MOBILEDEV-BENCH: A BENCHMARK FOR ISSUE RESOLUTION IN MOBILE APPLICATION DEVELOPMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) achieve strong performance on repository-level issue resolution, yet existing benchmarks focus on library-style repositories and rarely capture mobile applications, where fixes must build under framework-specific toolchains and often coordinate changes across source, resource, and build files. This paper introduces MobileDev-Bench, a benchmark of 407 verified issue-resolution tasks from 19 production Android Native (Java/Kotlin), React Native (TypeScript), and Flutter (Dart) applications, each paired with the developer-written tests and a containerized build environment. Fixes modify 12.9 files and 334.6 lines on average, with 41% spanning multiple artifact types. MobileDev-Bench distinguishes tests that fail before the fix from those that become compilable only after the fix introduces required symbols. Under the single-pass Agentless pipeline, four LLMs (Claude Sonnet 4.5, GPT-5.2, Gemini 2.5 Flash, and Qwen3-Coder) resolve only 3.19%–4.18% of tasks with automated retrieval and at most 5.65% with oracle retrieval. With the interactive mini-SWE-agent, the best-performing model, GPT-5.2, resolves 12.53% of tasks, compared with 69.0% for the same model and agent on the SWE-bench Verified leaderboard. We release the benchmark tasks, evaluation harness, and containerized environments.

1 INTRODUCTION

Large language models (LLMs) such as GPT-4 (OpenAI et al., 2024), Claude (Anthropic, 2023), and Code Llama (Rozière et al., 2024) have made rapid progress on repository-level issue resolution. SWE-bench (Jimenez et al., 2024) evaluates this ability on real GitHub issues and counts a generated patch as correct when the repository's tests pass on the patched code. Later benchmarks add other programming languages and multimodal issues (Yang et al., 2024c; Zhang et al., 2024a; Kabir, 2025; Mhatre et al., 2025; Zan et al., 2024; 2025), but they draw their tasks mostly from general-purpose libraries and web applications.

Mobile application (app) development is largely absent from these benchmarks. In an app repository, a framework-specific build system such as Gradle or the Flutter toolchain compiles and tests the code, so a patch must build before any test runs (Android, 2025). A fix often spans several artifact types, such as source code, manifests, and resource definitions, and must keep them consistent within one patch (Android, 2024). App code also relies on declarative user interfaces, event-driven callbacks, and asynchronous state shared between interface and service components (Flutter, 2026; 2025).

This paper introduces *MobileDev-Bench*, a benchmark of 407 verified issue-resolution tasks from 19 production Android Native (Java/Kotlin), React Native (TypeScript), and Flutter (Dart) apps, each with at least 400 GitHub stars. Each task pairs a developer-reported issue with the merged fix, the developer's tests, and a containerized build environment. Two annotators with industry experience in mobile development reviewed every candidate task and estimated its resolution time. By these estimates, 138 tasks are easy, 139 medium, and 130 hard. Fixes modify 12.9 files and 334.6 lines on average, and 41% of them span more than one artifact type. The median fix modifies 5 files and 93 lines, more than in any benchmark in Table 1.

MobileDev-Bench runs each task's tests at the base commit, with the test patch only, and with both the test and fix patches, recording Fail-to-Pass (F2P), Non-executable-to-Pass (N2P), Pass-to-Pass, and Pass-to-Fail transitions. SWE-bench retains a task only if at least one test fails before the fix and passes afterward (Jimenez et al., 2024; Zan et al., 2025). In Java, Kotlin, and Dart, however, this

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Benchmarks	Domain	#Tasks	#Repos	Verified	Difficulty level	Multi-type artifacts	Median #Files	Median #Lines
SWE-bench	Python libraries	2294	12	✗	✗	✗	1.0	12.0
SWE-bench Verified	Python libraries	500	12	✓	✗	✗	1.0	7.0
SWE-bench Multimodal	JavaScript libraries	617	17	✓	✗	✓	2.0	35.0
SWE-PolyBench	Multi-lang. libraries	2110	21	✗	✗	✗	2.0	19.0
SWE-bench Multilingual	Multi-lang. libraries	300	42	✗	✗	✗	1.0	10.0
OmniGIRL	Multi-lang. repos	959	15	✗	✗	✗	1.0	16.0
SWE-bench-java	Java libraries	91	6	✓	✗	✗	2.0	21.0
SWE-Sharp-bench	C# libraries	150	17	✓	✗	✗	2.0	25.5
Rust-SWE-bench	Rust repositories	500	34	✓	✗	✗	4.5	40.5
Multi-SWE-bench	Multi-lang. repos	1632	39	✓	✓	✗	2.0	33.0
MobileDev-Bench	**Mobile Apps**	**407**	**19**	✓	✓	✓	**5.0**	**93.0**

Table 1: Comparison of issue-resolution benchmarks. MobileDev-Bench is the only benchmark in the table that covers mobile apps and the only one that combines human verification, difficulty levels, and multi-type artifact fixes. Median files and lines are computed by us from each benchmark's public dataset.

criterion excludes tasks whose test patch references symbols introduced only by the fix: a single unresolved reference can prevent the entire test target from compiling (Gradle, 2025a;b; CS 125, UIUC, 2025), so no test in that target can register a pre-fix failure (Zan et al., 2024).

MobileDev-Bench retains such tasks and classifies the blocked tests as N2P. N2P tests added or modified by the test patch validate the fix analogously to F2P tests. Other N2P tests are blocked only because unresolved references elsewhere in the same target prevent compilation (Zhang et al., 2019); for these tests, the pre-fix outcome is taken from the base run (Section 4).

This paper extends Agentless (Xia et al., 2024) to Java, Kotlin, TypeScript, and Dart and evaluates four LLMs with it. Under this single-pass scaffold, the models resolve 3.19%–4.18% of tasks with automated retrieval and at most 5.65% with oracle retrieval. Oracle retrieval raises file-level recall to 31.4%–55.4% but leaves resolution at or below 5.65%. Under the interactive mini-SWE-agent scaffold (Yang et al., 2024a), Claude Sonnet 4.5 and GPT-5.2 resolve two to four times as many tasks, and the best model (GPT-5.2) resolves 12.53%. On SWE-bench Verified, the public leaderboard reports 69.0% for GPT-5.2 and 70.6% for Claude Sonnet 4.5 with the same scaffold (SWE-bench Team, 2026), 5.5 and 8.0 times their rates on MobileDev-Bench (Table 18). Our analysis of failed instances shows that 7 of the 15 mobile-specific task categories never resolved task under either scaffold. These categories involve platform knowledge such as security, hardware access, and OS version differences and apply to 11.8% of the tasks (Section 6.5).

This work makes the following contributions:

- MobileDev-Bench, with its evaluation harness and containerized environments, is publicly available at `https://huggingface.co/datasets/MobileDev-Bench/mobiledev-bench`.
- A compilation-aware validation procedure extends the SWE-bench retention rule to compiled test targets by separating tests that fail before the fix from tests that cannot compile until the fix defines the symbols they reference.
- Under the interactive scaffold, the resolution rates of four LLMs are at most one fifth of the rates of the same models and scaffold on SWE-bench Verified, and seven mobile-specific task categories remain unresolved under both the single-pass and the interactive scaffold.

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, both non-agentic pipelines like Agentless (Xia et al., 2024) and interactive, multi-turn agentic scaffolds like SWE-agent (Yang et al., 2024b), AutoCodeRover (Zhang et al., 2024b), MAGIS (Tao et al., 2024), OpenDevin (Wang et al., 2025), CodeR (Chen et al., 2024), and MASAI (Arora et al., 2024), have demonstrated strong results on these benchmarks, yet none target mobile application repositories. We evaluate MobileDev-Bench under one non-agentic and one agentic scaffold (Section 5) to measure how strongly the results depend on the scaffold.

Mobile app development *introduces additional challenges* not represented in current benchmarks, including framework-specific constraints, 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 & Farooq, 2026), 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*. Two concurrent efforts are closest in spirit. SWE-Bench Mobile (Tian et al., 2026) evaluates coding agents on a single production iOS (Swift/Objective-C) codebase, with multi-modal PRD- and Figma-derived tasks that go beyond issue resolution to full feature implementation; the best agent configuration reaches only 12% task success. SWE-Bench Pro (Deng et al., 2025) targets long-horizon, multi-file enterprise software tasks across 41 general-purpose repositories (business applications, B2B services, developer tools), with model performance below 45% Pass@1, but does not target mobile-specific repositories or platform constraints. MobileDev-Bench is complementary to both: it spans 19 production repositories across three mobile frameworks (Android Native, React Native, Flutter) rather than a single iOS codebase or general-purpose enterprise software, and evaluates issue resolution against executable test suites rather than PRD-to-implementation or general long-horizon tasks.

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. A detailed comparison of MobileDev-Bench with existing issue-resolution benchmarks, including patch complexity analysis and per-repository statistics, is provided in Appendix C.

3 MobileDev-Bench Construction

MobileDev-Bench is constructed from real issue-pull request (PR) pairs in open-source mobile app repositories. Figure 1 presents an overview of the five-phase construction pipeline.

3.1 Phase 1: App Repository 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.

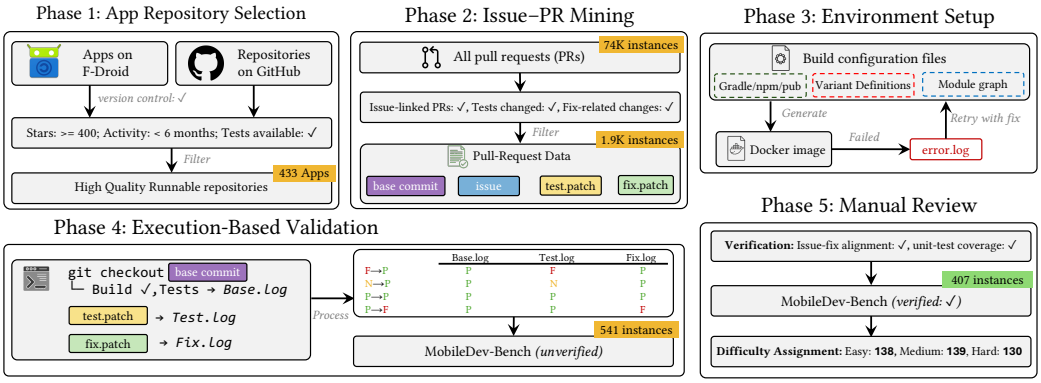

Figure 1: Overview of MobileDev-Bench construction pipeline. The framework consists of five phases: (1) filtering for high-quality repositories, (2) mining issue-linked PRs, (3) reproducing mobile builds via Docker, (4) executing compilation-aware validation, and (5) manually reviewing data to yield 407 verified, difficulty-stratified benchmark instances.

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 non-test implementation 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, restricting evaluation to host-side, headless unit test suites (e.g., `:testDebugUnitTest`). These assert against mocked view models, presenter logic, or programmatic state rather than the live rendered display, so MobileDev-Bench evaluates mobile repositories through hermetic host-executable test

suites rather than on-device or emulator GUI instrumentation; we discuss this construct-validity trade-off in Appendix I.

For each execution, we record four states: `PASS`, `FAIL`, `NON-EXECUTABLE` (test cannot be executed in that configuration, typically because it cannot compile against a symbol the fix patch has not yet introduced), and `SKIP` (see Appendix A for more details). We derive test-state transitions and retain instances satisfying three criteria: *(i)* at least one test exhibits a *Fail-to-Pass (F2P)* or *Non-executable 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 definitions in Appendix A). This execution-based filtering reduces the candidate pool from 1,939 to 541 instances.

These retention criteria give two reference points for free, by construction, rather than as separate evaluation runs. An *empty patch* (the test patch applied with no fix) cannot exhibit the required F2P or N2P transition, so it resolves 0% of instances by construction; this confirms the retained test suites are not trivially satisfiable. Conversely, the *gold patch* (the developer's original fix) is exactly the patch criterion *(i)* is checked against, so it resolves 100% of instances by construction; this serves both as an internal consistency upper bound and, since model-generated patches can be compared against it for suspicious verbatim similarity, as one input to the leakage analysis in Appendix E.10.

3.5 Phase 5: Human Verification

Following SWE-bench Verified (OpenAI, 2024), we perform a manual review to ensure benchmark quality. Two annotators, with several years of industry experience in mobile application development, independently examine each of the 541 candidate instances that passed execution-based filtering and resolve disagreements through discussion. Inter-annotator agreement is reported in Appendix B. Review focuses on four 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, a holistic judgment based on conceptual complexity, number of files modified, and required framework knowledge, which annotators operationalize as an estimated resolution-time band (<15min, 15min–1hr, 1–4hr, or >4hr); and *(iv)* Task Category by classifying each instance into one of the task types: Bug Fix, New Feature, or Feature Optimization. Where difficulty is reported as a three-tier Easy/Medium/Hard split elsewhere in the paper (e.g., Section 4), Easy and Medium correspond to the <15min and 15min–1hr bands respectively, and Hard combines the 1–4hr and >4hr bands; the four-band annotation is the canonical rubric, and the three-tier view is a coarsening of it for headline reporting.

Instances with severely underspecified issues or misaligned tests are excluded. In total, 134 instances (24.8%) fail manual verification, yielding a final benchmark of 407 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

Composition. The 19 repositories range from 337 to 7,923 source files (332K lines on average) and contribute between 1 and 100 tasks each (Appendix D.1). Bug fixes account for 44.2% of the tasks, new features for 35.9%, and feature optimizations for 19.9%. Nearly all fixes modify source files (99.5%), and many also modify resources (20.4%), build files (18.4%), localization files (17.2%), or manifests (3.7%). In total, 41.0% of the fixes span two or more artifact types, 20.9% span three or more, and 20.7% modify source files in more than one language, most often Kotlin together with Java (Appendix D). By per-repository medians, fixes modify 3.1 times as many files, 3.9 times as many hunks, and 6.3 times as many changed lines as fixes in SWE-PolyBench, the benchmark closest in repository count (Figure 6).

Validation Evidence. Of the 407 tasks, 348 (85.5%) have at least one N2P test, and the 12,310 traced N2P test entries fall into two kinds (Appendix A.6). *Primary* N2P tests (12.4%) belong to test files that the test patch adds (7.5%) or modifies (4.9%), and they exercise the fix. *Collateral* N2P tests (87.6%) are unchanged by the test patch and fail only because missing references elsewhere in their target cause compilation errors, so their number grows with the size of the blocked target (zulip-flutter averages 53.7 N2P tests per task). Their outcome before the fix comes from the base run, where they

compile. F2P tests, primary N2P tests, and collateral tests that fail at the base commit check behavior that the fix introduces or changes, while collateral tests that pass at the base commit show only that the fix restores compilation without regressions. F2P tests alone validate 59 tasks (14.5%), F2P and N2P tests together validate 88 (21.6%), and N2P tests alone validate 260 (63.9%), a share that rises from 47.2% for Bug Fix to 74.0% for New Feature and 82.7% for Feature Optimization (Table 5), since new functionality has no earlier test that can fail on it.

5 Experimental Setup

Scaffolds. MobileDev-Bench is evaluated with two scaffolds that differ in how the model interacts with the repository. Agentless (Xia et al., 2024) is a single-pass pipeline that localizes the files to edit and then generates a patch without executing code. This paper extends it to Java, Kotlin, TypeScript, and Dart by replacing its Python-specific AST parser with tree-sitter (Brunsfeld et al., 2018) (Appendix E.2). mini-SWE-agent (Yang et al., 2024a) is an interactive multi-turn scaffold in which the model reads files, runs shell commands, and revises its patch inside the task container. It runs for at most 250 steps with a cost cap of $2 per task, and its prompt templates add per-framework instructions for building and testing each project, whose commands may run for up to 30 minutes (Appendix E.6). Both scaffolds run the same four models on all 407 tasks, and one containerized harness validates every patch. Appendix E.7 compares the mini-SWE-agent results with the leaderboard results of the same agent and models on SWE-bench Verified (SWE-bench Team, 2026).

Retrieval Settings. Agentless runs under two retrieval settings, while mini-SWE-agent explores the repository itself and needs neither. *(i)* Automated retrieval: the model must identify which files to modify before generating a patch given the issue description. This end-to-end setting reflects the realistic repair scenario. *(ii)* Oracle retrieval (Jimenez et al., 2024): the ground-truth set of files to modify is provided to the model in addition to the issue description. While less realistic, it can provide an upper bound on repair performance achievable under ideal file-level localization conditions. 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.

Metrics. Following prior work (Jimenez et al., 2024; Xia et al., 2024), two metrics are reported for both scaffolds. *(i) Resolution Rate*: the percentage of tasks where the generated patch passes the full test suite. *(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. Metric definitions are provided in Appendix E.4.

Models. Both scaffolds run four code-capable models, three proprietary and one open-weight, namely 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).

Evaluation Integrity. Each repository is checked out at the instance base commit inside the evaluation container without the test or fix patch applied. Before evaluation, we remove the `origin` remote, delete branches and tags beyond the base commit, and expire the reflog, so the gold fix is not reachable through `.git` history. Network egress is disabled by default at the Docker network level; an allowlisted proxy permits only required package-registry traffic (Maven, Gradle Plugin Portal, npm, and pub.dev), while GitHub and general web access are blocked. To further assess potential memorization of gold patches, we additionally report resolution rate on a post-cutoff subset for each model, restricted to instances whose linked PR was created after that model's reported training cutoff; details and results are in Appendix E.10.

6 Results

6.1 Resolution Performance

Table 2 reports resolution rates under both retrieval settings and under mini-SWE-agent (Section 6.2). Under automated retrieval, overall performance remains low, ranging from 3.19% to 4.18% across models. Performance degrades with task difficulty: easy tasks achieve 8.0% across all models, medium tasks drop to 1.4%–4.3%, and hard tasks are not resolved by any model.

Table 2: Resolution rates (%) by difficulty level and artifact diversity under automated retrieval, oracle retrieval, and the agentic (mini-SWE-agent) baseline. Results show that resolution rates decline as difficulty and artifact complexity increase, across all three settings.

		Automated Retrieval				Oracle Retrieval				Agentic (mini-SWE-agent)			
Difficulty / artifact types	**Tasks**	**Claude**	**Qwen**	**GPT**	**Gemini**	**Claude**	**Qwen**	**GPT**	**Gemini**	**Claude**	**Qwen**	**GPT**	**Gemini**
Easy	138 (33.9%)	8.0	8.0	8.0	8.0	13.0	6.7	10.9	4.3	21.0	6.5	**24.6**	1.4
Medium	139 (34.2%)	4.3	1.4	1.4	1.4	2.9	0.7	2.2	1.4	5.0	2.2	**11.5**	0.7
Hard	130 (31.9%)	0.0	0.0	0.0	0.0	**0.8**	0.0	0.0	0.0	0.0	0.0	**0.8**	0.0
Single-type artifact	240 (59%)	6.7	5.0	5.4	5.0	9.6	4.2	7.1	3.3	12.9	4.6	**18.8**	1.3
Multi-type artifact	167 (41%)	0.6	0.6	0.0	0.6	0.0	0.0	0.6	0.0	3.0	0.6	**3.6**	0.0
Overall	407	4.18	3.19	3.19	3.19	5.65	2.46	4.42	1.97	8.85	2.95	**12.53**	0.74

Artifact diversity further exacerbates this trend. For single-type artifacts, resolution rates remain modest (5.0%–6.7%), but performance collapses to 0%–0.6% when multiple artifact types are involved. As shown in Figure 2, resolution rates also decline steeply with structural complexity: single-file tasks achieve 12.7%–15.5%, whereas performance approaches zero for tasks involving 6+ files or patches exceeding 50 lines.

Under oracle retrieval, Claude Sonnet 4.5 and GPT-5.2 improve to 5.65% and 4.42% overall, respectively. In contrast, Qwen3 Coder and Gemini 2.5 Flash degrade relative to automated retrieval (from 3.19% to 2.46% and 3.19% to 1.97%, respectively). Our manual analysis of the generated patches shows that, when given a full set of files to modify, models often introduce changes beyond the required fix, and these additional edits are associated with regressions; Claude Sonnet 4.5 and GPT-5.2 exhibit this pattern to a lesser extent. However, the Oracle setting also gives models a single greedy generation attempt rather than Automated retrieval's multi-sample, test-reranked budget, so we cannot fully attribute the regressions to over-editing rather than this reduced generation budget. Appendix F.3 provides analysis and examples of over-editing alongside this caveat.

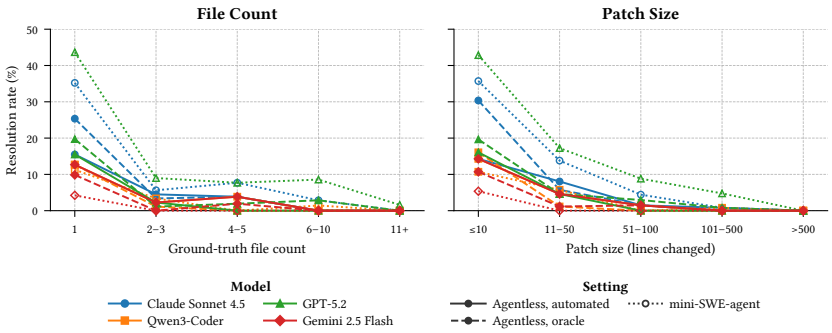

Figure 2: Resolution rates by ground-truth file count and patch size for Agentless with automated retrieval (solid), Agentless with oracle retrieval (dashed), and mini-SWE-agent (dotted, hollow markers). Single-file tasks reach 13–16% (Automated), 10–25% (Oracle), and 4–44% (mini-SWE-agent), and rates collapse for multi-file and large-patch tasks.

6.2 Scaffold Comparison

Under mini-SWE-agent, Gemini 2.5 Flash, Qwen3-Coder, Claude Sonnet 4.5, and GPT-5.2 resolve 0.74%, 2.95%, 8.85%, and 12.53% of the 407 tasks (Table 16).

Claude Sonnet 4.5 and GPT-5.2 resolve two to four times as many tasks as under Agentless (Table 2), while Qwen3-Coder resolves about as many and Gemini 2.5 Flash fewer, so the scaffold changes both the absolute rates and the ranking of the models. Gemini 2.5 Flash submits no patch on 168 tasks, 158 of them because of tool-call format errors, and resolves 1.26% of the tasks it submits (Appendix E.6). With the same agent, GPT-5.2 and Claude Sonnet 4.5 resolve 69.0% and 70.6% of SWE-bench Verified on the public leaderboard (SWE-bench Team, 2026), 5.5 and 8.0 times their rates on MobileDev-Bench (Table 18). The leaderboard runs use the same 250-step limit with a $3 cost cap instead of $2. Claude Sonnet 4.5 exhausts its step or cost budget on 38 tasks, but if every task that a model failed to submit had been resolved, GPT-5.2 would reach at most 14.5% and Claude

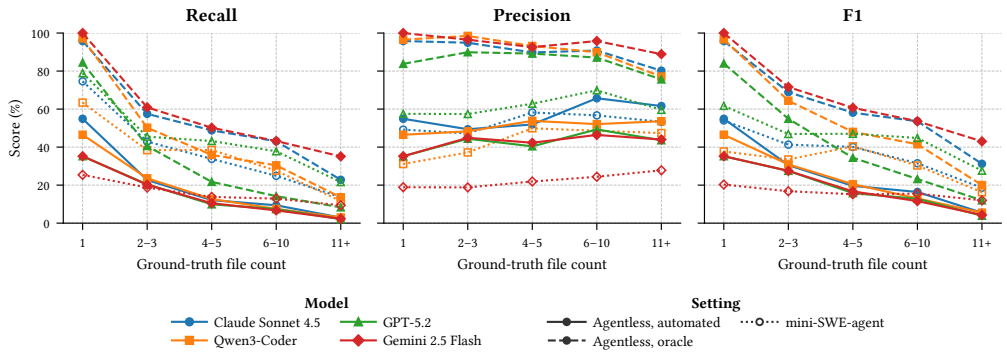

Figure 3: File-level recall, precision, and F1 by ground-truth file count for Agentless with automated retrieval (solid), Agentless with oracle retrieval (dashed), and mini-SWE-agent (dotted, hollow markers). Recall and F1 decrease as the number of files to modify grows, and oracle retrieval outperforms automated retrieval at every file count.

Sonnet 4.5 at most 18.7% (Table 16). The cost cap and submission failures therefore do not explain the gap. The comparison does not separate the mobile platform from other differences between the two benchmarks, such as language, build system, difficulty mix, and patch scope (Appendix E.7). mini-SWE-agent gives the model a raw shell without code navigation or automatic build diagnostics, so an agent framework with such tools may resolve more tasks. Appendix F.5 classifies why submitted patches fail, and Appendix E.10 reports a post-cutoff contamination check under this scaffold.

6.3 Resolution Characteristics

Problem Difficulty. Resolution rates decline with task difficulty in all three settings (Table 2). Under mini-SWE-agent, GPT-5.2 resolves 24.6% of easy tasks and 11.5% of medium tasks, and Claude Sonnet 4.5 resolves 21.0% and 5.0%. No model resolves a hard task under automated retrieval, and oracle retrieval and mini-SWE-agent each resolve a single hard task (0.8%, by Claude Sonnet 4.5 and GPT-5.2, respectively).

Artifact Type Diversity. Resolution success is strongly associated with artifact diversity. Single-artifact tasks constitute 59% of the benchmark and achieve resolution rates of 5.0%–6.7% under automated retrieval, with Claude Sonnet 4.5 and GPT-5.2 rising to 9.6% and 7.1% under oracle retrieval and to 12.9% and 18.8% under mini-SWE-agent. Multi-artifact tasks constitute 41% of the benchmark but reach at most 0.6% under either retrieval setting and at most 3.6% under mini-SWE-agent, indicating that current models struggle to coordinate fixes across heterogeneous artifact types.

File Count Effects. Figure 2 (left) shows that resolution rates drop steeply as the number of files to modify grows. Single-file tasks reach 12.7%–15.5% under automated retrieval, up to 25.4% under oracle retrieval, and 43.7% and 35.2% for GPT-5.2 and Claude Sonnet 4.5 under mini-SWE-agent. No model resolves a task with 6 or more files under automated retrieval, oracle retrieval resolves 2 of the 195 such tasks, and mini-SWE-agent resolves at most 8.6% of the tasks with 6–10 files and 1.6% of the tasks with 11 or more (both GPT-5.2). Multi-file coordination thus remains a major source of difficulty under both scaffolds.

Patch Size Effects. Figure 2 (right) shows the same decline with patch size in all three settings. Tasks whose fix changes at most 10 lines reach 14.3%–16.1% under automated retrieval, and 42.9% and 35.7% for GPT-5.2 and Claude Sonnet 4.5 under mini-SWE-agent. Beyond 50 changed lines, Agentless resolves at most 2.9% of the tasks in any bin, and mini-SWE-agent at most 8.8% for 51–100 lines and 4.8% for 101–500 lines (both GPT-5.2). No setting resolves a task whose fix changes more than 500 lines, including oracle retrieval, which supplies the files to edit, so patch generation fails as fixes grow even when localization is given.

Validation Type. At least one model resolves 32.2% of the 59 tasks validated only by F2P tests and 12.6% of the 348 tasks with N2P tests under either scaffold (Fisher exact test, $p < 0.001$), and each of the four models resolves the first group more often (Table 6). N2P validation therefore does not make tasks easier to pass.

6.4 Fault Localization Performance

Figure 3 reports recall, precision, and F1 by ground-truth file count under both retrieval settings and under mini-SWE-agent. Under automated retrieval, overall recall ranges from 13.8% to 18.6%, with precision substantially higher (42.9%–57.2%), indicating that when models do retrieve relevant context they tend to modify appropriate files, but they miss a large fraction of required modification files. Recall declines with file count, from 35–55% on single-file tasks to 2–3% on tasks with 11 or more files.

Under oracle retrieval, recall rises substantially, reaching 84.5–100% on single-file tasks, confirming that models can modify the correct files when those files are provided directly. However, oracle resolution rates peak at only 5.65% (Table 2), indicating that models edit the correct files but still fail to produce patches that pass the test suite. Recall under oracle also degrades with file count, falling to 8.3–35.1% on tasks with 11 or more files, pointing to an additional difficulty in generating changes across all provided files simultaneously.

Under mini-SWE-agent, which chooses the files to edit itself, overall recall reaches 32.7%–42.3% for Claude Sonnet 4.5, Qwen3-Coder, and GPT-5.2, about two to three times their recall under automated retrieval, and 15.4% for Gemini 2.5 Flash (Table 24). For GPT-5.2, recall under mini-SWE-agent exceeds its recall under oracle retrieval on every bin with more than one file, for example 21.6% against 8.3% on tasks with 11 or more files. As under oracle retrieval, higher recall does not produce passing patches, since the best model still resolves only 12.53% of tasks. Detailed resolution rate breakdowns, per-model analysis, and additional resolution characteristics are provided in Appendix F.

6.5 Mobile-Specific Failure Patterns

Beyond file count and patch size, we characterize failures using a taxonomy of 15 mobile-specific task labels grouped into 5 dimensions (Appendix D.6), and cross-reference it against resolution outcomes across all models and both scaffolds (Appendix F.6). Seven categories (MOBILE_SECURITY, DECLARED_CAPABILITY, MEDIA_ASSET, MOBILE_PLATFORM_SERVICES, PLATFORM_DIVERGENCE, HARDWARE_MEDIATED, and VERSION_GATED), covering 48 of 407 instances (11.8%), have zero resolved instances across every completed model-scaffold evaluation we ran, suggesting these require domain knowledge (store billing, cryptography and auth details, codec behavior, OS- and hardware-specific behavior) that is largely absent from the code itself. Resolution also falls sharply with difficulty tier, from 29.7% (Easy) to 10.8% (Medium) to 0.8% (Hard), and with the number of mobile-specific labels an instance carries, from 20.9% at one label to 0.0% at four. UI_RENDERING, the largest single category (130 instances) with a difficulty mix close to the benchmark average, resolves at only 0.22–0.75$\times$ each model's own overall rate regardless of difficulty tier; since our execution oracle uses host-side unit tests rather than emulator-based instrumentation (Appendix I), this gap reflects some mixture of the pipeline's inability to observe rendered UI state and the difficulty of navigating the mocking harnesses these tests rely on. Task category also predicts resolution success: under mini-SWE-agent (Section 6.2), unique resolved instances (union across all four models) concentrate on Bug Fix tasks (36), compared with New Feature (14) and Feature Optimization (10), despite Bug Fix being a minority of the benchmark by instance count (44.2%). Full taxonomy definitions, label distributions, and the complete failure-pattern breakdown are in Appendix D.6 and F.6.

7 Conclusion

We introduced MobileDev-Bench, a benchmark of 407 verified issue-resolution tasks from 19 production Android Native, React Native, and Flutter repositories. Four LLMs resolve at most 5.65% of tasks under a single-pass Agentless pipeline, and under mini-SWE-agent the best model (GPT-5.2) resolves 12.53%, compared with 69.0% for the same model and agent on SWE-bench Verified. Oracle retrieval raises file-level recall with little gain in resolution, so progress requires better multi-file patch generation as well as better localization. The agentic scaffold helps the two strongest models, yet under both scaffolds hard tasks and platform-dependent task categories remain almost entirely unresolved, and neither the cost cap nor submission failures account for the gap to SWE-bench Verified. MobileDev-Bench provides a reproducible basis for measuring progress in AI-assisted mobile app development.

AI USE STATEMENT

AI-based writing tools were used only for minor grammatical and wording improvements to the manuscript text. We did not use generative AI tools to generate research ideas, experimental results, analyses, or code, and did not use them for literature discovery, citation generation, or dataset curation. All technical content, experiments, and analyses were produced and verified by the authors, who take responsibility for the final content of this work, including text, claims, and artifacts.

ETHICS STATEMENT

MobileDev-Bench is built entirely from repositories, issues, and pull requests that are publicly available under open-source licenses on GitHub; no private, proprietary, or personally identifying data was collected. All included repositories satisfy an explicit open-source license filter (Appendix A.1), and per-repository license information is reported alongside the released dataset. As discussed in Appendix J and Appendix I, model-generated patches evaluated on MobileDev-Bench are not verified for security or correctness beyond passing the repository test suite, and should not be deployed to production mobile applications without human review, particularly in accessibility-, localization-, or security-sensitive components.

REPRODUCIBILITY STATEMENT

We release MobileDev-Bench publicly with all task instances, the evaluation harness, and containerized execution environments to support independent reproduction of our results. The benchmark construction pipeline, filtering criteria, and annotation procedures are described in Section 3 and Appendix A–B, including the inter-annotator agreement statistics in Appendix B.4. The execution-based evaluation protocol, retrieval settings, and metric definitions used to produce the results in Section 6 are detailed in Section 5 and Appendix E, which also documents the mini-SWE-agent configuration (Appendix E.6), API cost and token usage (Appendix E.9), and the contamination and leakage controls used during evaluation (Appendix E.10). Full per-model, per-language, per-difficulty, and mobile-specific-category resolution-rate breakdowns underlying our figures and tables are provided in Appendix F.

REFERENCES

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.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents, 2024. URL https://arxiv.org/abs/2406.11638.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick 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*, PLDI '14, pp. 259–269, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594299. URL https://doi.org/10.1145/2594291.2594299.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

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

Jialun Cao, Zhiyong Chen, Jiarong Wu, Shing chi Cheung, and Chang Xu. Javabench: A benchmark of object-oriented code generation for evaluating large language models, 2024. URL https://arxiv.org/abs/2406.12902.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. Coder: Issue resolving with multi-agent and task graphs, 2024. URL https://arxiv.org/abs/2406.01304.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, Krishna Haridasan, Ahmed Omran, Nikunj Saunshi, Dara Bahri, Gaurav Mishra, Eric Chu, Toby Boyd, Brad Hekman, Aaron Parisi, Chaoyi Zhang, Kornraphop Kawintiranon, Tania Bedrax-Weiss, Oliver Wang, Ya Xu, Ollie Purkiss, Uri Mendlovic, Ilaï Deutel, Nam Nguyen, Adam Langley, Flip Korn, Lucia Rossazza, Alexandre Ramé, Sagar Waghmare, Helen Miller, Nathan Byrd, Ashrith Sheshan, Raia Hadsell, Sangnie Bhardwaj, Pawel Janus, Tero Rissa, Dan Horgan, Alvin Abdagic, Lior Belenki, James Allingham, Anima Singh, Theo Guidroz, Srivatsan Srinivasan, Herman Schmit, Kristen Chiafullo, Andre Elisseeff, Nilpa Jha, Prateek Kolhar, Leonard Berrada, Frank Ding, Xiance Si, Shrestha Basu Mallick, Franz Och, Sofia Erell, Eric Ni, Tejasi Latkar, Sherry Yang, Petar Sirkovic, Ziqiang Feng, Robert Leland, Rachel Hornung, Gang Wu, Charles Blundell, Hamidreza Alvari, Po-Sen Huang, Cathy Yip, Sanja Deur, Li Liu, Gabriela Surita, Pablo Duque, Dima Damen, Johnson Jia, Arthur Guez, Markus Mircea, Animesh Sinha, Alberto Magni, Paweł Stradomski, Tal Marian, Vlado Galić, Wenhu Chen, Hisham Husain, Achintya Singhal, Dominik Grewe, François-Xavier Aubet, Shuang Song, Lorenzo Blanco, Leland Rechis, Lewis Ho, Rich Munoz, Kelvin Zheng, Jessica Hamrick, Kevin Mather, Hagai Taitelbaum, Eliza Rutherford, Yun Lei, Kuangyuan Chen, Anand Shukla, Erica Moreira, 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.

CS 125, UIUC. Opportunistic compilation, 2025. URL https://github.com/cs125-illinois/eMPire/wiki/Opportunistic-Compilation. eMPire project wiki. Accessed 2026-09-25.

Shihan Deng, Weikai Xu, Hongda Sun, Wei Liu, Tao Tan, Jianfeng Liu, Ang Li, Jian Luan, Bin Wang, Rui Yan, and Shuo Shang. Mobile-bench: An evaluation benchmark for llm-based mobile agents, 2024. URL https://arxiv.org/abs/2407.00993.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J. Hellendoorn. Patching as translation: the data and the metaphor. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pp. 275–286, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450367684. doi: 10.1145/3324884.3416587. URL https://doi.org/10.1145/3324884.3416587.

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

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

Moshood Abiola Fakorede and Umar Farooq. Understanding and detecting platform-specific violations in android auto apps. In *Proceedings of the 7th ACM/IEEE International Conference on Automation of Software Test*, AST '26, pp. 123–133, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400724763. doi: 10.1145/3793654.3793745. URL https://doi.org/10.1145/3793654.3793745.

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.

Gradle. Building java & jvm projects, 2025a. URL https://docs.gradle.org/current/userguide/building_java_projects.html. Accessed 2026-09-24.

Gradle. Testing in java & jvm projects, 2025b. URL https://docs.gradle.org/current/userguide/java_testing.html. Accessed 2026-09-25.

Lianghong Guo, Wei Tao, Runhan Jiang, Yanlin Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. Omnigirl: A multilingual and multimodal benchmark for github issue resolution, 2025. URL https://arxiv.org/abs/2505.04606.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

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

Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pp. 23–26. IEEE Press, 2017. ISBN 9781538615898. doi: 10.1109/ICSE-C.2017.8. URL https://doi.org/10.1109/ICSE-C.2017.8.

Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014. ISSN 1075-3583. URL https://dl.acm.org/doi/10.5555/2600239.2600241.

Sanket Mhatre, Yasharth Bajpai, Sumit Gulwani, Emerson Murphy-Hill, and Gustavo Soares. Swe-sharp-bench: A reproducible benchmark for c# software engineering tasks, 2025. URL https://arxiv.org/abs/2511.02352.

Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), January 2018. ISSN 0360-0300. doi: 10.1145/3105906. URL https://doi.org/10.1145/3105906.

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.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

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

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents, 2025. URL https://arxiv.org/abs/2504.08703.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre

Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

SWE-bench Team. Swe-bench leaderboards: Verified, 2026. URL https://www.swebench.com/#verified. Accessed 2026-09-25.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution, 2024. URL https://arxiv.org/abs/2403.17927.

Muxin Tian, Zhe Wang, Zhenwei Tang, Blair Yang, Kunlun Zhu, Honghua Dong, Hanchen Li, Xinni Xie, Guangjing Wang, and Jiaxuan You. Swe-bench mobile: Can large language model agents develop industry-level mobile applications? In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '26, pp. 8077–8087, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722592. doi: 10.1145/3770855.3818488. URL https://doi.org/10.1145/3770855.3818488.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '24, pp. 543–557, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704895. doi: 10.1145/3636534.3649379. URL https://doi.org/10.1145/3636534.3649379.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

Jiahong Xiang, Wenxiao He, Xihua Wang, Hongliang Tian, and Yuqun Zhang. Evaluating and improving automated repository-level rust issue resolution with llm-based agents, 2026. URL https://arxiv.org/abs/2602.22764. ICSE 2026.

John Yang, Carlos E. Jimenez, and the SWE-agent team. mini-swe-agent: A 100-line ai agent that solves github issues. https://github.com/SWE-agent/mini-swe-agent, 2024a.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024b. URL https://arxiv.org/abs/2405.15793.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024c. URL https://arxiv.org/abs/2410.03859.

Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, Dezhi Ran, Muhan Zeng, Bo Shen, Pan Bian, Guangtai Liang, Bei Guan, Pengjie Huang, Tao Xie, Yongji Wang, and Qianxiang Wang. Swe-bench-java: A github issue resolving benchmark for java, 2024. URL https://arxiv.org/abs/2408.14354.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of*

Software Engineering, ESEC/FSE 2019, pp. 176–187, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338917. URL https://doi.org/10.1145/3338906.3338917.

Linhao Zhang, Daoguang Zan, Quanshun Yang, Zhirong Huang, Dong Chen, Bo Shen, Tianyu Liu, Yongshun Gong, Pengjie Huang, Xudong Lu, Guangtai Liang, Lizhen Cui, and Qianxiang Wang. Codev: Issue resolving with visual data, 2024a. URL https://arxiv.org/abs/2412.17315.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024b. URL https://arxiv.org/abs/2404.05427.

Table 3: 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

Test and fix patches are separated using a deterministic, path-based rule rather than subjective judgment. We fetch the full PR diff, parse it with the `unidiff` library, and classify each changed file using case-insensitive matching against the following framework-specific naming conventions during Phase 2 (PR Collection and Filtering):

- **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.

Matching files' hunks are assigned to the test patch, and all others are assigned to the fix patch. File-level classification is appropriate here because Android, Flutter, and React Native conventionally separate production and test code through dedicated source sets, directories, and naming conventions; mixed test/production files were not a factor in our collected repositories.

We audited this classification on the full diff for all 60 unique instances resolved by mini-SWE-agent (Section 6.2) across all four evaluated models, covering 98 files classified as test files. All 98 were confirmed to be genuinely test-related (100% precision), including edge cases such as test fixtures, mock-data helpers, and support code under test directories, and none of the corresponding 170 files classified into the fix patch were found to be misclassified test files (100% recall on this sample).

Because our Android execution harness runs only host-side JVM unit tests (`:testDebugUnitTest`) and not emulator-based instrumentation tests under `androidTest/` (Section 3.4), a candidate PR whose test changes touched *only* `androidTest/` files would yield zero dynamic test executions during Phase 4 validation and be discarded by our execution-based filtering, which requires at least one captured test transition. In practice, this case did not need to be filtered out at the candidate stage: of the 407 verified instances, 14 have test-patch changes touching an `androidTest/` file, and every one of these 14 also modifies at least one `src/test/` file that is exercised by `:testDebugUnitTest`, so validation for these instances is never based solely on instrumentation-only changes.

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 3 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 4 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 407 verified instances (0.5% retention rate). The primary bottleneck is attribute filtering (Phase 2), which discards PRs that lack an explicit issue link or do not modify test files (Section 3); no file-count ceiling is enforced at this stage, and the benchmark retains instances with up to 237 modified files (Table 10). Execution validation (Phase 4) constitutes the second largest reduction, as a substantial fraction of candidates fail to build or produce non-deterministic test outcomes.

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. Manual intervention is restricted to build-environment and tooling configuration and never modifies application code, tests, or benchmark labels. 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`.

A.6 NON-EXECUTABLE State and Non-executable to Pass Semantics

The `NON-EXECUTABLE` test state arises when a test class references a type, method, or field introduced only by the fix patch; the class cannot compile against the base commit, so the test runner cannot locate or execute it. Because compilation for these languages is scoped to a module

Table 4: 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	$303_{\downarrow 3596}$	$77_{\downarrow 226}$	$68_{\downarrow 9}$
futsch1/medtimer	643	$25_{\downarrow 618}$	$12_{\downarrow 13}$	$8_{\downarrow 4}$
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	$245_{\downarrow 2483}$	$110_{\downarrow 135}$	$100_{\downarrow 10}$
tuskyapp/tusky	2,110	$28_{\downarrow 2082}$	$19_{\downarrow 9}$	$12_{\downarrow 7}$
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	$23_{\downarrow 158}$	$15_{\downarrow 8}$	$11_{\downarrow 4}$
rocketchat/rocket.chat.reactnative	3,750	$59_{\downarrow 3691}$	$5_{\downarrow 54}$	$2_{\downarrow 3}$
Total	74,195	$1{,}939_{\downarrow 72256}$	$541_{\downarrow 1398}$	$407_{\downarrow 134}$

or source set rather than a single file (Gradle, 2025a;b; CS 125, UIUC, 2025), this one unresolved reference blocks the entire target from compiling, so every test file that would otherwise be compiled together with it, whether or not that file itself changed, is also recorded as `NON-EXECUTABLE`. This is not a rare or artificial failure mode: unresolved-symbol and missing-import errors together account for roughly 70% of compiler errors observed in a large-scale empirical study of open-source Java continuous-integration builds (Zhang et al., 2019), indicating that the compilation dependency underlying most N2P transitions in MobileDev-Bench reflects a common characteristic of real-world statically typed development, not an artifact of our construction pipeline. This is distinct from `FAIL`, where the test compiles and runs but produces an incorrect result. We distinguish two kinds of `NON-EXECUTABLE`→`PASS` (N2P) transitions (Section 4). *Collateral* N2P tests are pre-existing test files, unmodified by either the test or fix patch, that were blocked purely because compilation failed elsewhere in the same target, and become executable once the fix restores the missing symbol; these account for the large majority of N2P transitions in MobileDev-Bench. *Primary* N2P tests are test classes genuinely introduced or modified alongside the fix to exercise it directly. This distinction matters because the compilation-cascade mechanism behind collateral N2P is a property of statically typed, compiled languages, where a missing symbol is a compile-time failure that blocks an entire compilation target; it is not observed in dynamically typed languages such as Python, where the same missing symbol would instead surface as a runtime error at the specific call site rather than preventing unrelated files from executing at all.

Does N2P independently validate feature-request tasks? A natural concern is that N2P-heavy instances may disproportionately be feature-request tasks validated only by a test that was written alongside the very feature it checks, which would be weaker evidence than an independent, pre-existing test. Table 5 shows this is not the dominant pattern. Bug Fix tasks have both the highest F2P rate (52.8%) and a high N2P rate (79.4%), while New Feature and Feature Optimization tasks skew toward N2P (89.7% and 91.4%) with lower F2P rates, consistent with genuinely new functionality having no pre-existing failing test to capture it as F2P.

Restricting to the strictest cut of each N2P mechanism sharpens this further. Of the 348 instances with at least one N2P test, only 36 (10.3%, 8.8% of the full benchmark) have *exclusively* primary N2P tests, i.e., every traced N2P test is a newly added file with no collateral component; among these, New Feature is the largest category (20/36, 55.6%), followed by Bug Fix (14/36, 38.9%) and Feature Optimization (2/36, 5.6%). At the other extreme, 130 instances (37.4% of instances with

Table 5: F2P and N2P coverage by task category (407 instances). N2P only counts tasks with at least one N2P test and no F2P test.

Category	N	Has ≥1 F2P (%)	Has ≥1 N2P (%)	N2P only (%)
Bug Fix	180	95 (52.8%)	143 (79.4%)	85 (47.2%)
New Feature	146	38 (26.0%)	131 (89.7%)	108 (74.0%)
Feature Optimization	81	14 (17.3%)	74 (91.4%)	67 (82.7%)
Total	407	147 (36.1%)	348 (85.5%)	260 (63.9%)

N2P) have *exclusively* collateral N2P tests, i.e., every traced N2P test is a pre-existing, untouched file; among these, Bug Fix is the largest category (57/130, 43.8%), close to its 44.2% share of the full benchmark, followed by Feature Optimization (40/130, 30.8%) and New Feature (33/130, 25.4%). The self-validation concern is therefore confined to a small minority of instances (8.8% of the benchmark), while the much larger collateral-N2P population is validated by pre-existing tests unrelated to the fix and is not disproportionately feature-request work. Repositories that follow a test-driven development workflow may still exhibit exclusively N2P transitions with zero F2P counts, because their test patches introduce test classes that are entirely absent from the base commit; this is the newly-introduced-test case rather than the more common compilation-dependency case.

Are N2P-validated instances easier to resolve? If N2P provided a weaker or more easily satisfied validation signal than F2P, instances relying on it might show inflated resolution rates. We test this directly by comparing resolution rate on the 59 instances with zero N2P transitions (validated purely by F2P) against the 348 instances with at least one N2P transition, counting an instance as resolved if any model resolves it under either scaffold (Agentless or mini-SWE-agent). Table 6 shows the opposite of the inflation concern: every model resolves the non-N2P subset at a substantially higher rate than the N2P-having subset, and in aggregate the non-N2P rate (32.2%) is roughly $2.5\times$ the N2P-having rate (12.6%). N2P-validated instances are, if anything, harder to resolve than the benchmark's most strictly validated instances, not easier.

Table 6: Resolution rate by N2P reliance. Tasks with no N2P test are validated purely by F2P tests. A task counts as resolved for a model when that model resolves it under either scaffold, and for the last row when any model does. p is a two-sided Fisher exact test.

Model	Non-N2P (n=59)	Has N2P (n=348)	p
Claude Sonnet 4.5	11 (18.6%)	32 (9.2%)	0.038
GPT-5.2	15 (25.4%)	36 (10.3%)	0.003
Gemini 2.5 Flash	4 (6.8%)	10 (2.9%)	0.130
Qwen3-Coder	7 (11.9%)	13 (3.7%)	0.016
Any model/scaffold	19 (32.2%)	44 (12.6%)	<0.001

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 4 and 5 visualise the score distributions and dataset composition, respectively.

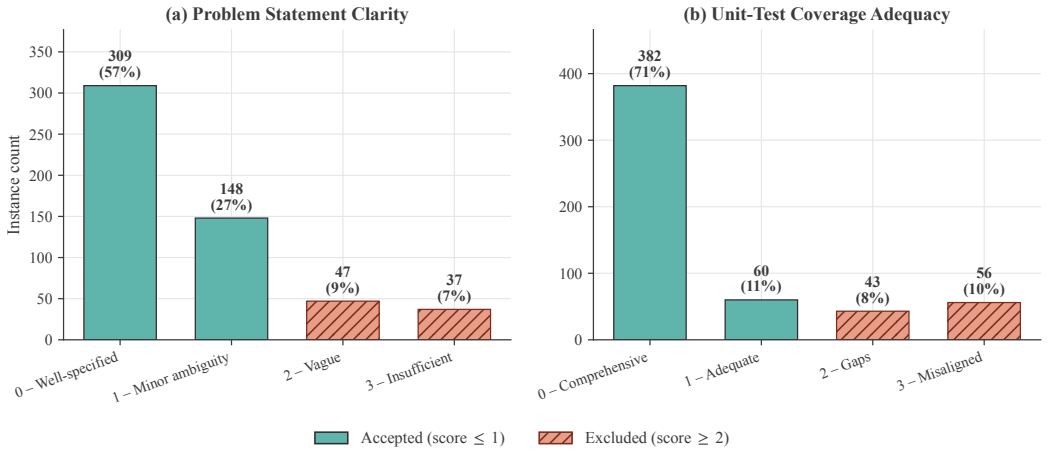

Figure 4: Distribution of manual annotation scores for **(a)** problem-statement clarity and **(b)** unit-test coverage adequacy, across all 541 annotated candidate instances. Green bars (scores 0–1) indicate instances accepted into the benchmark; hatched red bars (scores 2–3) indicate excluded instances. 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).

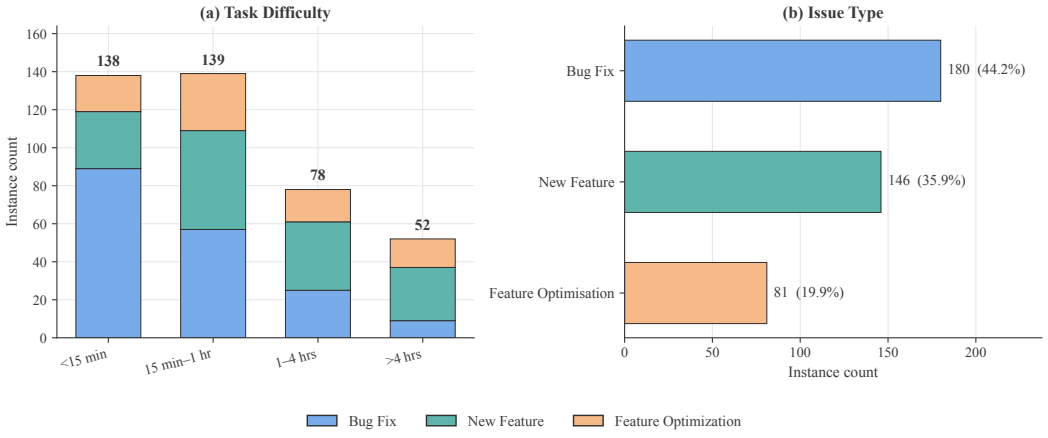

Figure 5: Task characteristics of the 407 verified benchmark instances. **(a)** Difficulty distribution stacked by issue type, using the four time-based difficulty labels assigned during annotation. **(b)** Overall issue-type breakdown. The benchmark spans the full difficulty range and covers all three major categories of mobile development activity.

B.1 Annotation Results

Among the 541 annotated candidates, 309 instances (57.1%) have unambiguously well-specified problem statements (score 0) and 148 (27.4%) have minor ambiguity but a clear interpretation (score 1). The remaining 84 (15.5%) are excluded by problem-statement score: 47 receive score 2 (vague) and 37 score 3 (insufficient information).

Test-coverage adequacy shows a higher baseline: 382 instances (70.6%) receive score 0 and 60 (11.1%) score 1. A total of 99 instances (18.3%) are excluded for inadequate test coverage, of which 43 score 2 and 56 score 3. Forty-nine instances (9.1%) are excluded for *both* reasons simultaneously; by inclusion-exclusion, this yields 84 + 99 − 49 = 134 total exclusions (24.8%) and 407 verified benchmark instances (75.2%).

B.2 Task Characteristics

Figure 5 summarises the 407 verified instances by estimated resolution time and issue type. Difficulty follows the four-band rubric described in Section 3 (Phase 5): 138 instances (33.9%) are estimated to take under 15 minutes, 139 (34.2%) 15 minutes and one hour, 78 (19.2%) between one and four hours, and 52 (12.8%) more than four hours. We report a three-tier Easy/Medium/Hard split, with Hard combining the 1–4hr and >4hr bands (130 instances). This grouping is used only for headline reporting and leaves the original four-band annotations unchanged. By issue type, 180 instances (44.2%) are Bug Fixes, 146 (35.9%) are New Features, and 81 (19.9%) are Feature Optimizations.

B.3 Difficulty Versus Scale

Because the difficulty rubric (Section 3, Phase 5) explicitly incorporates the number of files modified as one of its judgment criteria, difficulty and scale-related factors are correlated by construction: across all 407 instances, difficulty (as an ordinal Easy/Medium/Hard variable) has a Spearman correlation of $\rho = 0.67$ ($p < 0.001$) with file count and $\rho = 0.73$ ($p < 0.001$) with patch size (lines changed). This raises the question of whether difficulty carries information beyond what file count or patch size already provide, or is simply a redundant proxy for scale.

We find that it is not redundant. The raw correlation between difficulty and resolution (pooling resolutions across both the Agentless and mini-SWE-agent scaffolds and all four models) is $r = -0.34$ ($p < 0.001$). Controlling for file count via partial correlation leaves $r = -0.30$ ($p < 0.001$), and controlling for patch size leaves $r = -0.31$ ($p < 0.001$); in both cases the relationship is only modestly attenuated, indicating that difficulty predicts resolution success even after accounting for scale. Table 7 makes the same point non-parametrically: within fixed file-count strata, resolution rate still declines with difficulty tier. Under mini-SWE-agent, single-file Easy tasks resolve at 53.7% (29/54) versus 0% (0/2) for Hard; among tasks with 3 or more files, Easy still resolves at 7.5% (4/53) versus 0.8% (1/126) for Hard. Difficulty therefore captures conceptual and framework-knowledge complexity beyond what raw scale measures alone would predict.

Table 7: Resolution rate by difficulty tier within fixed file-count strata, Agentless (Automated retrieval, union across all four models) versus mini-SWE-agent (union across all four models).

	Agentless			Agentic (mini-SWE-agent)		
File-count stratum	**Easy**	**Medium**	**Hard**	**Easy**	**Medium**	**Hard**
1 file	33.3% (18/54)	20.0% (3/15)	0.0% (0/2)	53.7% (29/54)	40.0% (6/15)	0.0% (0/2)
2 files	12.9% (4/31)	0.0% (0/16)	0.0% (0/2)	25.8% (8/31)	6.2% (1/16)	0.0% (0/2)
3+ files	0.0% (0/53)	2.8% (3/108)	0.0% (0/126)	7.5% (4/53)	10.2% (11/108)	0.8% (1/126)

B.4 Inter-Annotator Agreement

Both annotators independently labeled all 407 verified instances, and the final labels reported throughout the paper are assigned after the two annotators discussed and resolved their disagreements; the statistics below are computed on the independent, pre-discussion labels.

Difficulty is an ordered four-level scale (<15min, 15min–1hr, 1–4hr, >4hr), for which Cohen's κ is not an appropriate statistic: it treats every disagreement as equally severe, penalizing a <15min vs.

15min–1hr disagreement exactly as much as a <15min vs. >4hr disagreement, which understates agreement on an ordered scale. We instead report quadratic-weighted κ and Krippendorff's α with an ordinal difference function, which scale the penalty to the distance between labels. Issue type is a nominal scale, for which Cohen's κ is appropriate, and we report it there.

We report two agreement percentages for difficulty. *Exact level* is the share of instances assigned the identical difficulty level by both annotators. *Adjacent level* is the share assigned either the identical level or an adjacent level (e.g., one annotator choosing <15min and the other 15min–1hr), measuring whether annotators place a task at the same point on the scale up to a single boundary.

Table 8: Inter-annotator agreement, pre-discussion labels (407 instances). 95% CIs are 5,000-sample bootstrap percentile intervals.

Dimension	Metric	Agreement	Coefficient
Difficulty (ordinal)	Exact level	48.9% (199/407)	Quadratic-weighted κ = 0.588 [0.521, 0.646]
	Adjacent level	92.1% (375/407)	Krippendorff's α = 0.592 [0.526, 0.651]
Issue type (nominal)	Overall	77.9% (317/407)	Cohen's κ = 0.658 [0.596, 0.719]
	— Bug fix	—	Dice = 0.749
	— New feature	—	Dice = 0.649
	— Feature Optimization	—	Dice = 0.540

The two ordinal coefficients for difficulty converge (0.588 and 0.592), which supports the reliability of the estimate. Per-class figures for issue type are Dice coefficients.

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 12.9 modified files and 334.6 changed lines.

Patch Complexity. Figure 6 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., 2024c), 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 per-repository median hunk count (27.5) and changed line count (342.6) across all five benchmarks. This is a repository-level statistic (the median, across MobileDev-Bench's 19 repositories, of each repository's own median patch size) and is distinct from the instance-level statistics in Table 10, which are computed across all 407 instances directly (mean 27.5 hunks / median 12 hunks; mean 334.6 lines / median 93 lines). The per-repository and per-instance hunk statistics coincide numerically (27.5 in both); the line statistics do not (342.6 vs. 334.6), since patch size varies substantially across repositories and the two aggregation levels are not expected to agree in general. 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.

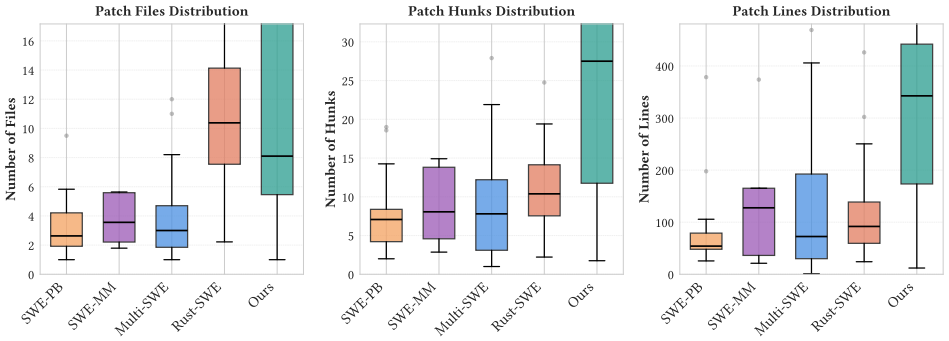

Figure 6: Distribution of patch complexity metrics (files modified, hunks, changed lines) across five benchmarks (SWE-PB: SWE-PolyBench; SWE-MM: SWE-bench Multimodal; Multi-SWE: Multi-SWE-bench; Rust-SWE: Rust-SWE-bench; Ours: MobileDev-Bench). Each data point is a repository's own median value across its instances, so the box for each benchmark shows the distribution of per-repository medians (the line inside each box is the median of those per-repository medians). Y-axes are capped at the 90th percentile for visual clarity.

Design Dimensions and Scale. MobileDev-Bench contains 407 manually verified instances from 19 repositories across four languages, comparable in scale to SWE-PolyBench Verified (Rashid et al., 2025) (382 instances, 20 repositories) and roughly four-fifths of SWE-bench Verified (OpenAI, 2024) (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 (308 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., 2024c) and multilingual variants (Zan et al., 2025; Rashid et al., 2025; Kabir, 2025), primarily measure an LLM's capacity to resolve isolated logic errors within generic programming environments. 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 fundamentally change this control flow. 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 excel at pure algorithmic synthesis often falter when required to maintain synchronization between a deeply nested source logic and external framework resources.

Scale of Modifications: As detailed in the comparison in Figure 6, 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, setting a new bar for evaluation environments emphasizing integration robustness.

Table 9: Framework-wise instance distribution and code change characteristics in MobileDev-Bench

	Instance Dist.		Fix Patch Statistics				Test Patch Statistics			
Framework	**Repos**	**Inst.**	**Avg. #F**	**Avg. #H**	**Lines+**	**Lines–**	**Avg. #F**	**Avg. #H**	**Lines+**	**Lines–**
Android Native	13	328	14.0	28.8	222.3	119.6	6.1	15.9	141.8	50.7
Flutter	2	63	7.6	18.3	204.6	50.6	4.0	13.9	234.3	43.5
React Native	4	16	9.5	36.2	314.8	181.8	3.4	15.2	102.7	70.1

Table 10: Summary statistics for MobileDev-Bench.

	Issue	Codebase		Fix Patch			Tests		
	Tokens	#Files	#Lines (K)	#Lines	#Files	#Hunks	F2P	N2P	P2P
Mean	308	4,605	389	334.6	12.9	27.5	1.1	31.1	11.9
Median	173	6,342	365	93	5	12	0	11	0
Max	11,501	7,923	2,652	11,915	237	592	30	375	267

D Dataset Analysis

This appendix reports the dataset statistics that Section 4 summarizes.

D.1 Summary Statistics

Per-repository statistics are reported in Table 11. 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 100 per repository, with the four largest contributors being thunderbird-android (100), wordpress-android (84), element-x-android (68), and zulip-flutter (51), together accounting for 74.4% of the dataset.

Table 10 reports aggregate statistics across the dataset. Issue descriptions average 308 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.

D.2 Framework-wise Distribution and Code Change Characteristics

Table 9 reports per-framework instance counts and patch statistics. Android Native dominates the dataset (328 instances, 80.6%), reflecting the larger number of mature open-source Android repositories. Flutter contributes 63 instances (15.5%) from two high-activity repositories, and React Native contributes 16 instances (3.9%) from four repositories. The fix patch statistics reveal that Android Native instances involve substantially more code changes on average (14.0 modified files, 28.8 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.

D.3 Distribution of Artifact Type Combinations

Mobile app fixes often require coordinated changes across heterogeneous artifact types. We classify modified files in each patch into eight categories:

- **Source:** Application source code (`.kt`, `.java`, `.dart`, `.ts`, `.tsx`, `.js`).
- **Resource:** UI and media assets (`res/` subtree, images, drawables, layouts).
- **i18n:** Localized string resources (Android `values-*/strings.xml`, Flutter `.arb`).
- **Build:** Build scripts (`build.gradle`, `libs.versions.toml`, `*.kts`).
- **Manifest:** App-level declaration files (`AndroidManifest.xml`, `pubspec.yaml`).
- **Config:** CI/CD workflows and configuration (GitHub Actions, linter and formatter configs).

Table 11: 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: Non-executable 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	68	159	368.4	40.0	17.8	2.0	27.9	23.4
medtimer	556	41k	8	152	454.0	33.2	17.2	0.5	4.6	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	100	411	342.6	27.5	16.4	0.8	30.1	3.1
tusky	1,593	169k	12	318	46.5	4.8	2.4	1.4	18.5	7.8
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	11	135	429.7	32.8	8.1	3.5	0.2	0.4
rocket.chat.reactnative	1,601	134k	2	442	24.5	9.0	5.0	3.5	3.5	0.5

- **Docs:** Project documentation (Markdown, changelog fragments, license files).
- **Other:** Files not matched by any of the above.

Source files appear in nearly all instances (99.5%). However, many fixes also require non-source changes: 20.4% modify resources, 17.2% modify localization files, 18.4% modify build files, and 3.7% modify manifest files. In total, 41.0% of instances modify two or more artifact types and 20.9% modify three or more, demonstrating that cross-artifact coordination is a common property of mobile app fixes.

Table 12 reports the distribution of artifact type combinations present in fix patches. The majority of instances (58.5%) modify only source files, and 59.0% are single-artifact overall, indicating that the primary challenge is code comprehension and logic repair. However, 41.0% 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 (8.1%), 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.4 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 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

Table 12: Distribution of artifact type combinations in MobileDev-Bench. 41.0% of instances require coordinated changes across multiple artifact types. Framework columns report count (% of that framework's own n=328/63/16 instances for Android Native/Flutter/React Native, respectively).

	Overall		% by Framework		
Combination	**#**	**%**	**Android Native**	**Flutter**	**React Native**
Single Artifact Type					
Src	238	58.5	183 (55.8%)	47 (74.6%)	8 (50.0%)
Res	1	0.3	1 (0.3%)	0 (0.0%)	0 (0.0%)
Build	1	0.2	1 (0.3%)	0 (0.0%)	0 (0.0%)
Subtotal	240	59.0	185 (56.4%)	47 (74.6%)	8 (50.0%)
Two Artifact Types					
Src + Build	33	8.1	33 (10.0%)	0 (0.0%)	0 (0.0%)
Src + Res	15	3.7	15 (4.6%)	0 (0.0%)	0 (0.0%)
Src + i18n	15	3.7	0 (0.0%)	13 (20.6%)	2 (12.5%)
Src + Docs	13	3.2	13 (4.0%)	0 (0.0%)	0 (0.0%)
Src + Other	4	0.9	3 (0.9%)	0 (0.0%)	1 (6.3%)
Other	2	0.5	1 (0.3%)	1 (1.6%)	0 (0.0%)
Subtotal	82	20.1	65 (19.8%)	14 (22.2%)	3 (18.8%)
Three or More Artifact Types					
Src + Res + i18n	27	6.7	27 (8.2%)	0 (0.0%)	0 (0.0%)
Src + Res + i18n + Build	11	2.7	11 (3.4%)	0 (0.0%)	0 (0.0%)
Src + Build + Docs	7	1.7	6 (1.8%)	1 (1.6%)	0 (0.0%)
Src + Manifest + Config	5	1.2	0 (0.0%)	1 (1.6%)	4 (25.0%)
Src + Res + i18n + Docs	5	1.2	5 (1.5%)	0 (0.0%)	0 (0.0%)
Other	30	7.4	29 (8.9%)	0 (0.0%)	1 (6.2%)
Subtotal	85	20.9	78 (23.8%)	2 (3.2%)	5 (31.2%)
Total (Single)	**240**	**59.0**	**185 (56.4%)**	**47 (74.6%)**	**8 (50.0%)**
Total (Multi)	**167**	**41.0**	**143 (43.6%)**	**16 (25.4%)**	**8 (50.0%)**

co-modification appears in 18.8% of React Native instances, reflecting gradual TypeScript adoption in existing JavaScript codebases.

D.5 Background Knowledge Required by Framework

Figure 7 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 (36.0%), reflecting widespread use of `ViewModel`, `Room`, `WorkManager`, and Navigation across the benchmark repositories. Declarative UI knowledge (Jetpack Compose) is needed in 32.9% of Android Native instances, Asynchronous/Concurrency patterns (Kotlin coroutines, `Flow`) in 29.6%, Platform SDK APIs in 16.8%, and Dependency Injection via Hilt or Dagger in 16.5%. 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 ($n = 16$): UI framework patterns appear in 43.8% of instances and Async/Concurrency in 25.0%, with Architecture Components in 18.8% and Platform/Native APIs in 12.5%; Dependency Injection and Networking domains register zero, plausibly reflecting these repositories' reliance on abstracted, library-level patterns not captured by source-level identifiers alone.

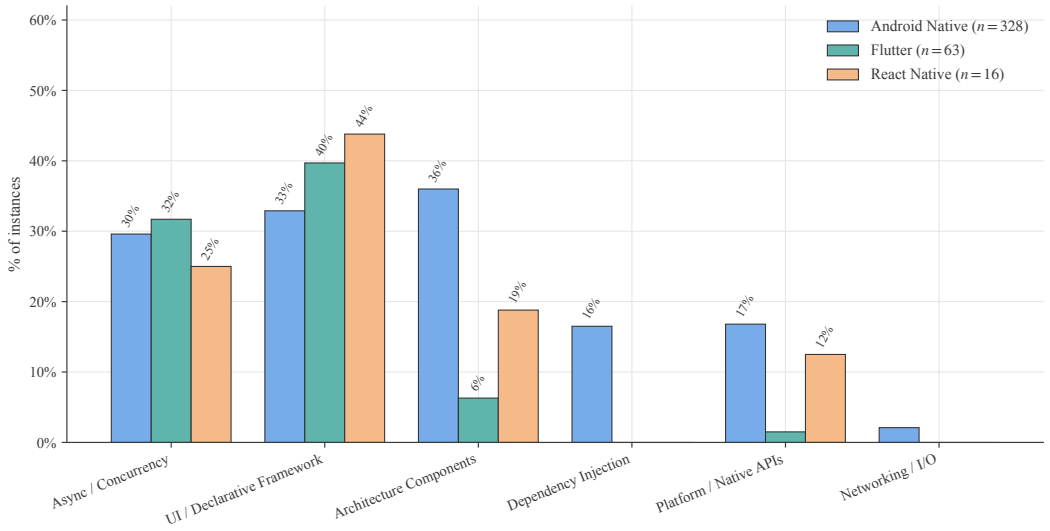

Figure 7: Percentage of instances per framework whose fix patches involve each technical knowledge domain, inferred from import statements, annotations, and identifiers in the added lines of fix patches. An instance may span multiple domains.

D.6 Mobile-Specific Task Taxonomy

Framework-level statistics (artifact type, language, background knowledge domain) characterize *what kind of file* a fix touches, but not *what mobile-specific concern* the fix addresses. To support the failure-pattern analysis in Section 6.5 and Appendix F.6, we introduce a complementary taxonomy of 15 fine-grained, mobile-specific task labels, grouped into 5 higher-level dimensions (Table 13). We jointly label all 407 instances; each instance carries one to four of the 15 labels, since a real fix usually addresses more than one mobile-specific concern at once. Instances carrying none of the 15 labels are classified as platform-agnostic (58 instances, 14.2%) — these include code refactors, library or dependency updates, and regex or text-parsing fixes that would look the same in any codebase regardless of platform. The remaining 349 instances (85.8%) carry at least one mobile-specific label.

Table 13: Mobile-specific task characterization taxonomy: 15 labels grouped into 5 dimensions. Instance counts are the number of instances carrying at least one label from that dimension; dimensions can overlap since an instance may carry labels from more than one dimension.

Dimension	Labels	Instances
Lifecycle, execution, and state management	APP_LIFECYCLE, BACKGROUND_EXECUTION_AND_ASYNC, MOBILE_STATE_SYNCHRONIZATION	192 (47.2%)
UI, resources, and multimedia presentation	UI_BINDING, UI_RENDERING, RESOURCE_QUALIFIED, MEDIA_ASSET	158 (38.8%)
External device, services, and runtime contracts	APP_BINDING/DYNAMIC_CONTRACT_BINDING, HARDWARE_MEDIATED, PUSH_NOTIFICATION, MOBILE_PLATFORM_SERVICES	110 (27.0%)
Platform and OS integration	VERSION_GATED, DECLARED_CAPABILITY, PLATFORM_DIVERGENCE	21 (5.2%)
Security and trust-sensitive workflows	MOBILE_SECURITY	16 (3.9%)

Table 14 gives the full definition used to apply each of the 15 labels during annotation.

Table 14: Full definitions for the 15 mobile-specific task labels.

Label	Definition
VERSION_GATED	Behavior must differ across OS versions live in the field. The change branches on API level at runtime or compile time.
DECLARED_CAPABILITY	A manifest declaration (permission / intent filter / exported flag / service entry) must match the runtime code that depends on it.
APP_LIFECYCLE	The change is triggered by or directly addresses a foreground / background / killed state transition or app-startup event. Covers Activity or Fragment lifecycle callbacks, Flutter `initState`/`dispose`, React Native `AppState`, process death recovery, back-stack management, and first-launch or upgrade detection.
RESOURCE_QUALIFIED	The change must be complete across all OS-selected resource variants (locale, density, night mode, RTL, screen width, API level).
APP_BINDING/DYNAMIC_CONTRACT_BINDING	Source code must match an external API or data contract with no end-to-end compile-time type check, so a field name, type, or structural mismatch passes compilation but causes silent runtime failures. Covers generated serialization artifacts (`json_serializable`, `@JsonKey`, Room, Pigeon-generated code) and untyped API contracts (GraphQL query/mutation strings that must match the server schema, REST response field names in manual `fromJson` parsing).
UI_BINDING	Source code must match a generated UI artifact with no end-to-end compile-time type check, so an ID typo or structural mismatch passes compilation but causes a silent runtime crash or missing binding. Covers ViewBinding, DataBinding, `R.id`, navigation-graph destination IDs, Compose slot parameters, and React Native `StyleSheet` keys.
UI_RENDERING	The change addresses what is visually rendered: layout, draw, animation, gesture recognition, viewport insets, view recycling, or mobile-native UI surfaces (bottom sheets, action sheets, modals, toasts).
BACKGROUND_EXECUTION_AND_ASYNC	The change addresses concurrency, ordering, cancellation, race conditions, retry-backoff, or polling behavior in async operations (coroutines, WorkManager, RxJava, Promises, long-poll).
MOBILE_STATE_SYNCHRONIZATION	The change addresses observable state held, derived, or propagated through the app, including adding or correcting state in a ViewModel, BLoC, Redux, or custom store, Room-backed model, or reactive event-driven state sync with a server.
MEDIA_ASSET	The change involves audio or video format compatibility, media metadata parsing (chapters, transcripts, podcast feeds), codec or audio-effect detection, or media session management.
HARDWARE_MEDIATED	Correctness depends on a sensor, radio, or peripheral the code cannot fully determine or simulate (GPS, camera, microphone, Bluetooth, NFC, barometer, cell radio, biometrics).
PUSH_NOTIFICATION	The change is specific to the FCM or APNs stack: token registration, notification channel configuration, sound or vibration settings, notification actions, grouping, or tap routing.
PLATFORM_DIVERGENCE	In a cross-platform app, the change must explicitly fork logic or UI by platform (iOS vs. Android) using platform checks or platform channels (Pigeon, native modules).
MOBILE_SECURITY	The change involves mobile-specific cryptography or authentication: end-to-end encryption with device-bound keys, OIDC/OAuth mobile flows, QR-code device verification, or mobile wallet security.
MOBILE_PLATFORM_SERVICES	UI or logic that integrates with a mobile platform's billing, subscription, in-app purchase, or app-store review system (Google Play Billing, Apple StoreKit, `SKStoreReviewController`); applies when the change exists specifically to drive store-mediated payment or review flows.

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

MobileDev-Bench is evaluated with two scaffolds, Agentless (Xia et al., 2024) and mini-SWE-agent (Yang et al., 2024a) (Appendix E.6), and one execution harness validates the patches of both. 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.

Patches from both scaffolds are validated with the same Docker-based test harness. For each task instance, we: (1) clone the repository at the base commit, (2) apply the test patch (Section 3) followed by the model-generated patch, (3) execute the repository test suite in an isolated Docker container, and (4) compare test outcomes against the ground truth. Applying the test patch first ensures that any test files it introduces or modifies exist before the test command runs. 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.

This protocol follows SWE-bench (Jimenez et al., 2024) and Multi-SWE-bench (Zan et al., 2025). Resolution rates on different benchmarks are comparable only under the same model and agent, and Appendix E.7 makes this comparison with SWE-bench Verified.

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.

Beyond the parser swap, the core Agentless BM25-plus-embedding retrieval pipeline is unchanged; we use a larger embedding model (`text-embedding-3-large`) in place of the original implementation's default, while retrieval and ranking logic are otherwise unmodified. Non-code artifacts rely on raw-text retrieval rather than the structured, tree-sitter-derived signals available for source files, since they lack function-level boundaries to index. Oracle retrieval, which bypasses localization entirely, improves the best-case resolution rate only from 4.18% to 5.65% (Table 15), and two of the four models regress under oracle retrieval, a pattern we associate with over-editing under full-file context, though a confounding reduction in generation budget under Oracle retrieval means we

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

Table 15: Resolution rates by ground truth file count under Automated retrieval, Oracle retrieval, and the agentic (mini-SWE-agent) baseline. Rate is resolved / 407 (the fixed benchmark size), with unsubmitted or empty-patch predictions counted as unresolved.

		Resolved				Rate (%)			
#Files	**Tasks**	**C. Sonnet 4.5**	**Qwen3**	**GPT 5.2**	**G. 2.5**	**C. Sonnet 4.5**	**Qwen3**	**GPT 5.2**	**G. 2.5**
Automated Retrieval									
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	125	0	0	0	0	0.0	0.0	0.0	0.0
Overall	407	17	13	13	13	4.18	3.19	3.19	3.19
Oracle Retrieval									
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	2.9	0.0
11+ files	125	0	0	0	0	0.0	0.0	0.0	0.0
Overall	407	23	10	18	8	5.65	2.46	4.42	1.97
Agentic (mini-SWE-agent)									
1 file	71	25	8	31	3	35.2	11.3	43.7	4.2
2–3 files	89	5	3	8	0	5.6	3.4	9.0	0.0
4–5 files	52	4	0	4	0	7.7	0.0	7.7	0.0
6–10 files	70	2	1	6	0	2.9	1.4	8.6	0.0
11+ files	125	0	0	2	0	0.0	0.0	1.6	0.0
Overall	407	36	12	51	3	8.85	2.95	12.53	0.74

cannot fully attribute the regression to over-editing alone (Appendix F.3); this suggests localization limitations contribute to, but do not fully explain, the low resolution rates.

E.3 Model Selection

We evaluate four contemporary language models: Claude Sonnet 4.5 (Anthropic, 2025) (Anthropic's proprietary model with 1M token context window), GPT-5.2 (OpenAI, 2025) (OpenAI's 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].

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.

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). This differs from the Automated setting not only in localization (full files vs. localized snippets) but also in generation budget (single greedy sample vs. greedy plus multiple diverse samples with test-based re-ranking); we discuss this as a confound in interpreting the Oracle regression observed for Qwen3-Coder and Gemini 2.5 Flash in Appendix F.3.

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.

[3] `https://openrouter.ai/`

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:

$$\text{Recall}_t = \frac{|F_t^{GT} \cap F_t^{Pred}|}{|F_t^{GT}|}$$

$$\text{Precision}_t = \frac{|F_t^{GT} \cap F_t^{Pred}|}{|F_t^{Pred}|}, \quad \text{Precision}_t := 0 \text{ if } |F_t^{Pred}| = 0$$

$$\text{F1}_t = \frac{2 \cdot \text{Precision}_t \cdot \text{Recall}_t}{\text{Precision}_t + \text{Recall}_t}, \quad \text{F1}_t := 0 \text{ if } \text{Precision}_t + \text{Recall}_t = 0$$

Every task in MobileDev-Bench has at least one ground-truth modified file, so $|F_t^{GT}| = 0$ never occurs and Recall_t is always well-defined.

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

$$\text{Recall} = \frac{1}{T}\sum_{t=1}^{T}\text{Recall}_t$$

$$\text{Precision} = \frac{1}{T}\sum_{t=1}^{T}\text{Precision}_t$$

$$\text{F1} = \frac{1}{T}\sum_{t=1}^{T}\text{F1}_t$$

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.

Table 16: mini-SWE-agent resolution rate by model, computed over the full 407-instance benchmark following standard SWE-bench-style reporting.

Model	Patches Submitted	Resolved	Resolution Rate (%)
Gemini 2.5 Flash	239	3	0.74
Qwen3-Coder	404	12	2.95
Claude Sonnet 4.5	367	36	8.85
GPT-5.2	399	51	12.53

- **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.

E.6 mini-SWE-agent Configuration

The interactive scaffold (Section 5) is mini-SWE-agent (Yang et al., 2024a), run with the same four models, accessed via OpenRouter, with `max_iterations=250` and a per-instance `cost_limit=$2.0`. An instance counts as evaluated if the agent produced a final patch (or exhausted its iteration/cost budget) within this configuration; evaluation coverage varies by model because some runs did not complete for every instance within these limits.

Coverage is not uniform across models. Table 17 breaks down, for each model, why the remaining instances never reached evaluation: the agent exhausted its iteration/cost budget without submitting (*Limits Exceeded*), exceeded the per-instance wall-clock budget (*Time Exceeded*), repeatedly failed to emit a well-formed tool call (*Tool-Format Error*), hit a malformed or missing API response (*API/Runtime Error*), or submitted with an empty diff (*Empty Patch*).

Gemini 2.5 Flash generated patches for only 239 of 407 instances (58.7%); 158 of the remaining 168 (94.0%) failed due to repeated tool-call formatting errors during tool interaction, a failure mode reported elsewhere in Gemini tool-use workflows, with a further 4 lost to malformed API responses. We include these failures in the overall evaluation, since the benchmark measures end-to-end resolution, but we also report the rate conditioned on successful patch generation to isolate cases where the agent produced a patch at all: 3 of 239 (1.26%) conditional on patch generation, versus 3 of 407 (0.74%) overall. The gap between these two numbers indicates that a meaningful share of Gemini 2.5 Flash's low resolution rate under this scaffold reflects tool-use reliability rather than repair capability conditioned on producing output. Claude Sonnet 4.5's 40 non-evaluated instances are almost entirely (38 of 40) budget exhaustion rather than tool-use failure, and GPT-5.2 and Qwen3-Coder rarely fail to reach evaluation at all (8 and 3 instances respectively), indicating that coverage failure is a Gemini-specific reliability issue rather than a property of the agentic scaffold itself.

E.7 Same-Model, Same-Harness Comparison with SWE-bench Verified

To isolate scaffold sensitivity from mobile-domain difficulty, we compare mini-SWE-agent's resolution rate on MobileDev-Bench against its publicly reported resolution rate on SWE-bench Veri-

Table 17: Breakdown of the 407 − Patches Submitted instances (Table 16) by why they never reached evaluation.

Model	Submitted	Limits Exceeded	Time Exceeded	Tool-Format Error	API/Runtime Error	Empty Patch
Claude Sonnet 4.5	367	38	1	0	0	1
GPT-5.2	399	0	4	0	4	0
Gemini 2.5 Flash	239	1	0	158	4	5
Qwen3-Coder	404	3	0	0	0	0

fied (SWE-bench Team, 2026) for the same four models. Table 18 shows the same agentic scaffold and the same underlying models resolve substantially more of SWE-bench Verified than MobileDev-Bench.

Table 18: mini-SWE-agent resolution rate on SWE-bench Verified (SWE-bench Team, 2026) versus MobileDev-Bench, same scaffold and same models.

Model	SWE-bench Verified (%)	MobileDev-Bench (%)	Ratio
Claude Sonnet 4.5	70.60	8.85	8.0×
GPT-5.2	69.00	12.53	5.5×
Qwen3-Coder	55.40	2.95	18.8×
Gemini 2.5 Flash	28.73	0.74	38.8×

Our mini-SWE-agent integration leaves the library's agent loop, bash tool-calling protocol, and retry logic unmodified. We use mobile-framework-aware system and instance prompts, set through the library's own configurable prompt-template fields, with `step_limit=250` and `cost_limit=$2.0` due to budget constraints, alongside benchmark-integrity infrastructure (network egress filtering, `.git` history scrubbing) that runs entirely outside the library around the Docker environment it manages.

E.8 Multimodal (Vision) Ablation Setup

We identify a fixed 98-instance subset of MobileDev-Bench whose linked GitHub issue includes at least one image attachment (screenshot, UI mockup, or crash capture), drawn from the same annotation pass used elsewhere in the paper. For each instance in this subset, we embed the attached image URL(s) directly into the task prompt using mini-SWE-agent's multimodal content extension, which expands an `<MSWEA_MULTIMODAL_CONTENT>` tag into an image content block the underlying model API accepts; all other scaffold settings (`max_iterations=250`, `cost_limit=$2.0`) are identical to the text-only mini-SWE-agent runs (Appendix E.6), so the only difference between the text-only and multimodal conditions is the presence of the image(s) in the prompt.

We restrict this ablation to Claude Sonnet 4.5 and GPT-5.2, the two evaluated models with native multimodal support in the scaffold configuration we use; we did not run a Gemini 2.5 Flash or Qwen3-Coder multimodal condition for this ablation. Coverage is not complete for either model: Claude Sonnet 4.5 submitted patches for 83 of 98 instances (15 did not reach evaluation, predominantly agent budget exhaustion), and GPT-5.2 submitted patches for 97 of 98 instances (1 instance, a `thunderbird-android` task, failed to evaluate after two independent attempts due to unrelated Docker image-pull and layer-extraction infrastructure errors, not a model or scaffold failure). Consistent with our fixed-denominator convention elsewhere in the paper, we score both conditions in Table 19 over the full 98-instance subset, counting any instance that did not reach evaluation as unresolved.

The text-only column in Table 19 is each model's resolved set from the main mini-SWE-agent run (Table 16), restricted to this same 98-instance subset, so the comparison isolates the effect of adding images while holding the model, scaffold, and instance set fixed. We did not anticipate the resolution-rate decline observed under the multimodal condition. Plausible contributing factors include: the added image content increasing prompt complexity without providing information the model can act on through a text-only patch interface; the model spending part of its limited iteration

Table 19: Resolution rate on the fixed 98-instance image-attachment subset: text-only mini-SWE-agent runs vs. the same instances with images attached (multimodal), both scored over the full 98-instance subset.

Model	Text-only	Multimodal	Rate (%), Text	Rate (%), Multimodal
Claude Sonnet 4.5	8	0	8.16	0.00
GPT-5.2	9	2	9.18	2.04

or cost budget reasoning about visual content that is not directly actionable for generating a code patch; and, for the specific case of resource-constrained instances, images consuming context or cost budget that would otherwise go toward exploration or patch iteration. We do not have evidence to adjudicate between these explanations from this ablation alone, and treat this as a negative result meriting further investigation rather than a definitive account of why visual grounding did not help in this setting.

E.9 Cost and Token Usage

Table 20 reports per-model API usage for the mini-SWE-agent runs (Appendix E.6), computed directly from each instance's mini-SWE-agent trajectory (summing OpenRouter's own per-call usage accounting): total API calls, prompt/completion/reasoning tokens, total cost, and average cost per instance with a completed trajectory (407 for Claude Sonnet 4.5, GPT-5.2, and Qwen3-Coder; 405 for Gemini 2.5 Flash).

Table 20: mini-SWE-agent API usage per model.

Model	API Calls	Prompt Tok.	Completion Tok.	Reasoning Tok.	Total Tok.	Total Cost ($)	Avg. $/Instance
Claude Sonnet 4.5	16,693	258,987,448	2,828,096	75,356	261,815,544	819.38	2.0132
GPT-5.2	25,571	870,949,709	10,070,002	7,209,803	881,019,711	327.86	0.8056
Gemini 2.5 Flash	10,884	186,322,313	2,593,472	1,701,051	188,915,785	27.01	0.0667
Qwen3-Coder	30,034	700,355,471	3,628,976	0	703,984,447	189.98	0.4668

Inference cost varies substantially across models: Claude Sonnet 4.5 has the highest average cost per instance ($2.0016), and Gemini 2.5 Flash the lowest ($0.0667). Reasoning token counts depend on provider-reported usage metadata; for Qwen3-Coder, OpenRouter does not return a separate reasoning-token value, so it is reported as 0 rather than omitted.

E.10 Contamination and Leakage Controls

A mined benchmark of popular, high-star repositories carries two related risks: models may have memorized the specific gold fix from pretraining data, or an evaluation container could reach the gold fix directly (e.g., through `.git` history or network access) rather than deriving it. We address both.

Repository state. Each repository is checked out at the instance base commit inside the evaluation container, without the test or fix patch applied. Before evaluation, we remove the `origin` remote, delete branches and tags beyond the base commit, and expire the reflog, so the gold fix is not reachable through `.git` history.

Network policy. Network egress is disabled by default at the Docker network level. An allowlisted proxy permits only package-registry traffic required to install dependencies (Maven, Gradle Plugin Portal, npm, and pub.dev); GitHub and general web access are blocked. We verified this behavior directly: package installation succeeds, while GitHub and external web requests fail.

Post-cutoff subset. To assess potential memorization of gold patches specifically (as opposed to container-level leakage), we report resolution rate on a post-cutoff subset for each model and scaffold, restricted to instances whose linked PR was created on or after that model's training-data cutoff date (Table 21): Gemini 2.5 Flash and Claude Sonnet 4.5, 2025-01-01; Qwen3-Coder, 2025-06-01; GPT-5.2, 2025-09-01, each taken from the model provider's publicly reported knowledge cutoff at the time of evaluation.

Table 21: Resolution rate on the post-cutoff subset, by model and scaffold. Post-cutoff instances a method never evaluated are counted as unresolved rather than excluded.

Model	Cutoff Date	Post-cutoff Instances	Agentic Rate (%)	Agentless Passes	Agentless Rate (%)
Gemini 2.5 Flash	2025-01-01	117	0.00	4	3.42
Qwen3-Coder	2025-06-01	74	2.70	3	4.05
Claude Sonnet 4.5	2025-01-01	117	7.69	5	4.27
GPT-5.2	2025-09-01	32	15.63	2	6.25

Three of the four models show comparable or lower resolution on the post-cutoff subset than on the full mini-SWE-agent evaluation (Table 16); GPT-5.2 is the exception, though its post-cutoff subset is small (32 instances). The Agentless pipeline shows the same mixed pattern (all four models' post-cutoff rates are comparable to or higher than their full-benchmark Agentless rates in Table 15). If memorization were a substantial driver of the reported resolution rates, we would expect a systematic drop across most models on unseen, post-cutoff instances under both scaffolds; instead, the pattern is mixed, and the one model that is higher on the agentic post-cutoff subset has by far the smallest post-cutoff sample. This does not rule out contamination entirely, and the small absolute pass counts (0–9) limit the statistical power of this check, but the pattern does not show what we would expect if memorization were a dominant factor.

F Evaluation Results

This section presents comprehensive evaluation results for four contemporary 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 15 shows resolution rates for both evaluation settings on 407 verified tasks (rate = resolved / 407, with unsubmitted or empty-patch predictions counted as unresolved). Under the Automated retrieval setting, rates range from 3.19% to 4.18% (13–17 resolved per model). Under the Oracle retrieval setting, rates range from 1.97% (Gemini 2.5 Flash, 8 resolved) to 5.65% (Claude Sonnet 4.5, 23 resolved). Tasks in MobileDev-Bench require modifying an average of 12.9 files (median: 5), reflecting the multi-file nature of mobile development.

Multi-File Modifications Show Lower Resolution Rates.

Figure 8 reveals a dramatic degradation in resolution rates as the number of required file modifications increases. Under the Automated retrieval setting, single-file tasks (71 tasks) achieve 12.7–15.5% resolution rates, substantially higher than the 3.19–4.18% overall rates. Under the Oracle retrieval 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 Automated retrieval and 0.0–3.4% under Oracle retrieval. Mid-complexity tasks requiring 4–10 files prove nearly unsolvable (0.0–3.8% Automated retrieval; 0.0–3.8% Oracle retrieval). This difficulty cliff highlights models' fundamental limitations in coordinating changes across multiple files.

Tasks requiring 11+ files (125 tasks, 30.7% of the benchmark) yield 0.0% resolution across all models under both Automated and Oracle retrieval settings.

Across all models, only 28 unique instances were successfully resolved under Automated retrieval, accumulating to 56 total resolutions due to overlap. The remaining 379 tasks (93.1%) were not resolved by any model under the Automated retrieval setting, underscoring the fundamental challenge of multi-file reasoning in mobile development contexts.

Larger Patches Correlate with Lower Success Rates.

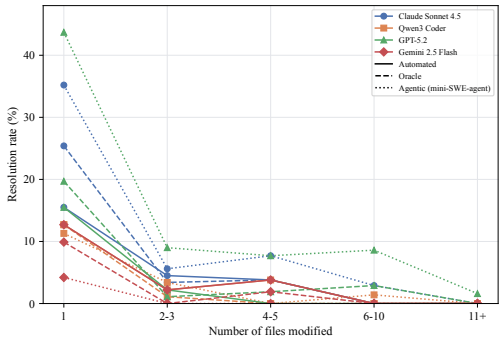

Figure 8: Resolution rates by number of files modified, for Automated (solid lines) and Oracle (dashed lines) retrieval settings and the agentic mini-SWE-agent baseline (dotted lines). Single-file tasks achieve 12.7–15.5% under Automated, up to 25.4% under Oracle, and up to 43.7% under mini-SWE-agent, dropping to near zero for 6+ files under Automated and Oracle.

Patch complexity, measured by lines changed in the ground truth fix, strongly predicts resolution success (Figure 9). Under Automated retrieval, tiny patches ($\leq$10 lines) achieve 14.3–16.1% resolution rates, while very large patches (>500 lines) yield 0.0% across all models. Under Oracle retrieval, rates for $\leq$10-line patches reach 10.7–30.4%, and >500-line patches remain at 0.0%. 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 7.0 lines versus 106.0 lines for unresolved instances (15.1x difference). Mean patch sizes show a similar pattern: 16.4 lines for resolved versus 358.1 lines for unresolved, a 95% reduction. This size difference indicates that successful repairs target localized, focused changes rather than broad refactorings.

Patch size distributions further emphasize this disparity (Figure 10). Small patches ($\leq$50 lines) comprise 92.9% of resolutions but only 35.1% of the benchmark, a 2.6x overrepresentation. Conversely, patches >50 lines represent 64.9% of the benchmark but only 7.1% of resolutions. Tiny patches ($\leq$10 lines) show the most extreme skew: 60.7% of resolutions versus 13.8% of benchmark (4.4x 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: 0.0–0.8%; >500 lines: 0.0%).

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 22 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.

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 59.0% of benchmark tasks are single-artifact, 96.4% of automated retrieval resolutions come from single-artifact tasks (1.63x overrepresentation). Multi-artifact tasks show severe underrepresentation (3.6% of resolutions vs. 41.0% of benchmark, 0.09x), highlighting difficulty coordinating changes across heterogeneous file types.

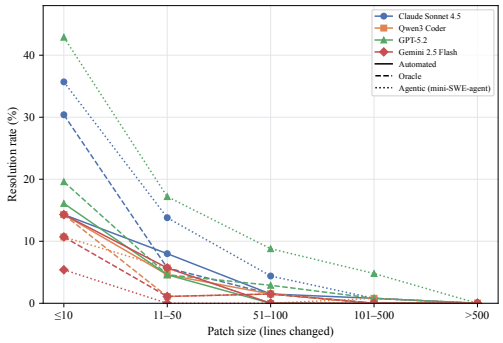

Figure 9: Resolution rates by patch size, for Automated (solid lines) and Oracle (dashed lines) retrieval and the agentic mini-SWE-agent baseline (dotted lines). Tiny patches ($\leq$10 lines) achieve 14.3–16.1% under Automated, 10.7–30.4% under Oracle, and up to 42.9% under mini-SWE-agent, dropping to near zero for patches $>$100 lines under Automated and Oracle.

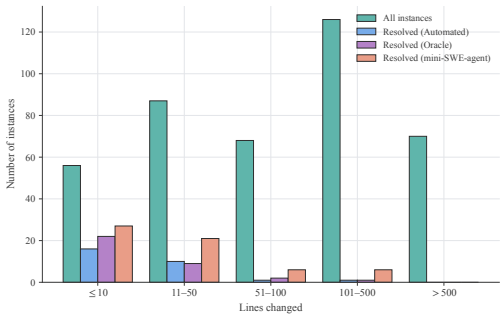

Figure 10: Patch size distribution for all instances versus instances resolved under Automated retrieval, Oracle retrieval, and the agentic mini-SWE-agent baseline. Resolved instances skew heavily toward patches $\leq$ 50 lines (92.9% vs. 35.1% under Automated/Oracle retrieval).

File Count Distribution. Single-file tasks dominate automated retrieval resolutions (75.0% of resolutions vs. 17.4% of benchmark, 4.31x overrepresentation), while multi-file tasks show severe underrepresentation. Tasks requiring 6–10 files exhibit 0.0% of automated retrieval resolutions versus 17.2% of benchmark (0.00x ratio). The 11+ file category also yields 0.0% resolutions (0.00x ratio), despite comprising 30.7% of the benchmark.

Framework Distribution. Framework distribution among resolved tasks broadly follows benchmark composition but with notable deviations (Table 22). Android Native tasks are slightly underrepresented (67.9% of Automated retrieval resolutions vs. 80.6% of benchmark, 0.84x), while Flutter tasks are overrepresented at 32.1% of resolutions despite comprising only 15.5% of the benchmark (2.07x), 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 under Automated retrieval (0.0% vs. 3.9%), 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 Oracle Regression for Qwen3 Coder and Gemini 2.5 Flash

Qwen3-Coder drops from 3.19% to 2.46%, and Gemini 2.5 Flash drops from 3.19% to 1.97%. Both models resolve the exact same 13 instances under Automated retrieval. The initial candidate-surfacing step of localization, BM25-plus-embedding retrieval (Appendix E.2), is model-independent: the same embedding model surfaces the same candidate files and functions regardless of which LLM is evaluated. However, the subsequent model-prompted narrowing and the patch itself are model-specific, so this shared candidate pool does not by itself guarantee identical final edited-file sets

Table 22: Resolution rate breakdown by task characteristics (407 tasks). Resolved column shows the share of uniquely resolved instances (28 under Automated, 34 under Oracle, 60 under mini-SWE-agent) in each category vs. the overall benchmark share. Ratio = Resolved% / Benchmark%.

		Automated Retrieval		Oracle Retrieval		Agentic (mini-SWE-agent)	
Category	**Bench.**	**Resolved**	**Ratio**	**Resolved**	**Ratio**	**Resolved**	**Ratio**
Artifact Type Diversity							
Single-artifact	59.0%	96.4%	1.63x	97.1%	1.65x	86.7%	1.47x
Multi-artifact	41.0%	3.6%	0.09x	2.9%	0.07x	13.3%	0.32x
File Count Distribution							
1 file	17.4%	75.0%	4.31x	76.5%	4.40x	58.3%	3.34x
2–3 files	21.9%	17.9%	0.82x	11.7%	0.53x	18.3%	0.84x
4–5 files	12.8%	7.1%	0.55x	5.9%	0.46x	10.0%	0.78x
6–10 files	17.2%	0.0%	0.00x	5.9%	0.34x	10.0%	0.58x
11+ files	30.7%	0.0%	0.00x	0.0%	0.00x	3.3%	0.11x
Framework Distribution							
Android Native	80.6%	67.9%	0.84x	82.4%	1.02x	81.7%	1.01x
Flutter	15.5%	32.1%	2.07x	14.7%	0.95x	16.7%	1.08x
React Native	3.9%	0.0%	0.00x	2.9%	0.74x	1.7%	0.42x

(Table 24 shows Qwen3-Coder and Gemini 2.5 Flash differ in overall recall, precision, and F1, since those metrics are computed over the files each model actually edits, not the raw retrieval candidates). The 13-instance overlap is consistent with a shared candidate pool making localization unambiguous on these particular, comparatively simple instances, so both models converge on the same correct file(s) despite localizing and generating independently.

In contrast, tracing the instances each model fails to resolve under the Oracle setting reveals two distinct failure mechanisms.

Incomplete Oracle context for new-file creations. The Oracle setting provides models with the set of files modified in the developer fix. However, when a fix requires creating a new file that does not exist at the base commit, that file cannot be included in the oracle context. One of the 10 instances that Qwen3-Coder fails to retain under Oracle retrieval falls into this category: `zulip-flutter-1340` requires creating `lib/model/actions.dart`, which is absent from the provided context. As a result, the model is unable to reproduce the intended fix within the constrained file set. In contrast, under Automated retrieval, the model successfully resolves this instance by patching a subset of existing files sufficient to pass the test suite, effectively bypassing the need for file creation. This option is unavailable in the Oracle setting, where the fixed file list implicitly constrains the repair scope. This failure mode does not affect Gemini 2.5 Flash, whose 10 Oracle regression instances all involve modification-only patches.

Over-editing under full-file context. The remaining 9 of Qwen3's and all 10 of Gemini's regression instances had a complete oracle file list, yet both models still failed. In the Automated retrieval setting, models are provided with localized code snippets (i.e., at the function or class granularity), which constrains the search space and encourages targeted, minimal edits. In contrast, the Oracle setting exposes the model to the full file content. We observe that under Oracle retrieval, Qwen3-Coder and Gemini 2.5 Flash consistently introduce additional changes beyond the ground-truth fix, co-occurring with the observed regressions; one plausible explanation is that the additional context dilutes the model's focus, resulting in over-editing, whereas the constrained context of Automated retrieval acts as an implicit regularizer. We cannot, however, isolate this explanation from a confound in our experimental design: the Automated and Oracle settings also differ in generation budget (Appendix E.3), with Automated retrieval generating a greedy sample plus multiple diverse samples and selecting the best via test-based re-ranking, while Oracle retrieval uses a single greedy sample only. Reduced sampling diversity and the absence of re-ranking under Oracle retrieval could themselves contribute to the observed regressions independent of over-editing, and we cannot fully separate the two effects with the current experimental design; we report the over-editing pattern as an observed association with the caveat that a budget-matched Oracle re-run would be needed to isolate its causal contribution.

Table 23: Average fix patch statistics for model predictions under Automated retrieval, Oracle retrieval, and the agentic (mini-SWE-agent) baseline. #L: total changed lines (added + removed). #A: added lines. #R: removed lines. #H: hunks. #F: files modified. Averages are over non-empty submitted patches only.

	Automated Retrieval					Oracle Retrieval					Agentic (mini-SWE-agent)				
Model	**#L**	**#A**	**#R**	**#H**	**#F**	**#L**	**#A**	**#R**	**#H**	**#F**	**#L**	**#A**	**#R**	**#H**	**#F**
Claude Sonnet 4.5	8.5	6.2	2.3	1.4	1.0	71.1	44.6	26.4	6.9	3.6	351.9	290.6	61.3	7.2	4.0
Gemini 2.5 Flash	8.7	5.9	2.8	1.4	1.0	115.5	93.9	21.6	9.5	4.9	756.7	524.0	232.7	11.5	5.6
GPT-5.2	10.2	7.3	2.9	1.8	1.1	37.6	26.4	11.2	2.6	1.3	526.2	477.5	48.7	11.8	6.6
Qwen3 Coder	12.7	9.0	3.7	1.6	1.1	57.1	41.7	15.4	4.8	2.3	468.6	393.4	75.2	9.4	5.6

Table 23 quantifies this effect across all models and settings. Under automated retrieval, all models produce compact patches (8.5–12.7 total changed lines, ≈1.0 files modified), reflecting the narrowly targeted edits that result from working with localized code snippets. Under oracle retrieval, patch sizes increase substantially (37.6–115.5 total lines, 1.3–4.9 files modified). Gemini 2.5 Flash shows the largest expansion (8.7→115.5 lines; 1.0→4.9 files), followed by Claude Sonnet 4.5 (8.5→71.1 lines), Qwen3 Coder (12.7→57.1 lines), and GPT-5.2 (10.2→37.6 lines). Notably, Claude Sonnet 4.5 expands substantially yet does not regress under oracle, suggesting that patch size alone does not determine whether a regression occurs; the size increase is most pronounced for Gemini 2.5 Flash and Qwen3 Coder, consistent with their oracle resolution rate drops, though as noted above we cannot rule out that their single-sample generation budget under Oracle retrieval also contributes to this pattern.

Listing 11 illustrates this for `thunderbird-android-6360` (Qwen3-Coder Oracle regression). The correct fix consists of a single insertion: adding `logBuffer.clear()` at the top of `readHelloResponse()`. The Automated retrieval patch applies exactly this change (one hunk, +1 line) and passes the test suite. In contrast, the Oracle retrieval patch includes the same insertion but, having access to the full `SmtpResponseParser.kt` file, additionally removes `logBuffer.clear()` from `readResponse()` and relocates it to `readResponseAfterReplyCode()`. This modification introduces a behavioral change across methods (three hunks, +1/−1 line) and leads to test failures.

Listing 12 shows a similar pattern in `thunderbird-android-9640` (observed for both models). The ground-truth fix introduces an early-return guard before any dependency injection occurs. The Oracle patch applies the same guard but places it one line later, after `DI.get<Clock>()` has already been invoked, which changes execution order relative to the intended fix and results in failure.

Gold / Automated *1 hunk, +1 line*

```
@@ -33,6 +33,8 @@
 fun readHelloResponse() {
+    logBuffer.clear()
     val replyCode = readReplyCode()
```

Oracle *3 hunks, +1/−1 lines*

```
@@ -33,6 +33,7 @@
 fun readHelloResponse() {
+    logBuffer.clear()
     val replyCode = readReplyCode()
@@ -153,7 +154,6 @@
 fun readResponse(...) {
-    logBuffer.clear()
     val replyCode = readReplyCode()
@@ -162,6 +162,7 @@
 private fun readResponseAfterReplyCode() {
     logBuffer.clear()
```

Figure 11: Patches for `thunderbird-android-6360` (`SmtpResponseParser.kt`). Gold inserts `logBuffer.clear()` in `readHelloResponse()`; Automated setting replicates this minimal change. Oracle makes the same correct insertion but, having seen the full file, also removes `logBuffer.clear()` from `readResponse()` and repositions it in `readResponseAfterReplyCode()`, altering runtime behavior.

Gold / Automated *guard before DI call*

```
@@ -85,6 +85,10 @@
     get() {
+        if (!isQuietTimeEnabled)
+            return false
         val clock = DI.get<Clock>()
         val quietTimeChecker = ...
         return quietTimeChecker.isQuietTime
```

Oracle *guard after DI call*

```
@@ -86,12 +86,14 @@
     get() {
         val clock = DI.get<Clock>()
+        if (!isQuietTimeEnabled)
+            return false
         val quietTimeChecker = ...
         return quietTimeChecker.isQuietTime
```

Figure 12: Patches for `thunderbird-android-9640` (`K9NotificationStrategy.kt`, repeated identically in `NotificationHelper.kt`). Gold and Automated setting prepend the early-return guard before `DI.get<Clock>()`, short-circuiting the dependency injection call. Oracle inserts the guard one statement too late: the DI call executes unconditionally, failing tests that assert no DI interaction when quiet time is disabled.

Across the 7 instances lost by both models under complete oracle context, the Oracle setting patches are consistently larger than the corresponding Automated retrieval patches despite addressing the same underlying tasks. In these cases, the Automated patches pass while the Oracle patches fail. This pattern suggests that full-file context can lead Qwen3-Coder and Gemini 2.5 Flash to incorporate additional, non-essential modifications beyond the minimal fix, resulting in behaviorally different patches that no longer preserve correctness.

In contrast, Claude Sonnet 4.5 and GPT-5.2 do not exhibit this effect to the same extent, which is consistent with their positive Oracle delta.

F.4 Fault Localization Performance

Under automated retrieval, overall retrieval performance ranges from 13.8–18.6% recall, 42.9–57.2% precision, and 17.6–23.3% F1 (Table 24), 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.

Recall Degradation by File Count. Under automated retrieval, 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. Figure 3 (Section 6) plots recall, precision, and F1 against ground-truth file count across all four models; precision remains comparatively stable (35–66%) while recall collapses, widening the precision-recall gap as complexity increases and confirming that F1 degradation is driven almost entirely by recall failure.

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** (6–10% of tasks): Almost exclusively single-file changes with strong semantic signals in issue descriptions.
- **Zero localization** (43–57% of tasks): Models edit entirely incorrect files, common when ground truth modifies non-code artifacts or peripheral modules. 111 tasks (27%) show zero recall across all four models simultaneously.
- **Partial localization** (37–47% of tasks): Models find some relevant files but miss cascading modifications across architectural layers.

Table 24: File-level retrieval metrics (Recall / Precision / F1, macro-averaged, %) by ground-truth file count under Automated retrieval, Oracle retrieval, and the agentic (mini-SWE-agent) baseline (407 tasks).

		Claude Sonnet 4.5			**Qwen3-Coder**			**GPT 5.2**			**Gemini 2.5 Flash**		
#Files	n	R	P	F1	R	P	F1	R	P	F1	R	P	F1
Automated													
1 file	71	54.9	54.9	54.9	46.5	46.5	46.5	35.2	35.2	35.2	35.2	35.2	35.2
2–3 files	89	22.7	49.4	30.5	23.6	48.3	30.9	20.4	44.4	27.5	20.2	44.9	27.5
4–5 files	52	12.1	51.9	19.6	12.7	53.8	20.5	9.9	40.4	15.8	10.5	42.3	16.7
6–10 files	70	9.4	65.7	16.4	7.5	52.1	13.0	7.4	49.3	12.8	6.7	46.4	11.6
11+ files	125	2.9	61.6	5.5	3.0	53.6	5.5	2.1	43.6	3.9	2.3	44.0	4.3
Overall	407	18.6	57.2	23.3	17.1	51.0	21.4	13.8	42.9	17.6	13.8	42.9	17.6
Oracle													
1 file	71	95.8	95.8	95.8	97.2	96.5	96.7	84.5	83.8	84.0	100.0	100.0	100.0
2–3 files	89	57.5	94.9	68.9	50.2	98.5	64.4	40.6	89.9	54.9	60.9	96.5	71.6
4–5 files	52	48.8	89.8	58.1	35.9	93.2	48.0	21.7	89.2	34.4	50.2	92.6	60.7
6–10 files	70	43.1	90.8	53.6	30.4	89.9	41.5	14.2	87.1	23.2	43.2	95.8	53.6
11+ files	125	22.8	80.2	31.2	13.6	77.1	20.0	8.3	75.6	12.0	35.1	88.9	43.0
Overall	407	50.0	89.2	58.0	41.9	89.4	50.4	31.4	83.9	38.7	55.4	94.2	63.3
Agentic (mini-SWE-agent)													
1 file	71	74.6	49.2	54.0	63.4	31.1	37.7	78.9	57.5	61.7	25.4	18.9	20.3
2–3 files	89	42.9	47.2	41.3	38.4	37.2	33.5	45.5	57.4	46.9	18.7	18.8	16.8
4–5 files	52	33.8	58.2	40.1	38.5	49.9	40.4	43.2	62.8	47.0	13.8	21.9	15.2
6–10 files	70	24.9	56.7	31.5	27.6	48.6	30.3	37.8	69.9	44.8	12.9	24.4	15.5
11+ files	125	13.6	53.3	18.6	11.6	47.4	16.3	21.6	59.6	27.6	9.4	27.8	12.0
Overall	407	35.2	52.4	34.7	32.7	42.9	29.3	42.3	60.9	43.2	15.4	22.9	15.5

F.4.1 IMPLICATIONS

Both file-level localization and patch generation are substantial bottlenecks. Under automated retrieval, recall (13.8%–18.6%) sets a hard ceiling on resolution: actual resolution (3.19%–4.18%) falls below this ceiling, so localization is one binding constraint. Under oracle retrieval, recall rises to 31.4%–55.4% overall and 84.5%–100% on single-file tasks, yet resolution still tops out at 5.65%. Generation is therefore the larger remaining constraint: improving localization to perfect lifts resolution from 4.18% to 5.65% (a +1.46pp gain), whereas closing the oracle generation gap on already-localized cases offers substantially more headroom.

Improving repair performance, therefore, requires advances on both axes. Prior work on automated repair has primarily focused on patch generation (Monperrus, 2018; Ding et al., 2021); our results suggest that fault localization in mobile codebases warrants comparable attention. Promising directions for localization include dependency-aware retrieval to capture inter-file relationships, incorporation of architectural priors (that is, leveraging knowledge of common software design patterns and component interactions such as MVVM or repository-based data flow), and specialized retrieval strategies for non-code artifacts that are underrepresented in embedding spaces. For generation, the persistent oracle gap points to multi-file consistency constraints, framework-aware reasoning about lifecycle and resource semantics, and explicit modeling of cross-artifact dependencies between source code, resources, and build configurations.

F.5 WHY SUBMITTED PATCHES FAIL TO RESOLVE

Section 6.2 and Appendix E.6 account for instances that never reach evaluation (budget exhaustion, tool-call formatting errors, and similar coverage failures). Here we classify, for each of the four models under mini-SWE-agent, why the patches that *are* submitted and evaluated still fail to resolve. For every unresolved instance, we mechanically inspect its execution report and fix-patch test log and assign it to exactly one of six categories, checked in the following priority order:

- **Apply Failure**: the submitted patch could not be applied to the repository (git apply and all fallback strategies fail), so no test execution is possible.
- **Build/Compile Failure**: the patch applies, but the test command captures zero test results, indicating a build or compilation error before any test can run.
- **No-Op**: the patch applies and the build succeeds, but no test transitions from failing or absent to passing, indicating the change does not affect the behavior under test.
- **Partial Fix**: the patch fixes at least one required test with no regressions on previously-passing tests (satisfying the instance-level validity check in our harness), but does not fix every test the dataset's gold fix requires, so it does not count as resolved under the strict, SWE-bench-style all-tests-pass criterion.
- **Regression**: at least one previously-passing test fails after the patch is applied, disqualifying the fix regardless of any tests it does fix.
- **Anomalous**: a rare, inconsistent transition pattern (e.g., a test with no baseline result that only fails after the fix) that does not fit the categories above.

Table 25 reports this classification by model.

Table 25: Semantic classification of unresolved instances under mini-SWE-agent, by model.

Model	Unresolved	Build/Compile	No-Op	Partial Fix	Apply	Regr./Anom.
Claude Sonnet 4.5	331	241 (72.8%)	60 (18.1%)	18 (5.4%)	8 (2.4%)	4 (1.2%)
GPT-5.2	348	228 (65.5%)	82 (23.6%)	31 (8.9%)	4 (1.1%)	3 (0.9%)
Gemini 2.5 Flash	236	178 (75.4%)	44 (18.6%)	8 (3.4%)	3 (1.3%)	3 (1.3%)
Qwen3-Coder	392	263 (67.1%)	72 (18.4%)	17 (4.3%)	36 (9.2%)	4 (1.0%)

Build/compile failure is the dominant failure mode for every model. Between 65.5% and 75.4% of unresolved instances never reach a working test run at all: the submitted patch applies cleanly but the resulting code does not compile or the build otherwise fails before tests execute. This is the single largest category by a wide margin for all four models, indicating that the primary obstacle is not test-level correctness reasoning but producing code that builds successfully within the target repository's toolchain (Gradle, pub, or npm/yarn).

No-Op is the second-largest category (18.1%–23.6%). These patches apply and build successfully but do not change the outcome of any relevant test, indicating the model edits code without addressing the behavior the test suite checks.

Partial fixes are a meaningful minority, most pronounced for GPT-5.2 (8.9%). These are cases where the model's patch is not spurious: it genuinely fixes at least one required behavior with no regressions, but leaves other required behavior broken. GPT-5.2 produces such partial fixes roughly twice as often as Claude Sonnet 4.5 or Qwen3-Coder and over twice as often as Gemini 2.5 Flash, suggesting GPT-5.2 more frequently identifies the right area of the fix but does not complete it.

Apply failure is rare except for Qwen3-Coder (9.2%, versus 1.1%–2.4% for the other three models). Qwen3-Coder's patches are substantially more likely to be rejected outright by the patch-application step, pointing to a diff-formatting reliability gap relative to the other models rather than a reasoning failure.

F.6 Mobile-Specific Failure Pattern Analysis

This section cross-references the mobile-specific task taxonomy (Appendix D.6) against resolution outcomes to identify failure patterns that file count, patch size, and difficulty tier alone do not explain. We evaluate four models under two repair scaffolds, Agentless and mini-SWE-agent (Section 5), and count an instance as resolved if either scaffold resolves it with any model.

All models fail in the same seven categories. MOBILE_SECURITY, DECLARED_CAPABILITY, MEDIA_ASSET, MOBILE_PLATFORM_SERVICES, PLATFORM_DIVERGENCE, HARDWARE_MEDIATED, and VERSION_GATED have zero resolved instances across all four models and both scaffolds — 48 of 407 instances, 11.8% of the benchmark. Coverage across these 48 instances is not uniform: of the 384 possible model-scaffold evaluations (48 instances $\times$ 4 models

$\times$ 2 scaffolds), 342 were actually submitted and completed, predominantly short of the full 384 due to Gemini 2.5 Flash's tool-call formatting failures under the agentic scaffold (Appendix E.6) and a small number of instances each Agentless run did not evaluate; every one of the 342 completed evaluations fails, with zero successes. A plausible explanation for the underlying pattern is that these categories require domain knowledge models rarely see in training — store billing, cryptography and authentication detail, codec behavior, and OS or hardware behavior — none of which is discoverable from the code alone. Three further categories come close to this pattern: BACKGROUND_EXECUTION_AND_ASYNC, UI_BINDING, and RESOURCE_QUALIFIED resolve at 5.0%, 5.9%, and 6.2% respectively, the lowest rates of any category with at least one success.

Resolution rate tracks task difficulty closely. Easy tasks resolve at 29.7%, Medium at 10.8%, and Hard at 0.8%. The three difficulty tiers are close to equal in size (138, 139, and 130 instances), so this drop is not an artifact of one tier having far fewer examples than the others. Resolution rate falls by a factor of about 37 from Easy to Hard; only 1 of 130 Hard-tier instances is resolved, across all four models and both scaffolds combined.

Resolution rate falls as an instance touches more mobile-specific concerns. Resolution rate falls from 20.9% on instances carrying one taxonomy label to 0.0% on instances carrying four. Platform-agnostic instances (those carrying none of the 15 labels) resolve at 1.05–1.80$\times$ each model's baseline rate. Both patterns track difficulty tier closely: instances with more labels are judged difficult more often, and platform-agnostic instances skew toward the Easy tier. We do not treat label count and difficulty as independent effects here, since the current sample does not let us fully separate one from the other.

Visual rendering fails regardless of difficulty. UI_RENDERING (130 instances) has a difficulty mix close to the benchmark average, yet every model resolves it at 0.22–0.75$\times$ its own overall rate. This suggests the low rate is a property of the category itself rather than of difficulty. Our execution oracle uses host-side, headless unit tests rather than emulator-based instrumentation (Appendix I), so this evaluates programmatic view state and mocked bindings rather than rendered pixels directly; the low rate is therefore consistent with both models struggling to verify layout, animation, or gesture correctness through code alone, and with the added difficulty of correctly navigating the synthetic mocking frameworks these tests rely on.

Resolution concentrates on Bug Fix tasks. Table 26 reports, by task category, the number of unique instances resolved by at least one of the four models under mini-SWE-agent (Section 6.2); an instance resolved by more than one model is counted once. Bug Fix tasks account for the large majority of successful resolutions despite being a minority of the benchmark by instance count (44.2%, Section 4), consistent with New Feature and Feature Optimization tasks requiring different capabilities than repairing an existing, localized defect.

Table 26: Unique resolved instances by task category (union across all four models; an instance resolved by multiple models is counted once), under the agentic (mini-SWE-agent) baseline and Agentless Automated retrieval.

Task Category	Agentic	Agentless (Automated)
Bug Fix	36	22
New Feature	14	6
Feature Optimization	10	0
Total (union)	60	28

Taken together, these patterns indicate that current models fail most where a fix requires platform knowledge outside the code (store, security, hardware, or OS-version behavior) or where correctness depends on rendered UI state rather than code alone. Repair pipelines reason comparatively well about code they can read, and comparatively poorly about external contracts or the visual result of a change.

G Discussion

Under Agentless, overall resolution rates on MobileDev-Bench are 3.19%–4.18% with automated retrieval and at most 5.65% with oracle retrieval, and under mini-SWE-agent they reach at most 12.53%. Three architectural properties of mobile development explain this low performance. First, multi-artifact coordination: single-artifact tasks comprise 59% of the benchmark and account for 96.4% of resolved instances at per-task resolution rates of 5.0%–6.7%, while multi-artifact tasks comprise 41% of the benchmark and account for only 3.6% of resolved instances at 0.0%–0.6%. Second, framework-mediated execution: platform lifecycles obscure control flow from static analysis. Third, cross-module dependencies: under automated retrieval, recall drops from 35.2%–54.9% on single-file tasks to 6.7%–9.4% on tasks requiring 6–10 files, and further to 2.1%–3.0% on tasks with 11 or more files.

Both file-level localization and patch generation are substantial bottlenecks. Under automated retrieval, recall (13.8achievable resolution at the recall ceiling. Under oracle retrieval, recall rises to 31.4%–55.4% overall and 84.5%–100% on single-file tasks, yet resolution still reaches at most 5.65%, indicating that patch generation is the larger remaining bottleneck. Progress on mobile app issue resolution therefore requires advances on both axes: multi-file localization (dependency tracing, artifact-aware retrieval, framework-specific reasoning) and patch generation under multi-file and multi-artifact coordination constraints.

Under automated retrieval, resolution rates decline with file count: single-file tasks achieve 12.7%–15.5% resolution, 2–3 file tasks drop to 2.2%–4.5%, mid-complexity tasks (4–10 files) fall to 0.0%–3.8%, and tasks with 11 or more files yield 0% across all models. This pattern suggests models struggle with architectural complexity, that is, interconnected changes across heterogeneous components, rather than raw file count alone.

Interpreting the mobile-specific failure patterns. The failure-pattern analysis (Section 6.5, Appendix F.6) points to two failure modes that file count and patch size alone do not explain. The first is a knowledge gap: the seven categories with zero resolutions across every model and scaffold (MOBILE_SECURITY, DECLARED_CAPABILITY, MEDIA_ASSET, MOBILE_PLATFORM_SERVICES, PLATFORM_DIVERGENCE, HARDWARE_MEDIATED, VERSION_GATED) share a common property: correctly resolving them requires knowing something about the platform, store policy, cryptographic primitive, codec, or hardware behavior in question that is not derivable from the surrounding code. Unlike a multi-file coordination failure, where the model can in principle see everything it needs and simply fails to use it, these are cases where the necessary information plausibly is not represented, or is only sparsely represented, in the training data the models draw on for mobile-platform specifics, as distinct from general-purpose programming knowledge. The second is a structural, not a difficulty, limitation: UI_RENDERING resolves at 0.22–0.75$\times$ each model's own baseline rate despite an average difficulty mix. Because our execution oracle is restricted to host-side unit tests rather than emulator-based instrumentation (Appendix I), this is consistent with both a text-only patch pipeline being unable to observe the actual rendered result of a change, and with models struggling to correctly navigate the mocked view-model and presenter harnesses these host-side tests rely on in place of the live display. Both point toward the same remedy discussed below: repair pipelines for mobile code need access to information beyond the source diff itself, whether platform documentation for the knowledge gap or rendered UI state (e.g., screenshots or accessibility trees) for the visual-correctness gap, rather than further improvements to code-only localization and generation.

H Future Work

Our findings suggest several high-priority research directions. First, localization limits resolution under automated retrieval, so techniques combining semantic search with static analysis for multi-file dependency tracing warrant investigation. Oracle retrieval and mini-SWE-agent, however, raise file-level recall without a matching gain in resolution, so multi-file patch generation needs equal attention. 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, and giving repair

agents a way to observe rendered UI state directly (e.g., screenshots or accessibility trees), may improve UI-related bug localization and address the structural UI_RENDERING gap identified in Section 6.5. Fourth, the categories where every model and scaffold fails outright (Appendix F.6) suggest giving repair agents access to platform-specific documentation (store policy, platform security and hardware APIs) as external context, rather than relying solely on what is derivable from the repository. Extending to iOS and developing framework-aware representations encoding lifecycle semantics provide additional promising directions.

I LIMITATIONS

MobileDev-Bench covers Android Native, React Native, and Flutter repositories, but not iOS-native projects in Swift or Objective-C. This exclusion reflects reproducibility constraints. iOS builds depend on Xcode and macOS, and the Xcode and Apple SDKs Agreement (Apple Inc., 2024) prohibits execution on non-Apple-branded hardware. As a result, our containerized build-and-test workflow cannot be reproduced for iOS using standard Linux infrastructure. Supporting iOS would require macOS CI runners or hardware-in-the-loop setups, reducing environment uniformity and reproducibility.

Evaluation relies on repository test suites, so issues with incomplete test coverage may be missed. The benchmark is also built from issue–pull request pairs in open-source repositories and may underrepresent activities such as UI design iteration or refactoring without associated issues. Although we evaluate both Agentless and mini-SWE-agent (Section 6.2), we do not exhaustively explore scaffold configurations or longer debugging and human-in-the-loop workflows.

The framework distribution is imbalanced: Android Native accounts for 80.6% of instances, Flutter 15.5%, and React Native 3.9% (Appendix D). This reflects the larger number of Android repositories meeting our criteria for active maintenance, issue-linked PRs, and reproducible execution, rather than an intentional sampling choice. Broader Flutter and React Native coverage remains future work.

Finally, MobileDev-Bench is motivated by mobile-specific challenges such as declarative UI rendering, asynchronous state management, and framework-mediated lifecycles, but its execution oracle is limited to host-side, headless unit tests (Section 3.4). Android instrumentation tests require an emulator and are excluded for reproducibility. These tests therefore exercise mocked view models, presenter logic, or programmatic state rather than the rendered UI. This trade-off enables hermetic Docker execution, but the observed difficulty of UI_RENDERING tasks (Appendix F.6) may reflect both visual-reasoning difficulty and the challenge of navigating mocking frameworks.

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.