IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 3 June 2026, accepted 3 July 2026, date of publication 13 July 2026, date of current version 22 July 2026.

Digital Object Identifier 10.1109/ACCESS.2026.3713096

APPLIED RESEARCH

PR4Code: A Pull Requests Dataset for AI Code Generation

BENEDETTA DONATO, (Student Member, IEEE), LEONARDO MARIANI, (Senior Member, IEEE), DANIELA MICUCCI, (Member, IEEE), AND OLIVIERO RIGANELLI
University of Milano-Bicocca

Corresponding author: Benedetta Donato (benedetta.donato@unimib.it)

This work was supported in part by MUR under the Grant "Dipartimenti di Eccellenza 2023–2027" of the Department of Informatics, Systems and Communication, University of Milano-Bicocca, Italy; and in part by the Engineered MachinE Learning-intensive IoT systems (EMELIOT) National Research Project funded by MUR under the PRIN 2020 Program under Contract 2020W3A5FY.

ABSTRACT AI-driven code generation is evolving from handling small, localized code-completion tasks toward supporting the implementation of full software features. To support this transition, this paper introduces PR4Code, a dataset of **4,508 Java** and **8,831 Python** curated Pull Requests (PRs) collected from GitHub, each enriched with metadata, commit histories, and detailed code changes. Unlike resources focused on isolated code fragments, PR4Code captures the feature implementation process, delivering a large and diverse collection of real-world development tasks that can be used to assess AI-driven solutions. Our analysis reveals substantial variability in PR structure, commit granularity, and textual content. In addition, we release scripts to regenerate and update the dataset, ensuring reproducibility and maintainability.

INDEX TERMS Pull requests, feature development, AI-driven code generation, LLMs, AI assistants, commits, code changes.

I. INTRODUCTION

Software evolution is a continuous process that results in a stream of code changes produced by developers. Each modification reflects a decision that embodies how a problem was understood and solved, for instance, to add, modify, or improve functionalities. In recent years, AI-based systems have been increasingly adopted to support and accelerate a wide range of software development activities.

Recent advances in Large Language Models (LLMs), such as GPT-5 [1], Claude [2], and specialized code-oriented models such as CodeLlama [3] and CodeGemma [4], have demonstrated remarkable capabilities in generating code from natural-language specifications. These models are now embedded in widely adopted industrial tools, including GitHub Copilot [5], Amazon CodeWhisperer [6], and Tabnine [7]. Empirical studies report measurable productivity gains associated with the adoption of such tools in real-world development settings [8], [9], [10].

Despite these advances, current AI-based approaches remain largely effective only for relatively small-scale tasks, such as code completion [11], [12], [13] and method-level generation [14], [15], [16]. Their performance drops significantly when the goal is to generate larger and more complex code units, including full features, modules, or applications. Such tasks require a deeper understanding of software architecture, cross-file dependencies, and integration with existing components, aspects that current models still struggle to handle effectively [14], [17], [18], [19].

A major obstacle to systematically studying these limitations is the lack of datasets that capture feature-level software development in realistic settings. Existing benchmarks and corpora often focus on isolated code changes [20], single commits [21], or narrowly scoped bug-fixing tasks [22], and therefore fail to represent the broader development context in which real-world changes are conceived, iteratively refined, and integrated across multiple artifacts.

The associate editor coordinating the review of this manuscript and approving it for publication was Claudia Raibulet.

In fact, feature development is typically documented through natural-language descriptions that may be incomplete or cross-referenced, implemented via multiple commits, and realized through coordinated changes spanning source code, tests, configuration files, and documentation.

To address this gap, this paper introduces *PR4Code*, a large-scale dataset of real-world pull requests collected from popular Java and Python GitHub repositories. PR4Code adopts pull requests as the primary unit of analysis, as they naturally encapsulate the lifecycle of a development task, from its textual description to the sequence of commits and the final integrated changes. For each pull request, the dataset provides rich contextual information, including natural-language descriptions at both PR and commit levels, fine-grained and cumulative code diffs, and metadata that make it possible to reconstruct the development history and the surrounding context.

By construction, PR4Code enables the investigation of fundamental research questions concerning AI-based code generation in realistic software engineering scenarios. In particular, the dataset makes it possible to study whether current AI-based approaches can autonomously implement non-trivial software features within existing projects, how effectively they can handle development tasks involving heterogeneous artifacts beyond source code (e.g., tests and configuration files), and to what extent cross-referenced or non-self-contained natural-language descriptions affect the quality and correctness of generated code changes. These questions are difficult to address using existing benchmarks, yet are central to understanding the practical limitations of LLM-based systems in feature-level development.

The contributions of this paper are threefold. First, we present PR4Code, a curated dataset comprising 13,339 pull requests and 46,334 associated commits that span 1,055 Java and Python repositories. Second, we provide a detailed characterization of the dataset, analyzing both textual descriptions and code changes to highlight the diversity and complexity of real-world feature development tasks. Third, we release the scripts used to construct the dataset, enabling reproducibility and facilitating future extensions aligned with the evolving landscape of AI-based software development tools.

The paper is organized as follows. Section II discusses our work in the context of related studies. Section III describes the methodology we used to build the PR4Code dataset. Section IV presents the PR4Code dataset. Section V discusses some exemplary uses of the dataset. Section VI reports early results from a small-scale experiment. Finally, Section VII provides final remarks.

II. RELATED WORK

The problem of evaluating LLMs and AI-based systems on realistic software engineering tasks has motivated the creation of a variety of datasets and benchmarks, each targeting different granularities of code change and development contexts.

Early resources mainly focused on isolated code fragments or single-function modifications. HumanEval [23] evaluates the ability of models to generate correct function implementations from short natural-language prompts using test-based metrics such as pass@k. Recent extensions such as HumanEval-XL [24] broaden this setting to multilingual scenarios while still preserving a function-level generation paradigm. More recently, benchmarks such as HumanEval Pro and MBPP Pro [25] introduced self-invoking code generation tasks, requiring models to reuse previously generated code to solve more complex problems. These works highlight the growing interest in evaluating reasoning and compositional capabilities beyond isolated function synthesis. However, such benchmarks still do not fully capture repository-level software evolution and realistic pull-request-based development workflows.

SWE-bench [17] is a curated collection of 2,294 issue–pull request pairs extracted from 12 popular Python repositories, designed to assess whether models can automatically resolve software issues by producing the corresponding reference pull request. Each instance in SWE-bench includes the textual description of the issue and the validated code patch, and correctness is assessed by executing the project test suites. This benchmark represents an important step toward realistic evaluation scenarios compared to earlier synthetic code-generation datasets; however, it remains limited in scale (only a few thousand examples from 12 repositories) and is largely monolingual (Python).

PR4Code broadens both the scope and diversity of curated pull-request datasets. It collects more than 13,000 real pull requests from both Java and Python projects, each enriched with associated commits, NL descriptions, and metadata, with the goal of supporting large-scale, multi-language evaluation of code generation techniques. PR4Code covers heterogeneous types of development activities, including bug fixes, new features, and refactoring across two distinct language ecosystems, thus providing a broad and diverse benchmark in terms of task difficulty and content. Moreover, PR4Code is primarily intended for evaluation rather than large-scale training, preserving its role as an independent testbed to evaluate model generalization.

Beyond issue-centered benchmarks, datasets with large collections of commits and code diffs that can be used to train models and evaluate code understanding and automated bug-fixing capabilities are available. A representative example is MegaDiff [21], a corpus of more than 663,000 Java code differences extracted from repositories (largely derived from Defects4J [20]) through strict filtering based on commit messages and diff size. MegaDiff supports tasks such as commit comprehension, fault localization, and automated program repair, and offers a large number of changes (all in Java) categorized by magnitude. However, datasets such as MegaDiff typically do not preserve the full structure of a pull request (e.g., multiple commits or rich textual descriptions), as they mainly focus on isolated diffs paired with

commit messages. PR4Code, instead, retains the complete PR context: not only the final diff, but also the sequence of commits, the descriptive text at both PR and commit level, and additional metadata (e.g., authorship, timestamps, links to issues). This richer context makes PR4Code more suitable for simulating realistic code integration scenarios, where a model must understand and generate changes that are coherent with both a natural-language description and a development history.

Recent evaluation methodologies [13], [17] have also emphasized the importance of large-scale, realistic benchmarks that go beyond isolated code snippets, incorporating execution-based validation, repository-level context, and task-oriented evaluation metrics. These approaches aim to better capture the complexity of real-world software development and provide more reliable assessments of model capabilities at scale.

Other datasets concentrate on very simple bug-fix patches, often at a function or single-statement level. For example, ManySStuBs4J [22] provides 153,652 single-statement bug fixes extracted from 1,000 Java projects, each labeled according to one of 16 frequent bug templates. This corpus is valuable for evaluating automated program-repair techniques on simple and repetitive defects, but it is highly homogeneous, being limited to Java, involving minimal code changes, and often accompanied only by short commit messages. In contrast, PR4Code includes pull requests of widely varying size, ranging from a few-line corrections to larger feature implementations affecting tens of lines and multiple files, and it exhibits substantial variability in the quality and length of textual descriptions, from short titles to detailed PR narratives. This ensures greater heterogeneity in both code and natural language, requiring evaluated models to cope with vague or informal descriptions and to generate or explain non-trivial code changes. Similarly to ManySStuBs4J for Java, the FixJS dataset [26] was proposed in the JavaScript domain, extracting approximately 300,000 before/after function pairs from about two million commits, while preserving original commits and modified files to provide rich context. Although FixJS represents a significant advance in scale and detail for training automated repair models, it remains focused on single-function changes within one language, whereas PR4Code targets multi-file, cross-artifact modifications that are more typical of real collaborative pull requests.

A complementary line of work investigates the interaction between AI agents and software development in the wild. Recent works [27], [28] on autonomous coding agents have further explored the ability of LLM-based systems to iteratively plan, generate, and refine code within realistic development environments, often combining multiple roles (e.g., planner, coder, reviewer) and interacting with external tools and repositories. These approaches highlight both the potential and the current limitations of agent-based software development, especially when dealing with complex, multi-step and multi-file tasks.

The recent AIDev dataset [29], for example, captures more than 456,000 pull requests generated by autonomous agents (e.g., Codex- or Copilot-based systems) across approximately 61,000 GitHub repositories, together with rich metadata on authorship (human vs. AI), review timelines, code review outcomes, and merge decisions. AIDev provides a unique observational perspective on how AI-generated contributions are received and how they differ from human-authored PRs in terms of complexity and quality. Although AIDev is not structured as a controlled evaluation benchmark, it highlights the importance of studying realistic development artifacts and the limitations of purely synthetic benchmarks such as SWE-bench for answering questions about in-the-wild behavior. In this respect, PR4Code shares the emphasis on realism and on pull requests as the central unit of analysis, but differs in its objective: PR4Code offers a fixed, curated set of historical PRs to support repeatable experimental evaluation, whereas AIDev serves as a living corpus for empirical observation. Both, however, stress the relevance of including rich metadata and heterogeneous file types, as real PRs typically involve not only source code but also tests, comments, and documentation.

PR4Code positions itself within the landscape of recent datasets and benchmarks for code generation and modification as a complementary and necessary resource. Compared to commit-level corpora and small-patch datasets, it provides a higher-level, pull-request-centric view of software change, including multiple commits and heterogeneous artifacts. Compared to issue-focused benchmarks such as SWE-bench, it offers greater diversity in terms of projects, programming languages (Java and Python), types of development activities, and styles of natural-language descriptions. By capturing realistic, context-rich pull requests, it provides the research community with a robust and realistic testbed for assessing the capabilities of LLMs and AI agents in feature-level software development.

III. DATASET CONSTRUCTION METHODOLOGY

The construction of the PR4Code dataset follows four sequential steps, as shown in Figure 1. The first step selects GitHub repositories that are relevant and remain active in terms of code changes and pull request creation. The second step selects closed pull requests that encapsulate feature development, bug fixing, or incremental improvements to the project. The third step analyzes the set of commits associated with each pull request. Finally, the fourth step analyzes the cumulative effect of the commits included in the same pull request. The output of this process is a curated and publicly available dataset of pull requests associated with the development of features extracted from relevant repositories.

A. REPOSITORY SELECTION

We selected all GitHub repositories with significant popularity (*>500 GitHub stars*), a threshold chosen to ensure a certain level of community adoption, relevance, and maturity, with recent pull requests created after the training cutoff

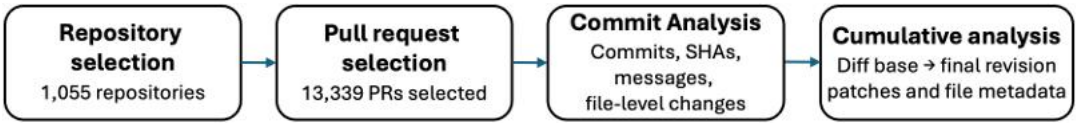

FIGURE 1. Methodology used to construct the PR4Code dataset.

date of several LLMs (after *October 3, 2023*, which is the cutoff date of ChatGPT 4o), in order to reduce the risk of data leakage and ensure that the selected PRs are unlikely to be included in the training data of such models. We further restricted the selection to repositories having as primary language either *Java* or *Python*, the two most used languages in GitHub [30].

This results in 1,055 repositories selected.

B. PULL REQUEST SELECTION

For each selected repository, we retrieved and analyzed all the *closed PR*s, that is, PRs that represent completed development activities. The goal of the analysis is to identify PRs representing feature development, bug fixing, or incremental improvement of the code, each accompanied by a description of the change, an essential component for constructing realistic input prompts for AI-based code generation. To this end, a PR is selected in PR4Code if it satisfies the following conditions:

- **Kind of development activity mentioned in the PR:** we select only the PRs whose title or body include at least one of the following keywords `requirement`, `feature`, `story`, `add`, `implement`, `enhance`, `remove`, `upgrade`, `change`, `create`, `resolve`, `develop`, `fix`, `update`, `support`, `merge`, `refactor`, `improvement` or `build`.
- **Sufficient text:** the combined length of the PR title and body must be of at least *100 characters*, ensuring a sufficiently detailed description of the developer's intent; this threshold was introduced to exclude trivial or uninformative PRs (e.g., very short descriptions such as "fix bug"), which do not provide enough context for constructing meaningful input prompts, as observed during a preliminary inspection of PR descriptions of varying lengths.
- **Semantic branch:** to select PRs that actually represent feature development, bug fixing, or incremental improvement of the code, we rely on branch naming conventions [31], [32]. That is, we retrieve the Git branch associated with the PR, and select only those PRs whose name of the associated branch includes a keyword among `feature`, `feat`, `story`, `task`, `fix`, `chore`, `enhancement`, `improvement`, or `spike`.

We selected a total of 13,339 PRs, by applying the criteria above, 4,508 from Java projects and 8,831 from Python projects.

C. COMMIT ANALYSIS

We retrieved the information about the set of commits associated with each PR, to precisely understand how the project was changed because of the PR. For each commit, we extracted the **unique SHA identifier** and the **timestamp** (author date) of the commit to enable the reconstruction of the development activities; the **commit message** describing the implemented change; the **list of modified files**; and, for each file, the number of added lines (`additions`), deleted lines (`deletions`), and total changes (`additions + deletions`). We also counted the **textual references** within the commit message, aggregating URLs, markdown links, and issue identifiers. These references are useful to discriminate self-contained commit descriptions (i.e., descriptions with no references) and commit descriptions that require a broader context to be interpreted (i.e., the descriptions of the referred entities). PR4Code stores all commit-level information alongside the corresponding pull request, allowing cross-referencing between commit history, textual descriptions, and code modifications.

D. CUMULATIVE ANALYSIS OF CODE CHANGES

To reconstruct the full set of modifications done by developers to implement a PR, we computed the diff between the project revision available before and after the completion of the PR. Specifically, we compared the **base snapshot** of the PR to the **final snapshot** associated with the merged changes. This comparison yields the complete set of changes introduced throughout the lifecycle of the pull request.

For each modified file, we retrieved its **path**; its modification **status** (*added*, *modified*, or *removed*); the number of lines **added**, **deleted**, and total **changed**; the textual **patch** (line-by-line diff); and the URLs for direct file retrieval (`blob_url`, `raw_url`, `contents_url`). This metadata provides a detailed and faithful representation of both the size and complexity of the modifications applied within the scope of each PR.

The dataset, the scripts used to build it, and the result of the selection process are available online [33].

IV. THE PR4Code DATASET

The PR4Code dataset is organized as a hierarchical collection of repositories and PRs. For each PR, PR4Code includes the files modified by the PR, both in the version before and after the changes have been applied, together with a JSON file containing metadata about the changes. Metadata is also aggregated at the level of the entire dataset, distinguishing between Java and Python projects. The metadata includes all the information extracted during the dataset construction process, as described in Section III.

To illustrate the structure and content of PR4Code, Figure 2 shows an example of a pull request as it appears on GitHub.

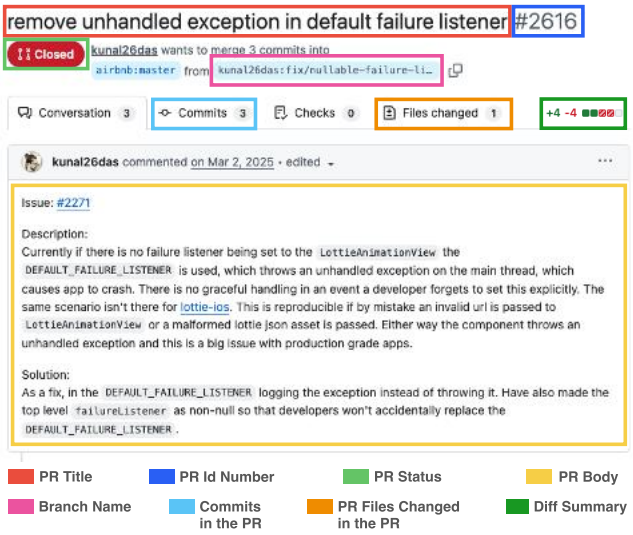

FIGURE 2. Example of a pull request included in PR4Code.

The figure highlights the main elements considered during dataset construction, including the PR title and identifier, the PR status, the associated branch name, the natural-language description (body), the number of commits in the PR, the number of files changed in the PR, and the diff summary associated with the changes.

These elements are systematically extracted and stored in PR4Code as structured metadata, enabling the reconstruction of the development task from its textual description to the corresponding code modifications. The example also illustrates the realistic nature of the PRs included in the dataset, which often involve detailed descriptions with references or links to external artifacts, multiple commits and modifications affecting several files.

In the following, we describe the content and structure of the dataset in more detail.

A. DESCRIPTIONS IN PRs AND COMMITS

We analyzed the descriptions provided by developers within PRs and commits, as this information is essential for defining realistic and context-aware prompts for AI-based code generation. In particular, we computed the length and content of 4,508 PRs and 14,639 commits from Java repositories, and 8,831 PRs and 31,695 commits from Python repositories. Below, we discuss the main findings of this analysis from multiple perspectives.

1) KEYWORD FREQUENCY

To characterize the purpose of PRs, we analyzed the occurrence of the keywords used for PR selection in the title and body of PRs. Figure 3 shows the results. The goal of this keyword-based analysis is not to measure the importance or complexity of PRs, but rather to characterize the diversity of development activities represented in the dataset. PR complexity is analyzed separately through other dimensions, such as the number of commits, modified files, and size of code changes.

To assess the reliability of the keyword-based characterization reported in Figure 3, we manually inspected a sample of 300 PRs balanced across Java and Python projects. Since a PR may contain multiple keywords, we validated individual keyword occurrences by checking whether they were semantically coherent with the actual intent of the PR. Overall, we validated 888 keyword occurrences, of which 78.49% were classified as coherent. The highest coherence values were observed for more specific terms such as *fix* (92%) and *refactor* (94%), while more generic keywords such as *update* (61%) or *merge* (39%) were more prone to ambiguous usages. These results suggest that the keyword distribution provides a meaningful, although not perfect, high-level characterization of development activities.

Consistently with this validation, the keywords *add* and *fix* are the most frequent ones, indicating that adding new functionality and fixing existing behavior are among the most common development activities captured in the PRs. This suggests that PR4Code reflects common and representative software evolution tasks, rather than niche or highly specialized changes. Requirement implementation and software updates are also relatively common, while the remaining keywords occur less frequently. The similarity between Java and Python further indicates a consistent way in which developers describe code evolution across different language ecosystems.

2) LENGTH AND STRUCTURE OF TEXTUAL DESCRIPTIONS

Figures 4 and 5 show the size of PR descriptions, considering the number of lines and characters. PR4Code includes a wide range of real cases, from extremely short to extremely long descriptions (e.g., thousands of lines and characters) revealing substantial variability in how developers document their work. This variability reflects the diversity of real-world development practices, where descriptions may range from minimal summaries to detailed narratives including context and rationale.

For Java projects, the PR body has an average length of 19.3 lines (median 13), while for Python projects, the average length is 21.3 lines (median 15). In terms of character count, Java PR bodies contain on average 986 characters (median 577), compared to 1,040 characters (median 669) for Python. The most verbose descriptions exceed 30,000 characters.

Titles are substantially more compact, with an average of 51 characters in Java and 48 in Python, reflecting a consistent tendency toward concise summarization practices across languages.

PR descriptions and titles report information about the nature and goal of changes, providing valuable context for assessing AI agents tasked with planning and generating code for full feature implementations.

Figures 6 and 7 show the size of commit descriptions, considering the number of lines and characters. Not surprisingly, commit descriptions are shorter than PR descriptions, as they are intended to document fine-grained and incremental changes. This distinction reinforces the hierarchical nature

FIGURE 3. Frequency of keywords in PR titles and bodies for Java and Python projects, highlighting the most common types of development activities captured in PR4Code.

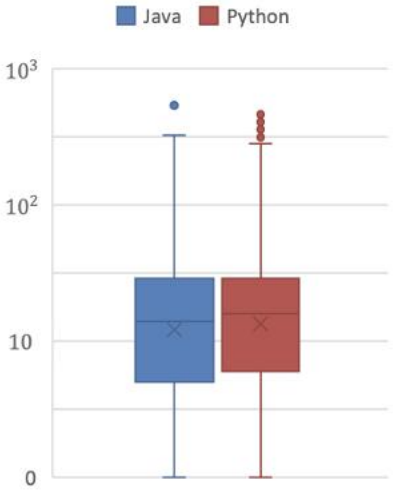

FIGURE 4. Distribution of the number of lines in PR descriptions.

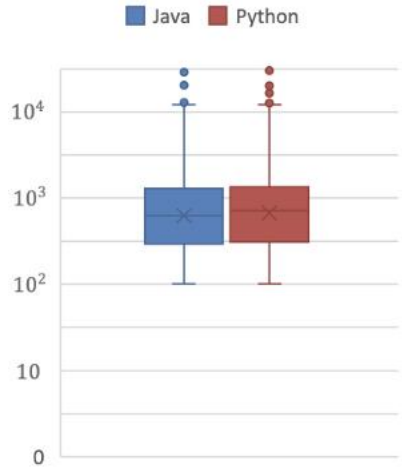

FIGURE 5. Distribution of the number of characters in PR descriptions.

of development information in PR4Code, where high-level intent is captured at the PR level and low-level actions at the commit level. The average message length is 86 characters in Java (median 48) and 105 in Python (median 43), typically spanning 1 or 2 lines. Yet, these descriptions document individual code-level changes that can be used to train and assess AI agents implementing features.

Overall, these quantitative figures indicate that PR descriptions typically provide an informative summary of the implemented changes, although with considerable variability in length and detail. In contrast, commit messages tend to offer a more concise and operational view of individual changes, highlighting the complementary roles of PR- and commit-level descriptions.

3) HYPERLINKS AND CROSS-REFERENCES

Sometimes descriptions are not self-contained, since they may include references to external sources. Expressions that automatically close issues (*Fixes #…*, *Closes #…*, *Resolves #…*) occur in 1,201 Java and 2,402 Python PRs, confirming the diffusion of GitHub automation syntax. Hyperlinks are predominantly located in the body of pull requests, where developers provide contextual information about the requirements and the functionalities to be implemented. In particular, 2,487 out of 4,508 Java PRs and 4,576 out of 8,831 Python PRs contain at least one external link.

The diffusion of hyperlinks and cross-references suggests that PRs typically consist of structured text, not just plain text. Overall, 55% of Java PRs and 52% of Python PRs include at least one reference to an external artifact. This indicates that fully understanding the nature of a PR may require (transitively) accessing several other sources, challenging, for instance, the construction of prompts for AI assistants.

Our analysis of template usage reveals a consistent adoption of standardized textual structures across PRs. In particular, markdown checklists (`- [ ]`) appear in 2,431

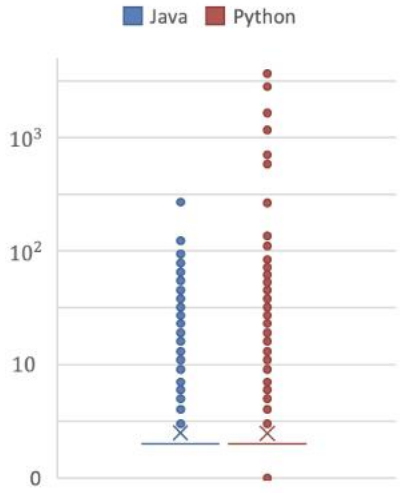

FIGURE 6. Number of lines per commit description.

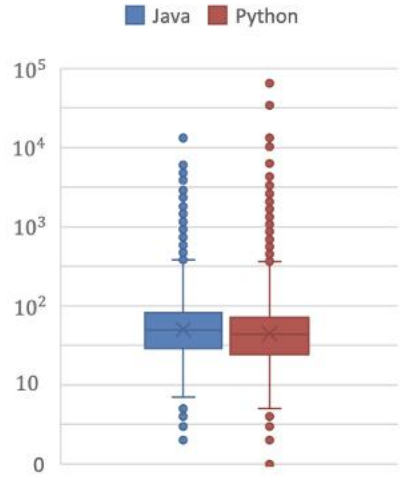

FIGURE 7. Number of characters per commit description.

out of 4,508 Java PRs and 5,106 out of 8,831 Python PRs, indicating a broader adoption of structured templates in Python projects. Collaboration markers such as *Signed-off-by* and *Co-authored-by* are also more frequent in Python, reflecting more explicit authorship tracking practices. In particular, Signed-off-by tags occur in 529 Java PRs and 492 Python PRs, while Co-authored-by markers appear in 147 Java PRs and 495 Python PRs. The use of structured information may simplify automated analysis and text understanding, offering additional analysis opportunities.

B. CHARACTERISTICS OF THE PRs

We analyzed the structure and content of PRs, considering their commits, file and line modifications, file types, and the occurrence of tests and comments in the changes.

1) COMMITS PER PR

Figure 8 shows the number of commits per PR. In both Java and Python the number of commits per PR is usually small,

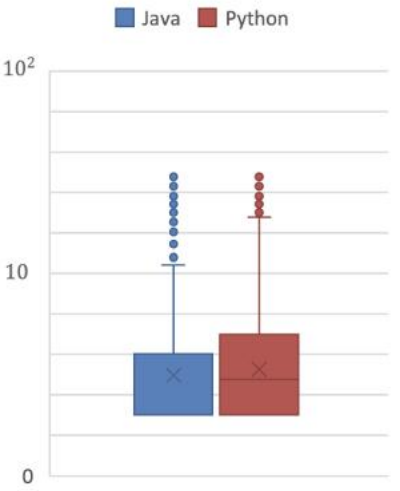

FIGURE 8. Distribution of the number of commits per PR.

indicating that most development tasks are implemented through a limited sequence of coherent changes. This suggests that many feature implementations are relatively structured and manageable in terms of development steps. In Java, the average number of commits per PR is 3.25 (median 1), with 54% of the PRs containing a single commit and 75% of the PRs no more than three. In Python, the average number of commits per PR is 3.59 (median 2), with 47% of the PRs containing a single commit and 75% of the PRs no more than four. The noticeable gap between mean and median values indicates a skewed distribution, where most PRs contain a few commits, while a limited number of PRs involve substantially larger commit sequences. Manual inspection of several high-commit PRs revealed that these cases are often associated with large-scale development activities, such as framework migrations, rendering-engine refactorings, infrastructure updates, and feature branches that underwent multiple review-driven revisions before merge. Only 7.7% and 10.5% of the PRs in Java and Python projects exceed eight commits (with a maximum of 30), indicating that most PR implementations are concise and composed of few logically distinct changes. As a consequence, the median better reflects the typical PR size in the dataset, whereas the mean is influenced by these less frequent but more complex development scenarios. This insight is helpful for the design of AI-based code generators that must calibrate the magnitude of suggested changes according to the description of the code-generation task.

2) FILES MODIFIED PER PR

Figure 9 shows the number of files modified by PR. In Java, PRs modify on average 3.47 files and in Python 4.26 files. Most PRs (75% for Java projects and 70% for Python projects) affect only one or two files, while a few involve hundreds, typically due to refactoring or large merges. The fact that many of the changes are localized to a few files

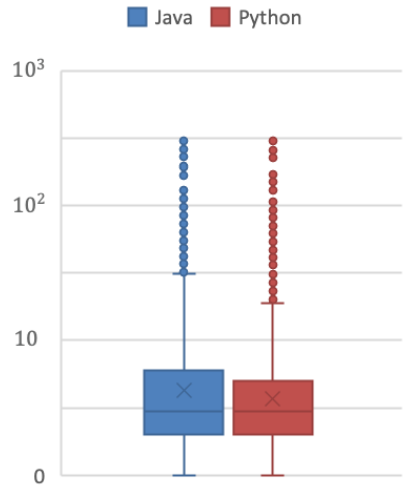

FIGURE 9. Distribution of the number of files modified per PR.

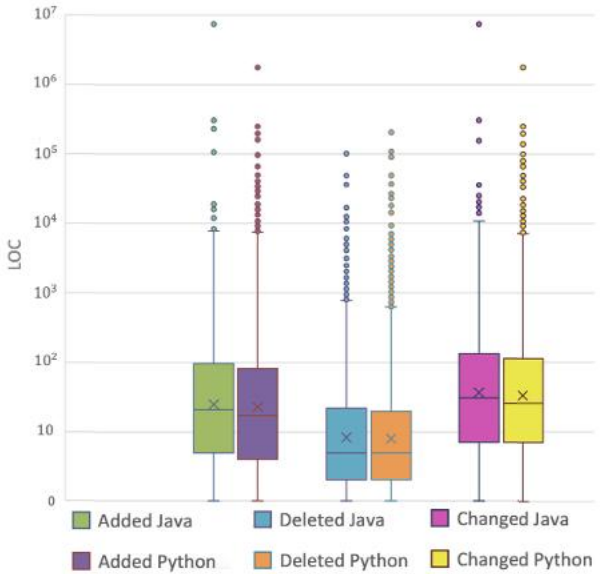

FIGURE 10. Distribution of added, deleted, and modified lines per PR.

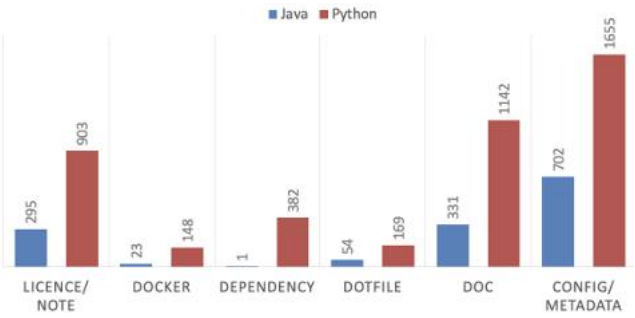

FIGURE 11. Number of PRs that modify non-code artifacts.

and distributed across a few commits only suggests that many features can be added to projects by implementing fairly localized changes. This also indicates that a substantial portion of feature implementations is modular and contained, which is particularly relevant for AI-assisted code generation techniques that must reason about localized modifications.

3) LINES ADDED, REMOVED, AND CHANGED

Figure 10 shows the number of lines added, removed, and changed per PR. The distribution is extremely skewed, with several outliers, reflecting the coexistence of small-scale maintenance commits and large-scale restructuring operations. This variability is particularly relevant for evaluation purposes, as it allows assessing model performance across tasks of different sizes and complexity levels.

Although changes are localized in a few commits and files only, the magnitude of changes within files is quite spread, with several PRs modifying more than 100 lines of code, which is a challenging magnitude for AI-based techniques.

4) TYPE OF MODIFIED FILES

Not all changes are limited to source code files. In particular, only 44.4% of Java PRs and 50.6% of Python PRs exclusively modify language-specific source files (i.e., `.java` files for Java projects and `.py` files for Python projects). We found that 23.8% of PRs in Java projects and 25.6% of PRs in Python projects modify both language-specific source files and other types of files. Surprisingly, a relevant portion of PRs, 31.8% of PRs in Java projects and 23.5% of PRs in Python projects, do not modify any Java or Python file. This highlights the prevalence of changes targeting non-code artifacts, which are essential components of modern software systems.

We analyzed the nature of changes not targeting source-code files. Figure 11 shows the results. Changes mainly target configuration and metadata files (e.g., `README.md`, `LICENSE`, `CHANGELOG.md`), followed by documentation, licensing, and dependency-related artifacts, highlighting the importance of non-code elements in real-world development. Changes to these files are more frequent in Python projects than in Java projects.

This result is particularly relevant for AI-based code generation, showing that effective feature development requires the ability to handle multiple types of files rather than merely generating source code. The PR4Code dataset provides a wide variety of representative cases that can be used to assess future AI-based solutions.

5) PRESENCE OF TEST FILES

Indeed, changes may affect not only the components of the applications hosted in the repositories, but also their test files. To estimate how often changes to tests occur, similarly to other studies [34], [35], we measured the percentage of changes affecting files whose names or hosting folders include the string `test`. We discovered that 38.2% of PRs in Java and 33.4% of PRs in Python include changes to at least one test file. These findings confirm that PR implementations also require the evolution of test files, as also reported in previous studies [34], [35].

6) COMMENT-RELATED CHANGES

Code evolution requires not only evolving the code, but also evolving the comments that describe the code. We detected code comment modifications as changes to comment lines defined with //, /*, or #. We discovered that nearly half of the changes affect comments (49.2% of PRs in Java, 54.6% of PRs in Python), while comment-only changes are rare (below 1%). These results highlight the importance of co-evolving code and documentation, and indicate that AI-based solutions must be capable of generating and updating comments alongside source code.

C. CHARACTERISTICS OF COMMITS

We briefly summarize here data about commits.

1) LINES ADDED, REMOVED, AND CHANGED

Commits intuitively represent small changes that evolve the project in a semantically coherent way. Those units of change serve as the basic granularity that AI-based tools have to mimic when attempting to emulate the work done by developers. The average number of modified lines per commit is non-trivial; in fact, it amounts to 37.8 lines modified across 9.38 files for Java projects and 42.7 lines modified across 5.97 files for Python projects.

2) TYPE OF MODIFIED FILES

Similarly to PRs, the percentage of commits modifying source code files only amounts to 56.1% and 50.8% for Python and Java projects, respectively. While mixed commits involving auxiliary files (e.g., configuration or documentation) account for about one-fifth. Between 26% and 30% of the commits modify non-code files only, such as build scripts or README documents. These results confirm that, although source code changes dominate, a significant portion of activity targets peripheral project elements. The higher share of code-only commits in Python reflects its simpler structure, whereas Java's ecosystem often results in more non-code changes due to its complex build and deployment configurations.

3) PRESENCE OF TEST FILES

Also, for changes to test files, the commit-level statistics resemble the PR-level statistics, with 29.1% of Java commits and 34.5% of Python commits involving at least one test file.

To further characterize the differences between Java and Python pull requests, we performed a statistical comparison across the main quantitative dimensions of the dataset using the Mann–Whitney U test, a non-parametric statistical test suitable for skewed and non-normally distributed software engineering data. To complement statistical significance testing and avoid overinterpreting minimal differences in large datasets, we additionally computed Cliff's Delta and Vargha–Delaney A12 effect sizes to quantify the magnitude of the observed differences. Table 1 summarizes the results.

Although several comparisons resulted in statistically significant differences, all observed effect sizes are negligible according to both Cliff's Delta and Vargha–Delaney A12 statistics. This indicates that, despite minor distributional variations, Java and Python PRs exhibit overall comparable characteristics in terms of commit structure, code churn, textual description size, and file modification patterns. The results therefore suggest that the dataset maintains a relatively balanced structure across the two language ecosystems, reducing the risk that subsequent analyses are disproportionately influenced by language-specific artifacts rather than by general software development behaviors.

The largest differences are observed for metrics such as files changed per commit and commit message length; however, even in these cases the effect sizes remain negligible. This further supports the view that the two subsets share broadly similar development dynamics despite small variations in coding and contribution practices.

D. EXTERNAL VALIDATION OF DATASET CHARACTERISTICS

To further validate the construction of PR4Code, we compared its characteristics with both an existing repository-level benchmark and a random sample of GitHub pull requests. This comparison aims to assess whether the dataset reflects realistic pull-request-based development activities and whether the keyword-based selection introduces an artificial characterization of PR intent.

First, we compared PR4Code with SWE-bench [17], one of the closest existing resources for repository-level evaluation. As summarized in Table 2, SWE-bench contains 2,294 issue–PR pairs from 12 Python repositories and focuses on executable issue-resolution tasks, PR4Code includes 13,339 pull requests from 1,055 Java and Python repositories. Moreover, PR4Code covers a broader range of development activities, including bug fixes, feature additions, updates, refactorings, and changes involving heterogeneous artifacts such as source code, configuration files, documentation, and tests. This comparison highlights the complementary nature of PR4Code: rather than focusing only on issue resolution, it provides a broader and more diverse collection of pull-request-level development tasks.

Second, we collected a random sample of 300 closed GitHub pull requests, balanced across Java and Python projects and subject to the same temporal constraint adopted for PR4Code, but without applying the keyword, branch-name, or minimum-description-length filters. As summarized in Table 3, 19% of the random PRs have an empty body and 27% have a combined title and body shorter than 100 characters. This supports the need for a minimum textual-content threshold when constructing input prompts for AI-based code generation. At the same time, 71% of the random PRs contain at least one of the selection keywords, indicating that terms such as ''add'', ''fix'', and ''update'' are common in general GitHub development activity rather than being artifacts introduced by our filtering process.

TABLE 1. Statistical comparison between Java and Python PRs across the main quantitative dimensions of PR4Code.

Metric	Java median	Python median	p-value	Cliffs delta	Effect size	Vargha–Delaney
Commits per PR	1	2	$< 10^{-3}$	-0.0662	negligible	0.4669
LOC added per PR	20	16	0.01	0.0274	negligible	0.5137
LOC deleted per PR	4	4	0.14	0.0154	negligible	0.5077
LOC changed per PR	30	25	$< 10^{-3}$	0.0349	negligible	0.5175
PR title length	49	46	$< 10^{-3}$	0.0780	negligible	0.5390
PR body length	576	668	$< 10^{-3}$	-0.0449	negligible	0.4775
PR body lines	13	15	$< 10^{-3}$	-0.0450	negligible	0.4775
Commit message length	48	43	$< 10^{-3}$	0.0854	negligible	0.5427
Commit message lines	1	1	$< 10^{-3}$	0.0209	negligible	0.5104
Files changed per commit	2	1	$< 10^{-3}$	0.1031	negligible	0.5515
Source-only PRs	0	1	$< 10^{-3}$	-0.0622	negligible	0.4689

TABLE 2. Comparison between PR4Code and SWE-bench.

Dimension	SWE-bench	PR4Code
Task unit	Issue PR pair	Pull request
Languages	Python	Java, Python
Repositories	12	1,055
Instances	2,294	13,339
Main focus	Issue resolution	Feature-level code changes
Repository diversity	Limited	Broad
Includes commits	Patch-level	Commit sequences and cumulative diff
Non-code artifacts	Limited	Explicitly characterized

TABLE 3. Characteristics of the random GitHub PR sample used for external validation.

Metric	Java	Python	All	PR4Code
Median title+body chars	436.5	823.5	661.0	683.5
Empty body	27%	11%	19%	None
Title+body $<$ 100 chars	37%	18%	27%	None
With selection keyword	67%	74%	71%	100%
With links/references	50%	59%	55%	53%
Median commits per PR	1	1	1	2
Median changed files per PR	2	3	3	2
Median changed lines per PR	69	119.5	95.5	28

This comparison suggests that PR4Code selects pull requests with more explicit development intent and sufficient textual context, while preserving terminology and structural characteristics commonly observed in real GitHub pull requests. Together with the manual validation of keyword occurrences reported in Section IV-A1, this provides additional evidence that the dataset construction is not solely heuristic-driven and that the selected PRs are suitable for realistic AI-based code generation studies.

V. DISCUSSION

The PR4Code dataset includes PRs, commits, and code changes that offer a wide range of meaningful challenges for AI-based code-generation techniques. For instance, it includes changes of various sizes (from a few to hundreds of thousands of lines), targeting a variable number of elements (from a few to hundreds of code locations and files), that modify artifacts of heterogeneous nature (e.g., code, configuration files, comments, dependencies, licenses, etc.), and documented through natural-language descriptions at both PR and commit level. PR4Code can thus be used to address several research questions about software evolution and feature development. In light of the growing need to evaluate AI-based approaches on feature-level code changes, we discuss below representative usage scenarios.

- **Feature-level code generation**. PR4Code can be used to evaluate the ability of AI systems to implement complete software features starting from natural-language descriptions. In this scenario, the model receives the PR description and must generate the corresponding multi-file changes. This represents a realistic setting, where the model must interpret the developer's intent and produce coherent modifications across the codebase. The PRs in PR4Code cover a wide range of real-world feature implementations, varying in size, complexity, and description detail.
- **Incremental development and commit modeling.** The availability of commit sequences enables the evaluation of incremental code generation. Models can be tasked with reproducing the evolution of a PR step by step, generating intermediate commits. This allows assessing whether the model can mimic realistic development workflows and maintain consistency across successive changes.
- **Multi-file and heterogeneous artifact handling**. Since PR4Code includes changes affecting different types of artifacts (e.g., source code, configuration files, documentation, and tests), it can be used to assess the ability of models to handle heterogeneous development tasks. Implementing a feature often requires dealing with multiple file types, and nearly half of the PRs in PR4Code include non-code elements, enabling evaluation beyond traditional source code generation.
- **Handling of cross-referenced and non self-contained descriptions.** PR descriptions are often not standalone, as they may include references to external artifacts (e.g., issues, links, or other PRs), as reported in Section IV-A. This introduces additional complexity, requiring models to interpret distributed and potentially incomplete information. PR4Code enables the evaluation of both prompt construction techniques and code generation approaches under these realistic conditions.

There are also limitations related to the PR4Code dataset. For instance, PR4Code focuses on pull requests from popular open-source Java and Python repositories selected using a popularity threshold. While this choice ensures realistic and non-trivial development scenarios, it may limit generalizability to less popular projects, other programming languages, or proprietary settings.

Moreover, this selection process may introduce biases toward well-structured and actively maintained repositories, where development practices (e.g., clearer documentation, standardized workflows, and more informative PR descriptions) are more mature than in smaller or less active projects. As a consequence, the dataset may underrepresent noisy, poorly documented, or irregular development processes that are also common in real-world settings.

Similarly, the filtering criteria applied to pull requests (e.g., minimum text length and keyword-based selection) may bias the dataset toward PRs with more explicit and descriptive natural-language content, potentially excluding shorter or less structured PRs that still represent valid development activities. This could affect the evaluation of AI-based approaches, as models may be exposed to clearer input descriptions than those encountered in practice.

Moreover, keyword-based categorizations may not fully reflect the actual complexity or practical relevance of PRs, as rare but critical development activities may appear less frequently in the observed distributions. For this reason, the keyword analysis should be interpreted as a descriptive characterization of development activity types rather than as a direct measure of task importance or difficulty.

Nevertheless, Java and Python are widely used, and the dataset exhibits substantial diversity in project characteristics, PR sizes, and artifact types.

Restricting PR4Code to pull requests created after October 3, 2023 mitigates the risk of training data leakage for models with a known cut-off date, but does not eliminate it. Different LLMs may rely on partially undisclosed or continuously updated training corpora, and newer models may already include some PRs in their training data. The temporal constraint should therefore be interpreted as a best-effort mitigation; future studies should explicitly report the training horizon of evaluated models and, where feasible, apply additional leakage controls.

VI. USING PR4Code

PR4Code is designed not only as a curated dataset, but also as a practical resource for evaluating AI-based code generation techniques in realistic software engineering scenarios. In this section, we discuss how the dataset can be used in practice, outlining a typical usage workflow, possible application scenarios, and a small-scale experiment that demonstrates its applicability.

A. USAGE WORKFLOW

PR4Code provides all the artifacts required to reconstruct real development tasks. For each PR, the dataset includes the natural-language description (title and body), the base version of the repository, the sequence of commits, and the cumulative code changes. A typical workflow proceeds as follows. A PR is selected and its description is used as input to the model. The corresponding base version of the repository is retrieved using the provided commit identifier, enabling reconstruction of the initial state. The model is then tasked with generating the required changes. The output is compared against the reference implementation, either at the level of cumulative diffs or individual commits. The availability of structured metadata and repository links supports flexible and reproducible experimentation.

B. SMALL-SCALE APPLICABILITY EXPERIMENT

To further demonstrate the practical applicability of PR4Code and complement the descriptive analysis with evaluation-oriented results, we conducted a small-scale repository-level code generation experiment using an agentic pipeline based on the DeepSeek-R1:8B LLM. The objective of this experiment is to show that PR4Code can be operationally used to instantiate realistic pull-request-driven code generation scenarios involving repository-level modification, compilation, testing, and manual inspection of generated patches. This experiment is intended to demonstrate dataset usability rather than to provide a statistically representative evaluation of model performance.

We selected 10 pull requests from PR4Code according to the following criteria: (i) the pull request described a localized modification that could be manually inspected, (ii) the repository could be compiled in isolation using Maven or Gradle, (iii) the pull request included either modified tests or executable tests related to the target functionality, and (iv) the change affected at most three files to enable detailed manual verification. The selected pull requests included bug fixes and small feature modifications involving real multi-file repository contexts.

For each pull request, we checked out the version of the repository immediately preceding the first commit associated with the PR. The agentic pipeline was then prompted to implement the feature from the pull request title and body. The pipeline was composed of four stages. First, a planning agent analyzed the pull request title and body to identify the intended modification and the repository files potentially involved in the change. Second, a context retrieval agent collected repository-level information related to the target files, including the original source code and surrounding project context. Third, a code generation agent produced modified source files starting from the original repository version and the retrieved contextual information. Finally, an evaluation agent automatically compiled the repository and executed the tests associated with the target functionality.

Each generated patch was evaluated according to four criteria:

1) Successful compilation of the modified repository,

TABLE 4. Summary of the small-scale applicability experiment conducted with PR4Code.

Repository	N. PR	LOC	N. Test	Compile	Test Passes
lottie-android	2457	362	1	Yes	No
mockito	3509	764	1	Yes	Yes
aeron	1734	823	1	No	No
junit4	1765	198	1	Yes	No
karatelabs	2415	493	2	No	No
grobid	1075	623	1	Yes	Yes
jadx	2326	353	2	No	No
pinpoint	12591	631	1	Yes	Yes
strimzi-kafka	11429	526	1	Yes	Yes
vavr	3045	975	1	No	No

2) Successful execution of the project tests,
3) Manual verification that the requested change described in the pull request was effectively implemented,
4) Manual identification of unrelated or unnecessary modifications outside the intended change scope.

The manual verification was performed by comparing the pull request description, the original developer patch, and the patch generated by the agentic pipeline. These criteria provide preliminary effectiveness metrics for repository-level code generation in realistic pull-request-driven development scenarios.

The experiment showed mixed results across the selected pull requests. Table 4 summarizes the selected PRs and the corresponding evaluation outcomes. Specifically, 6 out of 10 generated patches successfully compiled, while 4 out of 10 also passed the target tests associated with the pull request. Manual inspection showed that the requested modification described in the PR was correctly implemented in four cases, although the generated solution sometimes differed from the original developer implementation. This behavior is plausible since multiple semantically valid implementations may exist for the same pull-request requirement, especially in repository-level development scenarios involving refactoring decisions, alternative API usages, or different localization strategies for the modification. We also noticed three implementations that, although not passing the tests, were nearly identical to the developers' implementations. Furthermore, a very limited number of generated patches introduced unrelated or unnecessary modifications outside the intended change scope, suggesting that the pipeline generally remained focused on the requested repository changes.

These results highlight two important aspects. First, PR4Code can be effectively used to instantiate realistic repository-level code generation experiments involving pull-request interpretation, repository navigation, multi-file modification, compilation, testing, and qualitative inspection. Second, the results reported in Table 4 show a gap between the implementation of the requested modification and successful test execution, suggesting that the selected pull requests remain challenging for current agent-based code generation systems.

VII. CONCLUSION

AI-based code generation is gradually evolving toward increasingly automated tools capable of addressing code generation tasks at progressively larger scales. This paper presents PR4Code, a curated dataset of pull requests and associated data that provides a foundation for investigating key research questions in AI-based code generation and software evolution. As reported in the paper, PR4Code includes a large collection of heterogeneous cases, including large PRs spanning many files and locations, as well as modifications involving non-source code artifacts. The dataset is available publicly for use and experimentation [33]. Future work will explore the use of PR4Code to identify the capabilities and limitations of state-of-the-art AI-based techniques, and to support the development of next-generation models capable of handling realistic, feature-level software changes.

REFERENCES

[1] OpenAI. (2025). *GPT-5*. Accessed: Nov. 4, 2025. [Online]. Available: https://openai.com/gpt-5
[2] Anthropic. (2024). *Claude Haiku*. [Online]. Available: https://www.anthropic.com
[3] B. Rozière et al., "Code llama: Open foundation models for code," 2023, *arXiv:2308.12950*.
[4] C. Team et al., "CodeGemma: Open code models based on gemma," 2024, *arXiv:2406.11409*.
[5] GitHub and OpenAI. (2024). *GitHub Copilot*. Accessed: Nov. 4, 2025. [Online]. Available: https://github.com/features/copilot
[6] Amazon Web Services. (2024). *Amazon Q Developer: Generative AI Assistant for Software Development*. Accessed: Nov. 4, 2025. [Online]. Available: https://aws.amazon.com/q/developer/
[7] Tabnine. (2024). *Tabnine AI Code Completion*. Accessed: Nov. 4, 2025. [Online]. Available: https://www.tabnine.com
[8] N. Davila, I. Wiese, I. Steinmacher, L. L. da Silva, A. Kawamoto, G. J. P. Favaro, and I. Nunes, "An industry case study on adoption of AI-based programming assistants," in *Proc. 46th Int. Conf. Softw. Eng., Softw. Eng. Pract.*, Apr. 2024, pp. 92–102.
[9] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of AI on developer productivity: Evidence from GitHub copilot," 2023, *arXiv:2302.06590*.
[10] Z. K. Cui, M. Demirer, S. Jaffe, L. Musolff, S. Peng, and T. Salz, "The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers," Tech. Rep., 2025.
[11] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, "Language models for code completion: A practical evaluation," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Mali, Apr. 2024, pp. 1–13.
[12] R. A. Husein, H. Aburajouh, and C. Catal, "Large language models for code completion: A systematic literature review," *Comput. Standards Interfaces*, vol. 92, Mar. 2025, Art. no. 103917.
[13] Q. Wu, C. Peng, P. Gao, R. Hu, H. Gan, B. Jiang, J. Tang, Z. Deng, Z. Guan, C. Gao, X. Liu, and P. Yang, "RepoMasterEval: Evaluating code completion via real-world repositories," in *Proc. 40th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2025, pp. 3672–3683.
[14] V. Corso, L. Mariani, D. Micucci, and O. Riganelli, "Generating Java methods: An empirical assessment of four AI-based code assistants," in *Proc. 32nd IEEE/ACM Int. Conf. Program Comprehension*, Apr. 2024, pp. 13–23.
[15] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "LLM-based test-driven interactive code generation: User study and empirical evaluation," *IEEE Trans. Softw. Eng.*, vol. 50, no. 9, pp. 2254–2268, Sep. 2024.
[16] I. D. Fagadau, L. Mariani, D. Micucci, and O. Riganelli, "Analyzing prompt influence on automated method generation: An empirical study with copilot," in *Proc. 32nd IEEE/ACM Int. Conf. Program Comprehension*, Apr. 2024, pp. 24–34.
[17] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real-world GitHub issues?" in *Proc. Int. Conf. Learn. Represent.*, 2024.

[18] Z. Ságodi, G. Antal, B. Bogenfúrst, M. Isztin, P. Hegedus, and R. Ferenc, "Reality check: Assessing GPT-4 in fixing real-world software vulnerabilities," in *Proc. 28th Int. Conf. Eval. Assessment Softw. Eng.*, Jun. 2024, pp. 252–261.
[19] Y. Dong, X. Jiang, J. Qian, T. Wang, K. Zhang, Z. Jin, and G. Li, "A survey on code generation with LLM-based agents," 2025, *arXiv:2508.00083*.
[20] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2014, pp. 437–440.
[21] M. Monperrus, M. Martinez, H. Ye, F. Madeiral, T. Durieux, and Z. Yu, "Megadiff: A dataset of 600k Java source code changes categorized by diff size," 2021, *arXiv:2108.04631*.
[22] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur?: The ManySStuBs4J dataset," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, Jun. 2020, pp. 573–577.
[23] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.
[24] Q. Peng, Y. Chai, and X. Li, "Humaneval-XL: A multilingual code generation benchmark for cross-lingual natural language generalization," in *Proc. Joint Int. Conf. Comput. Linguistics, Lang. Resour. Eval.*, 2024.
[25] Z. Yu, Y. Zhao, A. Cohan, and X.-P. Zhang, "HumanEval pro and MBPP pro: Evaluating large language models on self-invoking code generation," 2024, *arXiv:2412.21199*.
[26] V. Csuvik and L. Vidács, "FixJS: A dataset of bug-fixing JavaScript commits," in *Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories (MSR)*, May 2022, pp. 712–716.
[27] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta programming for a multi-agent collaborative framework," in *Proc. Int. Conf. Learn. Represent.*, 2024.
[28] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "ChatDev: Communicative agents for software development," in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics (Volume 1: Long Papers)*, 2024, pp. 15174–15186.
[29] H. Li, H. Zhang, and A. E. Hassan, "The rise of AI teammates in software engineering (SE) 3.0: How autonomous coding agents are reshaping software engineering," 2025, *arXiv:2507.15003*.
[30] GitHut. (2024). *GitHut 2.0, A Small Place to Discover Languages in GitHub*. Accessed: Nov. 8, 2025. [Online]. Available: https://madnight.github.io/githut/#/pull_requests/2024/1
[31] GitHub. (2024). *Pull Request Naming Guide*. [Online]. Available: https://github.com/mozilla-mobile/firefox-ios/wiki/Pull-Request-Naming-Guide
[32] Greg Foster. (2024). *Best Practices for Naming Git Branches*. Accessed: Nov. 8, 2025. [Online]. Available: https://graphite.dev/guides/git-branch-naming-conventions
[33] PR4Code. (2025). *Open Science Framework (OSF)*. Accessed: Nov. 7, 2025. [Online]. Available: https://osf.io/v3tgw/overview?view_only=9c055172bb1f440f83646dbdec181b37
[34] C. Miranda, G. Avelino, and P. S. Neto, "Test co-evolution in software projects: A large-scale empirical study," *J. Softw., Evol. Process*, vol. 37, no. 7, 2025, Art. no. e70035.
[35] S. Levin and A. Yehudai, "The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 35–46.

BENEDETTA DONATO (Student Member, IEEE) received the master's degree in computer science from the University of Milano-Bicocca, in 2024, where she is currently pursuing the Ph.D. degree in computer science. Her master's thesis focused on the impact of generation parameters on the quality and correctness of code produced by LLMs. Her current research interests include AI-assisted software engineering, large language models for code generation, and empirical studies on developer–AI collaboration.

LEONARDO MARIANI (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of Milano-Bicocca, in 2005. He is currently a Full Professor with the University of Milano-Bicocca. His research interests include software engineering, in particular software testing, program analysis, automated debugging, AI4SE, and self-adaptive systems. He has authored more than 150 papers appeared at top software engineering conferences and journals. He received the ICST MIP Award for his ICST 2013 Paper. He was awarded the ERC Consolidator Grant, in 2015, and an ERC Proof of Concept Grant, in 2018. He is also active in several national and international projects. He is regularly involved in the organization of major software engineering conferences.

DANIELA MICUCCI (Member, IEEE) received the Ph.D. degree in mathematics, statistics, computational sciences, and computer science (informatics curriculum) from the University of Milan, in 2004. She is currently an Associate Professor with the Department of Informatics, Systems and Communication, University of Milano-Bicocca, where she leads the Software Architecture Laboratory (SAL). Her research interests include software architecture, software reliability, digital health systems, and the application of large language models to software engineering. She has participated and is also participating in several national and European research projects. She has authored numerous publications in leading international journals and conferences, e.g., IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, ICSE, ASE, and ISSRE. She serves as an editor and a reviewer for major venues in software engineering and intelligent environments.

OLIVIERO RIGANELLI received the Ph.D. degree in computer science and complex systems from the University of Camerino, Italy. He is currently an Associate Professor with the Department of Informatics, Systems and Communication, University of Milano-Bicocca, Italy. He has also conducted research with the University of Lugano, Switzerland, and Stony Brook University, USA. His research interests include the wide range of topics in software engineering, with a focus on methodologies and technologies to improve the correctness, reliability, and quality of software systems. In particular, his work covers software testing, program analysis, AI-assisted software development, cloud computing, and autonomous systems. He has participated in several national and international research projects, collaborating with academic institutions and industrial partners. He is actively engaged in the software engineering research community, serving as a reviewer and a program committee member for leading conferences and journals. His research contributions have been published in prominent international venues in software engineering.