Agentless-Java: Adapting Agentless Paradigm to Java Program Repair

Tianyi Huang, Wenqi Liao, Yiwei Wang, Yuyang Wang
University of Illinois Urbana-Champaign

ABSTRACT

Despite the growing success of Large Language Models (LLMs) in automated program repair, Java—one of the most widely used enterprise programming languages—remains underserved, with most solutions designed primarily for Python. We present Agentless-Java, an adaptation of the agentless paradigm for Java program repair that addresses Java's unique object-oriented paradigm and language-specific complexities. This midterm report details our progress on the initial fault localization phases of our pipeline Our approach implements a streamlined three-phase fault localization methodology: (1) identification of suspicious files through LLM-based and embedding-based retrieval, (2) localization to related elements using AST-based skeleton extraction, and (3) identification of precise edit locations. Evaluation on the swe-bench-java benchmark shows promising results, with over 95% touch accuracy for suspicious file identification. This report outlines our progress to date and future plans for completing the full repair pipeline. Our GitHub repository containing the implementation can be found at: https://github.com/upb3y/Agentless/tree/main

1 PROBLEM

Despite significant advances in automated program repair using Large Language Models (LLMs), there exists a considerable gap in effective repair solutions for Java applications. Current research and tooling predominantly focus on Python ecosystems, creating a significant bias that has left Java—one of the most widely used enterprise programming languages—underserved in the automated repair domain.

Agentless-Java addresses these critical limitations by applying an agentless paradigm to Java program repair. The challenges we face are twofold:

First, the research community's Python-first approach has resulted in repair techniques that fail to account for Java's unique characteristics. This bias has created a situation where Java repair solutions are often adaptations of Python-oriented methods rather than purpose-built approaches.

Second, Java's inherent complexities present substantial technical barriers to effective repair. These include its rigid object-oriented design, strict type system, compilation requirements, complex dependency management, extensive standard library, and platform compatibility considerations—all of which demand high precision in patch generation that existing tools struggle to deliver.

In this midterm report, we focus on our progress implementing the fault localization components of our pipeline, which consists of:

- Localization to Suspicious Files: Identifying the Java source files most likely containing the bug
- Localization to Related Elements: Narrowing focus to specific methods, classes, and dependencies
- Localization to Edit Locations: Pinpointing the exact code segments requiring modification

By eliminating unnecessary agent layers and focusing directly on core bug resolution processes, Agentless-Java aims to overcome the limitations of current approaches while respecting Java's unique characteristics.

2 SOLUTION

In this project, we aim to adapt the AGENTLESS pipeline [4]—originally designed for Python—to work effectively with Java codebases. Our goal is to build an end-to-end automated bug fixing system that can take an issue description and a Java project as input, and output a ranked list of suspicious files, along with potential patches. While retaining the overall structure of the Agentless workflow, we make several key modifications to accommodate Java's unique characteristics.

Our implementation follows a five-step approach:

2.1 Repository Structure Analysis

As the first step of our pipeline, we focus on hierarchical bug localization. Specifically, we take the GitHub issue description and the corresponding codebase as input, and begin by constructing a tree-like representation of the project structure. Since Java repositories vary significantly in layout and organization, it is difficult to extract a consistent "core" portion of the codebase. To generalize across projects, we adopt a lightweight heuristic: we retain all .java source files and the root-level README.md, while excluding other irrelevant files such as configuration scripts, build files, documentation, and .git metadata.

2.2 LLM-Based File Localization

Once the repository structure is extracted, we combine it with the issue description and apply a hierarchical prompting strategy. To avoid exceeding the token limits of the LLM, we split the full structure into smaller chunks (each with approximately 300–500 file paths). For each chunk, we prompt the LLM to return a list of suspicious files potentially relevant to the issue. After collecting results across all chunks, we aggregate and deduplicate the returned paths. We standardize the output by retrieving the top 50 suspicious files for each issue instance, ensuring consistency in downstream processing and facilitating comparison across different pipeline stages to identify high-confidence candidates.

2.3 Embedding-Based Retrieval

The third component employs an embedding-based retrieval approach leveraging semantic representation of source code. We selected Gemini text embedding "text-embedding-004" after comparing its performance with local embedding models from Sentence-BERT, specifically the "all-MiniLM-L6-v2" model[1]. This selection

was motivated by performance benefits, though we considered local embedding models for their ability to operate independently from external APIs, thereby circumventing rate limits associated with cloud-based services. Each repository's Java source files are structurally parsed into granular semantic units—classes, interfaces, and methods—using Tree-Sitter after comparing its performance with Javalang[3]. These units are converted into vector embeddings representing their semantic content, generated in batches to optimize computational throughput. To determine relevance, embeddings of issue descriptions are compared against code unit embeddings using cosine similarity, with the maximum similarity score across all code units within a file serving as its aggregate relevance score. Additionally, caching mechanisms based on MD5 hashing were incorporated to eliminate redundant computations. This step generates a list of the top 50 suspicious files, which are then intersected with suspicious files from the previous step to produce a final consolidated list of candidates for further analysis.

2.4 Element-Level Localization

To address Java's object-oriented nature, we implemented a two-phase localization approach. The primary phase uses AST-based skeleton extraction through a custom Java Abstract Syntax Tree parser utilizing the javalang library[2]. A secondary regex-based parsing mechanism serves as a fallback to handle potential AST parsing failures in complex or incomplete Java files.

To handle large repositories, we implemented batch processing that groups files into manageable chunks (default: 15 files per batch), preventing token limit overflows while ensuring comprehensive analysis. For each Java file, we generate detailed skeleton representations capturing package declarations, import statements, class declarations with inheritance relationships, field declarations, method signatures, and JavaDoc comments. These skeletons are deliberately formatted with visual structuring to enhance LLM comprehension, highlighting the hierarchical nature of Java code.

We construct specialized prompts that include the issue description, batch metadata, and structured skeleton representations, designed to elicit structured JSON responses containing file paths, class names, method names, confidence scores, and rationales. The responses undergo careful processing and filtering, retaining only elements with confidence scores of 3 or higher on a 5-point scale, improving precision while maintaining acceptable recall rates.

2.5 Line-Level Localization

In the final localization step, we implement fine-grained fault localization to identify the exact lines requiring modification within previously identified suspicious methods. This process transforms method-level suspicion into precise edit locations through several key steps: First, we parse Git patch diffs by extracting hunks that modify the target file and method. Using lightweight heuristics, we identify candidate line numbers from the patch—focusing specifically on newly added lines (those starting with "+") within the relevant hunks.

Secondly, we enrich these code snippets with line number annotations and submit them to the LLM along with a natural language explanation of the suspected issue. The prompt explicitly asks the LLM to identify the exact line numbers where errors are most likely located. To improve prediction stability, we employ multiple LLM invocations (typically 5) and aggregate the results using a frequency counter to identify the most commonly suggested lines.

Finally, we combine the LLM predictions with our heuristic-based candidates, ensuring suggested fixes align with actual modified code regions. The output is structured as a JSON-compatible dictionary containing precise line numbers requiring modification, the actual content of these lines, and both raw LLM and heuristic-based predictions for transparency. This detailed line-level localization serves as the foundation for subsequent repair generation.

3 EVALUATION SETUP

3.1 Dataset and Ground Truth

To assess the effectiveness of our suspicious file localization pipeline, we conducted comprehensive evaluations on the SWE-Bench-Java benchmark [5], which contains 91 real-world issue instances across diverse Java projects. Each instance is paired with its corresponding ground truth fix files derived from developer-submitted patches. We constructed the ground truth by extracting modified file paths and line numbers from these patches. Due to the structural complexity of some patches, we were only able to extract approximately 10

3.2 Evaluation Scope

Our evaluation is structured hierarchically, addressing multiple levels of localization granularity:

3.2.1 File-Level Evaluation. For file-level localization, we evaluate the combined output of three critical pipeline stages:

- **Project Structure Extraction**: The hierarchical representation of the repository's Java source files
- **LLM-Based File Localization**: The suspicious files identified through LLM reasoning over project structure
- **Embedding-Based Retrieval**: The ranked list of files based on semantic similarity to issue descriptions

These three components collectively produce a consolidated set of suspicious files that serve as candidates for subsequent fine-grained bug localization and repair.

3.2.2 Element and Line-Level Evaluation. For the more granular element-level (classes and methods) and line-level localization stages, we conduct separate evaluations as described in the subsequent sections. These evaluations focus on the precision of identifying specific code elements and line numbers requiring modification.

3.3 Evaluation Metrics

We adopt multiple complementary metrics to provide a comprehensive assessment of our pipeline's effectiveness:

- **Superset Accuracy**: The percentage of instances where all ground truth fix files are covered by the predicted suspicious file set. This metric measures the pipeline's ability to capture the complete set of relevant files.
- **Binary Touch Accuracy**: The percentage of instances where at least one ground truth fix file is included in the predicted list. This represents the pipeline's basic effectiveness in identifying relevant files.

- **Touch Accuracy**: The percentage of ground truth fix files that are successfully included in the predicted suspicious file list. This provides a more granular view of the pipeline's recall capabilities.

These metrics are specifically designed to reflect whether our pipeline successfully includes relevant files at this early localization stage, which is critical for downstream fine-grained bug fixing. The emphasis on recall-oriented metrics aligns with our goal of ensuring that all potentially relevant files are captured before proceeding to more computationally intensive analysis steps.

3.4 Experimental Configuration

For the full file-level localization evaluation, we utilized all 91 instances from the SWE-Bench-Java benchmark. However, for more granular localization at the element and line level, we excluded the "fasterxml/jackson-databind" repository due to high API costs. The element and line-level evaluations were therefore conducted on the remaining 42 instances.

In the final evaluation stage, we assessed the end-to-end effectiveness of our approach by running the reproduction tests for each issue and verifying whether the generated patches successfully resolved the identified bugs. This verification process provides the ultimate measure of our system's practical utility in real-world bug fixing scenarios.

4 RESULT ANALYSIS

4.1 File-Level Localization

After merging the LLM and embedding-based suspicious files, we observe the following performance across all 91 issue instances from the SWE-bench-Java benchmark:

- **Superset Accuracy**: 0.4176 (41.76%) → Our final suspicious file set fully contains all fix locations in 38 out of 91 instances.
- **Binary Touch Accuracy**: 0.9560 (95.60%) → At least one correct fix file is captured in over 95% of the cases.

These results demonstrate that our hybrid approach combining hierarchical prompting and embedding-based retrieval is highly effective at identifying relevant fix locations. The exceptionally high binary touch accuracy (95.60%) is particularly notable, as it indicates our pipeline rarely misses all relevant files. This is crucial for downstream tasks, as missing the relevant files entirely would make subsequent repair steps impossible. The superset accuracy of 41.76%, while lower, is still impressive considering the complexity and diversity of Java repositories in the benchmark. This metric represents cases where our approach identified *all* relevant files, which is a more stringent criterion.

The performance gap between binary touch and superset accuracy suggests that for many instances, our approach successfully identifies some but not all of the files requiring modification. This pattern aligns with the typical characteristics of complex bug fixes that span multiple files with varying degrees of semantic connection to the issue description.

4.2 Element-Level Localization

Building upon file-level localization, we further analyze performance at the element level (classes and methods):

- **Superset Accuracy**: 0.3800 (38.00%) → Our element-level localization fully contains all fix locations for 38% of the evaluated instances.
- **Binary Touch Accuracy**: 0.9000 (90.00%) → At least one correct fix element is captured in 90% of the cases.
- **Touch Accuracy**: 0.5800 (58.00%) → On average, 58% of the ground truth elements are covered in our predictions.

The element-level results follow a similar pattern to file-level localization but with slightly reduced performance, which is expected given the increased granularity. However, when examining specific element types more closely, we observe significantly lower performance metrics:

- **Class-Level**: Touch accuracy of 0.06 (6%) and binary touch accuracy of 0.10 (10%)
- **Method-Level**: Touch accuracy of 0.05 (5%) and binary touch accuracy of 0.10 (10%)

This performance drop at finer granularity levels stems from several challenges. First, our ability to extract complete ground truth data from patches was limited, with only approximately 10% of class and method information successfully recovered from patches. Second, the semantic gap between natural language issue descriptions and code elements increases at finer granularity levels. Third, Java's object-oriented nature introduces additional complexity when localizing specific elements within class hierarchies.

4.3 Line-Level Localization

At the finest granularity, line-level localization presents the most challenging aspect of the task:

- **Superset Accuracy**: 0.05 (5%) → Our approach fully captures all relevant lines in 5% of instances.
- **Binary Touch Accuracy**: 0.1 (10%) → At least one correct line is identified in approximately 10% of cases.
- **Touch Accuracy**: 0.07 (7%) → On average, 7% of the ground truth lines are covered in our predictions.

The substantial decrease in superset accuracy (5%) compared to file and element levels illustrates the inherent difficulty of pinpointing exact lines requiring modification. This challenge arises from several factors. Determining precise line-level modifications often requires deep understanding of control and data flow dependencies within the code. Additionally, multiple syntactically different lines could represent semantically equivalent fixes, creating ambiguity in what constitutes a "correct" line prediction. Furthermore, many patches involve complex transformations that add, remove, or modify multiple lines with intricate interdependencies, making it difficult to isolate individual lines without considering their relationships.

4.4 Overall Assessment

Our hierarchical bug localization pipeline demonstrates a clear performance gradient across granularity levels: At the file level, we achieve a superset accuracy of 41.76% and a binary touch accuracy of 95.60%. Moving to the element level, performance slightly decreases to 38.00% superset accuracy and 90.00% binary touch accuracy, with a touch accuracy of 58.00%. The class-level and method-level accuracy is around 10%. At the finest granularity (line

level), we observe the most significant drop, with only 5% superset accuracy, 10% binary touch accuracy, and 7% touch accuracy. This performance gradient aligns with our expectations given the increasing complexity at finer granularity levels. The consistently high binary touch accuracy across all levels demonstrates that our approach reliably identifies at least some relevant code components, which is critical for effective automated program repair. The gradual performance decrease from file to line level suggests that future work should focus on enhancing the precision of element and line-level localization. However, the current performance already provides a solid foundation for downstream repair tasks, as the identified candidates significantly reduce the search space for potential fixes. Notably, our approach achieves these results without requiring expensive test case executions or dynamic analysis, relying instead on efficient static analysis and LLM reasoning. This makes our approach particularly valuable for practical deployment scenarios where computational efficiency is important.

5 FUTURE PLAN

The following research phases outline subsequent steps for completion of the project:

5.1 Refine Ground Truth Extraction

Including finer-grained element localization in the ground truth would be a valuable next step for our research. While file-level identification has shown promising results with a more than 90% touch rate, the significant drop in performance at the method and class levels highlights a critical area for improvement. Future step should focus on both improving the construction of a complete ground truth file by localizing the methods and classes based on the line number and enhancing the element-level localization.

5.2 Patch Repair

A targeted period of approximately 1 to 2 weeks is allocated to develop an automated mechanism for generating proposed code patches based on the identified suspicious files and reported software issues. This phase involves algorithmic development, testing, and preliminary validation.

5.3 Patch Validation

Subsequent to patch generation, a comprehensive validation phase spanning approximately 2 to 3 weeks is planned. This will involve rigorous empirical evaluations of generated patches using standard testing frameworks and validation metrics to assess correctness, efficacy, and robustness.

Concurrently, efforts will be dedicated to the completion and submission of the final research paper, ensuring detailed documentation of methodologies, results, discussions, and conclusions in a structured and scientifically rigorous manner.

REFERENCES

[1] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. https://sbert.net/. Accessed: April 2025.
[2] Chris Thunes. 2024. Javalang: Pure Python Java parser and tools. https://github.com/c2nes/javalang. Accessed: April 2025.
[3] Tree-sitter. 2024. Tree-sitter Python bindings. https://github.com/tree-sitter/py-tree-sitter. Accessed: April 2025.
[4] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. arXiv:2407.01489 [cs.SE] https://arxiv.org/abs/2407.01489
[5] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, Dezhi Ran, Muhan Zeng, Bo Shen, Pan Bian, Guangtai Liang, Bei Guan, Pengjie Huang, Tao Xie, Yongji Wang, and Qianxiang Wang. 2024. SWE-bench-java: A GitHub Issue Resolving Benchmark for Java. arXiv:2408.14354 [cs.SE] https://arxiv.org/abs/2408.14354