An Empirical Analysis of Cross-OS Portability Issues in Python Projects

Denini Silva
Federal University of Pernambuco
Recife, PE, Brazil
dgs@cin.ufpe.br

MohamadAli Farahat
North Carolina State University
Raleigh, NC, USA
mfaraha3@ncsu.edu

Marcelo d'Amorim
North Carolina State University
Raleigh, NC, USA
mdamori@ncsu.edu

Abstract

While Python is designed as a cross-platform language, real-world applications encounter portability failures when deployed across different operating systems.

We present the first large-scale empirical study of cross-OS portability issues in Python, analyzing 2,042 open-source repositories using two complementary approaches: systematic cross-OS test re-execution and manual analysis of GitHub issues. Our cross-platform testing of 500 projects reveals that 11.2% exhibit OS-dependent test failures. Through systematic analysis of 240 GitHub issues, we confirm 102 genuine portability problems spanning 95 additional projects. We develop a comprehensive taxonomy identifying 7 primary failure categories—with file/directory operations, process management, and library dependencies being most prevalent—along with 24 distinct sub-categories, 15 diagnostic signatures, and 4 systematic repair patterns. Our evaluation reveals that existing static analysis tools provide minimal support for portability detection, while large language models achieve 40–79% accuracy in identifying issues and 50–77% success in generating fixes when provided with structured guidance. Through 33 contributed pull requests, we demonstrate practical applicability and developer acceptance (17 merged, zero rejected) of our findings.

This work establishes the first comprehensive baseline for understanding and addressing cross-OS portability issues in Python, providing actionable insights for developers, tool designers, and the broader research community.

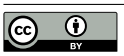
CCS Concepts

• **Software and its engineering → Software verification and validation**; **Functionality**.

Keywords

Cross-Platform Portability Issues, Python Tests, Large Language Models.

ACM Reference Format:
Denini Silva, MohamadAli Farahat, and Marcelo d'Amorim. 2026. An Empirical Analysis of Cross-OS Portability Issues in Python Projects. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3793302.3793373



MSR '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793373

1 Introduction

A *portability issue* refers to an observable difference in a program's behavior that arises from variations in the underlying platform—for example, inconsistencies in API implementations across platforms. Python is widely regarded as a cross-platform language, powering applications across diverse domains such as data science, web development, and automation. It consistently ranks among the most popular programming languages today [1, 2, 36]. However, platform portability is not guaranteed. Mince et al. [27], for instance, reported that roughly 40% of the functionality in popular machine learning libraries (e.g., PyTorch [30]) behaves inconsistently across hardware platforms such as CPUs and TPUs. Similar challenges occur at the operating-system level; for example, the function `os.geteuid()` is unavailable in Windows distributions of Python [11].

Portability issues in software projects often arise from dependencies on platform-specific APIs, system libraries, and environment assumptions. These issues commonly lead to deployment failures when software is executed in environments differing from those used during development or testing. Consequently, developers must spend substantial time diagnosing and adapting code, which negatively impacts productivity and delays releases. Prior work has shown that the availability and usage of platform-specific APIs can directly affect the portability and reliability of software systems [19]. Industry reports further emphasize that ensuring code portability requires deliberate cross-platform design, testing, and maintenance practices [20]. Systematic investigations indicate that defining and quantifying portability remains a persistent challenge in software engineering, with no widely adopted metrics or automated measurement frameworks [15]. Moreover, empirical studies have connected portability-related inconsistencies to unstable or flaky test behavior, complicating continuous integration pipelines and further burdening developers [24, 43]. Container technologies such as Docker [8] and Singularity (now Apptainer) [21] can address software portability challenges by encapsulating code, dependencies, and execution environments into reproducible units. They minimize configuration discrepancies across platforms and facilitate consistent testing and deployment. Nevertheless, despite the clear benefits of containerization, most open-source Python projects remain non-containerized. In our analysis of 2,042 GitHub repositories, we find that only 19.9% included a Dockerfile or Docker Compose configuration.

This paper presents a study of cross-OS portability issues in Python projects. Specifically, we analyze 2,042 repositories to assess the prevalence of such issues, their characteristics (e.g., symptoms, root causes, and fixes), the effectiveness of existing tools (e.g., static analyzers and large language models) in detecting and repairing them, and developers' reactions to the 33 pull requests

we submitted to address portability issues in their code. Our analysis reveals that these portability failures, while widespread (11.2% of tested projects), are not chaotic or unpredictable. Instead, they cluster around 7 main root-cause categories and 24 well-defined sub-categories—with file and directory operations, process management, and library dependencies being most prevalent. Results indicate that a combination of test re-execution and LLMs can complementarily detect issues, with LLMs achieving 40–79% accuracy in detection and 50–77% success in repair when provided with structured guidance. We conclude with a set of lessons we learned.

This paper makes the following contributions:

- ⋆ A study about the prevalence of portability issues that use two complementary methods to identify issues: test re-execution and mining of issues with subsequent manual analysis;
- ⋆ A detailed categorization of symptoms, root causes, and fix patterns of portability issues;
- ⋆ An evaluation of the ability of static analyses and LLMs to detect and repair portability issues;
- ⋆ An evaluation about the reaction of developers about the pull requests we open to fix portability issues in their GitHub projects;
- ⋆ A dataset of portability issues across 2,042 Python projects, including test re-execution results, GitHub issues, code examples, and LLM evaluation data.

Our artifacts, including code and data, are publicly available at **https://github.com/ncsu-swat/portability-issues-py**.

2 Projects and Questions

This section describes the projects we used (2.1) the research questions we posed (2.2).

2.1 Projects

Dataset. We identified candidate repositories through a broad GitHub API query, then filtered this pool by verifying Python was the predominant language and that repositories contained executable tests with at least one passing test. From this set, we sampled 2,042 projects, a size both substantial and feasible for cross-platform execution, and randomly partitioned them for our analyses.

Dataset partitioning. We partitioned the dataset to balance human effort with obtaining a representative sample. We allocated 900 projects to a *learning set* for identifying and categorizing portability issues through detailed analysis. This set was divided by detection method: 500 projects (nearly 25% of the total) through cross-OS test re-execution, and 400 projects (an amount realistically inspectable manually) through systematic GitHub issue mining. These subsets are non-overlapping, ensuring independent observations. We extract our taxonomy from these 900 projects (=500+400). The remaining 1,142 projects (=2,042-1,142) constitute an *application set* for validation through pull requests.

Table 1: Stats on 2,042 evaluated projects: no. of tests (#Tests), size (SLOC), no. of commits (#Sha), age in years (Age), no. of GitHub stars (⋆).

Statistic	#Tests	SLOC	#Sha	Age	⋆
Mean	572.51	41,635.39	2,883.25	6.24	6,147.88
Median	72	7,478	426.50	5.87	83.50
Min	1	2	2	0.3	2
Max	36,588	3,237,895	194,153	17.27	366,472
Sum	1,169,056	85,019,457	-	-	-

Table 1 summarizes key characteristics of the 2,042 evaluated projects. On average, each project comprises approximately 573 tests and 41.6K lines of Python code (SLOC), with a median of 72 tests and 7.5K SLOC, indicating a distribution skewed by a few very large systems. Regarding repository activity, projects have on average 2.9K commits, a median of 427, and span a mean lifetime of 6.2 years. The number of GitHub stars varies widely, with an average of 6.1K, a median of only 84, and a maximum of ~366K, reflecting the presence of both niche and highly popular projects.

Overall, the dataset covers a diverse range of software systems, from small-scale repositories with only a handful of commits to large, long-lived, and widely adopted projects.

2.2 Research Questions and Methodology

We aim to answer the following key research questions:

RQ1: How prevalent are portability problems in Python projects?. This question aims to assess how frequently portability problems occur in practice. If such issues are very rare, or if developers generally disregard them, the study's practical value would be limited. To address this question, we analyze issue reports and execute tests in the wild across multiple virtual machines.

RQ2: What are the causes, symptoms, and fixes of portability problems?. Assuming that identifying and resolving portability issues is an important problem, it is also essential to understand how these issues manifest (i.e., their symptoms), what their root causes are, and how they are typically repaired. For instance, automated solutions become more feasible when a small number of recurring fix patterns can be observed and documented.

RQ3: How good are existing tools at detecting and fixing portability issues?. This question examines the extent to which existing tools (e.g., static analyzers and LLMs) can detect and fix portability problems. Re-executing test suites across different OSes (e.g., through virtual machines) can reliably identify portability issues, but it is limited by the ability of the existing tests to exercise the affected code locations.

RQ4: How Do Developers Respond to Reported Portability Issues?. This question aims to further understand technical and social dimensions of portability issues, namely, what are the common fixes developers employ and how developers react to pull requests we submit to fix portability issues in their projects.

3 Results

This section provides answers to the posed research questions.

Table 2: Prevalence of portability problems.

(a) Problems detected with test re-execution.

Number of projects selected	500
Number of projects with OS differences	56
Number of tests executed	440,728
Number of test runs with OS differences	9,508

(b) Problems detected in issues with manual inspection.

Number of projects selected	400
Number of projects with OS differences	95
Number of issues mined	2,617
Number of issues analyzed	240
Number of issues documenting OS differences	102

3.1 Answering RQ1: How prevalent are portability problems in Python projects?

3.1.1 Methods. We use two complementary methods to identify portability issues. The first method, *cross-OS test re-execution*, systematically reveals behavioral differences across platforms but it is inherently incomplete as tests may not cover important code paths. The second method, *issue mining*, complements test re-execution by including developer perspectives and discussions, though it is costly as it requires manual analysis of discussion threads to rule out spurious cases. We discuss both methods in detail below.

Cross-OS test re-execution. For each project we run the test suite on multiple operating systems and record divergent outcomes. We then triage failures via logs and small code inspections, mapping each instance to a concrete category (i.e., a classification of the root cause of the portability issue, such as file handling, missing APIs, or library dependencies; the full taxonomy is presented in answering RQ2: What are the causes, symptoms, and fixes of portability problems?). We execute tests on ephemeral virtual machines provided by GitHub-hosted runners [16]. Each virtual machine is automatically provisioned with 4 CPUs and 16 GB of RAM. We implemented a single GitHub Actions workflow (YAML) that uses a matrix strategy to run the test-suite on Ubuntu 24.04 LTS, macOS 15, and Windows Server 2025 (each job runs in a fresh VM image). Each job installs dependencies, runs the Python tests with `pytest` [41] and the `pytest-json-report` plugin [28] to create JSON reports, and finally uploads the JSON files as workflow artifacts for later analysis. This approach ensures reproducible, parallel cross-platform testing while GitHub manages VM provisioning and maintenance.

Issues Mining. To complement test-based findings, we selected 400 projects from the learning set and mined GitHub issues and pull requests from them. We wrote a "candidate finder" combining multiple *types* of keyword: (a) OS/platform indicators (e.g., Windows, Linux, macOS, specific distros/architectures), (b) failure/-fix language (e.g., fails, error, bug, fix, workaround), (c) testing/CI context (pytest, CI, GitHub Actions), and (d) common portability causes (e.g., path separators, chmod/permissions, encodings/UTF-8, dynamic libraries like `.dll`/`.so`). We combine these keywords with "OR" within each type and "AND" across types. We searched titles, bodies, and comments, with extra weight to title and close-proximity matches. We employ a lightweight triage (brief summaries and negative filters for off-topic mentions) to filter spurious results. Candidate issues were normalized into consistent records (project, link, date, summary, compact tags like OS=, FIX=, TEST=, CAUSE=), duplicates removed, and full text (title, description, comments) archived for analysis.

3.1.2 Results. We applied both methods described in Section 3.1.1 to the learning set. Table 2 reports on the prevalence of portability problems in GitHub Python projects. Table 2a details results from test re-execution, while Table 2b details results from issue mining.

Table 2a shows that of the 500 projects analyzed, we found discrepant behavior in 56 (11.2%) of them, i.e., projects where test suites had at least one test manifesting OS differences. These projects included a total of 440,728 test cases, of which 9,508 showed discrepant outcomes, corresponding to 2.16%.

Table 2b shows the results from issue mining. From the 400 projects selected (see Section 3.1.1), we mined a total of 2,617 candidate issues. From this set, we randomly sampled 240 of the most recent issues for manual inspection. Through careful manual analysis, we confirmed that 102 of these were genuine portability issues, spanning 95 distinct projects. Combined with the test re-execution results, we identified **151** Python projects with portability issues considering both methods.

RQ_1 (prevalence): We find that portability issues are relatively prevalent. For example, 11.2% of the 500 projects we analyzed with cross-OS test re-execution have portability problems. Additionally, our issue mining approach shows that 95 of the projects we analyzed have genuine portability issues.

3.2 Answering RQ2: What are the causes, symptoms, and fixes of portability problems?

3.2.1 Method. We conducted a qualitative study to categorize the root causes, symptoms, and the fixes of portability problems.

We used cross-OS test re-execution and the analysis of issues to find root causes and symptoms. For fixes, we needed to rely on the discussion in the issue and the corresponding fix commits. In cross-OS test re-execution, we analyzed error messages, stack traces, and the failing test code to understand the root causes and symptoms. For issue mining, we analyzed issue titles, descriptions, developer discussions, and comments to understand the root causes and symptoms. Through iterative open coding [37] across both data sources, we identified recurring patterns and grouped instances with similar underlying causes. The coding process was conducted independently by two co-authors, with regular discussion to resolve ambiguities and refine category definitions. To ensure reliability, we computed inter-coder agreement on a subset of the data, achieving 96.4% agreement with a Cohen's kappa of 0.89, reflecting almost perfect inter-coder reliability. During the later phases of the coding process, new issues no longer produced new categories, indicating that the taxonomy had reached theoretical saturation.

The resulting taxonomy of root causes consists of **7** high-level categories (e.g., FILE, PROC, LIB) with **24** distinct sub-categories.

For each instance, we documented observable symptoms (error messages, behaviors) and examined corresponding fixes in merged pull requests or commits to identify common repair patterns. We identified **15** unique symptom signatures and **9** context-dependent symptoms across the 24 sub-categories, and **4** general fix patterns that collectively address all observed portability problems. Table 3 summarizes the primary data sources used to identify each component of our taxonomy. Root-cause categories were derived from patterns observed across both test failures and issue discussions. Symptom signatures were primarily extracted from test re-execution, where concrete error messages and stack traces provide direct evidence of portability problems. Fix patterns were identified by examining code changes and discussions in both test-related commits and issue-related pull requests.

Table 3: Data sources for portability taxonomy components.

Component	Test Re-execution	Issue Mining
Root-cause categories	✓	✓
Symptoms	✓	✗
Fix patterns	✓	✓

In the following subsections, we elaborate on the characteristics of portability problems: root-causes, symptoms, and fixes.

3.2.2 *Answering RQ2.1: What are the root causes of portability problems in Python?*

Table 4 presents our taxonomy of portability issues organized by root-causes. Each row groups related sub-categories under a main category. The columns show: (i) the root-cause category name and acronym, (ii) specific sub-categories within that category, (iii) a brief description of each sub-category, (iv) the number of distinct projects where the issue appeared in test execution, (v) the number of distinct projects where the issue appeared in mined GitHub issues, and (vi) the total count across both sources.

We identify seven main root-cause categories that explain why portability problems occur:

1. **File and Directories (FILE):** The most prevalent category (62 projects) stems from OS differences in file operations. Key issues include path separator differences (backslash vs. forward slash), line ending mismatches (`CRLF` vs. `LF`), file locking semantics, and text encoding defaults.
2. **Process and Signals (PROC):** This category (25 projects) captures failures due to OS process management differences. For example, shell execution varies across platforms (`cmd.exe` vs. `bash`), Windows lacks many Unix signals (`SIGHUP`, `SIGKILL`), and port binding produces different error codes.
3. **Library Dependencies (LIB):** These failures (24 projects) occur when code assumes platform-specific libraries. Issues include Unix-only modules (e.g., `fcntl`), missing dependencies, different dynamic library extensions (`.dll` vs `.so` vs `.dylib`), and architecture-specific binary wheels.
4. **API Availability (API):** This category (17 projects) covers Python APIs not universally available. Many OS module methods are Unix-specific (`os.uname()`, `os.geteuid()`, `os.getpgid()`) and fail on Windows, while optional modules like readline and resource may be missing.
5. **Environment and Display (ENV):** These problems (14 projects) arise from runtime environment differences. For example, headless CI lacks display servers for GUI testing, window managers behave differently, terminal capabilities vary, and the curses module is unavailable on Windows (not included in standard Python distributions).
6. **Permissions and Limits (PERM):** These issues (7 projects) reflect OS differences in access control. For example, permission models differ (Unix chmod versus Windows ACLs), file descriptor limits vary, and symbolic link creation requires admin privileges on Windows.
7. **System Information (SYS):** This category (2 projects) includes failures from platform-specific system information mechanisms. For example, the /proc filesystem exists only on Linux, and timezone databases may not be installed by default.

$RQ_{2.1}$ (root causes): We identified 7 root-cause categories with 24 sub-categories. File and Directories (FILE) is most prevalent (62 projects), followed by Process and Signals (25 projects) and Library Dependencies (24 projects).

3.2.3 *Answering RQ2.2: What are the observable symptoms of portability issues?*

Understanding how portability issues manifest during execution is essential for rapid diagnosis and triaging. In this section, we examine the observable symptoms of portability problems—the concrete error messages, exceptions, and behavioral differences that appear in test logs, and CI output. Table 5 maps the root causes (of portability issues) to the corresponding symptom signature and fix pattern.

Symptom categories. We categorize symptoms into two types based on their diagnostic value (color-coded in Table 5): **(i) Unique signatures** are error messages that unambiguously identify a specific portability sub-category, enabling immediate diagnosis from logs alone; and **(ii) Context-dependent symptoms** are manifestations that require additional code inspection or contextual understanding to distinguish from other failure modes.

Unique signatures. The majority of sub-categories (15 out of 24, 62.5%) produce unique, actionable error signatures. These include missing API methods (e.g., `AttributeError: module 'os' has no attribute 'geteuid'`), unavailable modules (e.g., `ModuleNotFoundError: No module named 'fcntl'`), encoding errors (`UnicodeDecodeError: 'charmap' codec...`), specialized exceptions like `TclError` for display issues or `ZoneInfoNotFoundError` for timezone problems, and platform-specific errors like `OSError: [Errno 24] Too many open files` for file descriptor limits. These signatures facilitate more direct diagnosis from logs, reducing the need for extensive code inspection required by context-dependent symptoms.

Context-dependent symptoms. The remaining 9 sub-categories (37.5%) do not produce unique error signatures, requiring developers to inspect the code context to diagnose the portability issue. These include cases where the same error can arise from multiple root causes, where failures are silent or produce generic errors, or where the symptom manifests as behavioral differences rather

Table 4: Taxonomy of portability issues showing root-cause categories, sub-categories, brief descriptions, and number of distinct projects observed in test execution and mined GitHub issues.

Root-cause category (Acronym)	Sub-categories	Description	# of projects in tests	# of projects in issues	Total
File and Directories (FILE)	Path separators	Forward vs. backslash differences	7	27	62
	Line endings	CRLF vs. LF mismatch	1	5	
	Case sensitivity	Filename case handling varies	1	0	
	File locking	Windows locks open files	7	4	
	Encoding	UTF-8 not default everywhere	2	7	
	Filesystem block size	Block size assumptions differ	0	1	
Process and Signals (PROC)	Shell command execution	Shell behavior differs across OS	5	14	25
	Missing signals	Signals unavailable on Windows	3	2	
	Address already in use	Port binding conflicts vary	1	0	
Library Dependencies (LIB)	Platform-specific libraries (e.g., fcntl)	Unix-only libraries missing elsewhere	2	1	24
	Missing libraries	Dependencies not universally available	2	11	
	Dynamic library loading	DLL vs. SO file issues	0	5	
	Binary wheel mismatch	Architecture-specific wheel incompatibilities	0	3	
API Availability (API)	Methods: os.uname(), os.geteuid(), os.getuid(), os.getpgid()	OS methods missing on Windows	13	2	17
	Modules: readline, resource	Optional modules not everywhere	0	2	
Environment and Display (ENV)	No display in GitHub CI (linux)	Headless CI lacks display server	3	2	14
	GUI differences	Window manager behavior varies	0	4	
	Missing curses	Curses unavailable on Windows	1	0	
	Terminal capabilities	Terminal features differ across systems	2	2	
Permissions and Limits (PERM)	Permission	Permission models differ by OS	3	1	7
	File descriptor limits	Resource limits vary across systems	0	1	
	Symlink privileges	Windows requires admin for symlinks	1	1	
System Information (SYS)	/proc filesystem	/proc missing on macOS/Windows	1	0	2
	Timezone database	Timezone data not always installed	1	0	
Total	-	-	**56**	**95**	**151**

than exceptions. For example, path separator issues often manifest as generic `FileNotFoundError` or silent failures where hardcoded paths like `"/tmp/file.txt"` work on Unix but fail on Windows. Similarly, case sensitivity issues may produce `FileNotFoundError` indistinguishable from actual missing files unless one examines the filesystem. The /proc filesystem is another case where generic `FileNotFoundError` occurs on macOS/Windows, requiring code inspection to determine if the issue stems from missing /proc paths rather than other file access problems. Shell command execution issues rarely throw exceptions but instead manifest as incorrect exit codes or output differences when commands are interpreted by different shells (`cmd.exe` vs. `bash`). GUI differences show as visual rendering variations or behavioral inconsistencies rather than explicit errors, and terminal capability mismatches may cause ANSI color codes to appear as literal text. Permission issues vary depending on the specific operation (chmod, file access, etc.), filesystem block size problems appear as test assertions on exact byte counts that differ across platforms, and binary wheel mismatches often succeed during installation but fail at runtime with ABI compatibility errors that vary by architecture.

Implications for diagnosis. Our findings reveal that while unique signatures enable rapid automated triage for 62.5% of portability issues, the remaining 37.5% require more sophisticated analysis. This suggests that detection tools must go beyond simple log parsing and incorporate contextual hints (e.g., hardcoded paths, shell command patterns, or platform-specific API usage). The presence of context-dependent symptoms also highlights the importance of comprehensive cross-platform testing, as these issues cannot be reliably predicted through static analysis alone.

RQ$_{2.2}$ (symptoms): Of 24 sub-categories, 15 (62.5%) produce unique error signatures enabling immediate diagnosis, while 9 (37.5%) exhibit context-dependent symptoms requiring code inspection to distinguish from other failure modes.

3.2.4 *Answering RQ2.3: What are fix patterns for addressing portability issues?*

Fix patterns. By examining source code, test logs, issue discussions, and merged pull requests from both detection approaches, we identified four general fix patterns that address portability problems across all 24 sub-categories (mapped in Table 5). Figure 1 illustrates these patterns with concrete examples:

1. **Environment handling** adapts code and tests to runtime conditions by detecting platform capabilities and adjusting behavior. Listing 2 shows this pattern using pytest's `skipif` decorator to conditionally skip tests on macOS. This allows test suites to adapt to platform-specific capabilities without maintaining separate test files for each operating system. This is the most prevalent pattern, accounting for 35.7% of all pattern applications.
2. **Defensive checks** add guards, fallbacks, or conditional imports to handle missing APIs or modules gracefully. Listing 1 demonstrates this pattern by wrapping the `readline` import (unavailable on Windows) in a `try/except` block with a fallback value, allowing the code to continue execution rather than

Table 5: Symptom signatures and general fix patterns for portability sub-categories. Symptom cells are color-coded: unique signatures enable immediate diagnosis; context-dependent require code inspection. Fix patterns are color-coded to show reuse across categories: Environment handling, Defensive checks, Normalization, Portable APIs.

Root-cause category	Sub-categories	Symptom (verbatim error / message)	General Fix Pattern
FILE	Path separators	Context-dependent: hardcoded paths fail silently or with generic file errors	Normalization — normalize paths with pathlib
	Line endings	`AssertionError: '\r\n' != '\n'`	Normalization — normalize line endings
	Case sensitivity	Context-dependent: file exists but name differs only in case	Normalization — case-normalize filenames
	File locking	`PermissionError: [WinError 32] The process cannot access the file because it is being used by another process` (Windows)	Environment handling — use context managers, explicit close
	Encoding	`UnicodeDecodeError: 'charmap' codec can't decode byte 0x{XX} in position N`	Normalization — set explicit UTF-8 encoding
	Filesystem block size	Context-dependent: tests assert exact byte counts that vary by OS	Normalization — normalize computations avoid assumptions
PROC	Shell command execution	Context-dependent: command syntax or exit codes differ across shells	Portable APIs — use subprocess.run shell=False
	Missing signals	`AttributeError: module 'signal' has no attribute 'SIGHUP'` (or SIGKILL, SIGTERM variants)	Defensive checks — guard signals per platform
	Address already in use	`OSError: [Errno 98] Address already in use` (Linux) / `[WinError 10048]` (Windows)	Defensive checks + Environment handling — catch OS-specific errors, use port 0
LIB	Platform-specific libraries	`ModuleNotFoundError: No module named 'fcntl'` (Windows)	Defensive checks — conditional import provide fallback
	Missing libraries	`ModuleNotFoundError: No module named '{library_name}'`	Defensive checks — optional import with fallback
	Dynamic library loading	`OSError: [WinError 126] The specified module could not be found` (Windows)	Defensive checks — guard loading prefer pure-Python
	Binary wheel mismatch	Context-dependent: installation succeeds but runtime fails with ABI errors	Environment handling — install platform-appropriate wheel
API	Methods (e.g., os.uname)	`AttributeError: module 'os' has no attribute '{method}'` (Windows)	Portable APIs + Defensive checks + Environment handling — use platform.uname, hasattr guards, OS detection
	Modules (e.g., readline)	`ModuleNotFoundError: No module named '{module}'` (Windows)	Defensive checks — guard import use alternatives
ENV	No display in GitHub CI	`TclError: no display name and no $DISPLAY environment variable` (Linux)	Environment handling — skip GUI tests in CI
	GUI differences	Context-dependent: widgets render or behave differently	Environment handling — skip or mock GUI features
	Missing curses	`ModuleNotFoundError: No module named '_curses'` (Windows)	Defensive checks — conditional import with stub fallback
	Terminal capabilities	Context-dependent: color codes appear as text or formatting ignored	Environment handling — detect capabilities degrade gracefully
PERM	Permission	Context-dependent: depends on operation (chmod, file access, etc.)	Defensive checks + Environment handling — check permissions, skip restricted tests
	File descriptor limits	`OSError: [Errno 24] Too many open files`	Environment handling — close files promptly, raise limits in CI
	Symlink privileges	`OSError: [WinError 1314] A required privilege is not held by the client` (Windows)	Environment handling — detect support and use copies
SYS	/proc filesystem	Context-dependent: /proc paths fail on macOS/Windows with generic FileNotFoundError	Portable APIs — use psutil or platform APIs
	Timezone database	`zoneinfo.ZoneInfoNotFoundError: 'No time zone found with key {...}'`	Normalization — install tzdata or fallback

```
- import readline
+ try:
+     import readline
+ except ImportError:
+     readline = None  # fallback if readline is unavailable
```

Listing (1) Defensive check for optional library

```
+ import platform
+ @pytest.mark.skipif(platform.system() == "Darwin", reason="Not
      supported on macOS")
def test_special_case():
    ...
```

Listing (2) Environment handling: Using pytest skip

```
- file_path = "data/file.txt"
+ import os
+ file_path = "data" + os.path.sep + "file.txt"  # consistent path
```

Listing (3) Normalization: Normalizing file paths

```
- hostname = os.uname()[1]  # unix-only function
+ import platform
+ hostname = platform.uname().node  # cross-platform
```

Listing (4) Portable APIs: Using platform module for OS detection

Figure 1: Representative examples of fixes for portability issues: (a) Defensive checks, (b) Environment handling, (c) Normalization, and (d) Portable APIs.

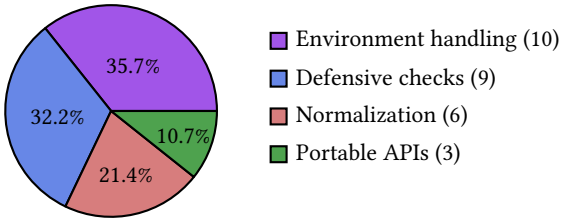


Figure 2: Distribution of fix patterns across 24 portability sub-categories.

crash. This pattern applies when functionality may be unavailable on certain platforms and accounts for 32.2% of pattern applications.

3. **Normalization** standardizes encodings, line endings, paths, or filenames to prevent spurious differences across platforms. Listing 3 illustrates this by constructing file paths using `os.path.sep` instead of hardcoded separators, ensuring correct behavior on both Unix (forward slash) and Windows (backslash) systems. This pattern accounts for 21.4% of applications.
4. **Portable APIs** replace OS-specific constructs with cross-platform abstractions. Listing 4 demonstrates replacing the Unix-only `os.uname()` function with the portable `platform.uname()` function, ensuring the code works correctly on both Unix-like systems and Windows. While used less

Table 6: LLMs for detection: metrics.

(a) Confusion matrices w/ 90 samples: 60 portable and 30 non-portable.

	Predicted					
	`llama-3.3`		`grok-4-fast`		`gpt-4o-mini`	
Actual	**Port**	**NonPort**	**Port**	**NonPort**	**Port**	**NonPort**
Port	10	50	53	7	21	39
NonPort	4	26	12	18	1	29

(b) Performance metrics.

Model	Precision	Recall	F1-Score	Accuracy
`llama-3.3`	0.34	0.87	0.49	0.40
`grok-4-fast`	0.72	0.60	0.65	0.79
`gpt-4o-mini`	0.43	0.97	0.59	0.56

frequently (10.7%), this pattern targets specific, well-defined problems where standard cross-platform abstractions exist.

Pattern distribution. Figure 2 shows how these patterns distribute across all pattern applications. Some sub-categories require multiple patterns, resulting in 28 total pattern applications across 24 sub-categories. Environment handling is the most prevalent strategy, accounting for 10 applications (35.7%), followed by Defensive checks with 9 applications (32.2%), together representing approximately two-thirds of all pattern applications. Normalization accounts for 6 applications (21.4%), primarily addressing file and path-related issues. Portable APIs, while used least frequently (3 applications, 10.7%), targets specific, well-defined problems where standard cross-platform abstractions exist. This distribution reflects the nature of portability issues: most problems arise from platform-environmental differences (explaining fixes with conditional handling) rather than fundamental API incompatibilities. The data also show that portability fixes are typically short and localized rather than extensive or widespread. Consequently, they tend to impose a low workload on developers for both implementation and review.

RQ$_{2.3}$ (fixes): We identified 4 general fix patterns that collectively address all 24 portability sub-categories. Some sub-categories require multiple patterns, resulting in 28 total pattern applications. Environment handling is most prevalent (35.7%), followed by Defensive checks (32.2%), Normalization (21.4%), and Portable APIs (10.7%). These patterns represent recurring, localized code modifications that provide systematic guidance for resolving portability issues across diverse contexts.

3.3 Answering RQ3: How good are existing tools at detecting and fixing portability issues?

This section evaluates performance of tools for detecting (§ 3.3.1) and for fixing (§ 3.3.2) portability issues.

3.3.1 LLMs for detection. **Alternative static analysis tools.** We examine the capabilities of various popular linters for Python, namely Ruff [39], Flake8 [6], Pylint [23], MyPy [22], Bandit [31], and Pyright [7]. We find that only Ruff provides a rule that directly addresses potential portability concerns, namely the `unspecified-encoding` check [42] that maps to the "Encoding" sub-category within the "File and Directories" category. The remaining tools focus primarily on style, type correctness, security, and general code quality. For that reason, we explored the use of Large Language Models (LLMs) to detect portability problems and repair them. We consider three LLMs of moderate model size, namely `llama-3.3` [25], `Grok-4-Fast` [45], and `GPT-4o-Mini` [29]

Setup. We sampled a total of 90 code snippets to evaluate performance of LLM: (i) 30 non-portable snippets containing OS-specific constructs, (ii) 30 portable snippets that had been fixed by developers (mined from issues), and (iii) 30 code we manually categorized as portable. This distribution attempts to reflect a more realistic division of code seen in the wild, where portability violations are less frequent. We used the openrouter.ai API [40] to interact with the models. Each model was prompted with a consistent template that included the code snippet and a question asking whether the code was portable across Windows, Linux, and Mac (see Table 7). We parse the answers to extract classification decisions.

Metrics. We used standard metrics to measure the performance of binary classifiers (e.g., precision, recall, accuracy, etc.) [33]. Table 6 shows results. Table 6a shows the confusion matrices associated with each LLM, illustrating the classification of the 90 samples. Table 6b shows the metrics for each model.

Analysis of results. The results we obtain indicate limited ability of LLMs to accurately reason about portability issues. llama-3.3 and GPT-4o-Mini are "pessimistic" and report too many alarms, most of which are false positives. In contrast, Grok-4-Fast is more cautious in reporting but misses lots of true positives.

Table 7: Prompts used for LLM evaluation.

(a) Generic prompt used for portability detection and fixing.

Generic Prompt
You are a Python expert. Check the following code and answer: 1. Is there any operation in the code that could fail on a specific operating system (Linux, Mac, Windows)? 2. If yes, explain why and on which OS it might fail. If it is fully portable, finish saying "Portable".

(b) Pattern-guided prompt template to fix portability issues.

Pattern-Guided Prompt Template
You are a Python expert. The following code has a portability problem: *<symptom-description>*. Fix the code, considering the following fix options *<list-of-possible-fixes-for-the-symptom>*.

3.3.2 LLMs for fixing. **Setup.** Following the classification experiment, we designed a complementary evaluation focused on the corrective capabilities of LLMs. In this phase, we used the 30 non-portable Python snippets identified earlier and prompted each model with two distinct repair configurations. In the first configuration (*generic prompt*), each model received a general instruction stating that the code exhibited portability issues and was asked to produce a corrected version (see Table 7). In the second configuration (*pattern-guided prompt*), each non-portable snippet was accompanied by the specific symptom (derived from our Unique

Table 8: Performance of LLMs in generating portability fixes across both prompt configurations (30 samples per model).

(a) Generic prompt.

Model	Correct	Incorrect	Accuracy
grok-4-fast	13	17	0.43
gpt-4o-mini	10	20	0.33
llama-3.3	8	22	0.27

(b) Pattern-guided prompt.

Model	Correct	Incorrect	Accuracy
grok-4-fast	23	7	0.77
gpt-4o-mini	21	9	0.70
llama-3.3	15	15	0.50

Table 9: Pattern-guided repair performance breakdown: cases with symptom descriptions vs. empty symptom fields.

(a) With symptom descriptions (n = 23).

Model	Correct	Incorrect	Accuracy
grok-4-fast	19	4	0.83
gpt-4o-mini	16	7	0.70
llama-3.3	12	11	0.52

(b) Without symptom descriptions (n = 7).

Model	Correct	Incorrect	Accuracy
grok-4-fast	4	3	0.57
gpt-4o-mini	5	2	0.71
llama-3.3	3	4	0.43

signature symptoms in 3.2.3) and the corresponding *General Fix Pattern* from Table 5.

The rationale for this approach is: (1) observe symptoms in the code, (2) look up the mapping in Table 5 (Symptom → General Fix Pattern), and (3) prompt the LLM with specific fix pattern guidance. For example, when encountering code that produces `AttributeError: module 'signal' has no attribute 'SIGHUP'`, we would provide that symptom together with corresponding fix(es) (Table 5). Table 7 shows the template structure used for these pattern-guided prompts. This design evaluates whether structured repair guidance improves the model's ability to generate functionally portable code. Each model attempted to fix all 30 snippets, resulting in 90 repair attempts per configuration. We also examine repair performance separately for cases with explicit symptom descriptions versus cases with empty symptom fields to understand how input characteristics influence model effectiveness.

Metrics. We evaluate the correctness of a generated fix with two criteria: (i) the modified code must execute successfully without errors across all target platforms (Linux, macOS, Windows), and (ii) the fix must preserve the intended functionality of the original program. Fixes that met both criteria were considered *correct*; otherwise, they were labeled as *incorrect*. Accuracy is the ratio of correct fixes to the total number of repair attempts.

Analysis of results. The results in Table 8 show that all models exhibited limited success under the generic prompt setting, with accuracies ranging from 27% to 43%. This indicates that, when provided with only minimal context, most LLMs struggle to infer the exact nature of portability issues and to produce appropriate cross-platform corrections. While the `grok-4-fast` model outperformed the others, its success rate remained below 50%, suggesting that general-purpose reasoning alone is insufficient for precise code repair in this domain.

In contrast, the pattern-guided configuration substantially improved repair performance across all models. The inclusion of explicit problem descriptions and generalized fix patterns increased accuracy by more than 30 percentage points on average. The `grok-4-fast` model achieved the highest accuracy (76.67%), demonstrating a clear ability to integrate structured repair hints into its reasoning process. Similarly, `gpt-4o-mini` reached 70%, producing fixes that were more consistent with the intended patterns while preserving functional correctness. The `llama-3.3` model also showed moderate improvement (50%), particularly in cases involving straightforward path and API adjustments, though it still struggled with more complex logic integration.

Table 9 reports on an experiment where we analyzed repair performance separately for cases with explicit symptom descriptions versus cases lacking such descriptions due to context-dependent symptoms. Results indicate that the availability of symptom descriptions significantly influences LLM performance. Among the 23 cases with explicit symptom descriptions, `grok-4-fast` achieves notably higher accuracy (0.83), while `gpt-4o-mini` and `llama-3.3` maintain similar or slightly improved performance (0.70 and 0.52, respectively). In contrast, for the 7 cases where symptom descriptions were not available due to context-dependent or unstable symptoms, the prompt contained only the code snippet and general instructions to apply the relevant fix pattern. In these cases, performance generally declines: `grok-4-fast` drops to 0.57, while `gpt-4o-mini` remains robust at 0.71 and `llama-3.3` falls to 0.43. This pattern demonstrates that explicit symptom descriptions provide valuable context that particularly benefits more sophisticated models, whereas some cases can still be addressed even without rich contextual information.

Overall, these findings highlight the strong impact of guided contextualization on LLM-based repair. When supplied with explicit repair patterns, the models produced significantly more reliable and semantically consistent fixes, confirming that structured guidance can substantially enhance cross-platform reasoning capabilities in code generation tasks.

RQ_3 (tool effectiveness): Static analysis tools show poor ability to detect portability problems. LLMs achieve moderate detection accuracy (40–79% across three models), with substantial variation in precision and recall. For repair tasks, LLMs show limited success with generic prompts (27–43% accuracy) but improve significantly with pattern-guided prompts (50–77% accuracy), yielding an average improvement of 30 percentage points. These results show that structured guidance using error signatures and fix patterns (Table 5) substantially enhances LLM-based repair.

3.4 Answering RQ4: How Do Developers Respond to Reported Portability Issues?

This section discusses the pull requests (PRs) we submitted.

3.4.1 Method. We use the cross-OS test re-execution method (§ 3.1.1) to detect portability problems in our *application set* (§ 2.1). For each identified issue, we manually apply one of the four fix patterns from our taxonomy (Table 5). We then validate each fix by re-running the affected tests on all three operating systems (Linux, macOS, Windows) to confirm consistent behavior across platforms. Once the fix is validated, we submit a pull request to the project, providing a clear description of the portability issue, the root cause, and an explanation of how the proposed change resolves the problem.

3.4.2 Results. We detected 47 portability issues; 33 PRs were submitted (70%), with 14 not submitted due to project policies. All of the PRs match one of the four fix patterns we discussed in Section 3.2.4. Table 10 describes the PRs. Of the 33 pull requests submitted, 17 were accepted at the time of writing. The overall acceptance rate of 51.5%, along with a zero rejection rate, indicates that the issues we identified correspond to genuine portability problems recognized by maintainers. Notably, "File and Directories" issues were the most frequent (15 PRs), reflecting the prevalence of path, encoding, and file handling inconsistencies across platforms. "Process and Signals" issues exhibited the highest acceptance rate (100%), likely due to their critical impact on application functionality. Considering the location of the fix, we observe that in 17 cases only the test code changes, in 10 cases only the program code changes, and in 6 cases both the test and program code changes. Note that many cases require changes to the application logic to ensure consistent behavior across platforms.

Next, we highlight three representative pull requests that illustrate how portability issues manifest in practice and how our identified fix patterns address them. These examples span multiple categories and demonstrate different sub-categories and fix patterns from our taxonomy.

Table 10: Summary of Pull Requests (PRs) by issue type, showing opened, accepted, and rejected. The last row shows totals.

Category	Opened	Accepted	Rejected
File and Directories	15	9	0
API Availability	12	4	0
Process and Signals	2	2	0
System Information	2	1	0
Library Dependencies	1	1	0
Environment and Display	1	0	0
Total	**33**	**17**	**0**

3.4.3 The webassets Pull Request. We illustrate how portability issues manifest and are fixed using the *webassets* project [9], a Python library for managing and bundling web assets such as CSS and JavaScript, with 932 stars. The problem we encountered falls into the **File and Directories (FILE)** category, specifically the "File locking" sub-category. The issue occurred in the test suite where `tempfile.NamedTemporaryFile` was used without disabling automatic deletion, causing failures on Windows due to platform-specific file locking behavior [14]. On Windows, the operating system locks open files more aggressively than on Unix-like systems, preventing the same file from being reopened while still in use. Listing 5 shows the diff of our fix applied to the test file. The original code created a temporary file within a context manager that automatically deleted the file upon exiting the `with` block. However, on Windows, attempting to reopen this file (which the test needed to do because it's a fixture) failed because the file was still locked by the previous file handle.

```
1  @pytest.fixture
2  def tmp_file():
3  -   with tempfile.NamedTemporaryFile(mode="wt") as f:
4  +   with tempfile.NamedTemporaryFile(mode="wt", delete=False) as f:
5        for _ in range(100):
6          f.write("\n")
7        f.flush()
8  -     yield f.name
9  +     tmp_path = f.name  # store path before file is closed
10 +   yield tmp_path
11 +   os.remove(tmp_path)  # clean up after the test
```

Listing 5: Approved PR for temporary file handling issue.

Our fix addresses this issue by modifying line 4 to disable automatic deletion using `delete = False`, storing the file path on line 9 before the file handle is closed, and manually cleaning up the temporary file on line 11 after the test completes. This pattern exemplifies the **Environment handling** fix strategy, where platform-specific behavior is handled through careful resource management. This case demonstrates how seemingly simple operations like temporary file creation can have subtle portability implications that only manifest under specific platform conditions, highlighting the importance of cross-platform testing for robust software development.

3.4.4 The hosteurope-letsencrypt Pull Request. We demonstrate another portability fix through the hosteurope-letsencrypt project [34], which has 67 stars and involves system administration functionality. The problem falls into the **API Availability (API)** category. The issue is due to the use of `os.geteuid()` to check for root privileges, but this function is unavailable on Windows, causing `AttributeError: module 'os' has no attribute 'geteuid'`.

```
1  +import platform
2  +def is_admin():
3  +    try:
4  +        if platform.system() == "Windows":
5  +            import ctypes
6  +            return ctypes.windll.shell32.IsUserAnAdmin() != 0
7  +        else:
8  +            return os.geteuid() == 0
9  +    except:
10 +        return False
11
12 -is_root = os.geteuid() == 0
13 +is_root = is_admin()
```

Listing 6: Cross-platform admin privilege detection fix.

Our fix creates a cross-platform `is_admin()` function that replaces the Unix-only `os.geteuid()` call. The fix uses **Portable APIs** (line 4: `platform.system()` for OS detection), **Environment handling** (lines 6, 8: conditional logic that calls `ctypes.windll.shell32.IsUserAnAdmin()` for Windows or `os.geteuid()` for Unix), and **Defensive checks** (lines 3–9: try-except wrapper with safe fallback).

3.4.5 The pydantic-extra-types Pull Request. We show an example from the pydantic-extra-types project [4], a library that provides additional type validators for Pydantic with 290 stars, used for extending Pydantic's [5] validation capabilities with specialized data types. The problem we encountered also falls into the **File and Directories (FILE)** category, specifically the "Path separators" subcategory. The issue occurred in test fixtures where `os.path.relpath` was used to create relative paths from absolute paths. On Windows, this function fails when the source and target paths are on different drives (e.g., `C:\` vs `D:\`), as Windows cannot create relative paths between different drive letters [12]. Listing 7 shows our fix applied to test fixtures. The original code assumed that `os.path.relpath` would always succeed, but on Windows this assumption may not hold. For example, when tests are executed from a project directory located on drive `D:` while the system temporary directory (used to create fixture files) resides on drive `C:`, the function call fails because it attempts to compute a relative path across drives. Since the location of temporary directories on Windows is determined by environment variables and may differ from the working drive [13], this behavior can lead to path resolution errors.

```
1   @pytest.fixture
2   def relative_file_path(absolute_file_path: Path) -> Path:
3 + cwd = Path.cwd()
4 + if os.name == 'nt' and absolute_file_path.anchor != cwd.anchor:
5 +     return absolute_file_path
6     return Path(os.path.relpath(absolute_file_path, os.getcwd()))
```

Listing 7: Patch for cross-drive relative path issue.

Our fix addresses this Windows-specific limitation by detecting when the OS is Windows (line 4) and checking whether the absolute path and current working directory have different anchors (drive letters). When this condition is met, the function returns the absolute path instead of attempting the problematic relative path conversion (line 5). This pattern exemplifies the **Environment handling** fix strategy from our taxonomy, where platform-specific limitations are accommodated through conditional logic.

This case demonstrates how path operations that work seamlessly on Unix-like systems can encounter fundamental limitations on Windows due to the multi-drive architecture, highlighting the importance of considering platform-specific filesystem constraints in cross-platform development.

RQ_4 (community validation): We submitted 33 pull requests (PRs) to open-source projects, achieving a 51.5% acceptance rate with no rejections. Issues related to files and directories were the most common (15 PRs), whereas process and signal issues had the highest acceptance rate (100%). These results confirm our initial observation that fixes generally involve localized changes—either to test code (51.5%) or application logic (30.3%)—and suggest a positive response from developers to the submitted PRs.

4 Lessons Learned

This section distills key insights from our study into actionable guidance for developers working on cross-platform Python projects.

1. **Porting code remains important despite the rise of containerization technology.** We find that a relatively small fraction of open-source projects (19.9%) from our dataset include Docker configurations. The majority of projects need to handle portability natively in code. *Recommendations.* (1) Maintain cross-OS CI test matrices [17] to catch issues early, regardless of deployment target; (2) Fine tune code agents [26] to detect (and fix) portability issues as code is written.
2. **Guided LLM repair using error signatures and fix patterns.** LLMs underperform with generic "fix this code" prompts (27–43% success) but achieve 50–77% success when guided with specific error messages and corresponding fix patterns from Table 5. *Recommendations.* Include the specific error message and suggest the corresponding fix pattern in your LLM prompt. This structured approach nearly doubles success rates compared to generic prompts.
3. **Portability fixes are localized, not architectural.** Our four fix patterns (Table 5) address issues through small, targeted modifications—adding conditional imports, normalizing paths, or wrapping platform-specific calls—rather than redesigning system architecture. *Recommendations.* Treat portability issues as localized problems. Consult Table 5 to identify the issue type and apply the corresponding fix pattern rather than deferring fixes as "too invasive".
4. **File and path issues dominate cross-platform failures.** File and Directories is the most prevalent category, affecting 62 projects (41% of all portability issues). Path separators, encoding, and file locking are the most common sub-categories. *Recommendations.* Use `pathlib.Path` for path construction, explicitly specify `encoding` when opening text files, and use context managers (`with` statements) to avoid Windows file locking issues.

5 Threats to Validity

Internal validity. Our cross-OS test re-execution relies on GitHub Actions runners, which may differ from physical machines. In addition, the manual analysis of GitHub issues involved subjective interpretation. Mitigation: multiple researchers independently conducted the manual analysis, and disagreements were resolved through consensus.

External validity. The 2,042 GitHub projects we analyzed may not fully represent all software domains or domain-specific applications. Mitigation: we sampled projects of diverse domains and sizes. Our taxonomy is derived from fundamental operating-system differences and thus transcends specific domains. Furthermore, the high pull-request acceptance rate supports the generalizability of our findings across multiple developer communities.

Conclusion validity. Sample sizes varied across research questions (500, 400, and 90 snippets), which may affect the reliability of cross-question comparisons. Mitigation: we triangulated results using multiple independent methods, validated the taxonomy with both detection approaches, and obtained a 51.5% pull-request acceptance rate with zero rejections.

Finally, our scope is limited to Linux, macOS, and Windows, which are representative platforms. We did not consider portability across hardware architectures or Python versions, for instance.

6 Related Work

Our work relates to prior research on software portability and cross-platform testing, but differs in scope and methodology.

6.1 Empirical Studies on Software Portability

Ghandorh et al. [15] performed a systematic literature review of metrics and methodologies for assessing software portability. Their work surveys existing approaches for measuring portability but does not provide empirical data on real-world portability issues or concrete solutions. In contrast, our study provides the first large-scale empirical characterization of Python portability problems through systematic testing and issue analysis, developing both a comprehensive taxonomy and practical repair patterns. While their review identifies the fragmented nature of portability research, our work addresses this gap by establishing a unified framework specifically for Python cross-platform issues.

Wang et al. [44] present an empirical study of compatibility issues in deep learning systems, analyzing thousands of posts to categorize common problems and their causes. While their findings relate to cross-environment inconsistencies, the scope is broader than portability and centered on DL ecosystems. Our work instead focuses directly on Python cross-platform portability, providing a targeted taxonomy and concrete repair strategies.

6.2 Cross-Platform Testing and Defect Studies

Vahabzadeh et al. [43] conducted an empirical study on test code defects, finding that approximately 18% of test bugs stem from environmental factors, particularly Windows vs. Unix platform differences. While their work acknowledges environment-related testing issues, it focuses on general test code quality and defect classification rather than systematically characterizing portability problems or developing repair strategies. Our work differs in three key ways: (1) we provide the first comprehensive taxonomy of Python portability issues with 7 categories and 24 sub-categories, (2) we develop systematic fix patterns that address these issues, and (3) we evaluate both detection and repair approaches using modern tools (LLMs) that were not available during their study.

6.3 Dynamic Analysis for Portability

Rasley et al. [32] proposed `CheckAPI`, a runtime framework that detects cross-platform API violations by comparing execution traces with platform-specific specifications. Unlike CheckAPI, which requires predefined API models and targets API-level compatibility, our work observes behavioral differences directly through cross-OS re-execution and covers broader portability issues such as file systems, encodings, dependencies, and environments.

6.4 API Specification Nondeterminism

Shi et al. [35] and Gyori et al. [18] introduced *NonDex*, a tool that systematically explores alternative valid behaviors of Java APIs to uncover bugs caused by deterministic assumptions about inherently nondeterministic specifications (e.g., iteration order in hash-based collections). Although NonDex focuses on intra-platform variability within the JVM, our study examines deterministic divergences that emerge across different operating systems. Both approaches highlight how subtle, often implicit, assumptions about platform behavior can compromise software reliability under diverse execution environments.

6.5 OS Variability and System Reliability

Sun et al. [38] proposed *Bear*, a framework that quantifies how nondeterministic OS behaviors (e.g., scheduling and I/O timing) affect application reliability. They show that even minor OS-level variations can propagate to failures or performance drops, revealing hidden fragility in applications that assume predictable OS behavior. While Bear focuses on statistical modeling of OS nondeterminism, our work addresses deterministic portability issues stemming from platform-specific APIs, file systems, and libraries, providing concrete fix patterns through cross-platform testing.

6.6 Static Analysis for Python

Popular tools such as Ruff [39], Flake8 [6], Pylint [23], MyPy [22], Bandit [31], and Pyright [7] focus on style, typing, and security, offering minimal support for portability detection—only Ruff includes `unspecified-encoding` [42]. Specialized analyzers like Pysa [10] target security (e.g., taint and injection). Our work instead addresses runtime portability issues caused by OS variability, mostly missed by existing tools.

6.7 Performance and Portability.

Zioga et al. [46] analyzed performance bugs across architectures. Our study differs from theirs in two aspects. First, we focus on portability issues across operaring systems not architectures. Second, we focus on functional correctness not performance. Awar et al. proposes framework to automatically translate Python in C++ code [3]. The work is orthogonal to ours as we do not focus on efficiency gains.

7 Conclusions and Future Work

This paper provides the first comprehensive characterization of cross-OS portability issues in Python code. We employ two methods to find these issues: cross-OS test re-execution and issue mining. We analyze 900 projects and find that 16.8% exhibit portability problems. Issues related to File and Directory operations are the most prevalent. Overall, our taxonomy captures seven categories of root causes of issues (with 24 sub-categories) and four fix patterns for those issues. We find that LLMs achieve 40-79% detection accuracy and 50-77% repair success when guided by structured patterns. Our validation through 33 pull requests showed an acceptance rate of 51.5%, indicating that developers care about these issues.

Future work can extend this study by developing hybrid detection tools that integrate cross-OS re-execution, symptom signatures, and lightweight static analyses. Another promising direction is the fine-tuning of code agents for portability issues using pattern-guided prompts. Finally, it is important to broaden the evaluation to cover a wider range of operating system versions, hardware architectures, Python versions and even other programming languages.

Data Availability

The artifacts are publicly available at the following link:

https://github.com/ncsu-swat/portability-issues-py

Acknowledgments

This work is partially supported by the NSF under Grant Nos. CCF-2349961 and CCF-2319472. Denini was supported by CNPq Grant No. 140220/2022-4. We thank Mohammed Yaseen for his assistance with experimental setup.

References

[1] 2025. PYPL PopularitY of Programming Language index. https://pypl.github.io/.

[2] 2025. Stack Overflow Developer Survey. https://survey.stackoverflow.co/2025/

[3] Kokkos Contributors. 2025. Kokkos C++ Performance Portability Programming Ecosystem: The Programming Model - Parallel Execution and Memory Abstraction. https://github.com/kokkos/kokkos

[4] Pydantic Contributors. 2025. Pydantic Extra Types. https://github.com/pydantic/pydantic-extra-types.

[5] Pydantic Contributors. 2025. Pydantic Validation. https://docs.pydantic.dev/latest/.

[6] Ian Stapleton Cordasco and contributors. 2025. Flake8: the modular source code checker for Python. https://flake8.pycqa.org/

[7] Microsoft Corporation. 2025. Pyright: Static type checker for Python. https://github.com/microsoft/pyright

[8] Docker Inc. 2013. Docker: Empowering App Development for Developers. https://www.docker.com/.

[9] Michael Elsdoerfer. 2025. webassets: Asset management for Python web development. https://github.com/miracle2k/webassets.

[10] Facebook/Meta Engineering. 2021. Pysa: Security-Focused Static Analysis Tool for Python. https://developers.facebook.com/blog/post/2021/04/29/eli5-pysa-security-focused-analysis-tool-python/

[11] Python Software Foundation. 2025. os.geteuid() - Return the current process's effective user ID. https://docs.python.org/3/library/os.html#os.geteuid

[12] Python Software Foundation. 2025. os.path.relpath — Return a relative filepath to path. https://docs.python.org/3/library/os.path.html#os.path.relpath.

[13] Python Software Foundation. 2025. tempfile — Generate temporary files and directories. https://docs.python.org/3/library/tempfile.html.

[14] Python Software Foundation. 2025. tempfile.NamedTemporaryFile - Create a named temporary file. https://docs.python.org/3/library/tempfile.html#tempfile.NamedTemporaryFile

[15] Hamza Ghandorh, Abdulfattah Noorwali, Ali Bou Nassif, Luiz Fernando Capretz, and Roy Eagleson. 2020. A systematic literature review for software portability measurement: Preliminary results. In *Proceedings of the 2020 9th International Conference on Software and Computer Applications*. 152–157.

[16] GitHub. 2025. About GitHub-hosted runners. https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners

[17] Inc. GitHub. 2025. *Running variations of jobs in a workflow*. https://docs.github.com/actions/writing-workflows/choosing-what-your-workflow-does/running-variations-of-jobs-in-a-workflow

[18] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. 2016. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 993–997.

[19] Ricardo Job and Andre Hora. 2024. Availability and usage of platform-specific APIs: a first empirical study. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 27–31.

[20] Kiuwan. 2025. A Guide to Code Portability. *Kiuwan Blog* (2025). https://www.kiuwan.com/blog/what-is-code-portability/

[21] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017. Singularity: Scientific Containers for Mobility of Compute. In *Proceedings of the 37th International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. ACM. https://doi.org/10.1145/3126908.3126923 Now maintained as Apptainer: https://apptainer.org/.

[22] Jukka Lehtosalo and contributors. 2025. MyPy: Optional static typing for Python. http://mypy-lang.org/

[23] Logilab and Pylint contributors. 2025. Pylint: Python code static checker. https://pylint.pycqa.org/

[24] Qingzhou Luo, Meiyappan Nagappan, Bogdan Vasilescu, and H. Malik. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*.

[25] Meta. 2025. Llama 3.3 70B Instruct. https://openrouter.ai/meta-llama/llama-3.3-70b-instruct

[26] Microsoft Corporation. 2025. *Microsoft 365 Copilot Tuning overview (preview)*. https://learn.microsoft.com/en-us/copilot/microsoft-365/copilot-tuning-overview

[27] Fraser Mince, Dzung Dinh, Jonas Kgomo, Neil Thompson, and Sara Hooker. 2023. The Grand Illusion: The Myth of Software Portability and Implications for ML Progress. *Advances in Neural Information Processing Systems* 36 (2023), 21217–21229.

[28] Numirias. 2025. pytest-json-report: A plugin to generate JSON reports for pytest. https://github.com/numirias/pytest-json-report

[29] OpenAI. 2025. GPT-4o Mini. https://openrouter.ai/openai/gpt-4o-mini

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[31] PyCQA. 2025. Bandit: A tool to find common security issues in Python code. https://bandit.readthedocs.io/en/latest/

[32] Jeff Rasley, Eleni Gessiou, Tony Ohmann, Yuriy Brun, Shriram Krishnamurthi, and Justin Cappos. 2015. Detecting latent cross-platform api violations. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 484–495.

[33] Takaya Saito and Marc Rehmsmeier. 2015. The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PloS one* 10, 3 (2015), e0118432.

[34] Stein Sebastian. 2025. Let's Encrypt Skripte für Hosteurope WebHosting. https://github.com/steinsag/hosteurope-letsencrypt.

[35] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *2016 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 80–90.

[36] GitHub Staff. 2024. Octoverse: AI leads Python to top language as the number of global developers surges. https://github.blog/news-insights/octoverse/octoverse-2024/.

[37] Anselm Strauss and Juliet Corbin. 1990. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Sage Publications, Newbury Park, CA.

[38] Ruimin Sun, Andrew Lee, Aokun Chen, Donald E Porter, Matt Bishop, and Daniela Oliveira. 2016. Bear: A framework for understanding application sensitivity to os (mis) behavior. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 388–399.

[39] Astral Team. 2025. Ruff: A fast Python linter, written in Rust. https://docs.astral.sh/ruff/

[40] OpenRouter Team. 2025. OpenRouter. https://openrouter.ai

[41] Pytest Development Team. 2025. pytest: Simple powerful testing with Python. https://docs.pytest.org/en/stable/

[42] Ruff Team. 2025. Rule Ruff: unspecified encoding. https://docs.astral.sh/ruff/rules/unspecified-encoding

[43] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An empirical study of bugs in test code. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 101–110.

[44] Jun Wang, Guanping Xiao, Shuai Zhang, Huashan Lei, Yepang Liu, and Yulei Sui. 2023. Compatibility issues in deep learning systems: Problems and opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 476–488.

[45] xAI. 2025. Grok 4 Fast. https://openrouter.ai/x-ai/grok-4-fast

[46] Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler. 2021. Productivity, portability, performance: Data-centric Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.