# OS-BENCH: EVALUATING SWE AGENTS ON CROSS-OS CODE MIGRATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Language-model (LM) agents are increasingly evaluated on repository-level software engineering (SWE) benchmarks derived from GitHub issues and pull requests. Although these benchmarks cover a broad range of development tasks, they provide limited insight into a common yet technically distinct challenge: migrating an existing implementation from one operating system (OS) to another while preserving behavior on both. We introduce OS-BENCH, an executable benchmark for cross-OS code migration between Linux and Windows. Motivated by case studies showing that cross-platform features are frequently implemented on one OS before support is added for another, OS-BENCH identifies migration opportunities by comparing execution coverage collected from the same repository on both systems. Platform-specific regions are then verified, organized into five migration scenarios, and transformed into repository-level tasks with target-platform tests and source-platform regression tests. The resulting benchmark contains 140 validated tasks from 111 repositories across C, C++, C#, Go, and Rust, comprising 91 Windows-to-Linux and 49 Linux-to-Windows migrations. We evaluate six models with SWE-agent, Claude Code, OpenHands, and Win-agent. The best configurations achieve only 26.4% success on Linux-target tasks and 20.4% on Windows-target tasks while preserving correctness across both systems. These results indicate that cross-OS migration is a challenging and underexplored capability, and OS-BENCH is a testbed for improving platform-aware SWE agents.

## 1 INTRODUCTION

Cross-platform support is essential for software libraries, developer tools, and applications that need to run across different computing environments. Supporting a new operating system (OS), however, requires more than recompiling the same source code. Developers often need to handle differences in system APIs, filesystem and path conventions, process and signal behavior, networking interfaces, text encodings, build toolchains, and command-line environments. A migration patch must therefore implement the required behavior on the target platform while preserving existing functionality on the source platform.

LM-based software engineering agents can explore repositories, execute tests, and modify multiple files in response to natural-language instructions. Their progress is typically evaluated on benchmarks built from real GitHub issues or pull requests (Jimenez et al., 2024; Jin et al., 2026; Zhang et al., 2026; Li et al., 2026a). While these benchmarks capture realistic repository-level development tasks, they usually evaluate each task in only one execution environment. As a result, they do not directly test whether an agent can identify an OS-specific implementation gap, adapt behavior across incompatible platform abstractions, work effectively in the target operating system, and preserve existing functionality on the original OS.

Our preliminary case studies illustrate how this challenge arises in real multi-platform repositories. In these cases, functionality is implemented for one platform first, such as Linux, and later ported to another, such as Windows, using a separate platform-specific implementation. Figure 1 shows an example from the asynchronous I/O library *libuv/libuv*. Support for configuring UDP sockets for multicast and broadcast was first introduced in the Unix backend and was later added to the Windows backend in a separate commit. The two implementations expose the same user-facing behavior, but they live in different code paths and rely on different operating-system interfaces.



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.




Figure 1: A cross-OS code migration example from repository *libuv/libuv*. The functionality was first implemented on Unix in an earlier commit and subsequently implemented on Windows in a later commit. During execution, each platform invokes only its corresponding platform-specific implementation.

Although such histories reveal migration work, migration tasks cannot be reliably extracted directly from individual commits. A single migration may span multiple commits, and each commit may mix migration changes with unrelated bug fixes, refactoring, or feature development. Using a commit as a task can therefore omit changes needed to complete the migration or include edits unrelated to it. Constructing a coherent task from commit history therefore requires substantial manual effort to trace the migration across multiple commits and disentangle relevant edits from unrelated changes.

We address this challenge requiring heavy manual workload and automate the task-construction process by using differences in execution coverage to identify platform-specific implementations. When a repository's tests are run on Linux and Windows, code associated with one backend is exercised on one OS but not the other. We build OS-BENCH around this observation using a differential-coverage pipeline. We first build and test candidate repositories in native Linux and Windows environments, compare their coverage traces, and verify which differences reflect platform-specific behavior. We then remove verified target-platform implementations to construct migration tasks, retaining tests that fail when the implementations are missing and regression tests that check unrelated behaviors are preserved. This procedure creates controlled restoration tasks without requiring migration changes to align with historical commit boundaries.

OS-BENCH contains 140 executable tasks from 111 open-source repositories spanning C, C++, C#, Go, and Rust. It includes both migration directions: 91 tasks migrate functionality supported on Windows to Linux, while 49 migrate Linux-supported functionality to Windows. Each task provides a repository at a reproducible base commit, a natural-language migration objective, an executable environment, fail-to-pass tests for the target functionality, and pass-to-pass tests for regression checking. We evaluate each patch on both the target and source platforms to check that it restores the missing functionality while preserving existing behavior under the test suites.

We evaluate six models using three Linux agent harnesses—SWE-agent, Claude Code, and OpenHands—as well as the Windows-native Win-agent harness. The strongest configuration solves only 26.4% of Linux-target tasks, while the best Windows configuration reaches 20.4% on Windows-target tasks when success requires the patch to work correctly on both systems. Performance also varies substantially across languages, migration directions, and agent harnesses. In particular, changing the harness can noticeably change the performance of the same underlying model. These results show that reliable migration remains difficult and that performance depends on both the model and the tools it uses to interact with the operating system.

Our main contributions are as follows:

- We formulate cross-OS code migration as an executable repository-level SWE task, where an agent must implement missing behavior on a target platform without breaking functionality on the source platform.
- We introduce an LM-assisted, scalable task-construction pipeline that uses differential execution coverage between Linux and Windows to identify and verify platform-specific implementations without relying on historical commit boundaries.
- We release OS-BENCH, containing 140 validated tasks from 111 repositories across five programming languages and both Linux-to-Windows and Windows-to-Linux migration directions.
- We evaluate six models with four agent harnesses and find low cross-platform success rates, with substantial variation across languages, migration directions, and agent designs.

## 2 RELATED WORK

### 2.1 SWE BENCHMARKS

**General SWE benchmarks.** Early code-generation benchmarks such as HumanEval (Chen et al., 2021) and LiveCodeBench (Jain et al., 2025) largely focus on self-contained, function-level programming problems. The SWE-bench family (Jimenez et al., 2024; Zhang et al., 2025; Deng et al., 2025; Jin et al., 2026) moves beyond this setting to repository-level issue resolution, where agents must identify relevant code, reason about dependencies, modify multiple files when necessary, and validate their changes in executable environments. Agentic systems such as RepoLaunch (Li et al., 2026b) further improve the scalability of constructing and reproducing these repository environments. While SWE-bench primarily centers on resolving reported software issues, ProgramBench (Yang et al., 2026) evaluates whether language models can reconstruct complete software repositories from scratch. These benchmarks typically execute and evaluate each task within a single OS environment, leaving the transfer of functionality across platform-specific implementations largely untested.

**Specialized SWE benchmarks.** More recent benchmarks have started to focus on specific forms of software maintenance and evolution. MigrationBench (Liu et al., 2026), for example, evaluates repository-scale migrations between Java versions. MigGPT (Dang et al., 2026) studies compatibility changes required by third-party Linux libraries across Linux versions, while ChainSWE (Jin et al., 2026) evaluates sequences of interdependent software engineering tasks that reflect how repositories evolve over time. OS-BENCH extends this direction to cross-OS migration, where an agent must translate functionality across different platform APIs and runtime behavior while preserving correctness on the original system. To the best of our knowledge, OS-BENCH is the first executable repository-level benchmark specifically designed for Linux–Windows code migration.

### 2.2 SWE AGENTS

SWE agents combine LMs with tools for navigating repositories, editing code, executing commands, and iterating on test results. Open-source systems such as SWE-agent (Yang et al., 2024; Zhang et al., 2026) and OpenHands (Wang et al., 2025) provide agent–computer interfaces that allow models to inspect files, run shell commands, apply patches, and respond to execution feedback. Commercial coding agents, including Codex (OpenAI, 2025) and Claude Code (Anthropic, 2025), further support capabilities such as task planning, long-context repository analysis, and delegated subtasks/subagents. More general computer-use agents, such as OpenClaw (Steinberger, 2026), extend this style of tool-mediated interaction beyond source-code repositories to broader OS workflows.

Despite recent progress, most SWE agent benchmarks and tool interfaces remain Linux-centric. Native Windows execution poses distinct challenges in shell semantics, path and escaping conventions, filesystem behavior, process management, compilation, and container support, affecting both code modification and agent–environment interaction. Recent incidents where coding agents unexpectedly deleted user data or exposed credentials in Windows environments further highlight the need for Windows-specific training, tooling, and evaluation. Existing benchmarks rarely assess cross-platform

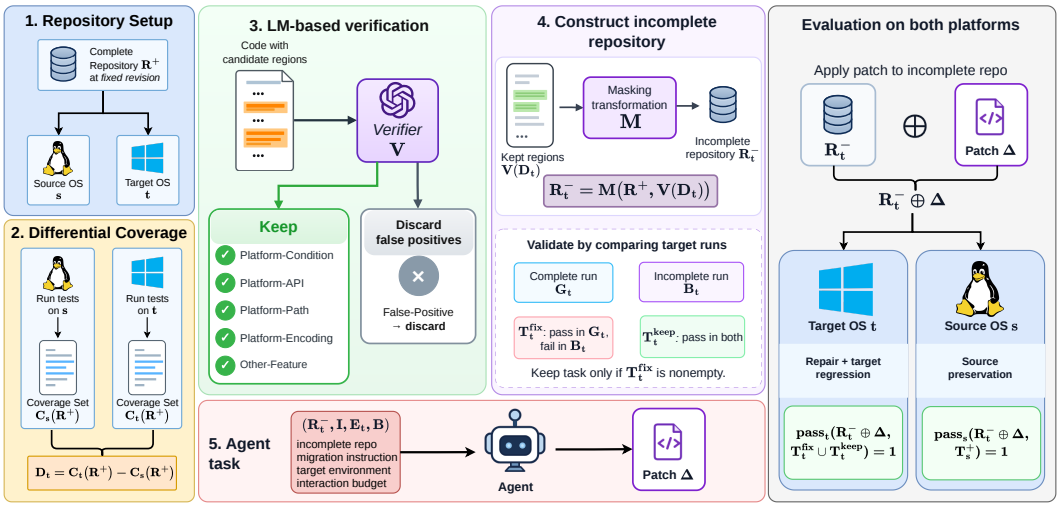


Figure 2: Overview of the OS-BENCH construction and evaluation pipeline.

correctness: implementing platform-specific behavior on Windows while preserving Linux compatibility, or vice versa. OS-BENCH addresses these gaps through native executable environments for both platforms, a Windows-oriented agent harness, and counter-platform regression testing.

## 3 OS-BENCH

An OS-BENCH task starts from a repository with part of its implementation for one operating system removed. The agent restores support for that target system, and evaluation checks the submitted patch on both the target and source platforms. Figure 2 summarizes the full OS-BENCH construction and evaluation pipeline.

### 3.1 TASK FORMULATION

Let $R^+$ denote the complete repository at a fixed revision, with executable test workloads on a source platform $s$ and a target platform $t$. Let $C_s(R)$ and $C_t(R)$ be the sets of source lines executed by the repository's test workload on the two platforms. We identify target-specific candidate code as

$$D_t = C_t(R^+) \setminus C_s(R^+). \tag{1}$$

An LM-based verifier $V$ examines the source code and filters out coverage differences unrelated to platform-specific behavior. A masking transformation $M$ then removes or replaces the verified target implementation:

$$R_t^- = M(R^+, V(D_t)). \tag{2}$$

The agent receives $(R_t^-, I, E_t, B)$: the incomplete repository, a natural-language migration instruction, the target execution environment, and a fixed interaction budget. It returns a patch $\Delta$.

Let $T_t^{\text{fix}}$ denote target tests that do not pass in $R_t^-$ but pass in $R^+$, $T_t^{\text{keep}}$ denote target tests that pass in both states, and $T_s^+$ denote tests that pass in the complete source environment. A patch solves the task only if it satisfies both

$$\text{pass}_t(R_t^- \oplus \Delta, T_t^{\text{fix}} \cup T_t^{\text{keep}}) = 1, \tag{3}$$

$$\text{pass}_s(R_t^- \oplus \Delta, T_s^+) = 1. \tag{4}$$

The first condition checks target repair and regression tests; the second checks source-platform preservation. We also require the source reference set $T_s^+$ to be nonempty. The same patch is evaluated in both native environments.

**Why test both platforms?** A target test suite cannot directly exercise code compiled only on the source OS. A patch can therefore pass the target suite while breaking a source-only branch or removing a guard needed by the source build. Running the patch on both systems checks behavior that would otherwise go unobserved. As with any test-based evaluation, passing these checks establishes correctness only for the behaviors they cover.

### 3.2 Repository and Environment Selection

We start from popular open-source repositories written in C, C++, C#, Go, and Rust whose continuous-integration configurations indicate support for both Linux and Windows. For each repository, RepoLaunch (Li et al., 2026b) checks out a fixed revision, resolves dependencies, builds the project in a native environment, identifies a stable test command, and constructs a parser that converts raw test output into test names and statuses. A repository is retained only when usable environments can be built on both operating systems.

The complete repository provides the reference behavior for task construction and evaluation. The environments retain platform-native shells and toolchains, since path handling, command execution, and system APIs are part of the migration problem.

### 3.3 Differential Coverage and Verification

We run each repository's tests with language-appropriate line-coverage instrumentation on both Linux and Windows. Coverage paths are normalized to the repository root while accounting for differences such as platform-specific separators, generated build prefixes, Go module paths, and Rust registry paths. For each file, we identify lines executed only on Linux or only on Windows and merge neighboring lines into candidate regions.

Coverage asymmetry alone does not establish that code is platform-specific. A line may execute on only one system because of differences in test selection, nondeterminism, dependency versions, or unrelated feature flags. We therefore use an LM-based verifier that receives the complete source file with Linux-only and Windows-only regions annotated and assigns each region one of six labels:

- Platform-Conditional: compile-time guards, build tags, platform-specific filenames, or runtime OS checks;
- Platform-API: system calls, headers, commands, libraries, constants, or other facilities specific to one OS family;
- Platform-Path: differences in separators, roots, drive prefixes, case sensitivity, or path-list semantics;
- Platform-Encoding: line endings, locales, console code pages, Unicode conversion, or other platform-specific text representations;
- Other-Migration-Feature: portability shims, dependency substitutions, capability probes, build adaptations, and related compatibility logic;
- False-Positive: asymmetric execution with no inherent platform-specific semantics.

Regions assigned to the first five categories were retained while those labeled False-Positive were discarded. The LM-retained lines were further filtered by human. For each migration direction, the retained regions in a repository form one task. A task may therefore involve several categories and require changes across multiple files.

### 3.4 Constructing the Incomplete State

For each retained repository, we first rebuild and test the complete version on the target platform and record its reference test status $G_t$. An LM-based masker is prompted to substitute every verified target-only region with a placeholder which would be silently bypassed during execution, while avoiding inserting obvious exceptions or a replacement implementation to maintain a sufficiently high level of task difficulty. Test files are excluded by default. The LM-masked fields are manually verified to ensure that: (1) It masked all verified target-specific regions selected for the task with

Table 1: Composition of OS-BENCH. The first four columns report counts, while the last four report per-task averages. Fail-to-pass (F2P) checks pass in the complete target repository and stop passing after masking; pass-to-pass (P2P) checks pass in both states. Files and estimated lines of code (LoC) are computed over the verified regions of each task; a file may therefore be counted in tasks from both migration directions.

| | **Counts** | | | | **Per-task averages** | | | |
|---|---|---|---|---|---|---|---|---|
| Language | Repos | Tasks | Win→Linux | Linux→Win | F2P checks | P2P checks | Files | LoC |
| C | 2 | 2 | 1 | 1 | 21.5 | 12.5 | 2.0 | 73.0 |
| C++ | 6 | 7 | 6 | 1 | 48.9 | 57.0 | 9.7 | 669.3 |
| C# | 31 | 34 | 24 | 10 | 675.1 | 3101.4 | 6.2 | 99.0 |
| Go | 28 | 38 | 20 | 18 | 509.8 | 624.8 | 9.0 | 324.3 |
| Rust | 44 | 59 | 40 | 19 | 253.6 | 77.2 | 6.1 | 295.2 |
| Overall | 111 | 140 | 91 | 49 | 411.9 | 958.4 | 7.0 | 271.0 |

no exceptions or re-implementations inserted; and (2) It did not touch source-platform codes and source-platform test suite $T_s^+$ remains passed. The violations are manually fixed.

We then rebuild the incomplete repository and run the same target test suite to obtain status $B_t$. The task labels are then derived directly from the difference between the complete and incomplete states:

$$T_t^{\text{fix}} = \{x \mid G_t(x) = \text{pass} \wedge B_t(x) \neq \text{pass}\}, \tag{5}$$

$$T_t^{\text{keep}} = \{x \mid G_t(x) = \text{pass} \wedge B_t(x) = \text{pass}\}. \tag{6}$$

A candidate is kept only when $T_t^{\text{fix}}$ is nonempty. Thus every retained task has at least one previously passing test that fails, is skipped, or is absent after masking. The reverse of the masking diff provides the reference (gold) repair patch.

To prevent recovery of the removed implementation from Git history, the original *.git* data including the Git remote URL were removed and the repo is re-initialized with a root commit.

We assign each task a construction-time difficulty label based on the size of the verified migration region. A task is *hard* if the region exceeds 100 estimated lines or spans more than five files, *medium* if it exceeds 30 lines or two files, and *easy* otherwise.

### 3.5 BENCHMARK COMPOSITION

Table 1 summarizes the released benchmark. OS-BENCH contains 140 tasks drawn from 111 repositories, with 29 repositories contributing one task in both migration directions. On average, each task spans seven files and 271 estimated platform-specific lines, while the largest task covers 152 files and 5,287 estimated lines. Across all tasks, the gold patches add 56,908 lines and remove 14,459 lines. The benchmark contains 44 easy, 38 medium, and 58 hard tasks.

## 4 EXPERIMENTS

We evaluate how reliably agents restore target-platform functionality while preserving source-platform behavior, and how success varies with migration direction, language, and agent harness.

**Models and harnesses.** We evaluate GPT-5.6-Sol, GPT-5.6-Terra, GPT-5.6-Luna, Claude-Opus-4.8, Claude-Sonnet-5, and DeepSeek-V4-Pro with medium thinking effort. For the 91 Linux-target tasks, each model runs with Claude Code (Anthropic, 2025), OpenHands (Wang et al., 2025), and SWE-agent (Yang et al., 2024). As the three Linux-based agents currently do not support running in Windows docker containers, for the 49 Windows-target tasks, each model runs with Win-agent (Li et al., 2026b), a native Windows harness with a tool set aligned with that of Linux SWE-agent: file viewing, creation and editing, PowerShell command execution, and patch submission. Section B compares the tool sets of the four agent harnesses. Each run is limited to 100 agent steps and five hours. The task instruction specifies the migration direction and asks the agent to fix compatibility issues and pass the target regression tests; each harness supplies its own workflow instructions. Section D reproduces the prompt templates.

Table 2: Cross-platform success rates grouped by model. Linux evaluations use SWE-agent, Claude Code, and OpenHands, while Windows evaluations use Win-agent. Thinking effort is set to *medium* for all models for a fair comparison.

| Model | Platform | Agent | C/C++ | C# | Go | Rust | Overall |
|---|---|---|---|---|---|---|---|
| **GPT-5.6-Sol** | Linux | SWE-agent | 0.0% | 12.5% | 5.0% | 32.5% | 18.7% |
| | | Claude Code | 0.0% | 12.5% | 5.0% | 45.0% | 24.2% |
| | | OpenHands | 0.0% | 12.5% | 15.0% | 32.5% | 20.9% |
| | Windows | Win-agent | 0.0% | 40.0% | 5.6% | 26.3% | 20.4% |
| **GPT-5.6-Terra** | Linux | SWE-agent | 0.0% | 12.5% | 15.0% | 32.5% | 20.9% |
| | | Claude Code | 0.0% | 12.5% | 10.0% | 35.0% | 20.9% |
| | | OpenHands | 0.0% | 12.5% | 15.0% | 30.0% | 19.8% |
| | Windows | Win-agent | 0.0% | 30.0% | 0.0% | 10.5% | 10.2% |
| **GPT-5.6-Luna** | Linux | SWE-agent | 0.0% | 12.5% | 0.0% | 17.5% | 11.0% |
| | | Claude Code | 0.0% | 8.3% | 15.0% | 27.5% | 17.6% |
| | | OpenHands | 0.0% | 8.3% | 15.0% | 25.0% | 16.5% |
| | Windows | Win-agent | 0.0% | 20.0% | 0.0% | 0.0% | 4.1% |
| **Claude-Opus-4.8** | Linux | SWE-agent | 0.0% | 16.7% | 15.0% | 40.0% | 25.3% |
| | | Claude Code | 0.0% | 12.5% | 10.0% | 32.5% | 19.8% |
| | | OpenHands | 0.0% | 12.5% | 15.0% | 45.0% | 26.4% |
| | Windows | Win-agent | 50.0% | 40.0% | 5.6% | 10.5% | 16.3% |
| **Claude-Sonnet-5** | Linux | SWE-agent | 0.0% | 0.0% | 0.0% | 25.0% | 11.0% |
| | | Claude Code | 0.0% | 4.2% | 5.0% | 32.5% | 16.5% |
| | | OpenHands | 0.0% | 12.5% | 10.0% | 40.0% | 23.1% |
| | Windows | Win-agent | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **DeepSeek-V4-Pro** | Linux | SWE-agent | 0.0% | 12.5% | 15.0% | 32.5% | 20.9% |
| | | Claude Code | 0.0% | 4.2% | 15.0% | 32.5% | 18.7% |
| | | OpenHands | 0.0% | 8.3% | 5.0% | 32.5% | 17.6% |
| | Windows | Win-agent | 0.0% | 20.0% | 0.0% | 0.0% | 4.1% |

**Evaluation.** We apply each submitted patch to the incomplete repository and evaluate it in two stages. Stage I requires every target fail-to-pass and pass-to-pass check to pass. Stage II applies the same patch in the native source environment and requires every test that passed in the complete source repository to remain passing, with a nonempty reference set. The primary metric is the percentage of tasks that pass both stages, using all 91 or 49 tasks in the corresponding split as the denominator. Runs without a validated success contribute no successes. Section A gives the full protocol.

## 5 Results

Table 2 reports success when both target repair and source preservation are required, while Section C reports Stage I success separately.

### 5.1 Overall Performance and Source Preservation

The strongest Linux configuration, Claude-Opus-4.8 with OpenHands, solves 24 of 91 tasks (26.4%). Claude-Opus-4.8 with SWE-agent follows at 25.3%, while GPT-5.6-Sol with Claude Code reaches 24.2%. On Windows, GPT-5.6-Sol with Win-agent leads with 10 of 49 tasks solved (20.4%), followed by Claude-Opus-4.8 at 16.3%. Thus, even the strongest configurations solve only about one quarter of Linux-target tasks and one fifth of Windows-target tasks.

Source-platform validation substantially changes the measured outcome. For GPT-5.6-Sol with Claude Code, 50 Linux-target patches pass Stage I, but only 22 pass both stages: success falls from 54.9% to 24.2%. Of the 50 target-accepted patches, 28 (56.0%) lack a validated source-platform pass. For GPT-5.6-Sol with Win-agent, 13 Windows-target patches pass Stage I and ten pass both stages (26.5% versus 20.4%). Passing the target suite alone is therefore insufficient evidence of a successful migration. Section C gives the Stage-I-only results and their gaps from cross-platform success.

Table 3: Failure rates by migration category for GPT-5.6-Sol with Claude Code (Linux) and Win-agent (Windows). Target is the percentage failing Stage I; Both is the percentage failing the combined Stage I–II criterion. The denominator is the task count in each row and direction. Labels overlap, so task counts should not be summed across rows.

| | Windows→Linux | | | Linux→Windows | | |
|---|---|---|---|---|---|---|
| Category | Tasks | Target | Both | Tasks | Target | Both |
| Platform conditionals | 71 | 43.7% | 76.1% | 38 | 68.4% | 76.3% |
| OS / library APIs | 61 | 52.5% | 78.7% | 28 | 71.4% | 75.0% |
| Paths | 56 | 44.6% | 78.6% | 32 | 78.1% | 81.2% |
| Encoding | 33 | 57.6% | 81.8% | 16 | 81.2% | 93.8% |
| Other compatibility | 45 | 57.8% | 82.2% | 24 | 66.7% | 70.8% |

### 5.2 Migration Direction and Programming Language

Every model scores lower on Windows-target tasks than with its strongest Linux harness, and the best Windows score trails the best Linux score by 6.0 percentage points. The model ranking also changes: GPT-5.6-Sol leads on Windows, whereas Claude-Opus-4.8 leads on Linux. GPT-5.6-Terra reaches 19.8–20.9% across Linux harnesses but 10.2% on Windows. Claude-Sonnet-5 reaches 23.1% with OpenHands on Linux but solves no Windows tasks with Win-agent, while GPT-5.6-Luna and DeepSeek-V4-Pro each solve two Windows tasks (4.1%). These comparisons reflect the combined differences in task composition, model capability, and harness on the two different platforms.

Success is uneven across languages. On Linux, Rust accounts for 18 of the 24 successes of Claude-Opus-4.8 with OpenHands and 18 of the 22 successes of GPT-5.6-Sol with Claude Code. No Linux configuration solves a C/C++ task on both platforms, and Go success never exceeds 15.0%. On Windows, GPT-5.6-Sol and Claude-Opus-4.8 each solve four of ten C# tasks (40.0%), while their Go success is only one of 18 tasks (5.6%). GPT-5.6-Sol also leads on Windows Rust tasks, solving five of 19 (26.3%). The language profile therefore changes with the migration direction: Rust accounts for most Linux successes, whereas C# contributes a larger share of Windows successes. The small C/C++ groups—seven Linux tasks and two Windows tasks—limit broader conclusions about these languages.

### 5.3 Sensitivity to the Agent Harness

Harness choice also matters when the model and task split are held fixed. Claude-Sonnet-5 ranges from ten Linux successes with SWE-agent (11.0%) to 21 with OpenHands (23.1%), an 11-task difference. Claude-Opus-4.8 and GPT-5.6-Luna each show a six-task difference between their strongest and weakest Linux harnesses. For Claude-Opus-4.8, OpenHands solves 24 tasks compared with 18 for Claude Code; for GPT-5.6-Luna, Claude Code solves 16 compared with ten for SWE-agent. No harness is best for every model: OpenHands leads for Claude-Opus-4.8 and Claude-Sonnet-5, Claude Code leads for GPT-5.6-Sol and GPT-5.6-Luna, and SWE-agent leads for DeepSeek-V4-Pro. GPT-5.6-Terra ties between SWE-agent and Claude Code. The differences also depend on language: GPT-5.6-Sol with Claude Code solves more Rust tasks than with OpenHands (18 versus 13), whereas OpenHands solves more Go tasks (three versus one). Model and harness should therefore be evaluated together.

Harness rankings can also change when source preservation is required. For GPT-5.6-Luna, OpenHands has the higher target-only success rate (46.2% versus 41.8% with Claude Code), but Claude Code has the higher both-platform rate (17.6% versus 16.5%). A harness that restores more target implementations can therefore produce fewer validated migrations. This reversal reinforces the need to evaluate the same patches on both systems when comparing agent configurations.

### 5.4 Failure Patterns

**Failures span platform behaviors.** Table 3 reports failure rates for GPT-5.6-Sol with Claude Code on Linux and Win-agent on Windows. The categories overlap: a task involving both system APIs and path handling appears in both rows, so they describe task contents rather than independent causes of failure. On Linux, 76.1–82.2% of tasks in each category remain unsolved under the combined

Table 4: Observed repair and verification patterns in the 35 recorded Stage I failures for GPT-5.6-Sol with Claude Code on Linux. Each share uses all 35 runs as its denominator. Labels overlap: a run can exhibit multiple patterns, so the shares do not sum to 100%. Definitions are given in Section 5.4.

| Failure pattern | Runs | Share |
|---|---|---|
| Unrestored implementation | 18 | 51.4% |
| Reduced verification scope | 19 | 54.3% |
| Turn budget exhausted | 2 | 5.7% |

criterion. For path-related tasks, the unsolved count increases from 25 of 56 under target-only evaluation to 44 of 56 when source preservation is required. On Windows, 15 of 16 tasks involving encoding remain unsolved, as do 26 of 32 involving paths. The difficulty therefore extends across conditional logic, APIs, paths, and encoding.

**Partial repairs leave migration work incomplete.** We audit the 35 Linux runs explicitly recorded as Stage I failures for GPT-5.6-Sol with Claude Code. This subset excludes incomplete or absent evaluations and patches that fail only at Stage II. Table 4 summarizes the audit; *turn budget exhausted* denotes runs that reach their interaction limit. In 18 of the 35 runs (51.4%), at least one untouched production file retains missing behavior. For example, in `yarp`, the agent restores certificate loading but leaves service registration incomplete: the reference repair also registers HTTP.sys delegation on Windows and a no-op fallback on Linux. In `yek`, the agent adds Unix permission checks to handle tests run as a privileged user, but leaves the output-path logic incomplete. When both an output directory and filename are supplied, the masked code still uses the filename alone rather than joining it with the directory.

**Local verification can omit required checks.** In 19 audited runs (54.3%) in Table 4, the agent's verification excludes tests, suites, or build modes required by the benchmark. In `simdjson`, the agent disables single-header generation in its local build, although `simdjson-singleheader` is a required F2P check. Its 108-test run therefore does not establish that the benchmark's required workload passes. In `difftastic`, the final test output marks `test_gzip_is_binary` as ignored, although this test belongs to the required F2P set and the benchmark explicitly runs ignored tests. These omissions can coexist with incomplete repairs; the audit labels are not mutually exclusive. The benchmark marks such runs unsuccessful when required checks do not pass, and the traces do not establish that the omissions were deliberate.

## 6 Conclusion

We introduce OS-Bench, a benchmark of 140 code migration tasks between Linux and Windows. Each task asks an agent to restore missing functionality on one platform while preserving behavior on the other. The best evaluated configurations solve 26.4% of Windows-to-Linux tasks and 20.4% of Linux-to-Windows tasks under this requirement. These results reveal that current agents still struggle with the demands of reliable cross-platform migration. Closing this gap requires agents to track the full scope of a migration, restore platform-specific behavior across files, and validate changes on both operating systems. OS-Bench provides a testbed for measuring progress toward these capabilities.

## Limitations

Our task-construction pipeline relies on cross-OS differences in execution coverage, so it cannot detect migration features implemented through cross-platform abstractions that follow the same code paths on both systems. Recovering such cases would require mining commit history to infer when and how cross-platform support was introduced. However, commits often mix unrelated changes and are rarely labeled as platform migrations, making similar large-scale automation like this work difficult and requiring substantial manual labeling effort. We therefore leave the use of historical signals to discover these additional migration patterns to future work.

### AI USE STATEMENT

In this work, we used generative AI tools for generating synthetic datasets (execution coverage diff filtering and platform-specific source code masking). We have not used generative AI tools to design or provide feedback on research methodology or experiments, implement methods, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, or interpret results, and using AI to help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses is not applicable to this work. We have reviewed all AI-assisted work. The coverage filtering and code masking results have all been manually verified. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

## REFERENCES

Anthropic. Claude code. https://code.claude.com/, 2025.

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.

Pucheng Dang, Di Huang, Dong Li, Kang Chen, Yuanbo Wen, Qi Guo, and Xing Hu. Miggpt: Harnessing large language models for automated migration of out-of-tree linux kernel patches across versions. *Advances in Neural Information Processing Systems*, 38:38818–38851, 2026.

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.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

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? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Qirui Jin, Lingching Tung, Kenan Li, Qiyang Shi, Yushi She, Huanzhong Jia, Harrison Zhao, Kejing Xia, Zhenbang Du, Jiaxin Pei, Zhenyu Zhang, Zhen Qi, Yuyan Duan, Wenke Lee, and Zijian Jin. Chainswe: Benchmarking coding agents on multi-bug software maintenance, 2026. URL https://arxiv.org/abs/2607.02606.

Kenan Li, Qirui Jin, Liao Zhu, Xiaosong Huang, Yijia Wu, Yikai Zhang, Xin Zhang, Zijian Jin, Yufan Huang, Elsie Nallipogu, Chaoyun Zhang, Yu Kang, Saravan Rajmohan, Qingwei Lin, Wenke Lee, and Dongmei Zhang. Oracle-swe: Quantifying the contribution of oracle information signals on swe agents, 2026a. URL https://arxiv.org/abs/2604.07789.

Kenan Li, Rongzhi Li, Linghao Zhang, Qirui Jin, Liao Zhu, Xiaosong Huang, Geng Zhang, Yikai Zhang, Shilin He, Chengxing Xie, Xin Zhang, Zijian Jin, Bowen Li, Chaoyun Zhang, Yu Kang, Yufan Huang, Elsie Nallipogu, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. Repolaunch:

Automating build and management of code repositories across languages and platforms, 2026b. URL https://arxiv.org/abs/2603.05026.

Linbo Liu, Xinle Liu, Qiang Zhou, Lin Chen, Yihan Liu, Hoan Nguyen, Behrooz Omidvar Tehrani, Xi Shen, Jun Huan, Omer Tripp, and Anoop Deoras. Migrationbench: Repository-level code migration benchmark from java 8. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '26, pp. 9427–9438, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722592. doi: 10.1145/3770855.3817526. URL https://doi.org/10.1145/3770855.3817526.

OpenAI. Codex. https://openai.com/codex/, 2025.

Peter Steinberger. Openclaw : The ai that really does things. https://github.com/openclaw/openclaw, 2026.

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, Daniel 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. In *The Thirteenth International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

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. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Programbench: Can language models rebuild programs from scratch?, 2026. URL https://arxiv.org/abs/2605.03546.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live! *Advances in Neural Information Processing Systems*, 38, 2025.

Yikai Zhang, Jiaxin Pei, Kenan Li, Qirui Jin, Maoquan Wang, Jin Pan, Yu Kang, Shengyu Fu, Elsie Nallipogu, Junjie Hu, Yufan Huang, and Zijian Jin. Swe-edit: Rethinking code editing for efficient swe-agent, 2026. URL https://arxiv.org/abs/2604.26102.

# A Detailed Experiments

## A.1 Budgets and Execution Environments

Each run receives a limit of 100 agent steps or model calls, according to the harness, with a five-hour wall-clock limit; SWE-agent stops at its 100-call limit. Linux agents work in native Linux environments, and Win-agent uses a native Windows environment with PowerShell. The prompts specify the migration direction and target-platform test objective, while the evaluator independently checks source-platform preservation. The wrappers in Section D also constrain test-file edits and Git use.

## A.2 Evaluation Stages

**Stage I: target-platform repair.** We apply the submitted patch to the incomplete repository, run the native build and test commands, and parse the output into individual test results. Every required fail-to-pass (F2P) and pass-to-pass (P2P) check must be recorded as passing. Let $\mathrm{status}_p(x;\Delta)$ denote the recorded status of test $x$ on platform $p$ after applying patch $\Delta$. The target check is

$$S_t(\Delta) = \mathbb{1}\left[\bigwedge_{x \in T_t^{\mathrm{fix}} \cup T_t^{\mathrm{keep}}} \mathrm{status}_t(x;\Delta) = \mathrm{pass}\right]. \tag{7}$$

A compilation failure, timeout, missing required test result, or failed required test makes the run unsuccessful.

**Stage II: source-platform preservation.** Patches that pass Stage I are tested on the source platform. We first run the complete repository in the native source environment and collect its passing tests in the reference set $T_s^+$. We then reverse the gold repair patch to recreate the incomplete repository, apply the agent's patch unchanged, and rerun the same tests. Stage II passes only if the reference set is nonempty and every test in it still passes:

$$S_s(\Delta) = \mathbb{1}\left[|T_s^+| > 0 \wedge \bigwedge_{x \in T_s^+} \mathrm{status}_s(x;\Delta) = \mathrm{pass}\right]. \tag{8}$$

Tests that already failed in the complete source repository are excluded. A migration is solved only when both stages pass:

$$S(\Delta) = S_t(\Delta) \cdot S_s(\Delta). \tag{9}$$

# B Agent Harness Tool Set Comparison

Table 5 compares the tool sets of the four agent harnesses under evaluation.

Table 5: Comparison of the tool sets of the four agent harnesses under evaluation.

| Agent Harness | Win-agent | SWE-agent | OpenHands | ClaudeCode |
|---|---|---|---|---|
| Supported docker container type | Linux&Windows | Linux | Linux | Linux |
| File view/create/edit | ✓ | ✓ | ✓ | ✓ |
| Shell command execution | ✓(Powershell) | ✓(Unix Bash) | ✓(Unix Bash) | ✓(Unix Bash) |
| Task planning | × | × | ✓ | ✓ |
| Sub-agent delegation | × | × | × | ✓ |

# C Single-Platform Success Rates

Single-platform success measures target repair alone: every required target F2P and P2P check must pass, without imposing the source-platform check. It uses the same patches, budgets, task splits, and denominators as the main evaluation. Table 6 reports the number and percentage of accepted tasks

under this criterion alongside success on both platforms. The gap is computed from the difference in accepted task counts before rounding, in percentage points. Claude-Sonnet-5 uses the union of two runs under each criterion, counting each task at most once. A both-platform success requires the same patch to pass both stages.

Table 6: Single-platform (Stage I) and cross-platform (Stages I–II) success. Cells give accepted tasks and percentages, with denominators of 91 for Linux and 49 for Windows. Gap is the difference in percentage points computed before rounding. Claude-Sonnet-5 pools two runs; see Section C.

| Model | Target | Agent | Target only | Both platforms | Gap |
|---|---|---|---|---|---|
| GPT-5.6-Sol | Linux | SWE-agent | 43 (47.3%) | 17 (18.7%) | 28.6 |
| | Linux | Claude Code | 50 (54.9%) | 22 (24.2%) | 30.8 |
| | Linux | OpenHands | 43 (47.3%) | 19 (20.9%) | 26.4 |
| | Windows | Win-agent | 13 (26.5%) | 10 (20.4%) | 6.1 |
| GPT-5.6-Terra | Linux | SWE-agent | 40 (44.0%) | 19 (20.9%) | 23.1 |
| | Linux | Claude Code | 42 (46.2%) | 19 (20.9%) | 25.3 |
| | Linux | OpenHands | 45 (49.5%) | 18 (19.8%) | 29.7 |
| | Windows | Win-agent | 11 (22.4%) | 5 (10.2%) | 12.2 |
| GPT-5.6-Luna | Linux | SWE-agent | 20 (22.0%) | 10 (11.0%) | 11.0 |
| | Linux | Claude Code | 38 (41.8%) | 16 (17.6%) | 24.2 |
| | Linux | OpenHands | 42 (46.2%) | 15 (16.5%) | 29.7 |
| | Windows | Win-agent | 6 (12.2%) | 2 (4.1%) | 8.2 |
| Claude-Opus-4.8 | Linux | SWE-agent | 49 (53.8%) | 23 (25.3%) | 28.6 |
| | Linux | Claude Code | 39 (42.9%) | 18 (19.8%) | 23.1 |
| | Linux | OpenHands | 50 (54.9%) | 24 (26.4%) | 28.6 |
| | Windows | Win-agent | 11 (22.4%) | 8 (16.3%) | 6.1 |
| Claude-Sonnet-5 | Linux | SWE-agent | 22 (24.2%) | 10 (11.0%) | 13.2 |
| | Linux | Claude Code | 41 (45.1%) | 15 (16.5%) | 28.6 |
| | Linux | OpenHands | 44 (48.4%) | 21 (23.1%) | 25.3 |
| | Windows | Win-agent | 4 (8.2%) | 0 (0.0%) | 8.2 |
| DeepSeek-V4-Pro | Linux | SWE-agent | 36 (39.6%) | 19 (20.9%) | 18.7 |
| | Linux | Claude Code | 32 (35.2%) | 17 (18.7%) | 16.5 |
| | Linux | OpenHands | 35 (38.5%) | 16 (17.6%) | 20.9 |
| | Windows | Win-agent | 4 (8.2%) | 2 (4.1%) | 4.1 |

**Target success substantially exceeds migration success.** For GPT-5.6-Sol with Claude Code, target-only evaluation accepts 50 of 91 Linux tasks (54.9%), while both-platform evaluation accepts 22 (24.2%). The 28-task difference is 30.8 percentage points, and 56.0% of target-accepted tasks lack a validated source-platform pass. On Windows, the same model with Win-agent drops from 13 of 49 tasks (26.5%) to ten (20.4%), a 6.1-point gap. These differences measure how much target-only evaluation overstates the number of validated migrations; they do not by themselves identify the cause of each unsuccessful source evaluation.

**The gap depends on the harness.** For GPT-5.6-Luna on Linux, OpenHands has the higher target-only rate (46.2% versus 41.8% with Claude Code), but Claude Code has the higher both-platform rate (17.6% versus 16.5%). Thus, a harness ranking based on target repair can change when source preservation is required. Across the reported configurations, Linux target-only rates range from 22.0% to 54.9%, compared with 11.0–26.4% on both platforms. On Windows, the corresponding ranges are 8.2–26.5% and 0.0–20.4%.

## D PROMPTS

This section reproduces the benchmark-specific prompt templates in the construction pipeline and evaluation harnesses.

### D.1 PLATFORM-SPECIFIC REGION VERIFICATION

**Verification Prompt**

```
You are a senior cross-platform migration specialist with deep expertise in {language}
↪ platform-specific code. Your task is to classify code regions that execute on only
↪ one platform (Linux or Windows) and determine whether they represent genuine
↪ platform dependencies or false positives.

## Source File: {filepath}

The complete source file is shown below with annotations:
- [L] = Line executes ONLY on Linux (not on Windows)
- [W] = Line executes ONLY on Windows (not on Linux)
- (no marker) = Line executes on both platforms or is not in coverage

```{language_ext}
{annotated_source}
```

## Coverage Analysis

Linux-only line ranges (execute on Linux, NOT Windows):
{linux_only_summary}

Windows-only line ranges (execute on Windows, NOT Linux):
{windows_only_summary}

## Classification Taxonomy

### Category Definitions ###

=== GENERAL CLASS DEFINITIONS ===

>>> PLATFORM_CONDITIONAL
  Code whose execution or compilation is selected by an explicit platform
  condition. This includes compile-time guards, build tags, platform-specific
  filenames, preprocessor branches, attributes, and runtime OS checks. Classify
  guarded code here even when the guarded statements are otherwise generic.

>>> PLATFORM_API
  Code that calls, imports, defines, wraps, or configures APIs, constants,
  commands, headers, crates, packages, namespaces, or system facilities that are
  specific to one OS family or behave incompatibly across OS families.

>>> PLATFORM_PATH
  Code that handles platform-dependent filesystem path syntax or semantics,
  including separators, drive prefixes, device names, root conventions, home or
  temp locations, path-list separators, and case-sensitivity behavior.

>>> PLATFORM_ENCODING
  Code that handles platform-dependent text, byte, locale, terminal, argument,
  environment, or line-ending representations, including Unicode conversion and
  console/codepage behavior.

>>> OTHER_MIGRATION_FEATURE
  Other platform-only code related to cross-platform migration or compatibility
  that does not fit the conditional/API/path/encoding classes. Use this for
  compatibility shims, fallback behavior, capability probing, dependency
  substitution, build/test adaptation, permission/process/filesystem behavior
  normalization, or other portability work needed for equivalent behavior across
  Linux and Windows.

>>> FALSE_POSITIVE
  Coverage differs between platforms, but the code is not inherently
  platform-specific. Use this for auto-generated files, test selection, timing,
  dependency/version skew, unrelated feature flags, incidental coverage gaps,
  or generic logic with no platform guard/API/path/encoding semantics.


### Language-specific Examples ###
{language_taxonomy}

## Task

For EACH contiguous block of platform-only lines listed above, classify it.

## IMPORTANT INSTRUCTIONS:
```

```
1. Analyze EVERY platform-only line block. Do not skip any.
2. Use the FULL file context to understand each region's purpose.
3. Code in directories like Interop/Linux/, Interop/Windows/, platform/, unix/, win32/
↪ is platform-specific by definition.
4. Consider the SURROUNDING code (guards, conditionals, includes) not just the marked
↪ lines themselves.
5. When a marked line is inside a platform-guarded block (e.g., inside #ifdef _WIN32 or
↪  #[cfg(unix)]), the entire block is platform-specific even if only some lines are
↪ marked.
6. Do NOT classify based on regex pattern matching alone. Use your understanding of
↪ what the code actually does.
7. For each region, also identify:
  - functional_area: the precise functionality this code provides (e.g., "POSIX symlink
↪  resolution", "sigaction handler setup", "terminal raw mode control", "Windows path
↪ separator normalization")
  - feature_group: a short snake_case key for grouping regions that require the same
↪ exact platform knowledge across files. Do NOT use broad same-type labels such as "
↪ posix_filesystem" when regions involve different knowledge like symlink resolution,
↪ directory iteration, chmod permissions, path separators, or timestamp semantics.
↪ Prefer precise keys such as "posix_symlink_resolution", "posix_directory_iteration",
↪  "windows_registry_lookup", "sigaction_handler_setup".
8. Return a JSON array with one entry per contiguous platform-only line block.
  - If Linux-only and Windows-only blocks are adjacent, still return them as
separate entries with the correct `platform` value.
9. If several adjacent discrete line numbers (such as {33, 35, 37, 40, 42}) are all
windows_only or linux_only, with no counterpart mixed in, and they either all belong to
↪  platform-specific or
all belong to non-platform-specific, then consider merging them into a single entry: [
↪ start_line=33, end_line=42]
in your answer list to compact your answer.
10. If there are too many diff lines so that your output is too long,
only output entries that are real platform-specific lines and ignore FALSE_POSITIVE
↪ lines in your answer.
The only requirement is that all the real platform-specific lines must be recorded in
↪ your answer.


Now, output a list[dict] in JSON format inside a ```json``` fence:
```json
[
  {
    "start_line": <int, boundary included>,
    "end_line": <int, boundary included>,
    "platform": "linux_only|windows_only",
    "is_platform_specific": <bool>,
    "classification": "PLATFORM_CONDITIONAL|PLATFORM_API|PLATFORM_PATH|
↪ PLATFORM_ENCODING|OTHER_MIGRATION_FEATURE|FALSE_POSITIVE",
    "functional_area": "<what this code does>",
    "feature_group": "<snake_case grouping key>",
    "explanation": "<cite specific APIs, guards, or patterns that led to this
↪ classification>"
  },
  {
    "start_line": <int>,
    "end_line": <int>,
    "platform": "linux_only|windows_only",
    "is_platform_specific": <bool>,
    "classification": "PLATFORM_CONDITIONAL|PLATFORM_API|PLATFORM_PATH|
↪ PLATFORM_ENCODING|OTHER_MIGRATION_FEATURE|FALSE_POSITIVE",
    "functional_area": "<what this code does>",
    "feature_group": "<snake_case grouping key>",
    "explanation": "<cite specific APIs, guards, or patterns that led to this
↪ classification>"
  },
  ......
]
```

For example:
```json
[
  {
    "start_line": 131,
    "end_line": 138,
    "platform": "windows_only",
    "is_platform_specific": false,
    "classification": "FALSE_POSITIVE",
    "functional_area": "terminal color reset dispatch",
```
```

```
    "feature_group": "termcolor_stream_wrapper",
    "explanation": "This block implements `WriteColor::reset` via straightforward
↪ delegation to the wrapped writer. There are no Windows-specific imports, cfg
↪ attributes, path semantics, or encoding handling in this code, so the Windows-only
↪ execution is an incidental coverage artifact."
  },
  {
    "start_line": 165,
    "end_line": 169,
    "platform": "windows_only",
    "is_platform_specific": false,
    "classification": "FALSE_POSITIVE",
    "functional_area": "unit testing of invalid utf8 byte patterns",
    "feature_group": "pattern_utf8_validation_tests",
    "explanation": "This test validates pattern_from_bytes using an invalid UTF-8 byte
↪ sequence. It is not platform-specific; it just happened to run only on Windows in
↪ the coverage data. The actually platform-conditional test in this file is the
↪ separate #[cfg(unix)] os() test, not this one."
  }
]
```

Return ONLY the JSON array, no commentary before or after.

**Language-specific taxonomy blocks.** Only the block matching the source file's language is inserted in each verifier call; C and C++ share a block.

Language Taxonomy Prompt

```
=== C/C++ PLATFORM-SPECIFIC TAXONOMY ===

>>> PLATFORM_CONDITIONAL -- Compile-time platform checks:
  #ifdef _WIN32, #ifdef __linux__, #ifdef __unix__, #ifdef __APPLE__
  #ifdef _MSC_VER (MSVC compiler), #ifdef __GNUC__ (GCC)
  #if defined(_WIN32), #if defined(__linux__)
  #ifndef _WIN32, #if !defined(_WIN32)
  #ifdef WINAPI, #ifdef _WINDOWS

>>> PLATFORM_API -- OS-specific API calls:
  Windows: CreateFile, ReadFile, WriteFile, CloseHandle
  Windows: GetLastError, FormatMessage, SetLastError
  Windows: WSAStartup, WSACleanup, SOCKET, WSADATA
  Windows: CreateProcess, WaitForSingleObject, CreateThread
  Windows: RegOpenKeyEx, RegQueryValueEx (Registry)
  Windows: LoadLibrary, GetProcAddress
  Linux/POSIX: fork(), exec*(), waitpid(), kill()
  Linux/POSIX: open(), read(), write(), close(), ioctl()
  Linux/POSIX: pthread_*, sigaction, signal()
  Linux/POSIX: mmap, mprotect, sysconf
  Headers: <windows.h>, <winsock2.h>, <unistd.h>, <sys/socket.h>

>>> PLATFORM_PATH -- File path and filesystem differences:
  Path separator \ vs /
  Drive letters (C:\, D:\)
  MAX_PATH vs PATH_MAX
  Case sensitivity in filenames
  Symlink handling differences

>>> PLATFORM_ENCODING -- Encoding/locale differences:
  Wide char (wchar_t) on Windows vs UTF-8 on Linux
  _wfopen vs fopen, wmain vs main
  Line ending \r\n vs \n
  Codepage handling

>>> OTHER_MIGRATION_FEATURE -- Other migration/compatibility features:
  Cross-platform compatibility wrappers or fallback implementations
  CMake/autotools/MSBuild source selection or configuration for portability
  Normalizing permissions, process, thread, terminal, or filesystem behavior
  Test helpers or adapters that validate equivalent behavior across platforms

>>> FALSE_POSITIVE -- Not platform-specific:
  Test infrastructure differences
  Compiler optimization differences affecting coverage
  Header inclusion order differences
  Googletest/CTest framework noise
```

```
  Standard library implementation differences (libc vs MSVCRT)
  that don't affect API behavior

=== C# PLATFORM-SPECIFIC TAXONOMY ===

>>> PLATFORM_CONDITIONAL -- Runtime platform checks:
  RuntimeInformation.IsOSPlatform(OSPlatform.Windows/Linux/OSX)
  OperatingSystem.IsWindows(), OperatingSystem.IsLinux(), OperatingSystem.IsMacOS()
  [SupportedOSPlatform("windows")], [UnsupportedOSPlatform("windows")]
  #if WINDOWS, #if LINUX
  PlatformID enum checks

>>> PLATFORM_API -- Windows-specific APIs requiring migration:
  Registry: Microsoft.Win32.Registry, RegistryKey, HKEY_*
  WMI: System.Management.*, ManagementObject, ManagementObjectSearcher
  COM: [ComImport], Type.GetTypeFromProgID, Marshal.ReleaseComObject
  GDI+: System.Drawing.* (Bitmap, Graphics, Image, Font) on .NET 6+
  ETW: TraceEventSession, ETWTraceEventSource
  Events: EventLog, PerformanceCounter, ServiceController, ServiceBase
  P/Invoke: [DllImport], kernel32, user32, advapi32, ntdll
  Identity: WindowsIdentity, WindowsPrincipal, LogonUser
  AD: System.DirectoryServices (non-LDAP)
  Audio: System.Media.SoundPlayer, NAudio
  Certificates: X509Store+StoreLocation.LocalMachine (Windows cert store)

>>> PLATFORM_PATH -- File path differences:
  Hardcoded C:\, D:\ paths, backslash-only path literals
  Path.DirectorySeparatorChar when used in platform-specific logic
  Environment.GetFolderPath with Windows-only SpecialFolder values

>>> PLATFORM_ENCODING -- Encoding differences:
  Console.OutputEncoding platform differences
  Line ending handling

>>> OTHER_MIGRATION_FEATURE -- Other migration/compatibility features:
  Cross-platform compatibility adapters or fallback implementations
  Target framework/runtime compatibility code for OS behavior parity
  Normalizing services, permissions, process, terminal, or environment behavior
  Test helpers or adapters that validate equivalent behavior across platforms

>>> FALSE_POSITIVE -- Not platform-specific:
  #if NETFRAMEWORK / #if NET6_0 (framework version, NOT platform)
  Standard BCL (System.IO.Path, System.Net.Http, System.Text.Json, LINQ)
  Async patterns (Task, async/await, CancellationToken)
  Modern frameworks (ASP.NET Core, EF Core, gRPC, HttpClient)
  Test infrastructure differences
  Auto generated files that are not actually source code written by human

=== GO PLATFORM-SPECIFIC TAXONOMY ===

>>> PLATFORM_CONDITIONAL -- Compile-time or runtime platform checks:
  runtime.GOOS == "linux" / "windows" / "darwin"
  //go:build linux, //go:build windows, //go:build unix
  // +build linux, // +build windows (legacy build tags)
  Files named *_linux.go, *_windows.go, *_darwin.go, *_unix.go
  Build constraints in file names (platform-specific file selection)

>>> PLATFORM_API -- OS-specific API calls:
  syscall.* (Syscall, RawSyscall, platform-specific constants)
  golang.org/x/sys/unix.*, golang.org/x/sys/windows.*
  os/signal handling differences
  os/exec with platform-specific commands
  os.Getenv("HOME") vs os.Getenv("USERPROFILE")
  net.Listen with platform-specific socket options

>>> PLATFORM_PATH -- File path and filesystem differences:
  filepath.Separator, os.PathSeparator, os.PathListSeparator
  filepath.ToSlash, filepath.FromSlash
  filepath.VolumeName (Windows drive letters)
  Hardcoded / vs \ in path construction
  Case sensitivity in file lookups

>>> PLATFORM_ENCODING -- Encoding/locale differences:
  Line ending normalization
  Console encoding/codepage differences
  Locale-dependent string operations

>>> OTHER_MIGRATION_FEATURE -- Other migration/compatibility features:
```

```
  Cross-platform compatibility wrappers or fallback implementations
  Build/module integration for target-specific dependencies
  Normalizing environment, permissions, process, terminal, or filesystem behavior
  Test helpers or adapters that validate equivalent behavior across platforms

>>> FALSE_POSITIVE -- Not platform-specific:
  Test infrastructure differences (test runner behavior)
  Environment variable differences unrelated to platform
  Timing-dependent coverage (goroutine scheduling)
  Template library functions that work identically on all platforms
  Pure computation / string manipulation that happens to have different coverage

=== RUST PLATFORM-SPECIFIC TAXONOMY ===

>>> PLATFORM_CONDITIONAL -- Compile-time or runtime platform checks:
  #[cfg(unix)], #[cfg(windows)], #[cfg(target_os = "linux")]
  #[cfg(target_family = "unix")], #[cfg(target_family = "windows")]
  cfg!(unix), cfg!(windows), cfg!(target_os = ...)
  #[cfg(not(windows))], #[cfg(any(unix, target_os = ...))]

>>> PLATFORM_API -- OS-specific API calls:
  std::os::unix::*, std::os::windows::*
  libc::* (fork, exec*, signal, pipe, ioctl, mmap, mprotect)
  winapi::*, windows_sys::*, windows::Win32::*
  nix::* (nix crate for Unix APIs)
  std::os::unix::fs::symlink, std::os::unix::fs::PermissionsExt
  std::os::windows::ffi::OsStrExt, OsStringExt
  std::process::Command with platform-specific args
  Signal handling (SIGTERM, SIGINT, SIGKILL, SIGHUP, SIGPIPE)

>>> PLATFORM_PATH -- File path and filesystem differences:
  std::path::MAIN_SEPARATOR
  Path handling with \ vs / differences
  Case sensitivity in path comparisons
  Home directory resolution (~/ vs %USERPROFILE%)
  /dev/null vs NUL, /dev/tty vs CON

>>> PLATFORM_ENCODING -- Encoding/locale differences:
  OsStr/OsString conversion differences (WTF-8 vs UTF-16)
  Line ending handling (\n vs \r\n)
  Console/terminal encoding differences

>>> OTHER_MIGRATION_FEATURE -- Other migration/compatibility features:
  Cross-platform compatibility shims or fallback implementations
  Cargo/build-script handling for target-specific dependencies
  Normalizing permissions, process, terminal, or filesystem behavior across OSes
  Test helpers or adapters that validate equivalent behavior across platforms

>>> FALSE_POSITIVE -- Not platform-specific:
  Test infrastructure noise (test setup/teardown timing differences)
  Feature flags unrelated to platform (#[cfg(feature = ...)])
  Code coverage gaps (lines not reached due to test selection, not platform)
  Library version differences unrelated to OS
```

### D.2 Masking the Target Implementation

The masking procedure labels verified target-specific lines with `<linux_specific>` for Windows-to-Linux tasks or `<windows_specific>` for Linux-to-Windows tasks.

**Masking Prompt**

```
You are provided with a file content where {marker} platform-specific lines have been
↪ labeled. You need to delete all {marker} platform-specific lines to create an
↪ incomplete code repository to test other software engineers' ability to re-implement
↪  it. Each of your str_replace tool call should satisfy ALL of the 3 requirements
↪ below:
    (1) Delete lines with prefix <{marker}> in some continuous code blocks.
    (2) Substitute the deleted lines and the contents surrounding the deleted lines
↪ with place holders to make the line deletion unobvious and make the remaining file
↪ content still look natural. This is to test other engineers' ability to find the
↪ hidden locations that have {marker}-specific problems from scratch. You must not
↪ substitute the deleted lines with `raise error` or `throw excpetion`, because `raise
↪  error / throw exception` is very obvious and easy to be found, which makes the
↪ created SWE task too trivial.
```

```
    (3) Do not re-implement the deleted locations, as these locations are deleted
↪ intentionally to test other engineers' ability to find and re-implement them. Ensure
↪  each substituted code block (your generated place holder) will still cause problems
↪  or errors in subsequent code blocks during execution due to the specified deleted
↪ lines. Only when a deletion/substitution can cause error at subsequent code blocks
↪ or can make some regression tests fail will the edit be considered valid. Do not fix
↪  bugs anywhere else, your task is only to delete blocks with <{marker}>.
we will ask you to remove labeled blocks in a loop until there is no line with <{marker
↪ }>.
code file content:
```{language}
{cur_content}
```
Now do str_replace.
```

### D.3 Migration Task Instruction

**Migration Task Prompt**

```
You are given a repository at /testbed. The repository was first developed on {source}
↪ but now you need to migrate it to {target}. Fix all platform compatibility issues on
↪  {target}. Your final goal is to pass all regression tests on {target}.
```

### D.4 SWE-agent

**System Prompt**

```
You are a helpful assistant that can interact with a computer to solve tasks.
```

**Instance Prompt**

```
<uploaded_files>
{{working_dir}}
</uploaded_files>
I've uploaded a code repository in the directory {{working_dir}}. Consider the
↪ following PR description:

<pr_description>
{{problem_statement}}
</pr_description>

Can you help me implement the necessary changes to the repository so that the
↪ requirements specified in the <pr_description> are met?
I've already taken care of all changes to any of the test files described in the <
↪ pr_description>. This means you DON'T have to modify the testing logic or any of the
↪  tests in any way!
Your task is to make the minimal changes to non-tests files in the {{working_dir}}
↪ directory to ensure the <pr_description> is satisfied.
Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to find and read code relevant to the <
↪ pr_description>
2. Figure out how to run regression tests in the repository.
3. Edit the source code of the repo to resolve the issue
4. Rerun the regression tests to confirm that the error is fixed!
5. Think about edgecases and make sure your fix handles them as well
Your thinking should be thorough and so it's fine if it's very long.

IMPORTANT NOTES:
1. Do not commit your edits, we will do it later. When you want to submit, just use the
↪  submit tool.
2. Do not use any git command. Git command is forbidden. If you use any git command
↪ your solution would be immediately judged as failed!
```

**Submission Review Prompt**

```
Thank you for your work on this issue. Please carefully follow the steps below to help
↪ review your changes.

1. If you made any changes to your code after running the reproduction script, please
↪ run the reproduction script again.
  If the reproduction script is failing, please revisit your changes and make sure they
↪  are correct.
  If you have already removed your reproduction script, please ignore this step.
2. Remove your reproduction script (if you haven't done so already).
3. If you have modified any TEST files, please revert them to the state they had before
↪  you started fixing the issue.
  You can do this with 'git restore /path/to/test/file.py' (this is the only place you
↪ can use git; do not use git anywhere else, otherwise your solution would be marked
↪ as failed). Use below <diff> to find the files you need to revert.
4. Run the submit command again to confirm.

Here is a list of all of your changes:

<diff>
{{diff}}
</diff>
```

### D.5 CLAUDE CODE

**Claude Code**

```
You are given a code repository in the current directory (pwd && ls ./).
The issue description is:
{description}
================================================
Your task is to fix the issue with the following steps:
(1) explore the source codes to find relevant files.
(2) figure out how to run regression tests.
(3) edit the source codes to fix the bug.
(4) rerun regression tests to see whether all linux-compatibility issues are fixed. If
↪ not, go back to explore and fix.
(5) remember to delete the test cases you write at last; if you modified the regression
↪  test files, git restore it. Note this is the only place you can use git command; if
↪  you use git command elsewhere your solution will be immediately judged as failed!
================================================
Notes:
(1) Please do not commit your edits. We will do it later.
(2) Do not use any git command except git restore the test files. Git command other
↪ than git restore the test files is forbidden. If you use any git command your
↪ solution would be immediately judged as failed!
(3) In our setting, each of your response must contain exactly one text content and one
↪  tool call. If you want to submit your result, just output one reponse with only
↪ text content and without tool call -- a response without tool call means stop and
↪ submission. If you have not finished, each of your response must have one tool call
↪ to make the session continue.
```

### D.6 OPENHANDS

**OpenHands**

```
I have access to a code repository in the directory {{ instance.repo_path }} . You can
↪ explore and modify files using the available tools. Consider the following issue
↪ description:

<issue_description>
{{ instance.problem_statement }}
</issue_description>

Can you help me implement the necessary changes to the repository so that the
↪ requirements specified in the <issue_description> are met?
I've already taken care of all changes to any of the test files described in the <
↪ issue_description>. This means you DON'T have to modify the testing logic or any of
↪ the tests in any way!
Also the development environment is already set up for you (i.e., all dependencies
↪ already installed), so you don't need to install other packages.
```

```
Your task is to make the minimal changes to non-test files in the {{ instance.repo_path
↪  }} directory to ensure the <issue_description> is satisfied.

Follow these phases to resolve the issue:

Phase 1. READING: read the problem and reword it in clearer terms
   1.1 If there are code or config snippets. Express in words any best practices or
↪ conventions in them.
   1.2 Hightlight message errors, method names, variables, file names, stack traces,
↪ and technical details.
   1.3 Explain the problem in clear terms.
   1.4 Enumerate the steps to reproduce the problem.
   1.5 Hightlight any best practices to take into account when testing and fixing the
↪ issue

Phase 2. RUNNING: install and run the regression tests on the repository
   2.1 Follow the readme
   2.2 Iterate and figure out how to run the regression tests

Phase 3. EXPLORATION: find the files that are related to the problem and possible
↪ solutions
   3.1 Use `grep` to search for relevant methods, classes, keywords and error messages.
   3.2 Identify all files related to the problem statement.
   3.3 Propose the methods and files to fix the issue and explain why.
   3.4 From the possible file locations, select the most likely location to fix the
↪ issue.

Phase 4. FIX ANALYSIS: state clearly the problem and how to fix it
   4.1 State clearly what the problem is.
   4.2 State clearly where the problem is located.
   4.3 State clearly how the test reproduces the issue.
   4.4 State clearly the best practices to take into account in the fix.
   4.5 State clearly how to fix the problem.

Phase 5. FIX IMPLEMENTATION: Edit the source code to implement your chosen solution.
   5.1 Make minimal, focused changes to fix the issue.

Phase 6. VERIFICATION: Test your implementation thoroughly.
   6.1 Run your reproduction script to verify the fix works.
   6.2 Add edge cases to your test script to ensure comprehensive coverage.
   6.3 Run existing tests related to the modified code to ensure you haven't broken
↪ anything.

7. FINAL REVIEW: Carefully re-read the problem description and compare your changes
↪ with the base commit {{ instance.base_commit }}.
   7.1 Ensure you've fully addressed all requirements.
   7.2 Run any tests in the repository related to:
     7.2.1 The issue you are fixing
     7.2.2 The files you modified
     7.2.3 The functions you changed
   7.3 If any tests fail, revise your implementation until all tests pass

NOTE:
 - Be thorough in your exploration, testing, and reasoning. It's fine if your thinking
↪ process is lengthy - quality and completeness are more important than brevity.
 - If you have modified test files, git restore them before submission. (Note this is
↪ the only place where you can use git command. If you use git command elsewhere your
↪ solution will be immediately judged as failed.)
 - You don't need to git commit. If you want to submit just use the `finish` tool.
 - Do not use any git command except git restore the test files. Git command is
↪ forbidden. If you use any git command your solution would be immediately judged as
↪ failed!
```

### D.7 WIN-AGENT

**Win-agent**

```
You are given a code repository at C:\testbed
You are in Windows powershell
You need to solve the following problem:
_PROBLEM_STATEMENT_
==================================================
Follow these steps to solve the problem:
1) Explore the repository to find relevant codes.
2) Figure out how to run regression tests in the repository.
```

```
3) Edit codes to solve the problem.
4) Run regression tests to validate your fix. If the test fails, go back to explore and
↪  edit again.
5) If all the tests can pass, remember to delete any test files you create and git
↪ restore the test files if you modified any. Then use submit tool call to submit.
↪ Note git restore the test files is the only git command you can use. Do not use git
↪ command anywhere else, otherwise your solution would be immediately judged as failed
↪ .
================================================
Note:
1) Each of your response should contain exactly one tool call.
2) Do not git commit, if you want to submit just use the submit tool.
3) Do not use git command anywhere except git restore the test files. Git command is
↪ forbidden. If you use git command, your solution would be immediately judged as
↪ failed.
4) As you are in a powershell terminal, DO NOT use `exit $LASTEXITCODE` in your command
↪  as it will terminate this conversation.
```