We provide a comprehensive evaluation of four frontier code-capable LLMs, showing that resolution rates drop to 3.90%–6.07% and that fault localization emerges as the dominant bottleneck for mobile app issue resolution.

2 Related Work

Early benchmarks such as HumanEval [Chen et al., 2021] and MBPP [Austin et al., 2021] established a foundation for assessing the programming capabilities of LLMs. These benchmarks primarily focused on small, self-contained problems: 95.8% target Python and over 83% evaluate function- or statement-level tasks [Cao et al., 2024], overlooking the real-world repository-level complexity.

SWE-bench [Jimenez et al., 2024] introduced repository-scale evaluation with 2,294 instances derived from real GitHub issues in Python projects, requiring models to generate patches validated through test execution. Subsequent work extended this direction to additional languages. Multi-SWE-bench [Zan et al., 2025] spans seven programming languages with 1,632 instances, while SWE-bench Java [Zan et al., 2024] provides 91 Java-specific tasks. SWE-PolyBench [Rashid et al., 2025] further shows that realistic repair tasks often require edits across multiple files. Despite this progress, existing benchmarks primarily focus on library-style repositories [Guo et al., 2025, Mhatre et al., 2025] and do not capture platform-specific app development environments. Alongside these benchmarks, LLM-based repair systems including SWE-agent [Yang et al., 2024a], AutoCodeRover [Zhang et al., 2024b], MAGIS [Tao et al., 2024], OpenDevin [Wang et al., 2024], CodeR [Chen et al., 2024], MASAI [Arora et al., 2024], and Agentless [Xia et al., 2024] have demonstrated strong results on these benchmarks, yet none target mobile application repositories.

Mobile app development *introduces additional challenges* not represented in current benchmarks, including framework-specific architectures, complex build systems, and coordination across heterogeneous artifacts such as source code, resources, and configuration files. Mobile app development spans Android, iOS, and cross-platform frameworks such as React Native [Facebook, 2015] and Flutter [Google, 2018]. Prior research in mobile apps has focused on static analysis and testing tools such as FlowDroid [Arzt et al., 2014], AutoComply [Fakorede and Farooq, 2025], and DroidBot [Li et al., 2017]. More recently, LLM-based approaches have targeted mobile UI task automation [Wen et al., 2024] and mobile agent evaluation [Deng et al., 2024], while benchmarks for evaluating AI-assisted *code issue resolution* in mobile repositories remain *largely unexplored.*

MobileDev-Bench addresses this gap by targeting production mobile app repositories with heterogeneous build environments and multi-artifact fixes, complementing existing library-centric benchmarks and broadening evaluation for AI-assisted software engineering systems.

3 MobileDev-Bench Construction

MobileDev-Bench is constructed from real issue-pull request (PR) pairs in open-source mobile app repositories. The construction process follows a five-phase pipeline as shown in Figure 1, designed to ensure that each instance corresponds to a real software engineering task and supports reliable execution-based validation.

3.1 Phase 1: App Selection

We begin by constructing a pool of candidate mobile app repositories spanning three major frameworks: Android Native (Java/Kotlin), React-Native (TypeScript/JavaScript), and Flutter (Dart). Repositories are collected from F-Droid [F-Droid, 2024], a catalog of free and open-source Android apps, and from GitHub searches filtered by mobile framework tags. We prioritize production apps distributed through app stores, as they typically represent mature codebases with active development and testing practices. This process yields an initial pool of 433 candidate repositories.

To improve quality and benchmark viability, we filter repositories based on community adoption ($\geq$400 GitHub stars), recent maintenance, an open-source license, a GitHub-hosted issue tracker, and verified buildability in a clean environment. The detailed filter criteria are provided in Appendix A.

3.2 Phase 2: PR Collection and Filtering

From the selected repositories, we retrieve merged pull requests (PR) using the GitHub API, collecting metadata such as issue descriptions, pull request text, commit history, and patch information. To ensure stable repository states, we only consider PRs merged into the repository's default branch.

To construct valid issue-resolution tasks, we apply the following filtering steps: *(i)* We retain only PRs that are explicitly linked to a GitHub issue using standard closure patterns (e.g., "fixes #123", "resolves #45"). This ensures that each task has a clear problem description derived from the issue report. *(ii)* PRs must modify or introduce test files so that the correctness of the fix can be verified through automated execution. *(iii)* For each retained PR, we extract two patches: a test patch containing the test changes and a fix patch containing the changes. These patches are anchored to the PR's base commit to enable controlled replay of the development history. Thus, each candidate instance contains four components: (a) the base commit, (b) the issue description, (c) the test patch, and (d) the fix patch. These filters yield a set of candidate issue–PR pairs for subsequent validation.

3.3 Phase 3: Environment Runtime Setup

Mobile app builds require precise environment configurations spanning language runtimes, framework SDKs, build tools, and dependencies. To support reproducibility, we automatically construct build environments by analyzing repository metadata and generating framework-specific Dockerfiles. For each candidate instance, we inspect the repository at the base commit and parse framework configuration files (`build.gradle/.kts` and `gradle/libs.versions.toml` for Android; `package.json` for React Native; `pubspec.yaml` for Flutter) to infer toolchains and dependencies. When configuration files are incomplete, we supplement them with README documentation and GitHub Actions workflow files. We then generate a per-repository Dockerfile. The environment remediation details are provided in Appendix A.

3.4 Phase 4: Execution-Based Validation

We validate candidate instances through execution-based filtering by running tests under three configurations within isolated Docker environments: *(i)* the base commit, *(ii)* the base commit with the test patch applied (`test.patch`), and *(iii)* the base commit with both test and fix patches applied (`fix.patch`). Rather than generic framework commands, we run instance-specific test commands derived from the PR's CI configuration, targeting only the tests relevant to each instance. Android instrumentation tests that require emulators are excluded due to reproducibility concerns.

For each execution, we record four states: `PASS`, `FAIL`, `NONE` (test absent in that configuration), and `SKIP` (see Appendix A for more). We derive test-state transitions and retain instances satisfying three criteria: *(i)* at least one test exhibits a *Fail-to-Pass (F2P)* or *None-to-Pass (N2P)* transition indicating the fix resolves the issue, *(ii)* no test exhibits a *Pass-to-Fail (P2F)* regression, and *(iii)* both the base and fix configurations build and execute tests successfully (we provide formal definitions in Appendix A). This execution-based filtering reduces the candidate pool from 1,939 to 541 instances.

3.5 Phase 5: Human Verification

Following SWE-bench Verified [OpenAI, 2024], we perform a manual review to ensure benchmark quality. Two annotators independently examine each of the 541 candidate instances that passed execution-based filtering and resolve disagreements through discussion. Review focuses on four

Table 2: Per-repository statistics for MobileDev-Bench. #Files and #LoC report codebase size at the base commit. # is the instance count. #Tok is the average issue description length in tokens. Fix patch columns (#L: changed lines, #H: hunks, #F: files) and test transition columns (F2P: Fail-to-Pass, N2P: None-to-Pass, P2P: Pass-to-Pass) report per-instance averages.

	Codebase		Instances		Fix Patch (Avg.)			Tests (Avg.)		
Repository	**#Files**	**#LoC**	**#**	**#Tok**	**#L**	**#H**	**#F**	**F2P**	**N2P**	**P2P**
Flutter (Dart)										
talawa	3,264	236k	12	280	256.6	18.0	5.9	1.0	4.2	3.8
zulip-flutter	514	141k	51	325	254.9	18.4	7.9	1.2	53.7	49.9
Android Native (Java/Kotlin)										
antennapod	1,143	115k	9	1,567	348.1	35.7	12.9	0.0	15.7	0.0
apps-android-commons	1,439	150k	10	548	1356.4	74.3	27.7	0.7	68.5	0.8
element-x-android	6,504	341k	72	154	387.1	41.8	18.7	1.9	31.5	22.8
medtimer	556	41k	9	162	472.0	32.1	16.6	0.4	4.7	0.6
geto	451	23k	1	156	172.0	33.0	5.0	0.0	52.0	0.0
jerboa	337	122k	4	218	12.8	1.8	1.2	4.0	11.8	1.0
neostumbler	627	39k	2	282	12.0	2.0	1.0	1.0	0.0	0.5
openhab-android	1,173	52k	8	228	584.1	13.5	8.1	0.6	10.9	5.6
voice	649	37k	4	259	273.0	10.0	7.5	0.3	4.8	0.5
streetcomplete	5,534	629k	18	411	625.4	52.4	27.1	0.6	68.3	7.1
thunderbird-android	6,389	365k	101	408	341.1	27.9	16.8	0.8	30.2	3.1
tusky	1,593	169k	13	464	234.8	11.2	5.4	1.3	17.9	7.2
wordpress-android	6,342	776k	84	159	174.9	17.4	7.2	0.3	28.9	0.8
React-Native (TypeScript)										
artsy/eigen	2,962	261k	1	287	2311.0	116.0	18.0	3.0	0.0	0.0
expensify/app	7,923	2,652k	2	547	429.0	42.0	17.5	1.5	1.0	2.0
NMF-earth/nmf-app	900	21k	12	143	439.8	40.3	17.3	3.9	0.2	0.4
rocket.chat.reactnative	1,601	134k	2	442	24.5	9.0	5.0	3.5	3.5	0.5

aspects: *(i)* Problem Statement Clarity to ensure that the GitHub issue provides sufficient context to understand the task; *(ii)* Test Coverage Appropriateness by verifying that the tests correctly correspond to the issue being resolved; *(iii)* Task Difficulty by labeling instances as Easy, Medium, or Hard based on conceptual complexity, number of files modified, and required framework knowledge; and *(iv)* Task Category by classifying each instance into one of the task types: Bug Fix, New Feature, or Feature Optimization.

Instances with severely underspecified issues or misaligned tests are excluded. In total, 126 instances (23.3%) fail manual verification, yielding a final benchmark of 415 instances. Difficulty and category labels are retained as metadata for analyzing model performance across task complexity levels and task types. More details are provided in Appendix B.

4 Benchmark Characteristics

4.1 Benchmark Statistics

MobileDev-Bench comprises 415 manually verified instances drawn from 19 production mobile repositories spanning three ecosystems: Flutter (Dart), Android Native (Java/Kotlin), and React-Native (TypeScript). Per-repository statistics are reported in Table 2. Repository sizes range from 337 to 7,923 source files (332K lines on average), reflecting the diversity of mature open-source codebases included in the benchmark. Instance counts vary from 1 to 101 per repository, with the four largest contributors being thunderbird-android (101), wordpress-android (84), element-x-android (72), and zulip-flutter (51), together accounting for 74.2% of the dataset.

Table 3 reports aggregate statistics across the dataset. Issue descriptions average 310 tokens (median 173), with a long tail reaching 11,501 tokens for the most detailed reports, reflecting the context-rich nature of app bug reports that typically include reproduction steps, platform versions, and device-specific conditions. Fix patches are larger than those in comparable benchmarks as shown in Figure 3: the mean patch modifies 344 lines across 13.5 files and 28.4 hunks, while the median is 97 lines, 5 files, and 12 hunks. The gap between mean and median reflects a right-skewed distribution, with outlier fixes in repositories such as apps-android-commons (avg. 1,356 lines) and openhab-android (avg. 584 lines) substantially inflating the mean. None-to-Pass is the dominant validation signal: 355

Table 3: Summary statistics for MobileDev-Bench.

	Issue	Codebase		Fix Patch			Tests		
	Tokens	#Files	#Lines (K)	#Lines	#Files	#Hunks	F2P	N2P	P2P
Mean	310	4,601	386	344	13.5	28.4	1.1	31.5	11.8
Median	173	6,342	341	97	5	12	0	11	0
Max	11,501	7,923	2,652	11,915	237	592	30	375	267

of 415 instances (85.5%) yield at least one N2P transition, with a mean of 31.5 and a median of 11. N2P transitions arise from two sources: tests that did not exist at the base commit and are introduced by the fix, and tests that were present but could not be executed due to compilation failures at the base commit, a common occurrence in statically typed languages such as Java and Kotlin where the fix itself is required to restore compilability. Additionally, 148 instances (35.7%) yield at least one Fail-to-Pass transition (mean 1.1), indicating cases where pre-existing tests directly captured the regression. The high N2P mean is driven by repositories with large test suites (e.g., zulip-flutter avg. 53.7 N2P per instance), where the fix enables previously non-executable test modules or introduces newly added tests.

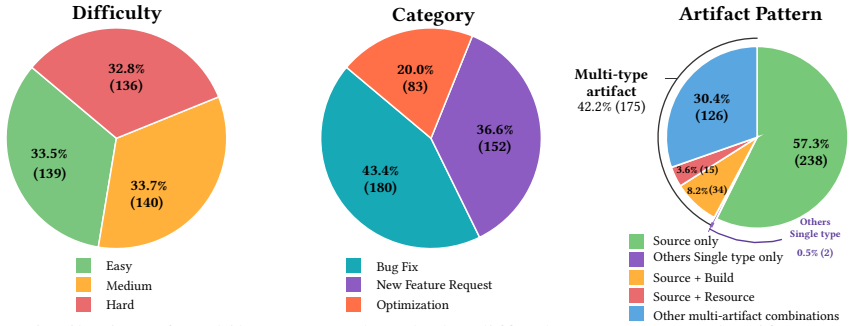

Figure 2: Distribution of MobileDev-Bench tasks by difficulty, category, and artifact pattern. Tasks are balanced across difficulty tiers, with bug fixes as the largest category. Most tasks modify source files only, while 42.2% require coordinated changes across multiple artifact types.

Figure 2 shows the distribution of instances in MobileDev-Bench across difficulty and task category. Difficulty is broadly balanced across Easy (under 15 min, 33.5%), Medium (15 min to 1 hr, 33.7%), and Hard (over 1 hr, 32.8%), ensuring the benchmark exercises a range of problem complexities. By task category, 43.4% of instances are Bug Fixes, 36.6% are New Features, and 20.0% are Feature Optimizations.

4.2 Benchmark Features

MobileDev-Bench captures characteristics of mobile app development that are absent from existing repository-level benchmarks. In particular, several fixes span multiple artifact types and may involve more than one language within a single patch.

Multi-Artifact Fix Patterns. Mobile app fixes often require coordinated changes across heterogeneous artifact types. We classify modified files in each patch into eight categories: (1) Source (`.kt`, `.java`, `.dart`, `.ts`, `.tsx`, `.js`); (2) Resource (`res/` subtree, image and media assets); (3) i18n (`values-*/strings.xml`, Flutter `.arb`); (4) Build (`build.gradle`, `libs.versions.toml`, `*.kts`); (5) Manifest (`AndroidManifest.xml`, `pubspec.yaml`); (6) Config (CI/CD workflows, linter config files); (7) Docs (Markdown, changelog fragments, license files); and (8) Other (files not matched by the above). Source files appear in nearly all instances (99.5%). However, many fixes also require non-source changes: 21.2% modify resources, 16.9% modify localization files, 19.3% modify build files, and 3.6% modify manifest files. As summarized in Figure 2, 42.2% of instances modify two or more artifact types and 21.9% modify three or more, demonstrating that cross-artifact coordination is a common property of mobile app fixes. Full combination-level counts with per-language breakdowns are reported in Table 9 in the appendix.

Intra-Patch Language Diversity. Fix patches also frequently span multiple languages. We identify cross-language instances by examining the set of source-language file extensions modified within each fix patch. In total, 20.7% of instances modify source files written in more than one language. The most common pattern is Kotlin–Java co-modification, representing 19.8% of all instances and

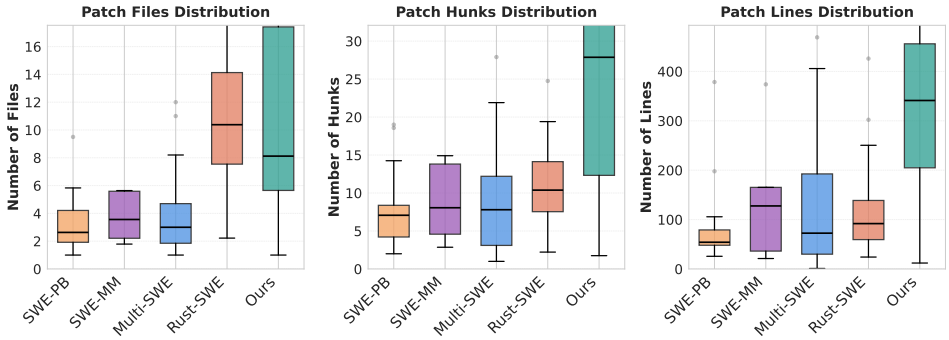

Figure 3: Distribution of patch complexity metrics (files modified, hunks, changed lines) across five benchmarks (SWE-PB: SWE-PolyBench; SWE-MM: SWE-bench Multimodal; Multi-SWE: Multi-SWE-bench; Rust-SWE: Rust-SWE-bench; Ours: MobileDev-Bench). Each data point is the per-repository average. Y-axes are capped at the 90th percentile for visual clarity.

25.2% of Kotlin-based fixes. This reflects the ongoing migration of Android projects from Java to Kotlin, with co-modification rates varying widely across repositories depending on their stage of migration: projects retaining residual Java code exhibit rates above 40%, while repositories that have fully transitioned to Kotlin show none. A second cross-language pattern occurs in React-Native repositories, where TypeScript–JavaScript co-modification appears in 18.8% of React-Native instances, reflecting gradual TypeScript adoption in existing JavaScript codebases.

Table 4: Resolution rates (%) by difficulty level under Agentless and Oracle settings.

		Agentless				Oracle			
Difficulty	**Tasks**	**Claude**	**Qwen**	**GPT**	**Gemini**	**Claude**	**Qwen**	**GPT**	**Gemini**
Easy	139	8.0	8.5	8.0	8.0	12.9	6.7	11.6	4.3
Medium	140	4.4	1.5	1.5	1.5	3.6	0.7	2.2	1.4
Hard	136	0.0	3.0	2.2	3.0	1.5	0.0	1.5	0.0
Overall	415	4.15	4.28	3.90	4.16	6.07	2.52	5.19	1.94

5 Experimental Setup

We evaluate models under two complementary settings designed to isolate the two stages of automated repair [Xia and Zhang, 2023]: fault localization and patch generation.

Agentless. We adopt Agentless [Xia et al., 2024] as our primary evaluation framework, selecting it as a competitive baseline whose modular architecture supports extension to new programming languages. We extend it to support Java, Kotlin, TypeScript, and Dart by replacing its Python-specific AST parser with tree-sitter [Brunsfeld et al., 2018] (see Appendix E.2 for details).

Oracle. We additionally evaluate an *oracle localization* setting [Jimenez et al., 2024], in which the ground-truth set of modified files is provided directly to the model, simulating a scenario where fault localization is perfect. The model receives the issue description and the exact files it must modify, and is asked solely to generate the patch. This setting serves two purposes. First, it establishes an upper bound on repair performance achievable under ideal localization conditions. Second, by comparing Oracle and Agentless resolution rates, we can attribute the performance gap to localization failure versus repair failure, isolating which component is the primary bottleneck.

In both settings, generated patches are applied to the base commit and validated by executing the repository test suite within our Docker-based [Merkel, 2014] containerized environment.

Models. We evaluate four frontier code-capable models spanning proprietary and open-weight systems: Claude Sonnet 4.5 [Anthropic, 2025], GPT-5.2 [OpenAI, 2025], Gemini 2.5 Flash [Comanici et al., 2025], and Qwen-3-Coder [Qwen Team, 2025]. To ensure deterministic patch generation and fair comparison, we disable extended reasoning modes where available, following the original Agentless evaluation protocol.

Metrics. Following prior work [Jimenez et al., 2024, Xia et al., 2024], we report two complementary metrics evaluated under both settings. *(i) Resolution Rate*: the percentage of tasks where the generated

Table 5: Resolution rate breakdown by artifact diversity. Resolved column shows the share of uniquely resolved instances in each category; Ratio = Resolved% / Benchmark%.

		Agentless		Oracle	
Category	**Bench.**	**Resolved**	**Ratio**	**Resolved**	**Ratio**
Single-type artifact	57.8%	84.4%	1.46x	86.8%	1.50x
Multi-type artifact	42.2%	15.6%	0.37x	13.2%	0.31x

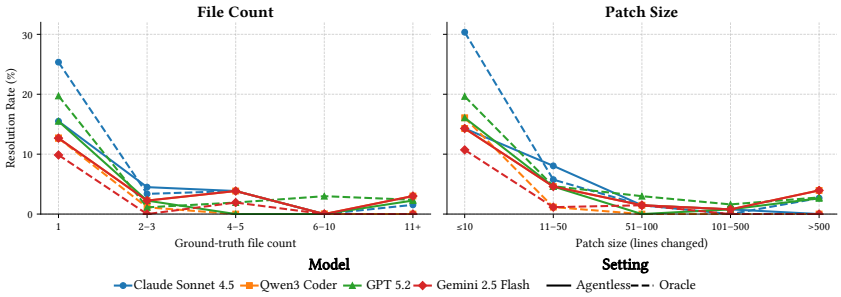

Figure 4: Resolution rates by ground-truth file count (left) and patch size (right). Single-file tasks achieve 13–16% (Agentless) and 10–25% (Oracle); rates collapse for multi-file and large-patch tasks.

patch passes the full test suite, measuring end-to-end repair success. *(ii) Localization Metrics*: file-level Precision, Recall, and F1 (macro-averaged across tasks), measuring how accurately the model's patch covers the ground-truth modified files. In the Agentless setting these metrics reflect the quality of automated fault localization; in the Oracle setting they reflect how completely the model generates edits across all files it was given. Comparing the two settings reveals whether the dominant failure mode is localization (finding the right files) or repair (generating correct changes once files are known). Formal metric definitions are in Appendix E.

6 Results

6.1 Repair Performance

Table 4 reports resolution rates across difficulty levels under both settings. Performance declines with task difficulty in both cases. Under Agentless, Easy tasks reach 8.0–8.5%, Medium tasks drop to 1.5–4.4%, and Hard tasks are rarely solved (0.0–3.0%). Under Oracle, Easy task rates rise to 4.3–12.9%, confirming that localization is a key bottleneck on simpler tasks, while Hard tasks remain difficult (0.0–1.5%) even with ground-truth files. More details are provided in Appendix F.

6.2 Resolution Characteristics

Table 5 analyzes how resolved instances differ from the overall benchmark distribution across artifact diversity. The comparison highlights structural factors that influence repair success.

Artifact Type Diversity. Resolution success is strongly associated with artifact diversity. While single-artifact tasks constitute 57.8% of the benchmark, they account for 84.4% of resolved Agentless instances, representing a 1.46x over-representation. In contrast, multi-artifact tasks require coordinated modifications across heterogeneous files and account for 42.2% of the benchmark but only 15.6% of resolved instances (0.37x). This gap indicates that current models struggle to coordinate fixes across multiple artifact types, such as source code, build configuration, and resource files.

File Count Effects. File count shows the strongest relationship with repair success. Single-file tasks, which represent only 17.1% of the benchmark, account for 65.6% of all resolved Agentless instances, a 3.84x overrepresentation. As the number of required file modifications increases, resolution success drops sharply. Tasks requiring 4–10 files are particularly challenging, accounting for 29.4% of benchmark tasks but only 6.3% of resolved instances. Tasks with 11+ files remain difficult but exhibit a modest recovery in resolution share (12.5%), a pattern further analyzed in Appendix F.3.

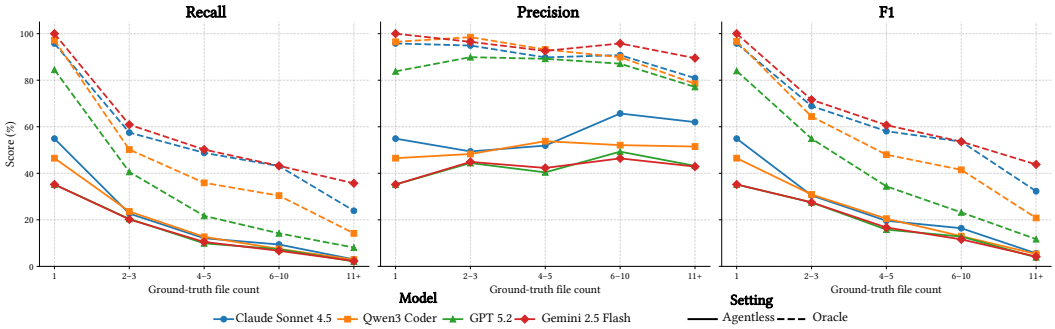

Figure 5: Recall decreases as the number of modified files increases.

6.3 Fault Localization Performance

Figure 5 shows file-level recall as a function of the number of ground-truth files under the Agentless setting. Across models, overall recall ranges from 13.5% to 18.3%, indicating that models frequently miss files that must be modified for a correct fix. In contrast, precision remains higher (42–58%, Appendix Table 12), suggesting that when models choose to edit a file, it is often relevant, but many required files are not identified.

Recall decreases as task complexity increases. Single-file tasks achieve the highest recall (35–55%), but recall drops to 20–24% for tasks with 2–3 files, 10–13% for tasks with 4–5 files, and 7–9% for tasks with 6–10 files. For the most complex tasks requiring 11+ files, recall falls to $\sim$ 2–3% across all models. This trend reflects the structural characteristics of mobile apps. Issues frequently require coordinated changes across multiple layers – source code, resources, and configuration artifacts. Current models rarely capture these cross-file dependencies, leading to missed modification sites.

Fault localization is the dominant bottleneck. Fault localization is widely recognized as a key determinant of automated repair success [Wong et al., 2016, Lou et al., 2020]. The low recall in the Agentless setting imposes a hard ceiling on resolution: even perfect patch generation could achieve at most 13–18% resolution rate (matching recall). Observed rates (3.90%–4.28% under Agentless) fall well below this ceiling, confirming that unresolved tasks fail first at localization.

The Oracle setting (Table 12) isolates the repair stage by providing ground-truth files directly. Even under these ideal conditions, recall reaches only 30.9–55.2% overall—models receive the exact files to modify but still generate patches that omit changes to many of them. Recall in the Oracle setting degrades similarly with file count: from 85–100% on single-file tasks to 8–36% on tasks with 11+ files. This repair-side failure is compounded on top of localization failure in the Agentless setting, where models must both discover the right files and modify all of them correctly.

Comparing the two settings quantifies each failure mode. The large gap between Oracle and Agentless recall (roughly 17–42 percentage points overall) confirms that fault localization is the dominant bottleneck. However, the Oracle recall falling well below 100% on multi-file tasks reveals a secondary challenge: even given perfect file knowledge, models struggle to generate coherent edits across many files simultaneously. Improving MobileDev-Bench resolution rates therefore requires advances on both fronts: dependency-aware multi-file retrieval for localization, and long-range patch coherence for repair. Comprehensive resolution rate breakdowns, per-model analysis, and additional resolution characteristics are provided in Appendix F.

7 Conclusion

We introduced MobileDev-Bench, a benchmark for evaluating LLM-based program repair on mobile app development tasks. It contains 415 verified instances from 19 production repositories across Android Native, React Native, and Flutter, capturing several properties of mobile development such as multi-artifact coordination and framework-driven execution. Evaluation of four frontier LLMs shows low resolution rates, with fault localization as the primary bottleneck. File-level recall remains limited and declines further for complex tasks, particularly multi-file and multi-artifact fixes. These results highlight the need for a better understanding of mobile app frameworks to support multi-file dependency reasoning and artifact-aware localization. MobileDev-Bench provides a reproducible benchmark and baseline results for future research in AI-assisted mobile app development.

References

Android. The activity lifecycle. 2020. URL https://developer.android.com/guide/components/activities/activity-lifecycle. Accessed 2026-02-12.

Android. App resources overview. 2024. URL https://developer.android.com/guide/topics/resources/providing-resources. Accessed 2026-02-12.

Android. Configure your build, 2025. URL https://developer.android.com/build. Accessed 2026-02-12.

Anthropic. Claude: A large language model by anthropic. 2023. URL https://www.anthropic.com.

Anthropic. Claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, 2025. Accessed: 2026-02-17.

Apple Inc. Xcode and Apple SDKs Agreement, 2024. URL https://www.apple.com/legal/sla/docs/xcode.pdf. Accessed 2026-02-12.

D. Arora, A. Sonwane, N. Wadhwa, A. Mehrotra, S. Utpala, R. Bairi, A. Kanade, and N. Natarajan. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638*, 2024.

S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2014.

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

M. Brunsfeld et al. tree-sitter: An incremental parsing system for programming tools. https://tree-sitter.github.io/tree-sitter/, 2018. Accessed: 2026-02-17.

J. Cao, Z. Chen, J. Wu, S.-C. Cheung, and C. Xu. Javabench: A benchmark of object-oriented code generation for evaluating large language models. *arXiv preprint arXiv:2406.12902*, 2024.

D. Chen, S. Lin, M. Zeng, D. Zan, J.-G. Wang, A. Cheshkov, J. Sun, H. Yu, G. Dong, A. Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024.

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.

G. Comanici, E. Bieber, M. Schaekermann, I. Pasupat, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. URL https://arxiv.org/abs/2507.06261.

S. Deng, W. Xu, H. Sun, W. Liu, T. Tan, J. Liu, A. Li, J. Luan, B. Wang, R. Yan, and S. Shang. Mobile-bench: An evaluation benchmark for llm-based mobile agents, 2024. URL https://arxiv.org/abs/2407.00993.

F-Droid. F-Droid. https://f-droid.org, 2024. Free and Open Source Android App Repository.

Facebook. React native, 2015. URL https://reactnative.dev.

M. Fakorede and U. Farooq. Understanding and detecting platform-specific violations in android auto apps, 2025. URL https://arxiv.org/abs/2503.04003.

Flutter. Start thinking declaratively, 2025. URL https://docs.flutter.dev/data-and-backend/state-mgmt/declarative. Accessed 2026-02-12.

Flutter. Introduction to declarative ui, 2026. URL https://docs.flutter.dev/flutter-for/declarative. Accessed 2026-02-12.

Google. Flutter, 2018. URL https://flutter.dev.

L. Guo, W. Tao, R. Jiang, Y. Wang, J. Chen, X. Liu, Y. Ma, M. Mao, H. Zhang, and Z. Zheng. Omnigirl: A multilingual and multimodal benchmark for github issue resolution, 2025. URL https://arxiv.org/abs/2505.04606.

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. URL https://arxiv.org/abs/2310.06770.

K. Kabir. Swe-bench multilingual, 2025. URL https://kabirk.com/multilingual.

Y. Li, Z. Yang, Y. Guo, and X. Chen. Droidbot: A lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017.

Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 75–87, 2020.

D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

S. Mhatre, Y. Bajpai, S. Gulwani, E. Murphy-Hill, and G. Soares. Swe-sharp-bench: A reproducible benchmark for c# software engineering tasks, 2025. URL https://arxiv.org/abs/2511.02352.

OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

OpenAI. Swe-bench verified. 2024. URL https://openai.com/swe-bench-verified.

OpenAI. Introducing GPT-5.2. https://openai.com/index/introducing-gpt-5-2/, 2025. Accessed: 2026-02-17.

Qwen Team. Qwen3-coder: A new era for code intelligence. https://qwen.ai/blog?id=qwen3-coder, 2025. Accessed: 2026-02-17.

M. S. Rashid, C. Bock, Y. Zhuang, A. Buchholz, T. Esler, S. Valentin, L. Franceschi, M. Wistuba, P. T. Sivaprasad, W. J. Kim, A. Deoras, G. Zappella, and L. Callot. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents, 2025. URL https://arxiv.org/abs/2504.08703.

B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng. MAGIS: LLM-based multi-agent framework for github issue resolution. *arXiv preprint arXiv:2403.17927*, 2024.

X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, et al. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024.

W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

C. S. Xia and L. Zhang. Automated program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179*, 2023.

C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

J. Xiang, W. He, X. Wang, H. Tian, and Y. Zhang. Evaluating and improving automated repository-level rust issue resolution with llm-based agents, 2026. URL https://arxiv.org/abs/2602.22764. ICSE 2026.

J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024a. URL https://arxiv.org/abs/2405.15793.

J. Yang, C. E. Jimenez, A. L. Zhang, K. Lieret, J. Yang, X. Wu, O. Press, N. Muennighoff, G. Synnaeve, K. R. Narasimhan, D. Yang, S. I. Wang, and O. Press. SWE-bench Multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024b.

D. Zan, Z. Huang, A. Yu, S. Lin, Y. Shi, W. Liu, D. Chen, Z. Qi, H. Yu, L. Yu, D. Ran, M. Zeng, B. Shen, P. Bian, G. Liang, B. Guan, P. Huang, T. Xie, Y. Wang, and Q. Wang. Swe-bench-java: A github issue resolving benchmark for java, 2024. URL https://arxiv.org/abs/2408.14354.

D. Zan, Z. Huang, W. Liu, H. Chen, L. Zhang, S. Xin, L. Chen, Q. Liu, X. Zhong, A. Li, S. Liu, Y. Xiao, L. Chen, Y. Zhang, J. Su, T. Liu, R. Long, K. Shen, and L. Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

L. Zhang, D. Zan, Q. Yang, Z. Huang, D. Chen, B. Shen, T. Liu, Y. Gong, P. Huang, X. Lu, G. Liang, L. Cui, and Q. Wang. Codev: Issue resolving with visual data, 2024a. URL https://arxiv.org/abs/2412.17315.

Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024b.

Table 6: Summary and licenses for all GitHub repositories represented in MobileDev-Bench.

Repository	Summary	License
palisadoesfoundation/talawa	Community organization management	GNU GPLv3
zulip/zulip-flutter	Zulip mobile apps for Android and iOS	Apache 2.0
antennapod/antennapod	Podcast manager for Android	GNU GPLv3
commons-app/apps-android-commons	Wikimedia Commons Android uploader	Apache 2.0
element-hq/element-x-android	Android Matrix messenger (Jetpack Compose)	GNU AGPLv3
futsch1/medtimer	Medication reminder app	MIT
jackeblan/geto	Per-app device settings manager	GNU GPLv3
lemmynet/jerboa	Native Android client for Lemmy	GNU AGPLv3
mjaakko/neostumbler	OpenStreetMap geolocation stumbler	MIT
openhab/openhab-android	openHAB client for Android	EPL 2.0
paulwoitaschek/voice	Minimalist audiobook player	GNU GPLv3
streetcomplete/streetcomplete	Easy-to-use OpenStreetMap editor	GNU GPLv3
thunderbird/thunderbird-android	Thunderbird for Android (fka K-9 Mail)	Apache 2.0
tuskyapp/tusky	Android client for Mastodon	GNU GPLv3
wordpress-mobile/wordpress-android	WordPress for Android	GNU GPLv2
artsy/eigen	Artsy's iOS/Android art marketplace	MIT
expensify/app	Financial collaboration platform	MIT
NMF-earth/nmf-app	Carbon footprint tracker for daily life	GNU GPLv3
rocketchat/rocket.chat.reactnative	Secure communications platform	MIT

A Additional Construction Details

This appendix provides supplementary details for the construction pipeline described in Section 3.

A.1 Repository Selection Filters

The following five filters are applied during Phase 1 (App Selection) to determine benchmark viability:

(i) A minimum of 400 GitHub stars to ensure community adoption and sustained development.

(ii) Recent maintenance within the past six months, verified through commit activity, merged pull requests, and responsive issue management.

(iii) A permissive open-source license (e.g., Apache 2.0, MIT, or GPL variants) allowing research use.

(iv) An issue tracker hosted on GitHub, enabling reliable retrieval of issue–PR mappings and contextual information.

(v) The latest commit must build and execute its test suite in a clean environment. For Android Native apps, we install the required JDK version (11/17/21) and run Gradle build and test tasks. For React Native apps, we verify Node.js compatibility and run tests via npm or yarn. For Flutter apps, we confirm Flutter SDK compatibility and execute the test framework. Repositories that fail this check are excluded.

A.2 Test File Identification Patterns

During Phase 2 (PR Collection and Filtering), test files are identified using the following framework-specific naming conventions:

- **Android Native:** `*Test.kt`, `*Test.java`, or files under `androidTest/` and `test/` source sets.
- **React Native:** `*.test.ts`, `*.test.tsx`, `*.test.js`, or `*.spec.ts` files.
- **Flutter:** `*_test.dart` files or any file under the `test/` directory.

A.3 Repository Composition

MobileDev-Bench draws from 19 open-source Android and cross-platform mobile repositories spanning utility apps, media players, productivity tools, and developer tooling. Table 6 lists all 19 repositories with a brief description of their purpose and their open-source license. The selection prioritises projects with active CI pipelines, high test coverage, and sustained PR activity to ensure a sufficient supply of execution-validated instances.

A.4 Pipeline Conversion Statistics

Table 7 reports per-repository counts at each stage of the conversion pipeline: PRs crawled, attribute-filtered candidates, execution-validated instances, and manually verified benchmark instances. Overall, the pipeline reduces 74,195 crawled PRs to 415 verified instances (0.6% retention rate). The primary bottleneck is attribute filtering (Phase 1), which discards PRs that lack an explicit issue link, do not modify test files, or touch more files than the complexity threshold permits. Execution validation (Phase 4) constitutes the second largest reduction, as a substantial fraction of candidates fail to build or produce non-deterministic test outcomes.

Table 7: Benchmark repository instance conversion pipeline showing progression from collected pull requests through attribute-based filtering, execution-based validation, and manual verification.

	PRs Crawled	**Filtered**	**Validated**	**Verified**
palisadoesfoundation/talawa	1,743	$212_{\downarrow 1531}$	$13_{\downarrow 199}$	$12_{\downarrow 1}$
zulip/zulip-flutter	1,051	$96_{\downarrow 955}$	$51_{\downarrow 45}$	$51_{.0}$
antennapod/antennapod	2,031	$89_{\downarrow 1942}$	$11_{\downarrow 78}$	$9_{\downarrow 2}$
commons-app/apps-android-commons	1,477	$200_{\downarrow 1277}$	$17_{\downarrow 183}$	$10_{\downarrow 7}$
element-hq/element-x-android	3,899	$303_{\downarrow 3596}$	$77_{\downarrow 226}$	$72_{\downarrow 5}$
futsch1/medtimer	643	$25_{\downarrow 618}$	$12_{\downarrow 13}$	$9_{\downarrow 3}$
jackeblan/geto	195	$20_{\downarrow 175}$	$6_{\downarrow 14}$	$1_{\downarrow 5}$
lemmynet/jerboa	1,048	$15_{\downarrow 1033}$	$5_{\downarrow 10}$	$4_{\downarrow 1}$
mjaakko/neostumbler	681	$20_{\downarrow 661}$	$5_{\downarrow 15}$	$2_{\downarrow 3}$
openhab/openhab-android	1,607	$27_{\downarrow 1580}$	$12_{\downarrow 15}$	$8_{\downarrow 4}$
paulwoitaschek/voice	1,364	$10_{\downarrow 1354}$	$6_{\downarrow 4}$	$4_{\downarrow 2}$
streetcomplete/streetcomplete	1,167	$96_{\downarrow 1071}$	$27_{\downarrow 69}$	$18_{\downarrow 9}$
thunderbird/thunderbird-android	2,728	$245_{\downarrow 2483}$	$110_{\downarrow 135}$	$101_{\downarrow 9}$
tuskyapp/tusky	2,110	$28_{\downarrow 2082}$	$19_{\downarrow 9}$	$13_{\downarrow 6}$
wordpress-mobile/wordpress-android	4,199	$438_{\downarrow 3761}$	$140_{\downarrow 298}$	$84_{\downarrow 56}$
artsy/eigen	10,020	$3_{\downarrow 10017}$	$2_{\downarrow 1}$	$1_{\downarrow 1}$
expensify/app	34,301	$30_{\downarrow 34271}$	$8_{\downarrow 22}$	$2_{\downarrow 6}$
NMF-earth/nmf-app	181	$23_{\downarrow 158}$	$15_{\downarrow 8}$	$12_{\downarrow 3}$
rocketchat/rocket.chat.reactnative	3,750	$59_{\downarrow 3691}$	$5_{\downarrow 54}$	$2_{\downarrow 3}$
Total	74,195	$1{,}939_{\downarrow 72256}$	$541_{\downarrow 1398}$	$415_{\downarrow 126}$

A.5 Environment Dockerfile Generation and Remediation

After generating a candidate Dockerfile from parsed configuration (as described in Phase 3, Section 3), we run a full build-and-test cycle to validate correctness. Failed builds are manually inspected; common fixes include adding missing system packages (e.g., `libssl-dev` for React Native native modules), pinning a breaking transitive dependency, or correcting a misidentified JDK version. Repositories requiring more than three remediation rounds are excluded.

For Android projects, GitHub Actions workflow files (`.github/workflows/*.yml`) often provide the most precise environment specification, pinning exact Gradle wrapper versions, JDK versions, and Android SDK command-line tool versions that are absent from `build.gradle` or `gradle.properties`.

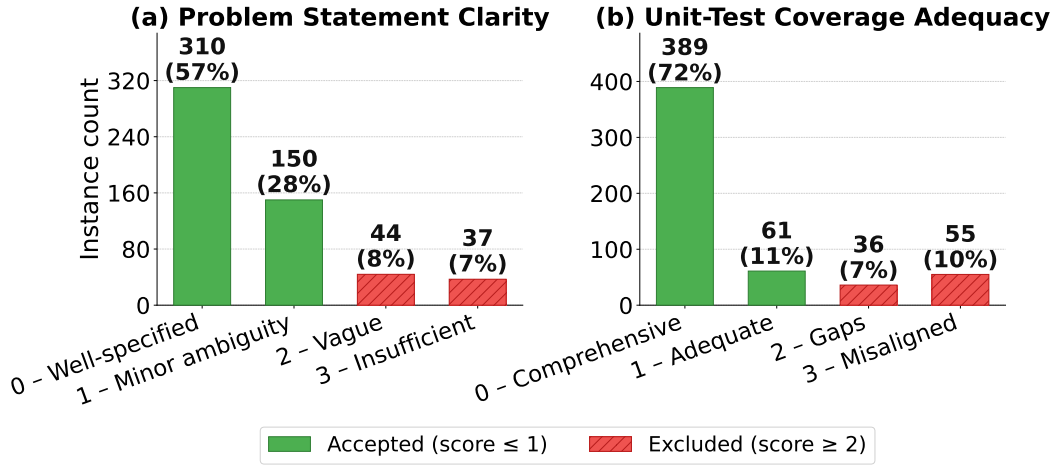

Figure 6: Distribution of manual annotation scores for **(a)** problem-statement clarity and **(b)** unit-test coverage adequacy, across all 541 annotated candidate instances. Green bars (scores 0–1) indicate instances accepted into the benchmark; hatched red bars (scores 2–3) indicate excluded instances. The majority of exclusions arise from vague issue descriptions (score 2/3 on dimension a) or tests that impose undue implementation constraints (score 2/3 on dimension b).

A.6 `NONE` State and None-to-Pass Semantics

The `NONE` test state arises when a test class references types or methods introduced only by the fix patch; the class is absent from the compiled base commit, so the test runner cannot locate or execute it. This is distinct from `FAIL`, where the test compiles and runs but produces an incorrect result. A `NONE`→`PASS` (N2P) transition therefore indicates a newly introduced test that becomes executable only once the fix is applied. Repositories that follow a test-driven development workflow may exhibit exclusively N2P transitions with zero F2P counts, because their test patches introduce test classes that are entirely absent from the base commit.

A.7 Multi-Module Gradle Test Targeting

For multi-module Android repositories, we execute module-specific Gradle tasks (e.g., `:feature:login:testDebugUnitTest`) rather than the top-level `test` task, targeting only modules containing files touched by each PR's test patch. Relevant modules are identified by mapping modified test files to their enclosing Gradle module via `settings.gradle` declarations. We default to the `Debug` build variant, which is the standard configuration used by the CI pipelines of the Android repositories in the benchmark.

B Manual Validation Details

Following SWE-bench-Verified [OpenAI, 2024], we perform human review of each instance that passes execution-based filtering. The review criteria are described in Section 3 (Phase 5). Figures 6 and 7 visualise the score distributions and dataset composition, respectively.

B.1 Annotation Results

Among the 541 annotated candidates, 310 instances (57.3%) have unambiguously well-specified problem statements (score 0) and 150 (27.7%) have minor ambiguity but a clear interpretation (score 1). The remaining 81 (15.0%) are excluded by problem-statement score: 44 receive score 2 (vague) and 37 score 3 (insufficient information).

Test-coverage adequacy shows a higher baseline: 389 instances (71.9%) receive score 0 and 61 (11.3%) score 1. A total of 91 instances (16.8%) are excluded for inadequate test coverage, of which

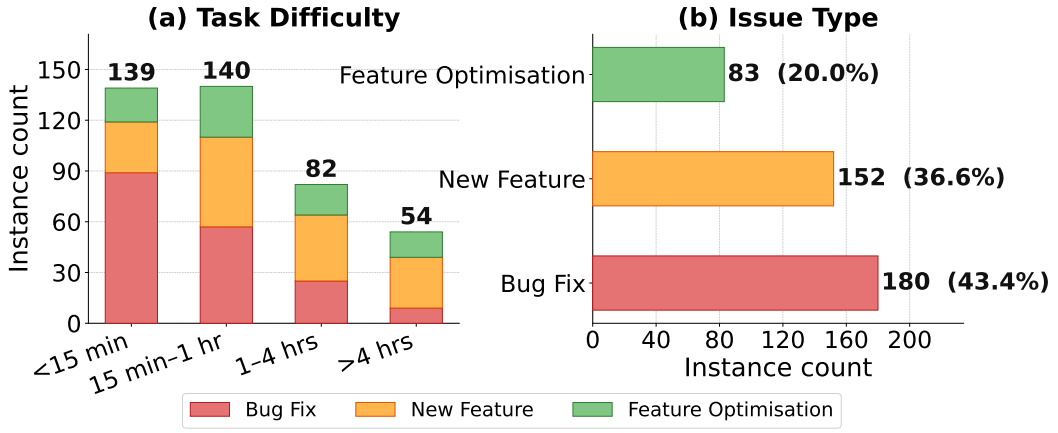

Figure 7: Task characteristics of the 415 verified benchmark instances. **(a)** Difficulty distribution stacked by issue type, using the four time-based difficulty labels assigned during annotation. **(b)** Overall issue-type breakdown. The benchmark spans the full difficulty range and covers all three major categories of mobile development activity.

36 score 2 and 55 score 3. Fifty-two instances (9.6%) are excluded for *both* reasons simultaneously, yielding 126 total exclusions (23.3%) and 415 verified benchmark instances (76.7%).

B.2 Task Characteristics

Figure 7 summarises the composition of the 415 verified instances by estimated resolution time and issue type. Difficulty is broadly distributed: 139 instances (33.5%) are estimated to take under 15 minutes, 140 (33.7%) 15 minutes to one hour, 82 (19.8%) one to four hours, and 54 (13.0%) more than four hours. By issue type, 180 instances (43.4%) are Bug Fixes, 152 (36.6%) are New Features, and 83 (20.0%) are Feature Optimisations.

C Benchmark Comparison

C.1 Comparison with Existing Benchmarks

Table 1 situates MobileDev-Bench among existing issue-resolution benchmarks across four dimensions: domain coverage, verification, difficulty stratification, and patch complexity. Most prior benchmarks focus on library-style repositories, often in a single language or a small set of languages. Their gold patches are typically small, averaging between 1.2 and 4.9 modified files and between 14 and 163 changed lines. In contrast, MobileDev-Bench targets production mobile apps and exhibits substantially larger fixes, averaging 13.5 modified files and 344.2 changed lines.

Patch Complexity. Figure 3 compares per-repository median patch size across five benchmarks using three metrics: files modified, hunks, and changed lines. SWE-PolyBench [Rashid et al., 2025], SWE-bench Multimodal [Yang et al., 2024b], and Multi-SWE-bench [Zan et al., 2025] occupy a compact low-to-moderate range, with median file counts of 2.6, 3.6, and 3.0, and median changed lines of 54, 128, and 73, respectively. Rust-SWE-bench [Xiang et al., 2026] is an outlier on the files axis (median 10.4 files) owing to the broad cross-module scope of typical Rust fixes, yet its median changed lines (92) remain well below those of MobileDev-Bench.

MobileDev-Bench presents the largest median hunk count (27.9) and changed line count (341.2) across all five benchmarks. Compared with SWE-PolyBench, the closest match on repository count, MobileDev-Bench fixes are 3.1× larger in files, 3.9× in hunks, and 6.3× in changed lines. Relative to Multi-SWE-bench, the ratios are 2.7×, 3.6×, and 4.7×, whereas relative to SWE-bench Multimodal, they are 2.3×, 3.5×, and 2.7×. While Rust-SWE-bench has a slightly higher median file count than MobileDev-Bench (10.4 vs. 8.1), MobileDev-Bench exceeds it by 2.7× in hunks and 3.7× in changed lines, indicating that mobile app fixes involve more densely modified changes rather than

touching more files. This pattern reflects the coordinated nature of mobile app fixes, where a single logical change often requires updates across source code, resources, and build configuration files.

Design Dimensions and Scale. MobileDev-Bench contains 415 manually verified instances from 19 repositories across four languages, comparable in scale to SWE-PolyBench Verified (382 instances, 20 repositories) and roughly four-fifths of SWE-bench Verified (500 instances, 12 repositories), while targeting a distinct application domain. Beyond scale, it differs from prior benchmarks in three design aspects. First, it covers production mobile apps spanning three frameworks (Android Native, React Native, and Flutter) across four programming languages, a domain absent from all existing benchmarks. Second, it combines human verification with difficulty stratification, enabling controlled evaluation across task complexity levels. Third, it explicitly captures multi-artifact fixes, where resolving an issue requires coordinated changes across heterogeneous project components rather than isolated source edits. Issue descriptions are also longer on average (310 tokens), reflecting the richer contextual information typical of mobile bug reports, such as reproduction steps, device configurations, and platform versions.

C.2 Conceptual Differences from Backend Benchmarks

Existing benchmarks, such as SWE-bench [Jimenez et al., 2024] and its multimodal [Yang et al., 2024b] and multilingual variants [Zan et al., 2025, Rashid et al., 2025, Kabir, 2025], Existing benchmarks primarily target issue resolution in library-style repositories with relatively direct control flow. These benchmarks often evaluate library-centric or backend repositories where control flow is direct and execution environments are relatively straightforward (e.g., standard Python or JavaScript runtimes).

Execution and Lifecycle: Mobile application frameworks change this control flow significantly. The operating system dictates the lifecycle of an application, instantiating UI components (such as Activities in Android or Widgets in Flutter) and maintaining complex asynchronous states. MobileDev-Bench tests an LLM's ability to reason over these implicit callbacks and event-driven architectures that are entirely absent in CLI or library benchmarks.

Artifact Diversity: Furthermore, a core distinction of MobileDev-Bench is its emphasis on multi-artifact coordination. While conventional benchmarks evaluate patches applied strictly to source code files, mobile fixes routinely demand coordinated changes across declarative user interface definitions (e.g., XML layouts), localized string resources, manifestation configurations, and heavy build scripts (Gradle). Models that perform well on algorithmic tasks may face additional challenges when synchronization across artifacts is required between a deeply nested source logic and external framework resources.

Scale of Modifications: As detailed in the comparison in Figure 3, the sheer scale of patches in MobileDev-Bench far exceeds those typical of SWE-bench paradigms. The necessity to touch multiple files reflects the cross-cutting operational reality of mobile deployments, reflecting a higher degree of integration complexity than prior settings.

D Dataset Analysis

We provide additional quantitative analyses of the MobileDev-Bench dataset that complement the summary statistics in Section 4.

D.1 Framework-wise Distribution and Code Change Characteristics

Table 8 reports per-framework instance counts and patch statistics. Android Native dominates the dataset (335 instances, 80.7%), reflecting the larger number of mature open-source Android repositories. Flutter contributes 63 instances (15.2%) from two high-activity repositories, and React Native contributes 17 instances (4.1%) from four repositories. The fix patch statistics reveal that Android Native instances involve substantially more code changes on average (14.5 modified files, 29.6 hunks) than Flutter (7.6 files, 18.3 hunks), reflecting the greater modularisation typical of large Android codebases. Notably, Flutter test patches add considerably more lines on average (234.3) than their fix counterparts (204.6), indicating that Flutter contributors tend to write comprehensive test suites alongside their fixes.

Table 8: Framework-wise instance distribution and code change characteristics in MobileDev-Bench

Metric	Android Native	Flutter	React-Native
Instance distribution			
Repositories	13	2	4
Instances	335	63	17
Fix patch statistics			
Avg. # files	14.5	7.6	15.9
Avg. # hunks	29.6	18.3	41.3
Lines added	234.5	204.6	312.5
Lines removed	118.6	50.6	187.2
Test patch statistics			
Avg. # files	6.6	4.0	3.9
Avg. # hunks	16.2	13.9	15.1
Lines added	143.5	234.3	98.4
Lines removed	50.1	43.5	67.4

D.2 Distribution of Artifact Type Combinations

Table 9 reports the distribution of artifact type combinations present in fix patches. The majority of instances (57.3%) modify only source files, and 57.8% are single-artifact overall, indicating that the primary challenge is code comprehension and logic repair. However, 42.2% of instances require coordinated changes across multiple artifact types, testing a model's ability to reason about the full mobile-development toolchain. The most common multi-artifact pattern is Source + Build (8.2%), where fixes also modify Gradle build scripts, dependency version catalogues, or build configuration files. Android Native accounts for most multi-artifact instances: patterns combining source with resources, internationalisation string files, or build scripts are exclusive to or heavily concentrated in Android projects, whereas Flutter instances are predominantly source-only (74.6%).

D.3 Background Knowledge Required by Framework

Figure 8 characterises the technical knowledge domains implicitly required by each framework's instances, inferred from import statements, annotations, and identifiers in the added lines of fix patches. The six domains span the main layers of modern mobile engineering: asynchronous programming, declarative UI frameworks, architecture components, dependency injection, platform and native APIs, and networking.

For Android Native, Architecture Components are the most frequently required domain (36.4%), reflecting widespread use of `ViewModel`, `Room`, `WorkManager`, and Navigation across the benchmark repositories. Declarative UI knowledge (Jetpack Compose) is needed in 33.7% of Android Native instances, Asynchronous/Concurrency patterns (Kotlin coroutines, `Flow`) in 30.4%, Platform SDK APIs in 17.3%, and Dependency Injection via Hilt or Dagger in 17.3%. Together, these figures confirm that Android Native instances stress a broad stack of modern Android engineering beyond core Kotlin syntax.

Flutter instances primarily demand Widget and UI framework knowledge (39.7%) alongside asynchronous Dart patterns (31.7%), consistent with Flutter's widget-centric programming model. Architecture and state-management patterns (BLoC, Provider) are needed in only 6.3% of Flutter instances, reflecting the dataset's focus on high-activity bug-fix PRs rather than large architectural changes.

React Native percentages are indicative only ($n = 17$): UI framework patterns appear in 41.2% of instances and Async/Concurrency in 23.5%, with Architecture Components in 17.6% and Platform/Native APIs in 11.8%; Dependency Injection and Networking domains register zero, plausibly reflecting these repositories' reliance on abstracted, library-level patterns not captured by source-level identifiers alone.

Table 9: Distribution of artifact type combinations in MobileDev-Bench. 42.17% of instances require coordinated changes across multiple artifact types. Framework columns show the % of instances within each framework exhibiting that combination (AN = Android Native, Fl = Flutter, RN = React Native).

	Overall		% by Framework		
Combination	**#**	**%**	**AN**	**Fl**	**RN**
Single Artifact Type					
Src	238	57.35	54.62	74.60	47.06
Res	1	0.24	0.30	0.00	0.00
Build	1	0.24	0.30	0.00	0.00
Subtotal	*240*	*57.83*	*55.22*	*74.60*	*47.06*
Two Artifact Types					
Src + Build	34	8.19	10.15	0.00	0.00
Src + Res	15	3.62	4.48	0.00	0.00
Src + i18n	15	3.62	0.00	20.63	11.77
Src + Docs	14	3.37	4.18	0.00	0.00
Src + Other	4	0.96	0.89	0.00	5.88
Other combinations	2	0.48	0.30	1.59	0.00
Subtotal	*84*	*20.24*	*20.00*	*22.22*	*17.65*
Three or More Artifact Types					
Src + Res + i18n	27	6.51	8.06	0.00	0.00
Src + Res + i18n + Build	11	2.65	3.29	0.00	0.00
Src + Build + Docs	7	1.69	1.79	1.59	0.00
Src + Manifest + Config	5	1.21	0.00	1.59	23.53
Src + Res + i18n + Docs	5	1.20	1.49	0.00	0.00
Other combinations	36	8.67	10.15	0.00	11.76
Subtotal	*91*	*21.93*	*24.78*	*3.18*	*35.29*
Total (Single)	**240**	**57.83**	**55.22**	**74.60**	**47.06**
Total (Multi)	**175**	**42.17**	**44.78**	**25.40**	**52.94**

E Experimental Setup

This section provides detailed information about the experimental framework, model selection, and evaluation metrics used to assess automated program repair performance on MobileDev-Bench.

E.1 Evaluation Framework

We evaluate MobileDev-Bench using the localization and repair pipeline from Agentless [Xia et al., 2024], a state-of-the-art autonomous code repair framework that has demonstrated strong performance on Python-centric benchmarks. Agentless decomposes the repair task into two sequential phases:

1. **Fault Localization:** Given an issue description and repository context, the model identifies potentially buggy files and functions. This phase employs retrieval-augmented generation to narrow the search space from the entire codebase to a manageable set of candidate locations.
2. **Patch Generation:** The model generates a patch as a unified diff format targeting the localized code elements. The generated patch must be syntactically valid and applicable to the base commit.

This two-phase decomposition enables us to separately measure localization capability (can the model identify where to fix?) and repair capability (can the model generate the correct fix?), providing diagnostic insights into failure modes.

We validate generated patches using a Docker-based test harness. For each task instance, we: (1) clone the repository at the base commit, (2) apply the model-generated patch, (3) execute the repository test suite in an isolated Docker container, and (4) compare test outcomes against the ground truth. A patch is considered successful if it passes all tests that pass with the developer's ground truth patch, including previously failing tests that should now pass.

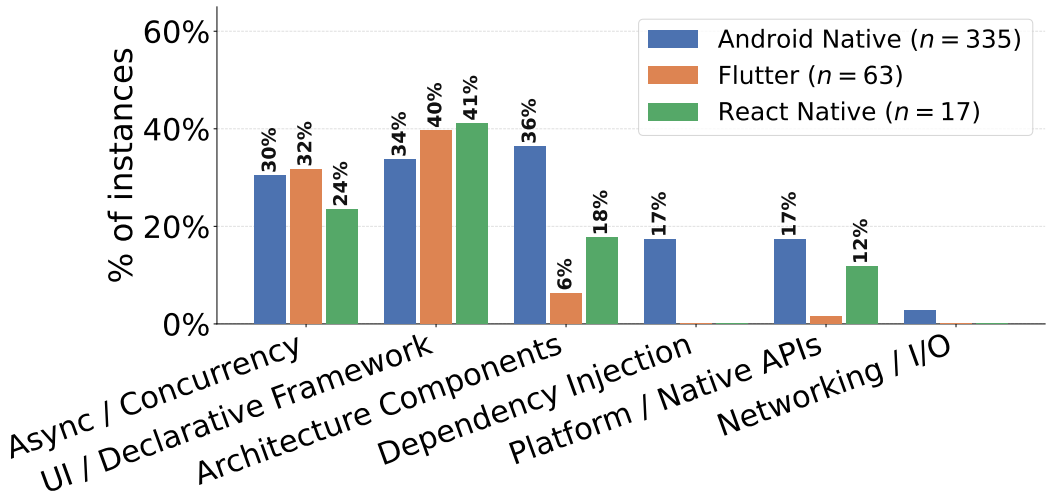

Figure 8: Percentage of instances per framework whose fix patches involve each technical knowledge domain, inferred from import statements, annotations, and identifiers in the added lines of fix patches. An instance may span multiple domains.

This evaluation approach mirrors the workflow used in SWE-bench [Jimenez et al., 2024] and Multi-SWE-bench [Zan et al., 2025], enabling direct comparison with existing benchmarks while avoiding the compounding errors and non-determinism of full agentic systems that include web browsing, terminal access, and multi-turn interactions.

E.2 Adaptation for Mobile Development Languages

The original Agentless implementation relies on Python's Abstract Syntax Tree (AST) module for parsing and extracting code structures. To support the diverse programming languages in MobileDev-Bench (Java, Kotlin, TypeScript, and Dart), we incorporate tree-sitter[1], a parsing library that provides unified syntax tree generation across multiple languages.

Similar to the MagentLess[2] adaptation for Multi-SWE-bench, we replace Python-specific parsing calls with tree-sitter-based parsing while maintaining the same localization and patch generation algorithms. This ensures that:

- Code structure extraction (classes, methods, functions) works consistently across Android (Java/Kotlin), Flutter (Dart), and React Native (TypeScript/JavaScript) codebases.
- Function-level localization operates with the same granularity across all four languages.
- The core Agentless algorithm remains unchanged, ensuring fair comparison with results on Python-based benchmarks.

For non-code artifacts (XML layouts, JSON i18n files, Gradle build scripts), which lack traditional function boundaries, we perform file-level localization only.

E.3 Model Selection

We evaluate four frontier language models: Claude Sonnet 4.5 [Anthropic, 2025] (Anthropic's proprietary model with 1M token context window), GPT-5.2 [OpenAI, 2025] (OpenAI's frontier model with enhanced reasoning capabilities), Gemini 2.5 Flash [Comanici et al., 2025] (Google's fast-inference model with native multimodal support), and Qwen-3-Coder [Qwen Team, 2025] (an open-weight model from Alibaba Cloud, fine-tuned for code tasks).

All models are accessed via OpenRouter[3].

[1] `https://pypi.org/project/tree-sitter-language-pack/`
[2] `https://github.com/multi-swe-bench/MagentLess`
[3] `https://openrouter.ai/`

Table 10: Resolution rates by ground truth file count under the Agentless and Oracle evaluation settings. Rate is resolved / evaluated instances (empty-patch predictions excluded from the denominator in the Oracle setting; instances without a submitted prediction excluded in Agentless). Column abbreviations: C=Claude Sonnet 4.5, Q=Qwen3 Coder, G=GPT 5.2, Ge=Gemini 2.5 Flash.

		Resolved				Rate (%)			
#Files	**Tasks**	**C**	**Q**	**G**	**Ge**	**C**	**Q**	**G**	**Ge**
Agentless									
1 file	71	11	9	11	9	15.5	12.7	15.5	12.7
2–3 files	89	4	2	2	2	4.5	2.2	2.2	2.2
4–5 files	52	2	2	0	2	3.8	3.8	0.0	3.8
6–10 files	70	0	0	0	0	0.0	0.0	0.0	0.0
11+ files	133	0	4	3	4	0.0	3.0	2.3	3.0
Overall	415	17	17	16	17	4.15	4.28	3.90	4.16
Oracle									
1 file	71	18	9	14	7	25.4	12.7	19.7	9.9
2–3 files	89	3	1	1	0	3.4	1.1	1.1	0.0
4–5 files	52	2	0	1	1	3.8	0.0	1.9	1.9
6–10 files	70	0	0	2	0	0.0	0.0	3.0	0.0
11+ files	133	2	0	3	0	1.5	0.0	2.4	0.0
Overall	415	25	10	21	8	6.07	2.52	5.19	1.94

Agentless setting. We use temperature 0.0 for fault localization. For patch generation, we generate one greedy sample (temperature 0.0) and multiple diverse samples (temperature 0.8), then rank and select the best patch. To ensure fair comparison, we disable extended reasoning modes: `reasoning_effort=low` for GPT-5.2, no extended thinking for Claude Sonnet 4.5, and default settings for Gemini 2.5 Flash and Qwen-3-Coder.

Oracle setting. The fault localization step is bypassed; models are given the exact set of files modified in the ground truth patch. Patch generation uses a single greedy sample (temperature 0.0). No reasoning mode overrides are applied.

E.4 Evaluation Metrics

Following prior work on automated program repair [Xia et al., 2024, Rashid et al., 2025], we report two complementary families of metrics that separately assess end-to-end repair capability and fault localization accuracy.

E.4.1 Resolution Rate

Let T denote the total number of tasks in the benchmark. For each task t, we define an indicator function $\mathbf{1}(\text{Pass}_t)$ that equals 1 if the model-generated patch passes all tests when applied to the base commit, and 0 otherwise. We compute Resolution Rate as:

$$\text{Resolution Rate} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{1}(\text{Pass}_t)$$

This metric measures end-to-end repair capability at the task level. A high Resolution Rate indicates that the model successfully localizes the bug, generates a syntactically correct patch, and produces semantically correct fixes that satisfy all test requirements.

Resolution Rate is the primary metric for comparing model performance, as it directly measures the practical utility of automated repair systems. However, it conflates localization and generation capabilities, providing limited diagnostic insight when rates are low.

E.4.2 Retrieval Metrics: Precision, Recall, and F1

To isolate fault localization performance from patch generation quality, we compute file-level retrieval metrics. For each task t, let F_t^{GT} denote the set of ground truth modified files (extracted from the developer's patch) and let F_t^{Pred} denote the set of files modified by the model-generated patch.

We compute per-task retrieval metrics as:

$$\begin{aligned}
\text{Recall}_t &= \frac{|F_t^{GT} \cap F_t^{Pred}|}{|F_t^{GT}|} \\
\text{Precision}_t &= \frac{|F_t^{GT} \cap F_t^{Pred}|}{|F_t^{Pred}|} \\
\text{F1}_t &= \frac{2 \cdot \text{Precision}_t \cdot \text{Recall}_t}{\text{Precision}_t + \text{Recall}_t}
\end{aligned}$$

We then report macro-averaged metrics across all T tasks:

$$\begin{aligned}
\text{Recall} &= \frac{1}{T} \sum_{t=1}^{T} \text{Recall}_t \\
\text{Precision} &= \frac{1}{T} \sum_{t=1}^{T} \text{Precision}_t \\
\text{F1} &= \frac{1}{T} \sum_{t=1}^{T} \text{F1}_t
\end{aligned}$$

These metrics measure fault localization performance independently of patch correctness:

- **Recall** captures the fraction of ground truth files that the model correctly identifies as requiring modification. Low recall indicates the model misses many necessary modification sites.
- **Precision** measures the fraction of predicted files that are actually relevant (i.e., appear in the ground truth). High precision indicates the model avoids editing irrelevant files, while low precision suggests over-editing or incorrect localization.
- **F1** provides a harmonic mean balancing both aspects, penalizing models that achieve high recall through indiscriminate file editing or high precision through overly conservative predictions.

Together with Resolution Rate, these metrics distinguish localization capability from repair capability. High recall with low Resolution Rate suggests that the model identifies the correct files but fails to generate correct fixes. Conversely, low recall limits achievable Resolution Rate, since a model cannot repair files it fails to localize. High precision indicates the model avoids over-editing irrelevant files, which is important for minimizing unintended side effects in production deployments.

E.5 Evaluation Infrastructure

Each task instance is executed in an isolated Docker container with repository-specific dependencies installed from the original project's build configuration (Gradle for Android, pub for Flutter, npm/yarn for React Native).

Test execution timeouts are set to 30 minutes per task to accommodate large test suites in production repositories. Tasks that exceed this limit are marked as timeouts and counted as failures.

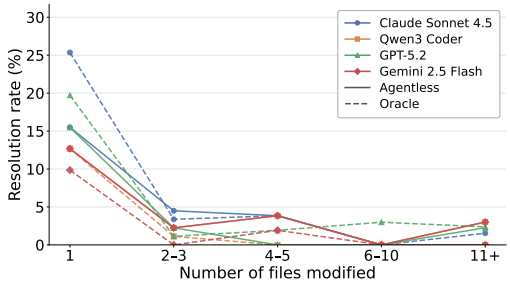

Figure 9: Resolution rates by number of files modified, for Agentless (solid lines) and Oracle (dashed lines) settings. Single-file tasks achieve 12.7–15.5% under Agentless and up to 25.4% under Oracle, dropping to near zero for 6+ files.

F Evaluation Results

This section presents comprehensive evaluation results for four frontier language models on MobileDev-Bench. We analyze resolution rates, fault localization performance, and the relationship between task characteristics and repair success.

F.1 Resolution Rate Analysis

Table 10 shows resolution rates for both evaluation settings on 415 verified tasks (rate = resolved / evaluated, excluding empty-patch predictions from the denominator). Under the Agentless setting, rates range from 3.90% to 4.28% (16–17 resolved per model, 397–410 evaluated). Under the Oracle setting, rates range from 1.94% (Gemini 2.5 Flash, 8/412) to 6.07% (Claude Sonnet 4.5, 25/412). Tasks in MobileDev-Bench require modifying an average of 13.5 files (median: 5), reflecting the multi-file nature of mobile development.

Multi-File Modifications Show Lower Resolution Rates.

Figure 9 reveals a dramatic degradation in resolution rates as the number of required file modifications increases. Under the Agentless setting, single-file tasks (71 tasks) achieve 12.7–15.5% resolution rates, substantially higher than the 3.9–4.3% overall rates. Under the Oracle setting, single-file rates rise further to 9.9–25.4%, reflecting that localization is the dominant bottleneck on simpler tasks. However, success rapidly diminishes with task complexity under both settings.

Tasks requiring 2–3 file modifications show a steep drop to 2.2–4.5% under Agentless and 0.0–3.4% under Oracle. Mid-complexity tasks requiring 4–10 files prove nearly unsolvable (0.0–3.8% Agentless; 0.0–3.8% Oracle). This difficulty cliff highlights models' fundamental limitations in coordinating changes across multiple files.

Tasks requiring 11+ files show modest recovery under Agentless (0.0–3.0%), with Qwen3 Coder and Gemini 2.5 Flash each resolving 4 of 133 such instances. However, detailed analysis (Section F.3) reveals this recovery stems from repository-specific architectural patterns rather than improved multi-file reasoning capabilities. Specifically, the majority of resolved 11+ file instances come from a single repository (`element-hq/element-x-android`), suggesting that certain codebases facilitate large-scale automated changes through consistent structure and comprehensive test coverage.

Across all models, only 32 unique instances were successfully resolved under Agentless, accumulating to 67 total resolutions due to overlap. The remaining 383 tasks (92.3%) were not resolved by any Agentless model, underscoring the fundamental challenge of multi-file reasoning in mobile development contexts.

Larger Patches Correlate with Lower Success Rates.

Patch complexity, measured by lines changed in the ground truth fix, strongly predicts resolution success (Figure 10). Under Agentless, tiny patches ($\leq$10 lines) achieve 14.3–16.1% resolution rates, while very large patches ($>$500 lines) drop to 0.0–3.9%. Under Oracle, rates for $\leq$10-line patches

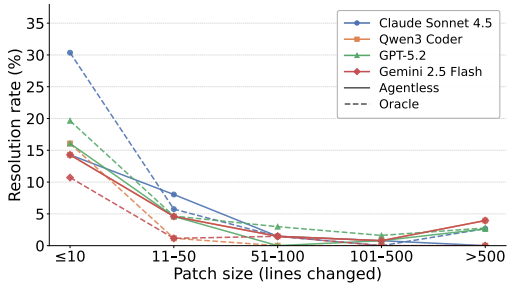

Figure 10: Resolution rates by patch size, for Agentless (solid lines) and Oracle (dashed lines) settings. Tiny patches ($\leq$10 lines) achieve 14.3–16.1% under Agentless and 10.7–30.4% under Oracle, dropping to near zero for patches $>$100 lines.

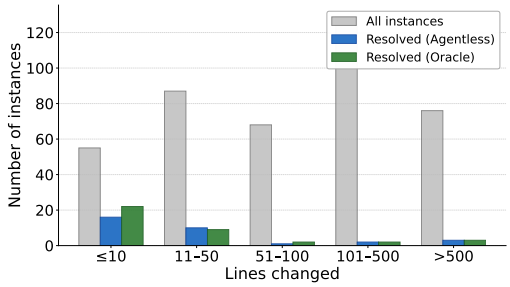

Figure 11: Patch size distribution for all instances versus resolved instances. Resolved instances skew heavily toward patches $\leq$ 50 lines (81% vs. 34%).

reach 10.7–30.4%, and $>$500-line patches remain low at 0.0–2.7%. This inverse relationship reveals that models struggle not only with multi-file coordination but also with generating extensive code changes within identified locations.

Resolved instances exhibit dramatically smaller patches than unresolved tasks. The median patch size for resolved instances is 9.5 lines versus 109.0 lines for unresolved instances (11.5x difference). Mean patch sizes show a similar pattern: 102.6 lines for resolved versus 364.4 lines for unresolved (0.28x ratio). This 72% size reduction indicates that successful repairs target localized, focused changes rather than broad refactorings.

Patch size distributions further emphasize this disparity (Figure 11). Small patches ($\leq$50 lines) comprise 81% of resolutions but only 34% of the benchmark, a 2.4x overrepresentation. Conversely, patches >50 lines represent 66% of the benchmark but only 19% of resolutions. Tiny patches ($\leq$10 lines) show the most extreme skew: 53.1% of resolutions versus 13.5% of benchmark (3.9x overrepresentation). Resolution rates collapse at medium patch sizes (51–100 lines: 0.0–1.5%) and remain near zero for larger patches (101–500 lines: 0.8–0.8%; >500 lines: 0.0–3.9%).

This finding complements the file count analysis, revealing two orthogonal dimensions of repair difficulty: spatial distribution (how many files) and change magnitude (how many lines). While file count measures coordination complexity, patch size measures generation complexity. Large patches require models to reason about extensive code interactions, maintain consistency across many modified lines, and ensure all changes compile and pass tests. Current models excel at small, surgical fixes but fail when extensive code generation is required.

F.2 Characteristics of Resolved Instances

Table 11 analyzes how resolved instances differ from the overall benchmark distribution across three dimensions: artifact type diversity, file count distribution, and framework. This analysis reveals structural factors that influence repair success.

Table 11: Resolution rate breakdown by task characteristics (415 tasks). Resolved column shows the share of uniquely resolved instances (32 Agentless, 38 Oracle) in each category vs. the overall benchmark share. Ratio = Resolved% / Benchmark%.

		Agentless		Oracle	
Category	**Bench.**	**Resolved**	**Ratio**	**Resolved**	**Ratio**
Artifact Type Diversity					
Single-artifact	57.8%	84.4%	1.46x	86.8%	1.50x
Multi-artifact	42.2%	15.6%	0.37x	13.2%	0.31x
File Count Distribution					
1 file	17.1%	65.6%	3.84x	68.4%	4.00x
2–3 files	21.4%	15.6%	0.73x	10.5%	0.49x
4–5 files	12.5%	6.3%	0.50x	5.3%	0.42x
6–10 files	16.9%	0.0%	0.00x	5.3%	0.31x
11+ files	32.1%	12.5%	0.39x	10.5%	0.33x
Framework Distribution					
Android Native	80.7%	71.9%	0.89x	81.6%	1.01x
Flutter	15.2%	28.1%	1.85x	13.1%	0.86x
React Native	4.1%	0.0%	0.00x	5.3%	1.29x

Artifact Type Diversity. We classify tasks as single-artifact (all files of one type) or multi-artifact (spanning source code, resources, build files, manifests). While 57.8% of benchmark tasks are single-artifact, 84.4% of Agentless resolutions come from single-artifact tasks (1.46x overrepresentation). Multi-artifact tasks show 0.37x underrepresentation (15.6% of resolutions vs. 42.2% of benchmark), highlighting difficulty coordinating changes across heterogeneous file types.

File Count Distribution. Single-file tasks dominate Agentless resolutions (65.6% of resolutions vs. 17.1% of benchmark, 3.84x overrepresentation), while multi-file tasks show severe underrepresentation. Tasks requiring 6–10 files exhibit 0.0% of Agentless resolutions versus 16.9% of benchmark (0.00x ratio), the steepest underrepresentation. The 11+ file category shows modest recovery (0.39x ratio), driven by repository-specific architectural regularity (Section F.3).

Framework Distribution. Framework distribution among resolved tasks broadly follows benchmark composition but with notable deviations (Table 11). Android Native tasks are near-proportionally represented (71.9% of Agentless resolutions vs. 80.7% of benchmark, 0.89x), while Flutter tasks are overrepresented at 28.1% of resolutions despite comprising only 15.2% of the benchmark (1.85x), likely reflecting the structural characteristics of Flutter repositories, which tend toward single-artifact changes with strong semantic signals in issue descriptions. React Native tasks see complete failure (0.0% vs. 4.1%), mirroring patterns seen across complex multi-layer architectures where cross-file dependency tracing breaks down. These differences reflect repository architecture and framework conventions rather than underlying language difficulty.

F.3 Analysis of 11+ File Resolved Instances

The modest recovery in resolution rates for 11+ file tasks (0.0–3.0%) stems from the concentration of resolved instances in a single repository rather than general model capability. All 4 unique resolved instances with 11+ files under the Agentless setting come from `element-hq/element-x-android`, whose modular architecture, consistent naming conventions, and comprehensive test coverage facilitate large-scale automated repair.

These instances share structural characteristics: averaging 33.5 files across 21.5 directories, predominantly multi-artifact changes (3/4), and 71% source code modifications. This finding indicates that repository architecture, not task scale alone, determines repair difficulty.

F.4 Fault Localization Performance

Table 12 shows that fault localization is the primary bottleneck. Overall retrieval performance ranges from 13.5–18.3% recall, 42.5–57.5% precision, and 17.3–23.0% F1, with Claude Sonnet 4.5 achieving the highest scores. The large precision-recall gap (3x across all models) indicates conservative file selection: models identify relevant files but miss many necessary modifications.

Table 12: Retrieval metrics by ground truth file count for Agentless and Oracle settings. All metrics are percentages, averaged across tasks in each file count bin. Column abbreviations: C=Claude Sonnet 4.5, Q=Qwen3 Coder, G=GPT 5.2, Ge=Gemini 2.5 Flash.

		Recall (%)				Precision (%)				F1 (%)			
#Files	**Tasks**	**C**	**Q**	**G**	**Ge**	**C**	**Q**	**G**	**Ge**	**C**	**Q**	**G**	**Ge**
Agentless													
1 file	71	54.9	46.5	35.2	35.2	54.9	46.5	35.2	35.2	54.9	46.5	35.2	35.2
2–3 files	89	22.7	23.6	20.4	20.2	49.4	48.3	44.4	44.9	30.5	30.9	27.5	27.5
4–5 files	52	12.1	12.7	9.9	10.5	51.9	53.8	40.4	42.3	19.6	20.5	15.8	16.7
6–10 files	70	9.4	7.5	7.4	6.7	65.7	52.1	49.3	46.4	16.4	13.0	12.8	11.6
11+ files	133	3.0	2.9	2.1	2.3	62.0	51.5	43.2	42.9	5.6	5.3	3.9	4.2
Overall	415	18.3	16.8	13.6	13.5	57.5	50.4	42.8	42.5	23.0	21.0	17.3	17.3
Oracle													
1 file	71	95.8	97.2	84.5	100.0	95.8	96.5	83.8	100.0	95.8	96.7	84.0	100.0
2–3 files	89	57.5	50.2	40.6	60.9	94.9	98.5	89.9	96.5	68.9	64.4	54.9	71.6
4–5 files	52	48.8	35.9	21.7	50.2	89.8	93.2	89.2	92.6	58.1	48.0	34.4	60.7
6–10 files	70	43.1	30.4	14.2	43.2	90.8	89.9	87.1	95.8	53.6	41.5	23.2	53.6
11+ files	133	23.9	14.2	8.1	35.7	80.9	78.5	77.1	89.5	32.3	20.8	11.7	43.8
Overall	415	49.8	41.6	30.9	55.2	89.2	89.6	84.2	94.3	57.8	50.0	38.1	63.1

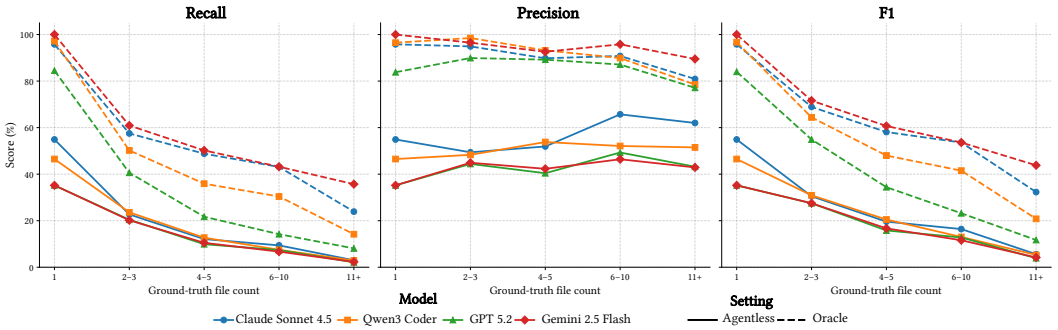

Figure 12: Retrieval metrics (recall, precision, F1) by ground-truth file count across all four models. Recall collapses from 35–55% on single-file tasks to 2–3% on 11+ file tasks, while precision remains comparatively stable (35–66%), widening the precision-recall gap with complexity. F1 degrades sharply, driven entirely by recall failure.

Recall Degradation by File Count. Recall degrades severely with file count: from 35–55% on single-file tasks to 2–3% on 11+ file tasks, representing a 33–53 percentage point decline. For tasks requiring 6–10 files, models identify fewer than 10% of necessary files. This catastrophic degradation reflects models' inability to trace cross-layer dependencies in mobile architectures, where changes cascade across network, data, presentation, view, and test layers.

Precision Trends. Precision increases with file count (35–55% for single-file tasks to 46–66% for 6–10 file tasks), indicating conservative behavior on complex changes. Localization errors stem primarily from missed files rather than false positives: models identify a core subset correctly but fail to expand to all necessary modifications.

Localization Patterns. Three distinct patterns emerge:

- **Perfect localization** (5–8% of tasks): Almost exclusively single-file changes with strong semantic signals in issue descriptions.
- **Zero localization** (46–59% of tasks): Models edit entirely incorrect files, common when ground truth modifies non-code artifacts or peripheral modules. 162 tasks (39.0%) show zero recall across all models.

- **Partial localization** (36–47% of tasks): Models find some relevant files but miss cascading modifications across architectural layers.

F.4.1 Implications

Low recall imposes an upper bound on resolution rates: even perfect patch generation would achieve only 13–18% resolution (matching the recall ceiling). Observed rates (3.90%–4.28% under Agentless) are lower still, indicating localization dominates repair difficulty. This inverts traditional assumptions in automated program repair, where patch generation is typically the primary challenge.

Improving performance requires advances in multi-file fault localization, including dependency-aware retrieval, architectural priors for mobile app patterns (MVVM, repository pattern), and specialized strategies for non-code artifacts underrepresented in embedding spaces.

G Discussion

Our evaluation reveals overall resolution rates of 3.90%–4.28% (Agentless) and 1.94%–6.07% (Oracle) on MobileDev-Bench. Three architectural properties of mobile development explain this low performance. First, *multi-artifact coordination*: models achieve 1.46x success on single-artifact tasks but only 0.37x on multi-artifact tasks requiring synchronization across source code, resources, build configurations, and manifests. Second, *framework-mediated execution*: platform lifecycles obscure control flow from static analysis. Third, *cross-module dependencies*: recall drops from 35–55% on single-file tasks to 7–9% on multi-file tasks (6–10 files), and further to 2–3% on large-scale tasks (11+ files).

Localization is the primary bottleneck. File-level recall of 13.5–18.3% imposes a hard ceiling on resolution rates. For mobile development, the challenge lies not in generating correct patches but in identifying which files need modification. This finding suggests research should prioritize multi-file dependency tracing, artifact-aware retrieval, and framework-specific reasoning over patch generation improvements.

The non-monotonic relationship between file count and resolution rate provides insight: single-file tasks achieve 13 to 16% resolution, mid-complexity tasks (4 to 10 files) drop to 0 to 3.8%, while 11+ file tasks show modest recovery (0.0 to 3.0%). This pattern suggests models struggle with *architectural complexity* (interconnected changes across heterogeneous components) rather than raw file count. Repository-specific phenomena hint that certain architectural patterns may facilitate automated repair.

H Future Work

Our findings suggest three high-priority research directions. First, given that localization is the primary bottleneck, techniques combining semantic search with static analysis for multi-file dependency tracing warrant immediate investigation. Second, repository-specific success patterns hint that certain architectural characteristics aid automated repair; systematically analyzing what makes codebases "LLM-friendly" could inform both tool design and best practices. Third, mobile issues often include screenshots and crash logs; evaluating multi-modal models' ability to ground repairs in visual artifacts may improve UI-related bug localization. Extending to iOS, developing framework-aware representations encoding lifecycle semantics, and testing iterative agentic workflows provide additional promising directions.

I Limitations

MobileDev-Bench focuses on Android Native and cross-platform apps, including React-Native and Flutter repositories, and does not include iOS-native projects written in Swift or Objective-C. This exclusion is a deliberate reproducibility constraint rather than a coverage gap. iOS builds are tightly coupled to Xcode and macOS: the Xcode and Apple SDKs Agreement [Apple Inc., 2024] explicitly prohibits execution on non-Apple-branded hardware, and macOS itself cannot be legally run in standard Linux container environments. Consequently, the containerized build-and-test execution that is central to our evaluation design is not achievable for iOS repositories without

proprietary infrastructure. Extending MobileDev-Bench to iOS would require macOS-based CI runners or hardware-in-the-loop setups, both of which introduce environment heterogeneity that undermines the reproducibility guarantees the benchmark is designed to provide. Evaluation relies on execution-based validation using repository test suites, which may not fully capture issues with insufficient or incomplete test coverage. Additionally, the benchmark is constructed from issue–pull request pairs in open-source repositories, which may not represent all mobile development activities, such as UI design iterations or refactoring tasks without associated issues. Finally, our evaluation uses a single-pass repair pipeline, whereas app development often involves iterative debugging and interactive workflows.

J Potential Societal Impact

MobileDev-Bench provides an execution-validated benchmark for mobile-specific program repair, exposing failure modes invisible to library-centric evaluations, guiding practitioners on where LLM assistants are and are not reliable in mobile CI pipelines, and highlighting the need for human oversight when deploying model-generated patches in accessibility-, localisation-, or security-sensitive components.

AI Assistance. AI-based writing tools were used only for minor grammatical and wording improvements. All technical content and analyses were produced and verified by the authors.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state three claims that are directly supported by the paper: (1) MobileDev-Bench is a benchmark of 415 verified mobile app issue-resolution tasks (Section 3, Table 2); (2) the evaluation framework is compilation-aware and execution-based across four languages (Section 3.4, Appendix E); and (3) four frontier LLMs achieve low resolution rates of 3.90–6.07%, with fault localization as the dominant bottleneck (Section 6, Table 4, Figure 5). No aspirational claims are presented as achieved results.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Appendix I discusses limitations explicitly, including the exclusion of iOS-native repositories (Swift/Objective-C) due to Apple Inc. licensing constraints that prevent containerized execution, reliance on repository test suites that may not fully cover fix correctness, the restriction to open-source issue–PR pairs which may not represent proprietary workflows, and the use of a single-pass repair pipeline that does not model iterative debugging.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.

- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [No]

   Justification: The paper makes no theoretical claims and presents no theorems, lemmas, or proofs. All contributions are empirical.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The benchmark instances, evaluation harness, and containerized Docker environments are released at `https://huggingface.co/datasets/MobileDev-Bench/mobiledev-bench`. Appendix E details the Agentless adaptation (tree-sitter parsing, language extensions), model identifiers and versions, temperature settings, reasoning-mode configurations, and the Docker-based test harness. Instance-specific test commands are included in each benchmark record.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example

(a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
(b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Task instances (problem statements, base commits, test patches, fix patches, test commands, and transition labels) are publicly released at `https://huggingface.co/datasets/MobileDev-Bench/mobiledev-bench` under a CC-BY 4.0 license. The evaluation harness and per-repository Dockerfiles are released alongside the dataset.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 5 and Appendix E cover all evaluation details: the two evaluation settings (Agentless and Oracle), the four models evaluated with their versions and access method (OpenRouter), temperature settings (0.0 for localization, 0.0 greedy and 0.8 diverse for Agentless patch generation, 0.0 greedy for Oracle patch generation), reasoning-mode disabling for Agentless (not applied under Oracle), the tree-sitter adaptation, file-level localization for non-code artifacts, and the 30-minute per-task timeout. No training is performed; the benchmark is evaluation-only.

   Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Error bars are not reported. Following prior work on issue-resolution benchmarks (SWE-bench, Multi-SWE-bench), we report deterministic resolution rates using temperature 0.0 for localization and a single greedy sample for the primary Agentless result. Rerunning the full evaluation across 415 tasks $\times$ 4 models $\times$ 2 settings would be prohibitively expensive.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Appendix E.5 specifies that each task instance is executed in an isolated Docker container with a 30-minute timeout, with repository-specific dependencies installed from the project's build configuration (Gradle for Android, pub for Flutter, npm/yarn for React Native). Model inference is accessed via the OpenRouter API, requiring no local GPU resources. The total number of evaluated instances (415 tasks $\times$ 4 models $\times$ 2 settings) and the per-task timeout bound the compute envelope.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.

- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The benchmark is constructed entirely from public open-source GitHub repositories under permissive licenses (Apache 2.0, MIT, GPL variants; listed in Table 6). No personal data, private repositories, or proprietary code is included. Human annotators reviewed benchmark instances; annotations were performed by the authors as part of their research duties.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Appendix J discusses positive impacts (exposing failure modes of LLM-based repair in mobile CI pipelines, guiding practitioners on where AI-generated patches can and cannot be trusted) and raises the need for human oversight when deploying model-generated patches in accessibility-, localisation-, or security-sensitive components. No direct negative societal impacts are identified, as the benchmark does not release a model or enable new harmful capabilities.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: MobileDev-Bench is a benchmark dataset derived from public GitHub pull requests and issues. It does not release a pre-trained model, image generator, or scraped dataset containing sensitive content. The constituent repositories are already publicly available under open-source licenses, and no new risk surface is introduced by their aggregation into benchmark instances.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example, by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: Table 6 lists all 19 source repositories with their licenses (GNU GPLv3, Apache 2.0, GNU AGPLv3, MIT, EPL 2.0, GNU GPLv2). The Agentless framework [Xia et al., 2024] and tree-sitter [Brunsfeld et al., 2018] are cited. All repositories are used in accordance with their open-source licenses, which permit research use.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: MobileDev-Bench is released at `https://huggingface.co/datasets/MobileDev-Bench/mobiledev-bench` with a dataset card describing fields, construction methodology, and intended use. Each instance includes structured metadata (instance ID, framework, base commit, difficulty, category, artifact types, test transitions, and test command). The evaluation harness and Dockerfiles are released alongside the dataset.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [Yes]

    Justification: Manual verification (Phase 5, Section 3.5 and Appendix B) was performed by two annotators who are co-authors of the paper, not external crowdworkers. Appendix B describes the four review criteria (problem statement clarity, test coverage appropriateness, task difficulty, task category), the scoring rubric (0–3 per dimension), and the inter-annotator disagreement resolution process. Figure 6 shows the resulting score distributions. No external compensation was involved.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The annotation work described in the paper was performed by the authors themselves and does not constitute research with human subjects under standard IRB definitions. No external participants were recruited, and no personal data was collected.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLMs (Claude Sonnet 4.5, GPT-5.2, Gemini 2.5 Flash, Qwen-3-Coder) are the evaluated systems and are central to the paper's experiments; their usage is fully described in Section 5 and Appendix E.3. AI-based writing tools were used only for minor grammatical and wording improvements and did not affect methodology or results, as noted in Appendix J.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.