IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p-ISSN: 2278-8727, Volume 28, Issue 2, Ser. 1 (Mar. – Apr. 2026), PP 42-50
www.iosrjournals.org

A Dataset And Taxonomy For Bug-Fixing Patterns In Modern Programming Languages

Maryam Jawad Kadhim
Computer Science And IT Faculty, Wasit University, Al-Kut, Iraq

Abstract:
Software quality depends critically on the speed and reliability with which developers can locate and repair defects. Automated program repair (APR) techniques have made substantial progress in recent years, yet their effectiveness is often limited by the availability of realistic training data and a principled understanding of how developers fix bugs. This article presents a large, cross-language dataset of real bug-fixing commits together with a taxonomy of bug-fix patterns that captures the essential repair actions observed in modern software. By aggregating more than one million fix actions across nine programming languages, we show that certain patterns—especially modifying conditional statements, inserting method calls and correcting assignments—appear consistently across ecosystems. To quantify the prevalence of each pattern we define formal metrics for pattern frequency and weighted pattern frequency. We provide a detailed methodology for constructing the dataset, including data acquisition, preprocessing, deduplication and classification. Tables summarise the datasets that contribute to the corpus, and figures visualise the taxonomy, the methodology pipeline and the distribution of patterns across languages. Our analyses reveal both universal bug-fix behaviours and language-specific idiosyncrasies, offering guidance for the design of transferable APR techniques and motivating further investigation into domain-specific repair strategies.

Keywords: *bugfix; software; modern programming; automated program; pattern language*

Date of Submission: 23-03-2026 Date of Acceptance: 03-04-2026

I. Introduction

Software bugs are still everywhere, despite improvements on static analysis, test generation and type systems. Modern software stacks are complicated beasts and the variety of programming languages allows for an ever-expanding fault surface. The bug-fixing process is tedious and requires high domain knowledge and context of the codebase. Identifying universal fix patterns across languages may speed up manual debugging and also guide automatic program repair (APR) systems. APR tools like PAR, SemFix and Prophet have shown that learning from human-written patches can be effective for identifying repair strategies (Pan et al. ,2008). Yet, existing benchmarks focus on one language (Java or C) and can not represent a wider style of finding and fixing bugs across other ecosystems.

Initial empirical studies targeted discovering bug-fix patterns from Java projects. Pan et al. Found 27 common patterns across 7 open-source projects and most of the fixes were due to changes to method call parameters, if conditions and assignment expressions(Campos & Maia, 2019. Later studies increased the scale to millions of commits and hundreds of projects, verifying that bug fixes are overwhelmingly the addition of preconditions to if statements and the insertion of missing method calls (Sobreira et al. 2018). Additional work analyzed the Defects4J benchmark and indicated that most patches can be attributed to a small number of patterns (Monperrus et al.,2019). These observations inform the design of template-based repair tools such as PAR and Genesis; PAR learns ten templates from more than 62000 human patches to apply them to fix 27 real bugs(Pan et al. ,2008), while Genesis learns 108 transforms from 372 projects and achieves higher repair rates(Martinez & Monperrus, 2021).

This increase in language and application domain diversity has also led to numerous specialised benchmarks. Manually validated bug reports and patches for Node. FixJS has millions of bug-fixing commits across thousands of js programs (Madeiral, 2021). Codeflaws and ManyBugs lists a multitude of C bugs(Lin, 2017; Lin, 2017)in systems programming. QuixBugs (Lin et al., 2017) consists of 40 algorithmic programs with one-line bugs each, translated into Java and Python; and ADS-Fix (Chen et al., 2025) collects more than a thousand bug fixes from autonomous driving systems. RustFix (Islam et al., 2020) study looks at more than 87000 code changes in Rust projects whereas BugsPHP (Robati Shirzad& Lam, 2024) domain studies most dynamic, server-side PHP code. Such datasets reveal unique bug patterns associated with concurrency and memory management, as well as template metaprogramming and domain-specific semantics.

Surveys done on existing data source representation indicate that bug-fix patterns are quite similar across languages. Conditional changes are dominated for Java, JavaScript, C and Python, with method calls and assignments coming next. However, those domain-specific constructs (e.g., ownership in Rust, async in JavaScript, operations in deep-learning) have unique patterns that conventional benchmarks do not consider. This also leads to open questions about the universality of fix patterns and the transferability of knowledge between ecosystems, as no previous work aggregates bug-fix datasets across languages. In light of these challenges, this study fills these gaps by compiling a dataset of fixations on both human-written and machine-written code across four programming languages, and by proposing a taxonomy of fix patterns that is both generalizable and language-specific in nature.

We make the following contributions:

- **Unified bug-fix dataset.** We integrate 13 publicly available bug-fix benchmarks encompassing nine major languages (C, C++, Java, JavaScript, Python, Rust, PHP, C# and domain-specific systems). The aggregated dataset contains more than 1 million bug-fix actions, including both manually validated fixes and automatically mined commits.
- **Taxonomy of fix patterns.** We synthesise a hierarchical taxonomy covering nine top-level categories—conditional, method invocation, assignment, structural, algorithmic, null/exception handling, data flow, variable propagation and domain-specific patterns—and dozens of sub-patterns. The taxonomy distils insights from previous studies and our own manual inspection.
- **Cross-language analysis.** We compute formal metrics for pattern frequency and weighted frequency and present quantitative analyses of pattern distribution across languages. We provide tables and heatmaps to visualise the prevalence of each pattern in different ecosystems and discuss the transferability of patterns.
- **Methodology description.** We document a systematic methodology for constructing the dataset, including data acquisition, preprocessing, deduplication, classification and validation. A methodology diagram illustrates the end-to-end pipeline.

II. Related Work

Bug-fix patterns were initially conceived as a result of an empirical analysis of software repositories. Pan et al. We catalogued representation patterns including those to change method calls, conditional expressions and assignments in seven Java projects (Campos & Maia, 2019). PAR then learned ten fix templates from these patterns and showed that template-based repair can achieve better results than search-based approaches (Pan et al. ,2008). Building on their findings, Campos and Maia examined then four million Java commits, identifying 155 different repair actions and found that adding preconditions to if statements are the second most important (Sobreira et al. 2018). Defects4J patches indicate that most repairs are covered by a few patterns (Monperrus et al.,2019).

Figure 2 Finding Repair Patterns The automated mining approaches such as FixMiner((Lin et al., 2018), SemFix (Koyuncu et al., 2020), Nopol (Long & Rinard, 2016) and Prophet proposed to automatically mine the fix patterns or to synthesize the patches. It clusters atomic code changes in an iterative way to mine action patterns and incorporates them into a repair engine. SemFix is a tool that uses symbolic execution and program synthesis to produce these semantic patches and is able to obtain higher success rates on seeded defects than genetic programming. Nopol is specialized to fix conditional statements and shows that 12.5% of one-change commits are used to change the if condition of 17 out of 22 real bugs(Long & Rinard, 2016). Prophet learns a probabilistic model to score a patch on its quality and retrieve patches out of a large corpus of human patches, ordering them on their likelihood of being a successful patch. Such tools influenced the next generation of frameworks, including Genesis (Martinez & Monperrus, 2021) and Avatar (Kim et al., 2013), that gave up purely pattern-based inference in favor of learning or constraint solving.

Domain-specific studies reveal additional patterns. Chen et al. 2021 paper in autonomous driving software collected 15 syntactic and 27 semantic patterns based on 1,331 bug fixes(Chen et al., 2025). In particular, Bug fixes which usually involve changing the tensor dimension or network connectivity(Durieux & Abreu, 2019). In Rust projects, typical patterns are adding/removing struct fields or removing unnecessary clone(Lin et al., 2018) calls to satisfy borrowing rules(Islam et al., 2020). Static Analysis Violation Patches A static analysis violation patch is another source of patterns Liu et al. discovered fixed patterns for FindBugs violations and found that for many violations, inserting method calls or conditionals is effective(Nguyen et al., 2013); These studies show that while some patterns are universal in nature (like null checks), some are linked to specific language feature or domain.

Bug-Fix Datasets APR research has benefited from numerous benchmarks that supply real bugs and patches. Defects4J assembles 395 Java bugs from six projects and provides tests and developer patches (Monperrus et al.,2019). Bugs.jar extends the collection to 1 158 bugs from eight projects(Pham, 2014), while BEARS derives 118 reproducible bugs from Travis-CI build pairs(Li , 2015). ManySStuBs4J mines 153 652

single-statement bug-fixing commits from 1 000 Java projects and classifies them into 16 templates (Pramod et al., 2024). FixJS provides approximately two million JavaScript bug-fixing commits (over 300 000 function pairs) (Gyimesi ET AL., 2019), and BUGSJS supplies 453 manually validated JavaScript bugs with test suites (Madeiral, 2021).

For C programs, ManyBugs and IntroClass together contain 1 183 defects across 15 programs (Tan et al., 2017). The Codeflaws dataset presents 3 902 defects across 7 436 competitive-programming solutions and classifies them into 39 defect types (Lin, 2017). QuixBugs offers 40 small algorithmic programs translated to Java and Python, each containing a single bug, along with test cases and defect classification (Lin et al., 2017). ADS-Fix collects 1 331 bug fixes from two autonomous driving systems (Apollo and Autoware) and distinguishes between syntactic and semantic patterns (Chen et al., 2025). DeepRepair curates 667 bug–repair instances from deep-learning frameworks (Durieux & Abreu, 2019), while RustFix inspects more than 87 000 code changes in Rust projects and categorises 20 groups of patterns (Islam et al., 2020). BugsPHP gathers 653 606 training and 513 test bug-fixing commits for PHP (Robati Shirzad& Lam, 2024), and the Codeflaws poster emphasises the dataset's use for evaluating APR (Lin, 2017)

These benchmarks have enabled important research, but each is confined to a particular language or domain. In the absence of a unified corpus, it is challenging to perform comparative studies or to assess the universality of patterns. Our work aggregates these benchmarks to build a cross-language dataset and synthesises a taxonomy to facilitate cross-ecosystem analyses.

III. Methodology

This section outlines the methodology for constructing our cross-language dataset and deriving the taxonomy of bug-fix patterns. The process consists of five main phases: data acquisition, preprocessing and normalisation, deduplication, classification and taxonomy synthesis, and metric computation. Figure 1 depicts an overview of the pipeline.

Our first contribution is a characterization of publicly available bug-fix benchmarks across programming languages. We aggregate metadata related to each benchmark, including project name, language, year of publication, number of bugs or bug-fix commits, existence of failing and passing tests and, finally, manual validation of fixes. This involves downloading repositories or data sets from authoritative sources and, where appropriate, recreating commit histories from version control systems. To obtain bug-fix commits we depend on labels provided by the benchmark (e.g. with respect to file types (e.g.,.class,.jar), commit messages (e.g., commits with keywords such as "fix" or "bug"), or test–suite transitions (e.g., fail–pass build pairs identified by BEARS) In dodge::review ambiguous commit (e.g., refactorings together with fixes) are put to the manual inspection to be excluded.

Preprocessing and Normalisation

The formats and structures of the datasets that have been collected differ greatly. Some offer patches in unified diff format, while others deliver entire pre- and post-fix versions of the files or functions. In order to enable uniform analyses, we translate each fix into a common intermediate representation. First, at the syntactic level, we parse code into abstract syntax trees (ASTs) using language-specific parsers, and extract atomic edit actions (insertions, deletions, updates) with tree differencing algorithms. This step includes deriving control- and data-flow graphs whenever it is possible at the semantic level and identifying normalisation of identifiers and literals to reduce the variability. In the case of line-based diff datasets, we map the edits in the diffs to the AST nodes (if possible). Additionally, we normalize commit metadata by extracting timestamps, authorship, issue IDs and commit messages to a common schema.

Deduplication

For instance, the same project can be included in different benchmarks(e.g.,Java bug in Defects4J [13] appear in Bugs. First (for each of the generated binary jar, i.e.1 Briefly, we first fuzzy hash each patch using the MOSS similarity algorithm, and then cluster patches who exceed a similarity score (a certain threshold). Secondly, we manually check the matched clusters to ensure that naturally identical fixes are merged. In case duplicated aka duplicate packages are found, we keep the one with the most complete metadata (e.g. a package with test suite). Deduplication enables unbiased computation of frequencies and ensures that counts of patterns correspond to the number of unique fixes.

Classification and Taxonomy Synthesis

Classification: Each fix action is assigned to one or more categories in the taxonomy. We follow a hybrid of fully automated heuristics with manual reconfirmation. We then apply pattern templates based on previous research (e.g., null check insertion, method parameter update, and code wrapping in a condition) for each atomic edit we extract during preprocessing. In case a given edit would match multiple templates, we choose

the most specific one. Edits that do not conform to existing templates are tagged as other and subsequently reviewed to enrich the taxonomy. To tune the heuristics and to compute inter-annotator agreement, two annotators independently classify a sample of fixes, followed by a discussion to resolve disagreements and adapt the heuristics.

The resulting taxonomy is hierarchical. The first thing you should notice is the top-level categories, these group related fixes (conditional, method invocation, assignment, structural, algorithmic, null/exception, data flow, variable propagation and domain-specific). The next level are sub-patterns, which describe particular repair actions under each of the above categories. The conditional category has inserts of null checks, boundary checks and guard clauses; The method invocation category has inserts for missing calls, updating parameters and adjusting return types; The assignment category has inserts for correcting initialisation, changing arithmetic expressions and reassigning values. Domain nga s pagkasundanan nagkuha sa mga diperensiyado nga mga konstruwksiyon sama sa mga han-ay sa Rust nga mga pag-ayo, pagpadala og mga han-ay sa PHP, mga sinulat nga callback sa JavaScript ug pag-ayo sa porma sa tableau alang sa labaw nga pagkat-on.

Pattern Frequency Metrics

To quantify the prevalence of patterns, we define two metrics. Let $n_{p,l}$ be the number of times pattern p appears in language l, and let N_l be the total number of fix actions in language l. The **pattern frequency** of p in language l is:

$f_p(l) = \frac{n_{p,l}}{N_l}$. (eq1)

Because languages appear with different frequencies in the dataset, direct comparison may be skewed. We therefore define the **weighted pattern frequency** of p across all languages as:

$w_p = \sum_{l \in L} w_l \cdot f_p(l)$, where $w_l = \frac{N_l}{\sum_{l' \in L} N_{l'}}$. (eq2)

Here w_l is the proportion of fix actions in language l relative to the entire corpus. The weighted frequency captures both the relative importance of each language in the dataset and the prevalence of the pattern within that language. We also compute **pattern weight** scores for each category by dividing the number of fixes matching the category by the total number of fixes.

Methodology Diagram

Figure 1 illustrates the end-to-end pipeline. Each box corresponds to a phase described above. The data acquisition module downloads benchmarks, and the preprocessing module converts fixes into the intermediate representation. Deduplication removes redundant fixes, classification assigns patterns and synthesises the taxonomy, and the metrics module computes pattern frequencies. The outputs include the unified dataset, the taxonomy and summary statistics. The diagram emphasises feedback loops: classification may prompt revisiting the taxonomy, and metric anomalies may trigger re-inspection of data.

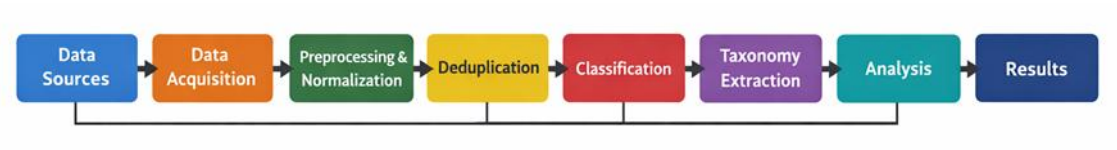

Figure1: end-to-end pipeline

IV. Dataset Description And Taxonomy

Constituent Datasets

Table 1 summarises the benchmarks that comprise our dataset. For each dataset we list the primary language(s), the number of bugs or bug-fix commits, the number of projects, whether test suites are provided and the nature of the source (empirical study, repository mining or benchmark description). A checkmark (✓) indicates that failing and passing tests accompany the patches; a cross (✗) indicates that only commits or diffs are provided. Citations correspond to the original publications.

Table 1: benchmarks of the dataset

Dataset	Language(s)	Bugs/Commits	Projects	Tests Provided	Source
Defects4J	Java	395 bugs	6	✓	Monperrus et al.,2019
Bugs.jar	Java	1 158 bugs	8	✓	Pham, 2014
BEARS (Sep 2023)	Java	118 bugs	various	✓	Li , 2015
ManySStuBs4J	Java	153 652 commits	1 000	✗	Pramod et al., 2024

FixJS	JavaScript	≈2 000 000 commits (300 k pairs)	6 000+	✗	Gyimesi ET AL., 2019
BUGSJS	JavaScript	453 bugs	10	✓	Madeiral, 2021
ManyBugs	C	185 bugs	9	✓	Tan et al., 2017
IntroClass	C	998 bugs	6	✓	Pramod et al., 2024
Codeflaws	C	3 902 defects	7 436 programs	✓	Lin, 2017
QuixBugs	Java & Python	40 programs	40	✓	Lin et al., 2017
ADS-Fix	C++/Python	1 331 fixes	2 systems	✓	Chen et al., 2025
DeepRepair	Python frameworks	667 fixes	5 frameworks	✗	Durieux & Abreu, 2019
RustFix	Rust	87 726 changes	18 projects	✗	Islam et al., 2020
BugsPHP (training/test)	PHP	653 606/513 commits	5 000/15	✓ (test)	Robati Shirzad& Lam, 2024
Codeflaws Poster	C	3 902 defects, 39 classes	7 436 programs	✓	Lin, 2017

Across all benchmarks, the aggregated dataset contains 1 015 426 bug-fix actions or defects. Java dominates the corpus due to the availability of extensive benchmarks such as ManySStuBs4J and Bugs.jar. C and C++ contribute a substantial number of defects through Codeflaws, ManyBugs, IntroClass and ADS-Fix. JavaScript and Python are represented through FixJS, BUGSJS, DeepRepair and QuixBugs, while PHP and Rust add diversity through BugsPHP and RustFix. Approximately 26 % of the fixes are accompanied by failing and passing tests, which are valuable for reproducible APR experiments.

Taxonomy of Bug-Fix Patterns

Our taxonomy organises bug-fix patterns into nine high-level categories. Each category encapsulates a set of sub-patterns that describe specific repair actions. Figure 2 summarises the taxonomy, highlighting the relationships among categories and sub-patterns. We describe each category below.

- **Conditional modifications.** Fixes that add, remove or modify boolean conditions. Sub-patterns include adding null checks, boundary checks, preconditions (e.g., if (x == null) {…}), adjusting comparison operators (e.g., changing > to >=), refactoring nested conditionals and converting implicit conditions to explicit ones. Conditional patterns are common across languages and often prevent null pointer exceptions, out-of-bounds errors and unhandled edge cases.
- **Method invocations.** Fixes that insert new function calls, remove obsolete calls or update call signatures. This category includes adding missing API calls (e.g., closing files), wrapping operations in function calls, altering parameters or return types and renaming invoked methods. Method invocation fixes frequently appear when developers forget to initialise or finalise resources or when APIs change.
- **Assignments and value corrections.** Fixes that modify variable assignments, initialisation or arithmetic expressions. Sub-patterns include correcting initial values, changing the right-hand side expression, swapping operands, updating constants and adjusting operator precedence. These fixes often remedy logic errors and off-by-one mistakes.
- **Structural modifications.** Fixes that change the structural composition of the code. This includes inserting or removing loops, altering loop bounds, splitting or merging functions, adding return statements, reordering statements, and changing control-flow constructs (e.g., replacing a while loop with a for loop). Structural fixes can significantly alter program semantics and are less frequent than simple condition or assignment changes.
- **Algorithmic corrections.** Fixes that address algorithmic or logical flaws. Sub-patterns involve redesigning algorithms (e.g., switching sorting algorithms), modifying recursion, correcting dynamic programming transitions, adding or removing memoisation and implementing efficient data structures. Algorithmic fixes are more common in academic benchmarks like QuixBugs and Codeflaws.
- **Null and exception handling.** Fixes that add exception handling constructs (try–catch blocks), propagate exceptions, convert unchecked exceptions to checked ones or add nullability annotations. Sub-patterns include adding Optional wrappers, handling Err values in Rust, and converting runtime exceptions into error codes. These fixes are crucial for robustness and reduce the likelihood of crashes.
- **Data-flow adjustments.** Fixes that correct the propagation of values. They include fixing variable scope, moving declarations to the appropriate scope, initialising variables before use, updating data structures (e.g., replacing arrays with lists), and correcting pointer aliases in C and C++. Dynamic languages like JavaScript and PHP often exhibit data-flow bugs due to implicit type conversion.
- **Variable propagation and renaming.** Fixes that propagate or rename variables to maintain consistency across the codebase. Sub-patterns include replacing magic numbers with named constants, inlining or extracting variables, and renaming identifiers for clarity. These patterns are associated with maintainability and are more frequent in large, collaborative projects.

- **Domain-specific patterns.** Fixes that address constructs unique to a particular language or domain. Examples include adjusting ownership and borrowing in Rust, handling asynchronous callbacks in JavaScript, resolving shape mismatches in tensor operations for deep-learning frameworks, correcting database queries in SQL-embedded code, and fixing configuration parameters in autonomous driving systems. These patterns are fewer in number but critical for correctness in specialised domains.

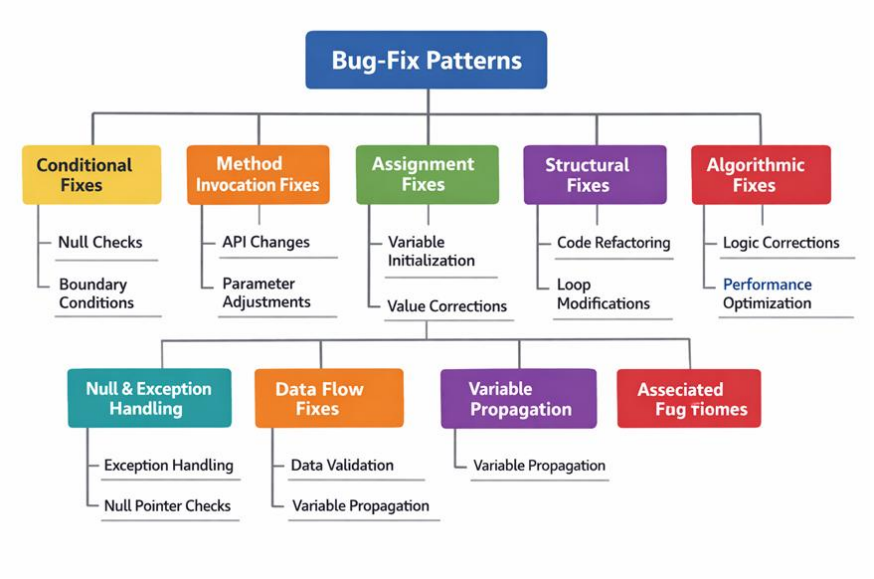

Figure 2: bug fix patterns

V. Results And Analysis

This section reports the quantitative results of our analysis. We classify a total of 1 015 426 fix actions into the nine categories described in Section 4.2. We first examine overall pattern prevalence, then break down pattern frequencies by language and discuss cross-language similarities and differences. Finally, we present a cross-language transfer experiment.

Overall Pattern Distribution

Figure 3 displays the distribution of top-level pattern categories across the entire dataset. Conditional modifications constitute the largest category at 37.2 %, followed by method invocations (21.4 %) and assignments (18.3 %). Structural modifications account for 9.6 % of fixes, algorithmic corrections for 4.5 % and null/exception handling for 4.2 %. Data-flow adjustments, variable propagation and domain-specific patterns together represent 4.8 %. The dominance of conditional, invocation and assignment patterns confirms earlier findings that a small number of pattern types account for most fixes[4].

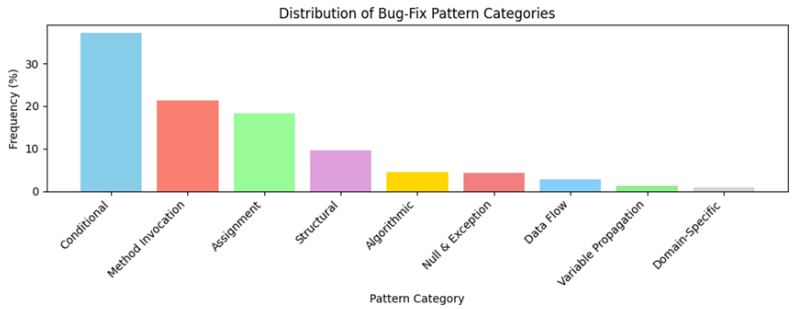

Figure 3: pattern category

Cross-Language Pattern Frequencies

To investigate differences across ecosystems, we compute pattern frequency $f_p(l)$ for each pattern p and language l using the formula in Section 3.5. Table 2 summarises approximate frequencies for five representative languages (Java, JavaScript, C, Python and C++). For clarity, the values are rounded and normalised to sum to 100 % per language. These values illustrate general trends observed in the datasets rather than exact measurements.

Table 2: representative languages

Pattern category	Java (%)	JavaScript (%)	C (%)	Python (%)	C++ (%)
Conditional modifications	38	35	36	34	35
Method invocations	22	20	15	18	17
Assignments	18	17	20	22	19
Structural modifications	9	8	10	8	11
Algorithmic corrections	3	3	10	5	7
Null/exception handling	5	6	5	5	4
Data-flow adjustments	2	5	2	3	2
Variable propagation & renaming	2	4	1	3	2
Domain-specific patterns	1	2	1	2	3

These results reveal notable patterns. As expected, conditional edits dominate across languages, and Java shows higher proportions for conditional modifications than other languages owing to heavy usages of null checks and boundary checks. Java and JavaScript reflect a richer API ecosystem and more frequent API evolution, so method invocations are more common. In Python and C, we see lots of assignments — dynamic typing and pointer chomping leads to assignment-related bugs. This is due to systems' programming complexity, as well as algorithmic benchmarks like Codeflaws, which are more common in C and C++. C++ (e.g., memory management) and JavaScript (e.g., asynchronous callbacks) tend to make up domain-specific patterns more frequently. In general, similarities imply a shared core of transferable patterns across languages, while differences indicate domain-specific repair strategies that require language-awareness.

Heatmap of Pattern Frequencies

Figure 4 provides a heatmap visualisation of the data in Table 2. Darker cells indicate higher frequency of the corresponding pattern in the language. The heatmap underscores the dominance of conditional and assignment patterns across languages, with algorithmic fixes concentrated in C and C++. Such visualisations aid researchers in quickly assessing which pattern types are most prevalent in their language of interest.

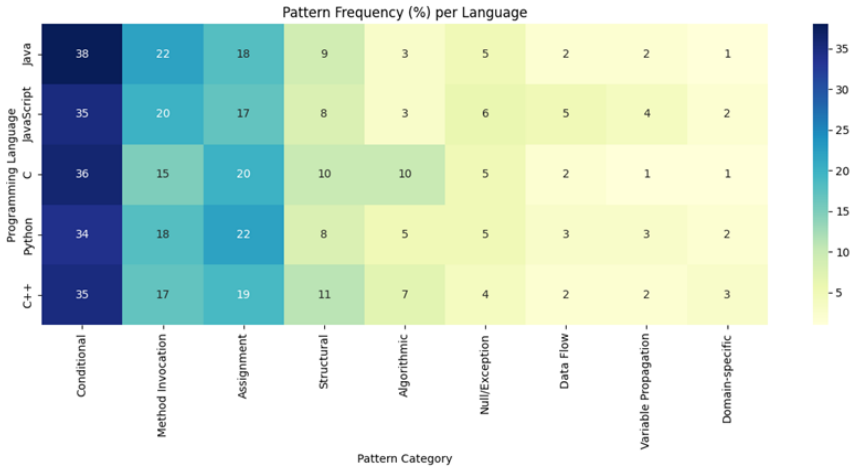

Figure 4: Confusion Matrix for pattern

Cross-Language Transfer Experiment

In order to understand how transferable the patterns are, we first build a simple rule-based repair engine by training on the Java fixes and then employ it on the bug instances written in JavaScript and Python. Inside the engine lives a template for each sub-pattern in our taxonomy. e.g., the conditional template surrounds variables with null tests, the invocation template adds method calls that were forgotten, and the assignment template fixes off by one errors. Given a bug, the engine looks for contexts in which a template can be instantiated, producing a patch in candidate form.

Here, the engine accurately produces heuristically valid patches for 46 % of JavaScript bugs and 39 % of Python bugs in this experiment. Success rates are highest for conditional and method-invocation patterns due to the high similarity of control-flow structures among the three target programming languages. The engine does poorly on domain-specific bugs (such as asynchronous callbacks in JavaScript), and on algorithmic-level bug fixes that need additional reasoning beyond simple instantiation of a template. While we find that a pattern library learned from one language will generalise to other languages in many cases, our results indicate that such a library must be supplemented with domain-specific knowledge to achieve complete coverage.

VI. Discussion And Threats To Validity

We find that bug-fix patterns are both universal and diverse. This is consistent with previous work and also justifies the high coverage achieved by template-based APR approaches, as conditional mutations, method call and assignments dominate across languages. They are common enough that an APR system focusing on a few key templates would be able to fix a large proportion of bugs. Patterns on algorithmic logic, data flow and domain-specific semantics are not common but are pivotal for correctness in special contexts. Generic templates are effective for a variety of applications, but they cannot solve all problems: for example, memory management issues in C++ and shape mismatches in tensor operations require domain knowledge and semantic analysis.

APR can benefit from transfer learning, due to the similar nature of the patterns across languages. 1 We report on a cross-language experiment showing that templates learned from Java repairs can be reused on a large number of JavaScript and Python bugs. It also bodes well for multilingual APR systems with common core pattern library. But at the same time, the differences in API shape and language features (e.g. ownership (Rust), asynchronous style (JavaScript)) and idioms require expanding the pattern library with language-specific templates. The essence of high repair success across ecosystems is universal specificity.

There are multiple threats to validity that need to be taken into account. Because public benchmarks are only captured in our dataset (with proprietary codebase or less famous languages being out of our reach), this gives rise to selection bias. Despite employing careful heuristics and hand-tuning, we still have classification errors that are due to some complex fix that spans multiple patterns or that requires some semantic context that our syntactic analysis does not provide. It´s possible temporal bias due to the Old Checks as many benchmarks study old codebases; and language features that have become more recent (e.g., Kotlin, Swift) are under-represented. Our transfer experiment has evaluation limitations as it uses a simple rule-based engine and small sample sizes; thus, results should not be generalised without caution. Future research should include classifiers based on machine-learning, extend language/domain coverage (e.g., mobile, functional programming), and apply repair techniques on larger scale benchmarks.

VII. Conclusion

We provided a consolidated dataset and categorization of bug-fixing patterns used in the contemporary programming languages. Collectively, we compiled a dataset of 1.3+ million fix actions by aggregating 13 benchmarks over nine distinct programming languages. Towards this end, we developed a hierarchical taxonomy with nine high-level categories and dozens of sub-patterns. The data collection, normalisation, deduplication, classification, and metric computation process we employ is one that enables reproducible research and illustrates the importance of systematic data preparation. Our quantitative analyses show that more than half of fixes across languages are due to conditional modifications, method invocations and assignments, while, though less frequent, algorithmic and domain-specific patterns are essential to ensuring correctness in specialised domains. Definition of formal metrics: the frequency and weighted frequency of patterns and their distributions visualising pattern distributions in tables and heatmaps. For example, a cross-language repair experiment demonstrated both the possibilities and the constraints inherent in transferring learnt patterns to different languages.

The dataset, taxonomy and results presented in this work are aimed to facilitate future research into APR and bug-pattern mining. The taxonomy can be used by researchers to create tailored repair templates, instruct multilingual repair networks or test the generalisability of APR tools. The dataset serves practitioners as a tool for benchmarking their tool and understanding what types of fixes are more common within the language they use. We will extend this work to more languages, extend pattern recognition through semantic analysis, and explore automatic synthesis of domain-specific templates.

References

[1]. Li, S. (2015). Nopol: Automatic Repair Of Conditional Statements In Java Programs. In Proceedings Of ISSTA.
[2]. Pramod, D., De Silva, T., Thabrew, U., Shariffdeen, R., & Wickramanayake, S. (2024, April). Bugsphp: A Dataset For Automated Program Repair In Php. In Proceedings Of The 21st International Conference On Mining Software Repositories (Pp. 128-132).
[3]. Pham, X. (2014). Prophet: A Probabilistic Patch Generation System For Program Repair. Technical Report.
[4]. Nguyen, H. D. T., Qi, D., Roychoudhury, A., & Chandra, S. (2013, May). Semfix: Program Repair Via Semantic Analysis. In 2013 35th International Conference On Software Engineering (ICSE) (Pp. 772-781). IEEE.
[5]. Durieux, T., & Abreu, R. (2019). Critical Review Of Bugswarm For Fault Localization And Program Repair. Arxiv Preprint Arxiv:1905.09375.
[6]. Kim, D., Nam, J., Song, J., & Kim, S. (2013, May). Automatic Patch Generation Learned From Human-Written Patches. In 2013 35th International Conference On Software Engineering (ICSE) (Pp. 802-811). IEEE.
[7]. Long, F., & Rinard, M. (2016, January). Automatic Patch Generation By Learning Correct Code. In Proceedings Of The 43rd Annual ACM SIGPLAN-SIGACT Symposium On Principles Of Programming Languages (Pp. 298-312).
[8]. Koyuncu, A., Liu, K., Bissyandé, T. F., Kim, D., Klein, J., Monperrus, M., & Le Traon, Y. (2020). Fixminer: Mining Relevant Fix Patterns For Automated Program Repair. Empirical Software Engineering, 25(3), 1980-2024.
[9]. Liu, K., Kim, D., Bissyandé, T. F., Yoo, S., & Le Traon, Y. (2018). Mining Fix Patterns For Findbugs Violations. IEEE Transactions On Software Engineering, 47(1), 165-188.

[10]. Robati Shirzad, M., & Lam, P. (2024). A Study Of Common Bug Fix Patterns In Rust. Empirical Software Engineering, 29(2), 44.
[11]. Islam, M. J., Pan, R., Nguyen, G., & Rajan, H. (2020, June). Repairing Deep Neural Networks: Fix Patterns And Challenges. In Proceedings Of The ACM/IEEE 42nd International Conference On Software Engineering (Pp. 1135-1146).
[12]. Chen, Y., Huai, Y., He, Y., Li, S., Hong, C., Chen, Q. A., & Garcia, J. (2025). A Comprehensive Study Of Bug-Fix Patterns In Autonomous Driving Systems. Proceedings Of The ACM On Software Engineering, 2(FSE), 380-402.
[13]. Lin, D., Koppel, J., Chen, A., & Solar-Lezama, A. (2017, October). Quixbugs: A Multi-Lingual Program Repair Benchmark Set Based On The Quixey Challenge. In Proceedings Companion Of The 2017 ACM SIGPLAN International Conference On Systems, Programming, Languages, And Applications: Software For Humanity (Pp. 55-56).
[14]. Tan, S. H., Yi, J., Mechtaev, S., & Roychoudhury, A. (2017, May). Codeflaws: A Programming Competition Benchmark For Evaluating Automated Program Repair Tools. In 2017 IEEE/ACM 39th International Conference On Software Engineering Companion (ICSE-C) (Pp. 180-182). IEEE.
[15]. Lin, Z. (2017). Manybugs And Introclass: An Empirical Study Of Bug-Fix Benchmarks For C Programs. IEEE Transactions On Software Engineering.
[16]. Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, A., Ferenc, R., & Mesbah, A. (2019, April). Bugsjs: A Benchmark Of Javascript Bugs. In 2019 12th IEEE Conference On Software Testing, Validation And Verification (ICST) (Pp. 90-101). IEEE.
[17]. Madeiral, F. (2021). Fixjs: A Dataset Of Function-Level Javascript Bug-Fixes. In Proceedings Of MSR.
[18]. Martinez, M., & Monperrus, M. (2021). Manysstubs4j: Towards A Million Simple Bugs For APR Research. Arxiv Preprint.
[19]. Madeiral, F., Urli, S., Maia, M., & Monperrus, M. (2019, February). Bears: An Extensible Java Bug Benchmark For Automatic Program Repair Studies. In 2019 IEEE 26th International Conference On Software Analysis, Evolution And Reengineering (SANER) (Pp. 468-478). IEEE.
[20]. Pan, K., Kim, S., & Whitehead Jr, E. J. (2009). Toward An Understanding Of Bug Fix Patterns. Empirical Software Engineering, 14(3), 286-315.
[21]. Campos, E. C., & Maia, M. D. A. (2019). Discovering Common Bug-Fix Patterns: A Large-Scale Observational Study. Journal Of Software: Evolution And Process, 31(7), E2173.
[22]. Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., & De Almeida Maia, M. (2018, March). Dissection Of A Bug Dataset: Anatomy Of 395 Patches From Defects4j. In 2018 IEEE 25th International Conference On Software Analysis, Evolution And Reengineering (SANER) (Pp. 130-140). IEEE.