Detecting, Ranking and Planning Issue-based Refactorings with a Multi-Agent Architecture

Henrique Peres
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, Brazil
hperes@inf.puc-rio.br

Alessandro Garcia
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Abstract

Repository issues often describe relevant maintenance tasks, which are available before the corresponding issue-resolution implementation commits are made. With design for change in mindset, structural refactorings should ideally be performed upfront based on issue descriptions before behavioral changes are implemented. For enabling issue-driven refactoring in this way, a system should detect whether an issue is demanding a refactoring, rank it against other issues, predict plausible refactoring types, while distinguishing the core files associated with the required refactoring changes. Existing studies do not assess whether the use of LLMs can contribute to support these refactoring-first activities. This paper presents RefactorFirstAgents, a LLM-based multi-agent architecture for these front-end activities. The architecture combines three agents: Issue-Selector, Refactoring-Predictor, and Planner. Issue-Selector assigns a distribution over four issue kinds and ranks these issues with a four-step priority order. Refactoring-Predictor predicts refactoring plans and abstains when the issue does not provide enough evidence for refactoring. Planner returns a bounded list of candidate files for the predicted refactorings. We evaluate the architecture with: (1) manual and LLM-based issue-kind inspection of the detection and ranking activities performed by the Issue-Selector, and (2) the use of RefactoringMiner 3.1.3 on five Java repositories to assess Refactoring-Predictor and Planner outputs. The latter is made in terms of the relationship between the issue-based recommended refactorings and refactorings actually performed by developers. In a 53-issue sample, the manual judge reports 87% overall directional accuracy, while ChatGPT 5.5 and Claude Code Opus 4.7 report 74% and 92%, respectively. In a larger 145-issue validation sample, the system reaches 72% overall directional accuracy. Pure refactoring issues reach 97%, while refactor+bug issues reached the lowest accuracy of 33%. For the Refactoring-Predictor, we compare the predicted refactoring types with refactorings detected by RefactoringMiner in commits linked to previously addressed issues. We report agreement for the primary prediction and for the bounded list of predicted plans. For the Planner, we analyze whether the returned candidate-file lists reduce the repository search space and whether the top-ranked files are specific to each refactoring-demanding issue. The results in general reinforce the promising ability of RefactorFirstAgents to support developers in performing major refactoring-first activities with high accuracy, with a few exceptions.

Keywords

Refactoring, Issue Ranking, Issue Classification, Issue-based Refactoring Prediction, Mining Software Repositories, Agent Systems

1 Introduction

Refactoring improves the internal structure of a software system without changing its observable behavior [10, 19]. To support a design-for-change mindset [10, 11], an automated pipeline must make front-end decisions before any refactoring-related code transformation starts. Various organizations, including Google [33] and Amazon [5], envisage maximizing the execution of refactorings in a single batch before critical behavioral changes. Beck [11] argues that structural refactoring tasks should be separated from behavioral changes so that refactorings can be performed first. If refactoring plans are prepared beforehand from issue descriptions, developers can better separate structural work from subsequent behavioral changes. We use *issue-driven refactoring* to denote the selection and planning of structural changes from a repository issue before the corresponding source-code modifications are performed.

However, this vision is difficult to realize for two reasons. First, repository issue descriptions often mix refactoring intent with feature requests, build changes, design discussions, tests, documentation changes, and bug reports [26, 32]. Second, many refactoring-demanding issues remain open for a long time [26], sometimes without a resolvable link to implementation commits. There is limited support for deriving, before those commits are available, a concrete plan containing plausible refactoring types and the files likely to be affected.

Prior work (e.g., [5, 33]) addresses only parts of this problem, but not a complete issue-driven front-end workflow. Studies on refactoring evidence tend to analyze commits, pull requests, or code reviews, where changes or patches are already available [1, 2, 8, 25]. Issue-type classification and issue-prioritization approaches rank or label issues for general maintenance goals, but they do not focus on detecting and prioritizing refactoring-demanding issues [7, 13, 15, 16]. Refactoring recommendation approaches usually start from code smells, code-level metrics, or other code-level evidence, assuming that these recommendations are useful from the perspective of developers who describe refactoring-relevant tasks as issues (e.g., [5, 6, 14, 17, 22, 33]). This leaves a gap before performing refactoring code transformations: deciding which issues should be treated as refactoring candidates, which refactoring types are plausible, and which core files should be inspected first.

This paper presents RefactorFirstAgents, an LLM-assisted multi-agent architecture for issue-driven refactoring planning before code transformation. The architecture has three agents. Issue-Selector detects and ranks refactoring-relevant issues. Refactoring-Predictor predicts plausible refactoring types from issue text and can abstain when the evidence is insufficient. Planner localizes candidate files to reduce the initial code search space. The goal is not to generate or assess patches, but

to evaluate whether these front-end decisions can prioritize refactoring-relevant issues, avoid unsupported refactoring plans, and reduce the amount of code that needs to be inspected. Given the automation of these activities, the use of RefactorFirstAgents would encourage more organizations to perform refactoring-first cycles in their project lifecycles.

This paper does not claim to fully describe each low-level refactoring transformation required to implement an issue. A recent LLM-based study [17] already shows evidence of using LLMs with high accuracy for this specific end. Our focus is the decision process that precedes transformation: selecting issues, estimating refactoring intent, and identifying candidate files. In industry-strength projects, such plans are more actionable when they are linked to problems already reported as issues by developers.

The architecture has four main design choices. First, Issue-Selector does not assign only one label to each issue. It computes a multi-label distribution over four issue kinds: `refactor`, `feature`, `other`, and `bug`. This labeling allows hybrid issues to be represented with multiple issue kinds, such as `refactor+feature` or `refactor+bug`, instead of forcing them into a single dominant category. It also captures how much refactoring evidence appears in comparison with feature, bug, or other evidence in the same issue. Second, Issue-Selector ranks issues with a four-step priority order based on bug-dominant exclusion, a 12-level combination ladder, dominance over the raw distribution, and a final reason-based score. Third, it uses boundary-aware text matching and a lexicon that separates strong and weak refactoring signals. Fourth, Refactoring-Predictor can abstain by returning `primaryPlan = null`, so issues with insufficient refactoring evidence or non-refactoring issues do not continue automatically to the file localization step.

The evaluation is organized around one research question for each front-end agent responsibility. RQ1 evaluates detection and ranking of refactoring-related issues. RQ2 evaluates whether predicted refactoring types agree with evidence from linked commits. RQ3 evaluates whether Planner reduces the search from the whole repository to a smaller set of issue-specific files. We consider a file issue-specific when it is mentioned in the issue text or is directly related to the predicted refactoring. The research questions are:

- **RQ1**: How effective is the proposed Issue-Selector at detecting and ranking refactoring-relevant issues?
- **RQ2**: How accurate is the Refactoring-Predictor when predicting the refactoring types and subcategories implied by an issue?
- **RQ3**: How well does Planner reduce the search space by localizing core candidate files for refactorings?

The evaluation has four parts. First, we assess issue-kind distributions with three outcomes: *Match*, *Borderline*, and *Wrong*. *Match* means that the predicted distribution agrees with the dominant issue kind or combination of issue kinds observed during validation. *Borderline* means that the prediction captures part of the issue intent, but misses or overemphasizes one relevant kind. *Wrong* means that the prediction misclassifies the main interpretation of the issue, such as treating a bug-dominant issue as a refactoring candidate. Second, we compare the consensus of two human validators with two LLM judges on the same sample of 53 issues. The two LLM judges are ChatGPT 5.5, accessed through ChatGPT [24], and Claude Code Opus 4.7, accessed through Claude Code [4]. We selected them because they represent two different validation settings: a general-purpose conversational LLM and a coding-oriented agentic LLM. Third, we compare Refactoring-Predictor outputs with RefactoringMiner 3.1.3 [31] detections on linked commits from five Java repositories: Checkstyle, JavaParser, Spring Framework, Lucene, and Mockito. Fourth, we analyze 172 issues processed by Planner to measure whether its candidate-file lists reduce the repository search space and avoid repeatedly selecting the same generic files. These sample sizes differ because each agent produces a different type of evidence: judge agreement is evaluated on 53 manually inspected issues, issue-selection behavior is evaluated on 145 ranked issues, and planner localization is evaluated on 172 issues for which Planner returned candidate-file lists.

The results indicate that the architecture is most reliable for clear refactoring cases and less reliable for hybrid refactor+bug issues. In the 53-issue sample, human consensus reports 87% directional accuracy, while ChatGPT 5.5 and Claude Code Opus 4.7 report 74% and 92%, respectively. In the 145-issue validation sample, the system reaches 72% directional accuracy under the Claude Code Opus 4.7 judge. Pure refactoring issues reach 97% directional accuracy, while the refactor+feature+bug and refactor+bug groups reach 20% and 33%, respectively. For Refactoring-Predictor, abstention is a precision-oriented behavior: when the issue text does not provide enough refactoring evidence, the agent avoids generating an unsupported plan instead of forcing a refactoring prediction. For Planner, the candidate-file lists are useful because they replace full-repository inspection with a limited set of files to inspect first. Thus, the localization strategy already provides practical search-space reduction, although repeated top-1 files show that issue-to-file evidence must still be strengthened.

This paper makes four contributions: (i) an LLM-assisted multi-agent pipeline that performs refactoring-demanding issue detection and ranking, refactoring prediction, and file localization before code transformations; (ii) a distribution-based policy for ranking refactoring-relevant and hybrid issues; (iii) an evaluation design that separates exact agreement, partial agreement, confirmed errors, and judge sensitivity; and (iv) an empirical evaluation across multiple repositories using manual validation, LLM-based validation, RefactoringMiner detections, and planner-output analysis. The results indicate that RefactorFirstAgents is effective for clear refactoring issues, but still needs stronger handling of hybrid refactor+bug issues and repeated generic files in planner outputs.

2 Related Work and Validation Support

This section positions RefactorFirstAgents against four lines of work: refactoring evidence in developer-written text, issue prioritization, refactoring prediction and recommendation, and validation support. These lines address parts of the problem, but they do not cover the full workflow studied in this paper: detecting refactoring-relevant issues, ranking them, predicting refactoring intent, and localizing candidate files before code transformations are made.

2.1 Mining Evidence in Developer-Written Text

Refactoring catalogs define structural transformations such as extract method, move method, rename method, and extract class [10, 19]. These catalogs describe code changes, but issue-driven automation must first infer refactoring intent from natural-language artifacts. This inference is difficult because issue text can describe refactoring, feature work, bug fixing, documentation, tests, build configuration, design discussion, or technical debt [34].

AlOmar et al. [1] study self-affirmed refactorings in commit messages. They show that developers often document refactoring activity with recurring terms such as restructuring, cleanup, rename, and move. This supports the use of textual evidence for detecting refactoring intent. However, their input is commit-message text after changes have already been made. RefactorFirstAgents uses issue descriptions before refactoring-related code transformations are performed, including cases where no commit is available yet.

AlOmar et al. [2] mine refactoring-related issues across Java projects and show that issue reports can contain useful refactoring intent. Their work characterizes how refactoring is documented during issue handling. Our work uses similar evidence operationally: Issue-Selector detects and ranks refactoring-relevant issues, and Refactoring-Predictor uses the issue text to estimate possible refactoring types. However, their study does not evaluate a workflow that combines issue detection, ranking, refactoring prediction, and file localization.

Paixão et al. [25] study refactoring in modern code review and analyze the intents behind refactoring changes. Their results show that refactoring activity can be connected to different development intents, including structural improvement, feature work, and bug fixing. This supports our decision to avoid a binary refactoring/non-refactoring label. In RefactorFirstAgents, hybrid issues are represented with multiple issue kinds, such as refactor and bug or refactor and feature. However, Paixão et al. analyze code reviews, where patches already exist. Our setting is earlier: the system must decide whether an issue should continue before code review or patch generation.

Coelho et al. [8] study refactoring-inducing pull requests and show that refactoring edits can appear after the initial commits, either due to review discussion or developer initiative. Their work connects pull requests, review discussion, and refactoring activity. RefactorFirstAgents starts from developers' reported issues in software repositories, where the available evidence is weaker and the system must decide whether to continue or abstain before a corresponding refactoring-related pull request exists.

2.2 Issue Prioritization and Issue Labels

Issue prioritization has been studied as a general maintenance problem. Izadi et al. [15] predict both the objective and priority of issue reports using textual, label, discussion, event, developer, and sentiment features. Their goal is to classify issue objectives and urgency. Our goal is narrower: to detect issues that are suitable for refactoring-oriented processing and rank them accordingly. Prioritization has also been studied in related software engineering tasks, such as requirements selection [28].

He et al. [13] study issue prioritization in GitHub and evaluate features and ranking strategies for deciding which issues should be handled earlier. Caddy and Treude [7] focus on priority labels and normalize GitHub priority labels into ranked categories. These works address prioritization in a project-management sense. RefactorFirstAgents ranks issues by refactoring suitability. It uses an issue-kind distribution over `refactor`, `feature`, `other`, and `bug`, and then applies an ordered policy over combinations of these kinds. For example, pure refactoring issues are ranked above mixed refactoring issues, and bug-dominant issues are moved to the end of the ranking.

Issue-type classification tools also support repository management. Ticket Tagger predicts GitHub issue types from issue titles and descriptions, and assigns labels such as bug report or feature request [16]. This is related to Issue-Selector, but RefactorFirstAgents does not reduce each issue to one label before ranking. Instead, it keeps a distribution over multiple issue kinds. This is important for mixed issues, such as refactoring requests that also mention feature, bug, documentation, or test context.

2.3 Predicting and Recommending Refactorings

Several approaches predict or recommend refactoring opportunities from code-level evidence. These approaches usually assume that the code artifact to be analyzed is already available. They often rely on code smells, metrics, change history, dependency information, or structural properties of classes and methods to decide where a refactoring should be applied. This line of work is related to Refactoring-Predictor, but they all start from source code analysis. In contrast, Refactoring-Predictor starts from issue text and predicts refactoring intent before detailed code analysis. We are going to discuss a representative set of these approaches here.

Oizumi et al. [22] recommend composite refactorings for smell removal, focusing on combinations of refactorings that can remove code smells. Bibiano et al. [6] study composite refactoring recommendations based on how these refactorings occur in practice. Both studies use code-level evidence: the refactoring opportunity is already represented as a smell or as a code-level quality problem. RefactorFirstAgents operates before this stage. It first identifies which issues are likely to demand refactoring and which files should be inspected. Other studies address refactoring safety, impact analysis, and developer comprehension of specific refactoring operations [9, 21].

Recent work also questions whether refactoring recommendation tools match the criteria used by practitioners. Ivers et al. [14] show a gap between industrial refactoring criteria and criteria commonly supported by refactoring recommendation tools. Their work reinforces the need to connect automated refactoring support with project context and developer-reported problems. RefactorFirstAgents contributes to this direction by using issue evidence to select refactoring candidates, predict refactoring intent, and localize candidate files likely to be impacted by the predicted refactorings.

LLM-based refactoring has also been evaluated for identifying refactoring opportunities, recommending refactoring solutions, and refactoring test smells [3]. Liu et al. [18] evaluate general-purpose LLMs for performing code refactoring transformations and report that narrowing the search space and making the expected refactoring subcategory explicit can improve LLM performance. This result

motivates the front-end architecture proposed in this paper. Issue-Selector, Refactoring-Predictor, and Planner are designed to produce the information that a refactoring model or developer would need before transformation: a prioritized issue, plausible refactoring types, and candidate files. However, Liu et al. start from code artifacts and evaluate LLMs as refactoring performers or recommenders. Our study evaluates the preceding issue-driven decisions: which issues should be considered, which refactoring types are plausible, and which files should be inspected first as part of refactoring-first planning.

2.4 Assessment Support

This study also relies on prior work to support the assessment of RefactorFirstAgents. First, RefactoringMiner is used as an automatic detector of Java refactorings in commits linked to closed issues [23, 30]. RefactoringMiner provides reproducible detections from version history and is useful for comparing predicted refactoring types with refactorings later performed by developers.

However, RefactoringMiner is not used as complete ground truth. It detects only the refactoring types covered by its catalog, and its output depends on the commit or commit range being analyzed. In addition, the refactorings detected in past commits may differ from the refactorings recommended by RefactorFirstAgents because implemented changes can mix refactoring with feature work, bug fixing, tests, or configuration changes. Therefore, the RefactoringMiner-based analysis is treated as an external agreement check, not as a definitive correctness oracle.

Second, the study uses LLM judges for part of the issue-kind validation. LLM-as-a-judge evaluation can scale qualitative assessment, but prior work shows that judge behavior is sensitive to model and prompt design [12, 35]. For this reason, our evaluation reports the judge and sample size for each accuracy value instead of presenting one judge as ground truth. In addition, we perform manual validation on a stratified sample of issues to provide a human reference point for the empirical results.

3 Architecture Scope

Figure 1 shows the architecture evaluated in this paper. The workflow has three agents: Issue-Selector, Refactoring-Predictor, and Planner. The study starts from GitHub issues and ends with a ranked list of candidate files. It does not evaluate patch generation or code transformation.

Figure 1 shows how raw GitHub issues are transformed into prioritized issues, predicted refactorings, and candidate files. In this study, an agent is a specialized component with a distinct responsibility, explicit input and output artifacts, and independently evaluable decision logic. The three agents respectively decide which issues are refactoring-relevant, which refactoring types are plausible, and which files should be inspected first.

The input is a list of GitHub issues collected from a target repository. Each issue is normalized before being processed. The normalized representation includes the title, body, labels, issue state, state reason, repository metadata, issue URL, creation date, update date, and comment count when these fields are available. The title and body provide the main textual evidence, while labels provide

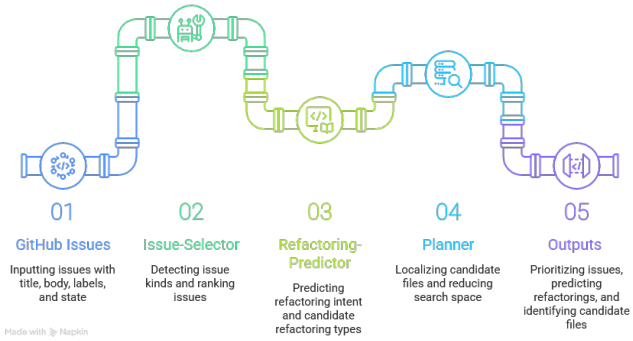

Figure 1: Evaluated issue-driven refactoring pipeline.

project-specific signals such as bug, enhancement, documentation, or refactoring.

Issue-Selector is the first agent. It receives the normalized issue list and classifies each issue using four issue kinds: `refactor`, `feature`, `other`, and `bug`. Instead of assigning only one label, it computes a distribution over these kinds. This allows the system to represent mixed issues, such as refactoring combined with feature or bug evidence. The agent then ranks the issues so that stronger refactoring candidates appear first. Its output is a ranked issue list with issue-kind distributions and ranking evidence.

Refactoring-Predictor is the second agent. It receives the ranked issues produced by Issue-Selector and decides whether each issue should continue in the workflow. The predictor can identify a refactoring candidate, a hybrid bug-fix-with-refactoring case, a non-refactoring task, or an uncertain case. When the evidence is not enough, it abstains and returns no primary plan. When the evidence supports refactoring, it returns a primary refactoring plan and, when applicable, alternative candidate plans.

Planner is the third agent. It runs only for issues for which Refactoring-Predictor does not abstain. It receives the issue information and the predicted refactoring intent, then localizes candidate files in the repository. Its output is a ranked list of candidate files and supporting context. This output reduces the search space from the full repository to a smaller set of files that can be inspected for the requested refactoring.

The implementation is written in TypeScript and executed with `ts-node` and `pnpm` [20]. The agents exchange auditable messages through Redis-backed queues [29]. In the evaluated configuration, Issue-Selector and Refactoring-Predictor fuse heuristic outputs with GPT-5.1 reviews constrained by JSON schemas, retaining the heuristic output on LLM failure. Planner is heuristic and does not call an LLM.

The rest of the paper details each part of the workflow. Section 4 describes issue-kind detection and ranking. Section 5 describes refactoring prediction and abstention. Section 5 describes file localization. Section 6 presents the assessment methodology used to evaluate each of these agents' outputs.

4 Issue Selection and Ranking

Issue-Selector is hybrid: heuristics produce a distribution over `refactor`, `feature`, `other`, and `bug`, and GPT-5.1 returns a second

distribution under a strict JSON schema. Both are fused before ranking; on LLM failure, the heuristic output is retained. Keeping a distribution represents issues that mix structural work with features, bugs, tests, or documentation.

The classifier uses textual and label-based evidence. Refactoring evidence includes terms such as *refactor*, *cleanup*, *extract*, *rename*, *move method*, *split class*, *remove duplication*, *long method*, and *technical debt*. Feature evidence includes more specific phrases such as *feature request*, *add support for*, *new check*, and *new rule*. Bug evidence includes terms such as *bug*, *fails with*, *false positive*, *false negative*, *runtime error*, and *stack trace*. Other evidence covers documentation, configuration, CI, build, website, onboarding, and tutorial work.

The text matching step avoids partial matches. A term is counted only when it appears as a separate textual unit, not as part of a dotted identifier, stack trace, file path, method name, or copied code fragment. This reduces false matches caused by issue bodies that contain exceptions, method names, file paths, or logs.

The lexicon also separates strong and weak signals. Strong refactoring signals include verbs such as *refactor*, *extract*, *rename*, *move*, *split*, *consolidate*, *inline*, *cleanup*, *convert*, *modernize*, *migrate*, and *remove*. Weak feature signals such as *add*, *improve*, *enable*, *support*, *allow*, and *enhance* require additional evidence in the issue body. This avoids classifying an issue as feature-related only because the title starts with a general verb. Documentation and meta-work are represented by terms such as *wiki page*, *tutorial page*, *how-to guide*, *onboarding documentation*, *minimal example*, and *coverage table*.

After computing the raw distribution, Issue-Selector applies an effective-kind floor of 10%. A kind whose share is below this floor does not define the issue group, although it still contributes to the ranking score. The issue group is the combination of issue kinds that remain after the floor. For example, an issue can be grouped as `refactor`, `refactor + feature`, `refactor + bug`, `feature`, `other`, or `bug`. This prevents a small incidental signal from changing the main group of the issue while preserving its effect as weak evidence.

Ranking uses four ordered criteria:

(1) **Bug-dominant exclusion**: issues with at least 75% bug share are moved to the end of the ranking. This threshold excludes cases where the issue is mostly a bug report, such as a 90% bug and 10% refactor distribution.
(2) **Issue-group priority**: effective kinds are mapped to a 12-level priority order. Pure refactoring has the highest priority. Mixed refactoring groups come next. Feature and other groups are placed below refactoring groups. Pure bug has the lowest priority.
(3) **Dominance score**: issues inside the same group are ordered with the raw distribution using $1.0R + 0.5F + 0O - 1.0B$, where R, F, O, and B are the refactor, feature, other, and bug shares. This score rewards refactoring evidence, gives smaller weight to feature evidence, ignores other evidence, and penalizes bug evidence.
(4) **Reason-based score**: remaining ties use a secondary score computed from the textual reasons collected for the issue. These reasons include maintainability terms, test-related terms, severity terms, issue specificity, issue age, and valid closed-issue status. This score does not move a non-refactoring group above a refactoring group; it only breaks ties after the main ranking criteria have been applied.

The 12-level priority order is: (1) only refactoring; (2) refactoring and feature; (3) refactoring, feature, and other; (4) refactoring, feature, and bug; (5) refactoring and other; (6) refactoring and bug; (7) feature; (8) feature and other; (9) feature and bug; (10) other; (11) other and bug; and (12) bug. The order is used only after applying the 10% floor.

The output of Issue-Selector is a ranked issue list. For each issue, the output stores the issue-kind distribution, dominant kind, effective kinds, issue group, dominance score, bug-dominant flag, ranking rationale, and the individual reasons that contributed to the ranking.

5 Refactoring Prediction and Planning

Refactoring-Predictor is hybrid: heuristic candidates are fused with GPT-5.1 recommendations constrained to the supported vocabulary, while heuristic candidates are preserved. Its output contains a primary plan when applicable and one of four kinds: refactoring candidate, hybrid bug fix with refactoring, non-refactoring task, or uncertain.

Prediction. The predictor builds an issue text from the title, body, and labels, then searches for refactoring expressions related to operations, names, smells, and quality attributes. Operation signals include expressions such as *extract*, *split*, *move*, *cleanup*, *simplify*, and *remove unused*. Naming signals include *rename*, *better name*, and *naming convention*. Smell signals include *duplicate code*, *long method*, *large class*, *dead code*, and *magic number*. Quality signals include *readability*, *maintainability*, *complexity*, *coupling*, *testability*, and *modularity*. Each matched signal is mapped to candidate refactoring types and subcategories. For example, duplicate-code evidence can suggest `extract_clone` or `extract_method`; long-method evidence can suggest `extract_method`; misplaced-behavior evidence can suggest `move_method`; type-checking evidence can suggest `replace_conditional_with_polymorphism`; and naming evidence can suggest rename-oriented candidates.

After extracting candidate refactoring types, the predictor assigns a final score to each candidate and orders the list. The score starts from the candidate's initial confidence and adds bounded support from the candidate source, textual refactoring evidence, quality-related terms, operation-related terms, and the current pipeline phase. Because detailed code analysis has not yet been used, the score is mainly based on issue text and labels. The highest-scored candidate can become the primary plan when the suitability decision allows the issue to continue. Other candidates are kept as alternative plans, up to the configured maximum number of plans.

The predictor can abstain. If the issue is not a refactoring task, or if the evidence is too weak, it returns `primaryPlan = null`. This prevents the workflow from forcing bug reports, feature requests, documentation changes, or build issues into refactoring plans. When the predictor does not abstain, it returns a list of candidate refactoring plans. In the evaluated configuration, the list is capped at six plans by `MAX_REFACTORING_PLANS`; it can contain fewer plans, and it is empty when the predictor abstains.

Planning. Planner is heuristic and does not invoke an LLM. It runs only when Refactoring-Predictor does not abstain. It receives the issue information and the predicted refactoring intent, then returns a ranked list of candidate files. This step reduces the repository search space from the full project to a smaller set of files that can be inspected for the requested refactoring. This paper evaluates file localization, but it does not evaluate generated code changes.

The planner uses three kinds of evidence: explicit textual evidence, such as file names, class names, package names, and backticked identifiers mentioned in the issue; repository evidence extracted from Java files, including paths, declared classes, packages, imports, test files, and file size; and refactoring-intent evidence from Refactoring-Predictor, such as whether the candidate plan suggests extraction, movement, renaming, duplication removal, or test-related work. Direct file-name or class-name matches receive stronger support than broad lexical overlap. The final output is a bounded ranked list of candidate files and supporting context.

The planner evaluation uses 172 issues from a nine-repository output set: Checkstyle, JavaParser, JGit, JUnit Framework, OkHttp, Spring Framework, Jetty, Lucene, and Mockito. This set is used for RQ3, which evaluates file localization. It is different from the five-repository set used in RQ2, where Refactoring-Predictor outputs are compared with RefactoringMiner detections. RQ3 therefore focuses on the size and specificity of the candidate-file lists, not on agreement with RefactoringMiner.

6 Assessment Methodology

The validation uses three outcomes: *Match*, *Borderline*, and *Wrong*. These outcomes are used to evaluate issue-kind distributions and the ranking behavior of Issue-Selector. **Match** requires three conditions. First, the dominant issue kind must agree with the judge's decision based on title, body, labels, and comments. Second, every kind with at least 10% share must have textual evidence. Third, no substantial issue kind can be absent. We treat a kind as substantial when the judge assigns it at least 30% of the issue-kind distribution.

Wrong is assigned when at least one of four conditions holds. The dominant issue kind differs from the judge's decision; a substantial kind is excluded by the floor; an extra kind is added without textual evidence; or a meta discussion, question, or planning issue is classified as implementation work. **Borderline** is used when the dominant issue kind is acceptable, but the exact distribution is debatable. This includes cases where a secondary kind is weakly supported, where the proportions differ from the judge's interpretation, or where the issue mixes more than one concern and more than one distribution can be justified. A case can be borderline only if it is not wrong. The primary metric is directional accuracy:

$$\textit{directional accuracy} = \frac{\textit{Match} + \textit{Borderline}}{\textit{Total}}. \tag{1}$$

Directional accuracy is used because borderline cases can still be useful inputs for the subsequent analysis, while wrong cases contaminate the pipeline. To avoid hiding this choice, the evaluation reports *Match*, *Borderline*, and *Wrong* separately. Reported results assess the integrated hybrid Issue-Selector and Refactoring-Predictor and heuristic Planner; the internal GPT-5.1 advisor is distinct from RQ1's LLM judges, and no ablation is reported.

For RQ2, we focus on closed issues because the evaluation requires implementation evidence. The predictor receives only the issue text, but its output is compared with refactorings detected by RefactoringMiner (Section 2.4) in commits that resolved or referenced the issue. Open issues, or closed issues without resolvable commits, do not provide the same external evidence. Therefore, closed issues are used to estimate the relation between predicted refactoring intent and refactorings later implemented by developers. Also, the evaluation separates internal classification assessment from external refactoring detection. The classification evaluation compares the system output with manual and LLM judges. The predictor validation compares Refactoring-Predictor outputs with RefactoringMiner detections in commits linked to closed issues. RefactoringMiner agreement is not treated as complete ground truth because it depends on commit resolution and on the detectors' catalog.

The manual validation for RQ1 uses a stratified sample of 53 issues selected across the issue-kind groups produced by the ranking policy. The goal is to inspect not only high-ranked pure refactoring issues, but also mixed refactoring issues, feature issues, other issues, and bug-dominant issues. The 145-issue validation sample complements this manual sample by reporting the same outcome categories by priority group. The 145-issue validation sample was judged with Claude Code Opus 4.7. The judge opened the GitHub URL for each issue, inspected the issue content, and applied the same three-outcome criteria. This procedure is reported because the result depends on the judge and on whether the issue URL was inspected directly.

For RQ2, we compare Refactoring-Predictor outputs with RefactoringMiner detections. This evaluation uses closed issues classified as refactoring-related by Issue-Selector in five Java repositories. For each issue, we try to resolve a closing commit by searching the repository history for references to the issue number. When a commit or commit range is found, RefactoringMiner 3.1.3 is executed on the corresponding range. The refactoring types detected by RefactoringMiner are then compared with the predictor output. We report two agreement measures for RQ2. *Top-1 agreement* means that the primary predicted refactoring type matches a type detected by RefactoringMiner. *Plan-list agreement* means that any refactoring type in the predictor's returned plan list matches a type detected by RefactoringMiner. The plan list is capped at six plans in the evaluated configuration, but it can contain fewer plans or be empty when the predictor abstains. Table 4 reports these measures together with the number of selected issues, issues without resolved commits, externally evaluable issues, abstentions, and agreement hits. This allows the analysis to distinguish low agreement caused by wrong predictions from low agreement caused by missing commit evidence or predictor abstention.

For abstention analysis, *no commit* means that no closing commit was resolved by the commit-search procedure. *Evaluable* means that the issue had commit evidence that could be compared with RefactoringMiner detections. *Abstained and no commit* is the intersection between abstained issues and issues without a resolved closing commit. Table 5 reports how abstentions are distributed

Table 1: Evaluation design.

RQ	Agent	Evidence	Main measure
RQ1	Issue-Selector	53 manually inspected issues and 145 issues judged by Claude Opus 4.7	Directional accuracy over Match, Borderline, and Wrong
RQ2	Refactoring-Predictor	Closed issues linked to commits in five Java repositories	Top-1 and plan-list agreement with RefactoringMiner
RQ3	Planner	172 planner outputs from the nine-repository artifact set	Search-space reduction and repeated top-1 analysis

between issues with and without external commit evidence. This distinction is necessary because an abstention in an issue without a resolved commit cannot be interpreted in the same way as an abstention in an externally evaluable issue.

7 Empirical Study

This section reports the results for the three evaluated agents and folds the main discussion points into the result narrative. RQ1 evaluates issue-kind classification and ranking by Issue-Selector. RQ2 evaluates Refactoring-Predictor outputs by comparing them with RefactoringMiner detections in commits linked to closed issues. RQ3 evaluates Planner localization results from the nine-repository output set. Table 1 summarizes the evaluation design. The three RQs use different evidence because they evaluate different outputs. RQ1 uses manual and LLM-based judgments over issue-kind distributions. RQ2 uses commit evidence and RefactoringMiner detections. RQ3 uses planner outputs over nine repositories. Therefore, the sample sizes differ across the evaluation: 53 issues for inter-judge validation, 145 issues for priority-group validation, and 172 issues for planner-output analysis. We do not use a single common sample because each agent produces a different output and requires different validation evidence.

The manual evaluation of Issue-Selector used a stratified sample across issue-kind priority groups instead of a purely random sample. The goal was to inspect not only high-ranked pure refactoring issues, but also mixed groups such as refactor+feature, refactor+bug, feature+bug, other, and bug-dominant issues. Larger groups were sampled with larger quotas, while smaller mixed groups were sampled with smaller quotas subject to availability. This strategy produced the 53-issue inter-judge sample in Table 2. The broader 145-issue validation in Table 3 reports the same outcome categories by priority group.

7.1 RQ1: Issue Selection Results

Table 2 reports the same 53 issues evaluated by three judges: one manual judge and two independent LLM judges. All judges used the same three-outcome criteria. The manual validation reaches 87% directional accuracy, with 21 Match cases, 25 Borderline cases, and 7 Wrong cases. This is the main human-assessed result for Issue-Selector and indicates that most inspected issues were classified and ranked in a useful direction, even when the exact distribution was debatable. The high number of Borderline cases also shows

Table 2: Inter-judge validation on 53 issues.

Judge	Match	Bord.	Wrong	Dir. acc.
Manual judge	21	25	7	87%
GPT-5.5	18	21	14	74%
Claude Opus 4.7	20	29	4	92%

that many issues are not purely refactoring, feature, bug, or other; they often combine more than one concern.

The two LLM judges produce different estimates on the same sample. GPT-5.5 reaches 74% directional accuracy, while Claude Opus 4.7 reaches 92%. The 18 percentage point difference shows that LLM-based validation is sensitive to the judge, even when the criteria and the evaluated issues are the same. The manual result is between the two LLM results. Therefore, the LLM judges are useful as independent checks, but they should not replace manual judgment or be treated as ground truth as we observed with the current versions of these LLMs. Table 3 reports the 145-issue validation sample by priority group. P1–P10 are the priority groups created by the combination ladder in Section 4; P1 is the highest-priority group. The excluded group contains bug-dominant issues moved to the end of the ranking.

The strongest result appears in P1, the pure refactoring group. This group contains 35 issues and reaches 97% directional accuracy, with 26 Match cases, 8 Borderline cases, and only 1 Wrong case. The excluded bug-dominant group also performs well, reaching 87% directional accuracy. These results show that the detection and ranking policy works best when the issue has clear refactoring evidence or clear bug evidence. The weakest groups are P4 and P6. P4, the refactor-feature-bug group, reaches 20% directional accuracy, while P6, the refactor-bug group, reaches 33%. The common pattern is that bug-related terms can be triggered by explanatory language in issues that describe cleanup, migration, movement, or simplification. The feature and other groups show mixed behavior: P8 reaches 91%, while P10 reaches 50%. Overall, the aggregate 72% directional accuracy hides important differences across groups, so group-level reporting is more informative than the aggregate alone.

The group-level view also clarifies how the ranking should be used in practice. P1 issues can be sent to the next agents with comparatively high confidence, whereas P4 and P6 issues should be treated as review-first cases. In these mixed groups, the ranking is still useful because it exposes a possible refactoring concern, but the downstream workflow should require an explicit confirmation step before prediction and planning. This avoids turning ordinary bug reports into refactoring tasks only because the issue body contains weak structural terms such as cleanup, simplify, warning, or broken. The distribution also identifies review-first cases: P4 and P6 should not proceed automatically to prediction and planning.

7.2 RQ2: Prediction and Abstention

Table 4 reports the RefactoringMiner-based validation of Refactoring-Predictor for RQ2. The validation uses five Java repositories. For each repository, we selected closed issues classified as refactoring-related by Issue-Selector, resolved linked commits when possible, and ran RefactoringMiner 3.1.3 on the

Table 3: Priority-group results for the 145-issue validation sample.

Group	Combination	Total	Match	Borderline	Wrong	Dir. acc.
P1	refactor	35	26 (74%)	8 (23%)	1 (3%)	97%
P2	refactor + feature	6	3 (50%)	0	3 (50%)	50%
P3	refactor + feature + other	0	–	–	–	–
P4	refactor + feature + bug	5	1 (20%)	0	4 (80%)	20%
P5	refactor + other	11	8 (73%)	1 (9%)	2 (18%)	82%
P6	refactor + bug	18	5 (28%)	1 (6%)	12 (67%)	33%
P7	feature	12	4 (33%)	4 (33%)	4 (33%)	67%
P8	feature + other	11	5 (45%)	5 (45%)	1 (9%)	91%
P9	feature + bug	12	6 (50%)	2 (17%)	4 (33%)	67%
P10	other	12	3 (25%)	3 (25%)	6 (50%)	50%
Excluded	bug-dominant	23	16 (70%)	4 (17%)	3 (13%)	87%
Total		145	77 (53%)	28 (19%)	40 (28%)	72%

resolved ranges. The table reports top-1 and plan-list agreement between Refactoring-Predictor and RefactoringMiner. The first relevant result is that the number of externally evaluable issues is much smaller than the number of selected issues. JavaParser has 13 selected issues, but no evaluable issue after commit resolution. Spring Framework has 7 selected issues and 3 evaluable issues. Lucene has 30 selected issues and 5 evaluable issues. Mockito has 11 selected issues and 3 evaluable issues. Checkstyle has the largest evaluable set, with 26 evaluable issues out of 53 selected issues. This shows that commit availability is a major constraint for external validation.

The agreement results should be interpreted at issue level rather than as stable repository-level estimates. Checkstyle reaches 7.7% top-1 agreement and 19.2% plan-list agreement. This means that two Checkstyle issues match the primary predicted type, and five issues match at least one type in the returned plan list. Mockito reaches 33.3% in both measures, but this corresponds to only one hit among three evaluable issues. Spring Framework reaches 66.7% plan-list agreement, but it also has only three evaluable issues. Lucene reaches 20% plan-list agreement, with one plan-list hit among five evaluable issues.

The difference between top-1 and plan-list agreement is important. Top-1 agreement evaluates only the primary predicted refactoring type. Plan-list agreement is less strict because it counts a hit when any predicted type in the bounded plan list matches a RefactoringMiner detection. The Checkstyle result illustrates this difference: plan-list agreement is higher than top-1 agreement, which suggests that the predictor sometimes includes the correct refactoring family as an alternative rather than as the primary plan. We report both measures because issue descriptions frequently express intent at a higher level than the RefactoringMiner catalog. For example, an issue can ask for a component to be simplified, decomposed, or made easier to maintain, while the eventual commit contains a mixture of rename, extract, move, and cleanup operations. In such cases, expecting the first predicted operation to match one detected operation exactly is a strict criterion. The bounded plan list is closer to how the output would be consumed by a developer: it presents a small set of plausible refactoring directions, and the next stage can inspect whether one of them is supported by code evidence. For this reason, RQ2 reports both measures instead of collapsing the result into a single accuracy value.

Table 4: Predictor agreement with RefactoringMiner.

Repository	Total	Eval.	Top-1	Plan-list
JavaParser	13	0	–	–
Spring fwk.	7	3	0%	66.7%
Lucene	30	5	0%	20%
Mockito	11	3	33.3%	33.3%
Checkstyle	53	26	7.7%	19.2%

Table 5 separates abstentions that occur in issues without a resolved closing commit from abstentions that occur in externally evaluable issues. The table is necessary because an abstention cannot be interpreted in the same way when there is no commit evidence available. In JavaParser, 10 of the 11 abstentions occur in issues with no resolved closing commit. In Spring Framework, 3 of the 4 abstentions occur in no-commit issues. In Lucene, 13 of the 17 abstentions occur in no-commit issues. Mockito also has 3 abstentions in no-commit issues. These cases cannot be treated as clear predictor errors because the external validation does not provide commit evidence that could confirm a missed refactoring.

Checkstyle is different. Among 35 abstentions, 10 overlap with no-commit issues, while 51.4% of abstentions occur in evaluable issues. These cases are the best candidates for future false-negative inspection. If RefactoringMiner detects refactorings in an evaluable issue where the predictor abstained, the abstention may indicate that the predictor missed refactoring evidence in the issue text. However, manual inspection is still needed because the detected refactoring may be unrelated to the issue. Therefore, the RQ2 numbers measure agreement with an automatic Java refactoring detector under commit-resolution limitations, not complete correctness. The abstention policy should also be read as a precision-oriented design choice. When the issue text contains only weak or generic maintenance evidence, forcing a refactoring type would contaminate the planner with unsupported localization requests. Abstention reduces that risk, but it introduces a recall trade-off: some true refactoring opportunities may stop too early. The most informative false-negative candidates are consequently the evaluable abstentions,

Table 5: Abstention and commit evidence.

Repository	Abst.	No commit	Abst.+no commit	Eval. abst.
JavaParser	11	11	10	0.0%
Spring fwk.	4	4	3	25.0%
Lucene	17	24	13	23.5%
Mockito	6	6	3	16.7%
Checkstyle	35	16	10	51.4%

Table 6: Dominant issue-kind distribution.

Repository	Total	Ref.	Bug	%Ref.
JavaParser	179	13	5	7.3%
Spring fwk.	176	7	16	4.0%
Lucene	890	30	396	3.4%
Mockito	120	11	11	9.2%
Checkstyle	399	53	60	13.3%

especially when a resolved commit contains a RefactoringMiner detection that is semantically close to the issue. These cases define a concrete inspection set for future prompt, lexicon, and threshold improvements.

7.3 RQ3: Planning and Localization

Table 6 reports the dominant issue-kind distribution for the five repositories used in the RefactoringMiner validation. The distribution varies substantially by project, which affects how many issues can reach the predictor evaluation. Checkstyle has the highest share of refactor-dominant issues among the five repositories, with 53 refactor-dominant issues out of 399 issues, or 13.3%. Mockito also has a relatively high refactor share, with 11 out of 120 issues, or 9.2%. JavaParser has 13 refactor-dominant issues out of 179, or 7.3%. Spring Framework and Lucene have lower refactor-dominant shares, with 4.0% and 3.4%, respectively.

The bug distribution also varies. Lucene has 396 bug-dominant issues out of 890, while Checkstyle has 60 bug-dominant issues out of 399. The smaller repositories show lower absolute bug counts, with 5 in JavaParser, 16 in Spring Framework, and 11 in Mockito. This variation affects pooled results because repositories contribute different proportions of refactoring, feature, other, and bug issues. The bug-dominant exclusion rule does not remove all bug-related issues, because mixed refactor-bug issues can remain in the ranked list when the bug share is not dominant. This explains why P4 and P6 still appear in Table 3 and why mixed bug/refactoring cases remain an important source of error.

The planner evaluation uses 172 Planner outputs from the nine-repository artifact set. These outputs come from nine open-source Java repositories: Checkstyle, JavaParser, JGit, JUnit Framework, OkHttp, Spring Framework, Jetty, Lucene, and Mockito. Each planner output contains a ranked list of candidate files. This means that the workflow starts from a bounded part of the repository instead of the full repository.

Table 7 reports the repetition of the same top-1 file across repositories. A repeated top-1 file means that the same file is ranked first for many issues in the same repository. In six of the nine repositories, the most frequent top-1 file appears in at least half of the issues. In four repositories, the same top-1 file appears in at least 80% of the issues. This result shows that the planner reduces the search space, but the reduction is not always issue-specific.

Table 7: Repeated top-1 planner files.

Condition	Repositories	Ratio
Top-1 repetition $\geq$ 50%	6/9	66.7%
Top-1 repetition $\geq$ 80%	4/9	44.4%

This repetition indicates that the planner is sometimes prioritizing repository-central files instead of issue-specific files. Therefore, the main RQ3 result is not only that the candidate lists reduce the number of inspected files, but also that this reduction can be too generic in repositories where the same top-ranked file appears for many unrelated issues. Thus, the planner reduces the number of files to inspect, but it does not yet provide reliable issue-specific localization.

A repeated top-1 file is not automatically an error. It can be reasonable when many issues genuinely refer to the same central file. However, repetition becomes a limitation when the top-ranked file is a broad repository file and the files explicitly mentioned in the issue are absent from the ranked list. A broad file is a file that is structurally central or frequently referenced by the repository, so it can receive high scores for many unrelated issues. In that situation, the planner reduces the search space, but the reduced space may not correspond to the files most related to the issue.

Representative examples show this pattern. Checkstyle repeatedly ranks `TokenTypes.java` first, and Mockito repeatedly ranks a broad test file first. Manual inspection shows that these files can differ from the classes named in the issue body. For example, Checkstyle issue #19206 names `BaseCellEditor.java` and `TreeTable.java`, but neither appears in its top-10. Therefore, RQ3 is mixed: the planner produces bounded candidate-file lists, but file localization still needs stronger issue-to-file evidence. In practice, the current planner output should be interpreted as triage support rather than as a complete localization oracle. Future planner versions should attach file-level evidence, such as explicit issue mentions, path matches, recent commits, tests, or detected smells. Repeated top-1 analysis is a simple way to expose when this explanation is missing: if many unrelated issues receive the same first file, the planner may be learning repository centrality more strongly than issue specificity. The next version should therefore combine textual mentions, path similarity, recent commits, and static-analysis evidence before assigning final file ranks.

8 Threats to Validity

Issue-kind ambiguity, judge bias, and metric definition. Some issues do not have a single clear kind. Examples include test migrations, cleanup tasks attached to bug fixes, and feature requests that also mention structural work. The *Borderline* category reduces the pressure to force these cases into a binary result, but another reviewer may still assign different shares to the same kinds. A related internal threat is judge bias. In the 53-issue sample, GPT-5.5

reports 74% directional accuracy and Claude Code Opus 4.7 reports 92%. This 18 percentage point spread shows that LLM judges can be more conservative or more permissive under the same criteria. We mitigate this threat by reporting the judge and the sample for each result and by keeping the manual 53-issue assessment as a human reference point. Directional accuracy combines *Match* and *Borderline*, which is less strict than exact match. We use it because borderline cases can still be useful for subsequent analysis, while wrong cases contaminate the pipeline; the tables therefore report *Match*, *Borderline*, and *Wrong* separately.

Repository scope and evaluable pools. The RefactoringMiner validation covers five Java repositories. These repositories have different dominant issue-kind distributions, and the results should not be generalized to other ecosystems without further evaluation. However, the five projects are diverse across domains and cover a wide range of refactoring-demanding issues, as presented before. Some repositories also have small evaluable sets in the predictor-versus-RefactoringMiner comparison. Spring Framework and Mockito have three evaluable issues each, while Lucene has five. These values should therefore be interpreted at issue level and not as precise repository-level estimates.

Construct validity. The evaluated outputs are front-end decisions rather than generated patches. This means that directional accuracy, plan-list agreement, and top-1 repetition do not measure the same property. Directional accuracy assesses whether the issue-kind distribution points the workflow toward the right decision; plan-list agreement assesses whether at least one predicted refactoring family appears in a bounded set; and top-1 repetition assesses whether file localization is issue-specific. These measures evaluate the front end only; they do not measure whether a generated refactoring patch would compile, pass tests, or be accepted by developers. A full end-to-end evaluation would also need patch generation, behavioral preservation, build results, test outcomes, and developer review. The integrated evaluation does not isolate LLM effects from heuristics.

External oracle, commit resolution, and planner evidence. RefactoringMiner is an automatic detector, not complete ground truth. It detects Java refactorings covered by its catalog, does not cover YAML, XML, Groovy, documentation, or configuration changes, and can miss refactorings outside its catalog. When run over a wide range, it can also include refactorings from commits unrelated to the analyzed issue. Commit resolution through `git log –grep` captures commits that mention the issue number, but it misses issues closed through pull requests or messages that use other conventions. This limitation explains part of the no-commit count and directly affects the interpretation of abstention. Finally, the planner analysis supports a limited claim: candidate-file lists reduce the search space, but repeated top-1 files show that the reduction is not always issue-specific, and the outputs do not expose enough validation evidence to assess test-based gating in detail. We therefore report abstentions, no-commit cases, and repeated planner files separately.

9 Conclusion and Future Work

This paper evaluated the front end of RefactorFirstAgents for issue-driven automated refactoring: the hybrid Issue-Selector and Refactoring-Predictor and the heuristic Planner. The system performs best when issues provide clear evidence, with P1 reaching 97% directional accuracy and bug-dominant exclusion reaching 87% in the 145-issue validation sample. The weakest results occur for mixed bug-related groups: P4 (refactor+feature+bug) reaches 20%, while P6 (refactor+bug) reaches 33%. The RefactoringMiner validation provides an external view of the predictor, but it is constrained by unresolved closing commits, non-Java changes, small evaluable pools, and wide commit ranges. The planner reduces the candidate-file search space, although repeated top-1 files indicate that localization is not always issue-specific.

The results support keeping the three agents separated as a staged and inspectable decision pipeline. Issue-kind distributions guide prediction, bounded prediction lists guide code inspection, and candidate-file lists reduce repository scope while still requiring validation before transformation. Keeping these artifacts separate also helps identify whether failures originate during issue selection, refactoring prediction, or file planning.

Future research should improve bug-signal handling in P4 and P6, add missing bug phrases that affect P10, revise the threshold for adding a third effective kind, and repeat multi-judge validation on more repositories. We also plan to improve commit resolution through the GitHub timeline API, run RefactoringMiner per commit, and strengthen planner evidence with static-analysis results, test-impact information, dependency relations, and files modified by recent issue-resolution commits. Finally, future experiments will compare hybrid and heuristic-only configurations and extend the architecture toward patch generation, review, execution, and behavioral validation.

Artifact Availability

To support reproducibility, the artifacts utilized and generated in this paper are publicly available in the companion replication package at https://doi.org/10.5281/zenodo.20031732 [27]. The package contains the evaluation datasets, the source code of the three-agent, intermediate outputs, and the scripts used to produce the reported results. It also documents the evaluated hybrid configuration, in which Issue-Selector and Refactoring-Predictor combine heuristic mechanisms with LLM-assisted reasoning, while Planner relies on heuristic and repository-based evidence without invoking an LLM.

The updated package additionally includes a complementary manual inspection of the same 145 Checkstyle issues considered in RQ1. These results are provided as additional evidence of judge sensitivity and as material for future analyses. They do not replace the Claude Code Opus 4.7 assessment or modify the tables and empirical results reported in this paper.

Acknowledgments

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence), https://www.ines.org.br, CNPq grant 408817/2024-0. Authors also are also supported by FAPERJ (200.510/2023, 211.033/2019, 202.621/2019), CNPq (310391/2025-3); and CAPES/Proex (88887.373933/2019-00).

References

[1] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Can Refactoring be Self-Affirmed? An Exploratory Study on How Developers Document Their Refactoring Activities in Commit Messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. 51–58. doi:10.1109/IWoR.2019.00017

[2] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D. Newman, and Ali Ouni. 2022. An Exploratory Study on Refactoring Documentation in Issues Handling. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. 107–111. doi:10.1145/3524842.3528525

[3] Gabriel Amaral, Henrique Gomes Nunes, Eduardo Figueiredo, Carla I. M. Bezerra, and Larissa Rocha. 2025. Improving JavaScript Test Quality with Large Language Models: Lessons from Test Smell Refactoring. In *Proceedings of the Brazilian Symposium on Software Engineering (SBES 2025)*. 776–782. doi:10.5753/sbes.2025.11568

[4] Anthropic. 2026. Claude Code Overview. https://code.claude.com/docs/en/overview. Accessed: 2026-05-04.

[5] Amazon (AWS). 2026. Preserve Deployed Resources when Refactoring CDK Code. https://docs.aws.amazon.com/cdk/v2/guide/refactor.html

[6] Ana Carla Bibiano, Daniel Coutinho, Anderson Uchôa, Wesley K. G. Assunção, Alessandro Garcia, Rafael Maiani de Mello, Thelma Elita Colanzi, Daniel Tenório Martins de Oliveira, Audrey Vasconcelos, Baldoino Fonseca, and Márcio Ribeiro. 2024. Enhancing Recommendations of Composite Refactorings Based on the Practice. In *Proceedings of the 24th IEEE International Conference on Source Code Analysis and Manipulation*. 83–93. doi:10.1109/SCAM63643.2024.00018

[7] James Caddy and Christoph Treude. 2024. Prioritising GitHub Priority Labels. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. 1–4. doi:10.1145/3663533.3664041

[8] Flávia Coelho, Nikolaos Tsantalis, Tiago Massoni, and Everton L. G. Alves. 2021. An Empirical Study on Refactoring-Inducing Pull Requests. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 9:1–9:12. doi:10.1145/3475716.3475785

[9] José Aldo Silva da Costa, Rohit Gheyi, José Júnior Silva da Costa, Márcio Ribeiro, Rodrigo Bonifácio, Hyggo Oliveira de Almeida, Ana Carla Bibiano, and Alessandro Garcia. 2026. Refactoring for Novices in Java: An Eye Tracking Study on the Extract vs. Inline Methods. *Journal of Systems and Software* 237 (2026), 112825. doi:10.1016/j.jss.2026.112825

[10] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[11] Martin Fowler. 2023. *Tidy First?: A Personal Exercise in Software Design (O'Reilly Media, 2023).* Addison-Wesley.

[12] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. 2024. A Survey on LLM-as-a-Judge. *arXiv preprint arXiv:2411.15594* (2024).

[13] Yingying He, Wenhua Yang, Minxue Pan, Yasir Hussain, and Yu Zhou. 2023. Understanding and Enhancing Issue Prioritization in GitHub. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 813–824. doi:10.1109/ASE56229.2023.00044

[14] James Ivers, Anwar Ghammam, Khouloud Gaaloul, Ipek Ozkaya, Marouane Kessentini, and Wajdi Aljedaani. 2024. Mind the Gap: The Disconnect Between Refactoring Criteria Used in Industry and Refactoring Recommendation Tools. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 138–150. doi:10.1109/ICSME58944.2024.00023

[15] Maliheh Izadi, Kiana Akbari, and Abbas Heydarnoori. 2022. Predicting the Objective and Priority of Issue Reports in Software Repositories. *Empirical Software Engineering* 27, 2 (2022). doi:10.1007/s10664-021-10085-3

[16] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. 2021. Predicting Issue Types on GitHub. *Science of Computer Programming* 205 (2021), 102598. doi:10.1016/j.scico.2020.102598

[17] Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2025. Exploring the potential of general purpose LLMs in automated software refactoring: an empirical study. *Autom. Softw. Eng.* 32, 1 (2025), 26. doi:10.1007/S10515-025-00500-0

[18] Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2025. Exploring the Potential of General Purpose LLMs in Automated Software Refactoring: An Empirical Study. *Automated Software Engineering* 32, 26 (2025). doi:10.1007/s10515-025-00500-0

[19] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139. doi:10.1109/TSE.2004.1265817

[20] Microsoft. 2026. The TypeScript Handbook. https://www.typescriptlang.org/docs/handbook/.

[21] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. 2014. Making Refactoring Safer Through Impact Analysis. *Science of Computer Programming* 93 (2014), 39–64. doi:10.1016/j.scico.2013.07.016

[22] Willian Nalepa Oizumi, Ana C. Bibiano, Diego Cedrim, Anderson Oliveira, Leonardo da Silva Sousa, Alessandro F. Garcia, and Daniel Oliveira. 2020. Recommending Composite Refactorings for Smell Removal: Heuristics and Evaluation. In *Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES)*. 72–81. doi:10.1145/3422392.3422423

[23] Jonhnanthan Oliveira, Rohit Gheyi, Leopoldo Teixeira, Márcio Ribeiro, Osmar Leandro, and Baldoino Fonseca. 2023. Towards a Better Understanding of the Mechanics of Refactoring Detection Tools. *Information and Software Technology* 162 (2023), 107273. doi:10.1016/j.infsof.2023.107273

[24] OpenAI. 2026. ChatGPT. https://openai.com/chatgpt/. Accessed: 2026-05-04.

[25] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 125–136. doi:10.1145/3379597.3387475

[26] Matheus Paixão, Anderson G. Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. ACM, 125–136. doi:10.1145/3379597.3387475

[27] Henrique Peres and Alessandro Garcia. 2026. RefactorFirstAgents — Replication Package for "Detecting, Ranking, and Planning Issue-Driven Refactorings". doi:10.5281/zenodo.20031732

[28] Antônio Mauricio Pitangueira, Rita Suzana Pitangueira Maciel, and Márcio de Oliveira Barros. 2015. Software Requirements Selection and Prioritization Using SBSE Approaches: A Systematic Review and Mapping of the Literature. *Journal of Systems and Software* 103 (2015), 267–280. doi:10.1016/j.jss.2014.09.038

[29] Redis. 2026. Redis Documentation. https://redis.io/docs/.

[30] Nikolaos Tsantalis, Akash Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. doi:10.1109/TSE.2020.3007722

[31] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 483–494. doi:10.1145/3180155.3180206

[32] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley K. G. Assunção, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. 2021. Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 471–482. doi:10.1109/MSR52588.2021.00059

[33] Hyrum K. Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan. 2013. Large-Scale Automated Refactoring Using ClangMR. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 548–551. doi:10.1109/ICSM.2013.93

[34] Laerte Xavier, João Eduardo Montandon, Fabio Ferreira, Rodrigo Brito, and Marco Túlio Valente. 2022. On the Documentation of Self-Admitted Technical Debt in Issues. *Empirical Software Engineering* 27, 7 (2022), 163. doi:10.1007/s10664-022-10203-9

[35] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, Vol. 36. 46595–46623.