B.Comp. Dissertation

Autonomous Program Improvement at the IDE

By

Fu Zanwen

Department of Computer Science

School of Computing

National University of Singapore

2024/2025

B.Comp. Dissertation

Autonomous Program Improvement at the IDE

By

Fu Zanwen

Department of Computer Science

School of Computing

National University of Singapore

2024/2025

Project No: H049560
Supervisor: Abhik Roychoudhury

Deliverables:

1 Report: Autonomous Program Improvement at the IDE
2 Programs: AutoCodeRoverSG/jetbrains-plugin, AutoCodeRoverSG/acr-interacitve

Abstract

This project brings an agent for autonomous program improvement namely AutoCodeRover (ACR) to the IDE environment. AutoCodeRover is an agent that leverages large language models with structured code analysis to diagnose and resolve software engineering tasks in natural language such as bug fixes and feature addition. Central to my contribution is a newly developed an IDE extension (JetBrains Plugin), an interactive interface for developers to prompt AutoCodeRover service from the IDE, which allows users to initiate tasks, receive real-time feedback, and apply generated patches seamlessly. Using the IDE extension developers can benefit from an agent with autonomous improvement capabilities in common development events inside the IDE, such as build failure troubleshooting, test failure diagnosis, and code-quality checks. Enhancements to the AutoCodeRover backend including interactive feedback and a Self-Fix capability of rectifying inapplicable patches by replaying relevant steps with self-initiated guidance. Experimental evaluations are conducted to demonstrate that this approach allows developers to remain inside a familiar IDE context while leveraging automated debugging and patch generation. This solution aims to ultimately increase developer productivity and accelerate the maintenance cycle in modern software development.

Subject Descriptors:

D.2.3	Coding Tools and Techniques
D.2.5	Programming Environments
I.1.2	Algorithms
I.2.2	Automatic Programming
K.6.3	Software Management

Keywords:

Autonomous Program Repair, Integrated Development Environment, Large Language Models, Software Maintenance, Patch Generation

Implementation Software:

Kotlin, Python, JetBrains, Visual Studio Code

Acknowledgement

I would like to express my heartfelt gratitude to everyone who supported me throughout the completion of this Final Year Project.

First and foremost, I am deeply grateful to Professor Abhik Roychoudhury for granting me the invaluable opportunity to undertake this project.

I also wish to sincerely thank my supervisor, Dr. Ridwan Shariffdeen, for his invaluable guidance, feedback, and encouragement at every stage. His expertise has greatly enhanced the quality of my work.

My appreciation also goes to Mr. Martin Mirchev, a PhD student at the National University of Singapore, for his detailed assistance, continuous support, timely responses and generous sharing of insights along the way.

I am also thankful to Mr. Guanlin Wang, formerly a research assistant at the National University of Singapore, for his prompt collaboration and support during implementation tasks.

Additionally, I would like to acknowledge the School of Computing, National University of Singapore, for providing the resources and a conducive learning environment.

Lastly, I am grateful to my family and friends for their unwavering support and encouragement throughout this journey.

Table of Contents

Chapter 1. Introduction

Modern software development faces a persistent challenge: developers spend an inordinate amount of time not only creating new features but also diagnosing, debugging, and fixing errors within complex, evolving codebases. Despite the promise of AI-assisted tools such as GitHub Copilot, current solutions often fall short when it comes to automating maintenance tasks like bug fixing and code improvement. This project addresses this gap by integrating an autonomous program improvement – AutoCodeRover, directly into the developer's Integrated Development Environment (IDE).

AutoCodeRover (ACR) is designed as a fully automated system for resolving software engineering tasks in natural language description, capable of generating code patches for both bug fixes and feature enhancements. At its core, AutoCodeRover combines the reasoning capabilities of large language models (LLMs) with traditional program analysis techniques, such as code search and debugging, to accurately identify and prioritize the code locations that require modifications. The resulting patches are intended not only to resolve the reported issues but also to align with the intended behavior of the software, thereby reducing the manual intervention required in the debugging process.

A key aspect of this project is the development of an IDE plugin that integrates seamlessly with ACR's backend. The plugin serves as the local interface for AutoCodeRover, enabling developers to initiate tasks directly from their IDE and receive real-time feedback from ACR. Besides prompting ACR for its normal usage (bug fixing and feature addition), the plugin flexibly leverages multiple IDE internal Application Program Interfaces (APIs) to extend ACR's capabilities, enabling it to resolve issues that arise during common development events within the IDE. To this end, the project involves substantial front-end and back-end work. The front-end component focuses on delivering a clean, modern, and user-friendly interface that fits naturally within the existing IDE environment. It is engineered to simplify complex interactions - presenting chat-based conversations, real-time progress updates, and intuitive controls for application of ACR messages - thus reducing the cognitive burden on developers.

On the back-end, significant effort has been directed toward refining the AutoCodeRover algorithm. Enhancements include integrating interactive feedback loops that allow developers to provide feedback for intermediate LLM responses during the ACR analysis, as well as the introduction of a *Self-Fix Agent* to self-fix and improve the analysis if ACR fails to resolve an issue at the first attempt. This new component is capable of autonomously analyzing and correcting the result by replaying parts of the pipeline with self-generated feedback to improve the old analysis, thereby moving the system closer to fully autonomous program repair.

By merging these advancements, this project aims to transform traditional maintenance workflows. The integration of autonomous program improvement within the IDE promises to significantly boost developer productivity and improve code quality by reducing the time and effort spent on manual debugging and repair. Furthermore, by incorporating continuous feedback and self-correction mechanisms, AutoCodeRover moves beyond the limitations of static code generators, offering a dynamic, responsive, and transparent solution that is well suited for the fast-paced environment of modern software development.

The AutoCodeRover WebConsole serves as the central, cloud-hosted management platform for AutoCodeRover services, offering code-analysis and issue-resolution capabilities via a *Software-as-a-Service* model. It orchestrates and monitors code-fixing tasks by initiating ACR Agent runs asynchronously and provides seamless integration with essential tools commonly used in modern CI/CD pipelines. The WebConsole supports connectivity with development environments (such as JetBrains IDEs), code repositories (e.g., GitHub), and monitoring solutions like Datadog and SonarQubeCloud, significantly enriching the Software Development Life Cycle through targeted, context-sensitive code suggestions.

Built using PHP with the Symfony framework, the WebConsole employs Symfony Messenger for internal asynchronous handling of requests, enhancing scalability and responsiveness. Docker Compose further facilitates consistency and ease of use across development, testing, and production environments by providing a unified containerized setup.

Figure 1 illustrates how ACR Plugin, ACR WebConsole, and the cloud-deployment of ACR Agent interact with each other. Within this Final Year Project, the ACR Plugin acts as the local interface integrated directly into the developer's IDE, communicating securely with the ACR WebConsole. To enable plugin interaction, users must first register and authenticate their accounts along with the corresponding target projects via the WebConsole. When a developer initiates tasks such as bug fixing or feature enhancement, the ACR Plugin compiles comprehensive project context into a structured JSON payload and transmits this securely via HTTPS to the WebConsole's dedicated endpoint.

Upon receiving this payload, the WebConsole authenticates the request using the provided API token and places it onto a Symfony-managed asynchronous queue. An internal request handler then spawns and configures a dedicated Docker container instance of the ACR Agent, deployed on cloud infrastructure (e.g., AWS EC2), passing it the necessary codebase and issue description for analysis. After successfully initiating the ACR Agent run, the WebConsole responds to the IDE plugin with critical identifiers (*runId*) and endpoints required for the plugin to poll or stream updates in real-time. Thus, ACR WebConsole efficiently bridges the local IDE interface with dynamically provisioned cloud-based agents, facilitating a smooth, interactive experience for developers as they track and manage ongoing code tasks.

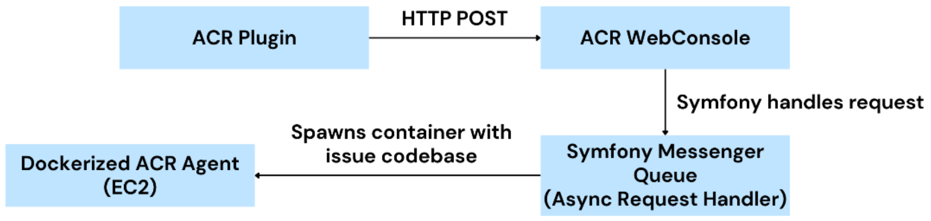

ACR Plugin
HTTP POST
ACR WebConsole
Symfony handles request
Spawns container with issue codebase
Dockerized ACR Agent (EC2)
Symfony Messenger Queue (Async Request Handler)

Figure 1: AutoCodeRover End-to-End Ecosystem

In summary, this Final Year Project seeks to deliver an autonomous program improvement solution that bridges the gap between human expertise and automated code repair. Through the careful integration of an interactive IDE plugin and an enhanced AutoCodeRover backend, the project paves the way for a future where software maintenance is not a bottleneck but a seamlessly integrated part of the development lifecycle.

Chapter 2. Literature Review

AI-driven tools in Integrated Development Environments (IDEs) have advanced significantly, now providing developers with real-time coding assistance. Popular tools such as GitHub Copilot and JetBrains AI Assistant support development by offering context-aware code suggestions, syntax corrections, and quick access to documentation.

2.1 Existing AI Tools in IDEs

GitHub Copilot, powered by OpenAI's Codex model, has notably improved coding efficiency by suggesting code in real time, primarily aiding in initial code generation. Studies reveal Copilot's ability to reduce coding time but highlight its limitations in debugging and maintenance (Zhang, Liang, Zhou, Ahmad, & Waseem, Study reveals common issues faced by GitHub Copilot users, 2023). Recent research by Zhang et al. (2023) further explores Copilot's functionality and user expectations, indicating that while Copilot is highly beneficial for generating code snippets and improving productivity, it struggles with integration and handling complex workflows, especially when it comes to debugging and optimizing existing code. Similarly, JetBrains AI Assistant provides in-IDE suggestions tailored to initial code creation but lacks tools for managing build failures or diagnosing bugs (Yuan, Nadi, Nguyen, & Sandoval, 2024). Sergeyuk et al. (2024) emphasize that while developers appreciate AI's assistance in simple tasks and code generation, they still face challenges when using AI tools for context-aware support and complex programming tasks. These studies suggest that while these tools enhance initial productivity, they lack robust support for complex maintenance workflows essential for software evolution.

2.2 The Need for Maintenance-Focused AI Tools

Although these AI tools boost coding speed, they fall short in maintenance tasks like bug resolution, build failure handling, and code quality checks, which are vital in real-world development (Bird, et al., 2023). According to Sergeyuk et al. (2024), developers' needs extend beyond basic code suggestions to include support for comprehensive debugging and seamless error handling within the IDE. The study (Zhang, Liang, Zhou, Waseem, & Ahmad, Demystifying Practices, Challenges and Expected Features of Using GitHub Copilot, 2023) also reveals developers' desire for tools that facilitate deeper integration with existing workflows, enhancing productivity in more than just code generation. This limitation has sparked a demand for AI solutions that can assist with ongoing maintenance and reduce the manual effort associated with debugging and code review.

2.3 AutoCodeRover: Bridging the Maintenance Gap

AutoCodeRover (ACR) was developed to address these needs, combining large language models (LLMs) with spectrum-based fault localization to offer automated debugging and code improvement (Zhang, Ruan, Fan, & Roychoudhury, 2024).Utilizing program representations like Abstract Syntax Trees (ASTs), ACR can analyze code structure, perform targeted diagnostics, and refactor code to support tasks like bug fixing and feature enhancement (Zhang et al., 2024). A recent study on in-IDE Human-AI Experience (Sergeyuk et al., 2024) highlights the importance of context-aware AI systems that adapt to developers' workflows and reduce cognitive load, which aligns with ACR's goals to improve IDE-based maintenance tasks.

This project integrates AutoCodeRover's backend capabilities into JetBrains IDE as a plugin, enabling developers to access ACR's maintenance features directly within their workflow. This integration facilitates tasks such as bug fixing, build failure diagnosis, and code quality analysis by offering real-time feedback and actionable code changes. By embedding ACR's maintenance-focused capabilities, this plugin not only extends initial coding support but also covers essential maintenance workflows, significantly reducing manual debugging and troubleshooting effort.

Chapter 3. Overview of AutoCodeRover

3.1 Overview

This chapter provides a comprehensive overview of the AutoCodeRover (ACR) algorithm and its associated products, with a particular focus on the ACR WebConsole. Understanding these components is essential for grasping the subsequent discussions in this report. In my work, the term "ACR Agent" is used interchangeably with AutoCodeRover to refer to the autonomous system that performs program improvement tasks. The ACR algorithm is designed to autonomously diagnose and repair software issues by leveraging large language models in conjunction with structured code analysis. The ACR WebConsole complements this approach by offering a *Software-as-a-Service* interface that facilitates interaction, monitoring, and integration with popular development and CI/CD tools. This integrated ecosystem lays the foundation for a seamless and efficient automated software improvement process.

3.2 AutoCodeRover Algorithm

At its core, ACR is an autonomous program improvement system that bridges the gap between abstract issue descriptions and concrete code modifications. The system is structured around a series of collaborating `Agents` that emulate a human software engineer's workflow - beginning with understanding the problem and culminating in the generation and validation of a `Patch`. As shown in Figure 2, This multi-agent architecture forms a *pipeline* when executing a task, which is designed not only to generate a candidate `Patch` but also to refine iteratively until it satisfies both functional correctness and developer intent (Ruan et al., 2024).

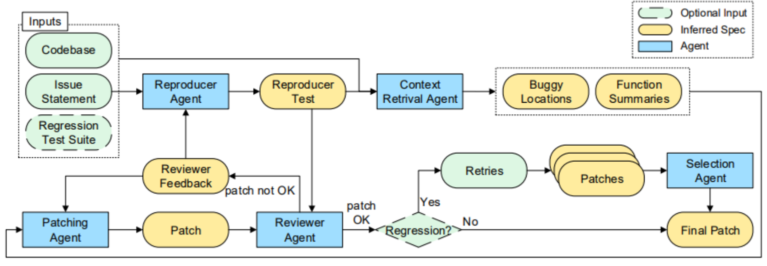
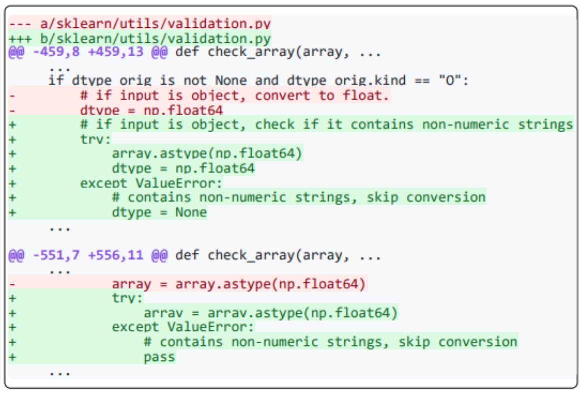


Figure 2: Overall workflow of AutoCodeRover (source: Ruan, Zhang, & Roychoudhury, 2024)

The process commences with the *Context Retrieval Agent*. When a GitHub issue is received, the `Agent` first analyzes the natural language description to extract relevant keywords - these may include names of classes, methods, or even code fragments. Using a predefined set of code search APIs (such as *search_class*, *search_method_in_class*, and *search_code_in_file*), the `Agent` navigates the project's *abstract syntax tree* (AST) to retrieve concise pieces of code context. Importantly, during this retrieval, the `Agent` not only collects raw code snippets but also infers function-level summaries - natural language specifications that describe the intended behavior of each function in relation to the issue at hand. This step is crucial: by converting low-level code artifacts into high-level specifications, the system provides the subsequent agents with a more human-understandable representation of the code's purpose, thereby reducing ambiguity in *patch generation*.

Once the retrieval phase produces a sufficiently rich context, the *Patch Generation Agent* takes over. This `Agent` integrates the gathered code context, the extracted function summaries, and the identified buggy locations into a comprehensive prompt that is fed to a large language model (LLM). The goal of the *Patch Generation Agent* is to synthesize a candidate `Patch` - a `diff` that (see Figure 3, source: Zhang et al., 2024), when applied, is expected to resolve the reported issue. Recognizing that a single pass may not yield a `Patch` that fully conforms to the desired specifications or passes all regression tests, the system incorporates an iterative retry loop. In this loop, if the `Patch` does not adhere to a predefined format or if it fails basic syntactic validation (for example, through a *linter* for Python code), the `Agent` re-issues the prompt with refined context. This process iterates a preconfigured number of times, ensuring that the final candidate is robust and aligns with the intended behavior.



```
--- a/sklearn/utils/validation.py
+++ b/sklearn/utils/validation.py
@@ -459,8 +459,13 @@ def check_array(array, ...
    ...
    if dtype_orig is not None and dtype_orig.kind == "O":
-       # if input is object, convert to float.
-       dtype = np.float64
+       # if input is object, check if it contains non-numeric strings
+       try:
+           array.astype(np.float64)
+           dtype = np.float64
+       except ValueError:
+           # contains non-numeric strings, skip conversion
+           dtype = None
    ...

@@ -551,7 +556,11 @@ def check_array(array, ...
    ...
-           array = array.astype(np.float64)
+           try:
+               array = array.astype(np.float64)
+           except ValueError:
+               # contains non-numeric strings, skip conversion
+               pass
    ...
```

Figure 3: Final patch generated by AutoCodeRover (source: Zhang, Ruan, Fan, & Roychoudhury, 2024).

Parallel to these stages, a *Reviewer Agent* is employed to evaluate the candidate `Patch`. The *Reviewer Agent* performs a dual function: it runs *reproducer tests* - if available - to verify that the `Patch` indeed resolves the issue, and it generates natural language feedback explaining its decision. This feedback not only validates the `Patch` but also serves as a meta-specification that can be incorporated into future iterations of the *patch generation* process. If the `Patch` fails the validation step, the feedback is sent back to the *Patch Generation Agent*, which then refines its approach. This iterative exchange is key to building trust in the autonomous repair process, as it mirrors the peer-review process of human-driven software engineering.

In summary, the AutoCodeRover backend algorithm unfolds as an orchestrated interplay among several specialized agents:

1. *Context Retrieval Agent*: extracts and refines code context and specifications from the project's AST based on the natural language issue.
2. *Patch Generation Agent*: synthesizes this context into a candidate `Patch` by iteratively prompting an LLM, with retry loops ensuring both correctness and adherence to style.
3. *Reviewer Agent*: validates the `Patch` through *reproducer tests* and provides actionable feedback, which in turn informs further iterations if necessary.

This architecture embodies a software engineering - oriented approach that leverages traditional debugging methods alongside modern LLM capabilities. The integration of explicit function summaries, iterative refinement, and natural language feedback distinguishes AutoCodeRover from purely AI-driven code generators. It provides a transparent, reliable, and effective workflow for automating program improvement, ultimately reducing the manual burden on developers while enhancing the quality and trustworthiness of automatically generated patches.

Chapter 4. AutoCodeRover Agent Enhancement

4.1 Overview

This chapter introduces the implementation done at the ACR Agent side - a critical component for the agent that, although separate from the ACR Plugin, is closely integrated with it to deliver a cohesive and efficient system. Our primary focus in the ACR backend has been to refine the underlying code structure and extend the existing algorithm of AutoCodeRover to boost both performance and interactivity.

To this end, we have refactored the core `Agents` to incorporate interactive feedback loops. These modifications enable the `Agents` to dynamically process and respond to user feedback, thereby allowing the ACR Agent to iteratively improve its output. In addition, we introduced a new *Self-Fix Agent* designed to autonomously address scenarios where a generated `Patch` proves inapplicable. This `Agent` functions by collecting diagnostic information from the failure, analyzing the root causes, and then triggering a *replay* of the affected stage with corrective feedback integrated. Together, these enhancements not only make the ACR Agent more interactive and robust but also ensure that it can adaptively self-correct, significantly enhancing the capability of resolving complicated tasks and reducing the need for manual intervention.

4.2 Interactive AutoCodeRover

4.2.1 Introduction of *replay* Mode

AutoCodeRover originally runs as a single-pass algorithm that, once triggered, would proceed through its agents (*Context Retrieval*, *Patch Generation*, etc.) and generate a final `Patch` with minimal input from the user beyond the initial issue description. While this was functional, it lacked the flexibility needed for real-world usage, where partial or additional instructions may be discovered after the process has begun. To address this gap, we introduce a more interactive model, underpinned by a new "*replay*" mode.

The fundamental shift lies in how intermediate agent outputs are tracked and subsequently revisited. Throughout an AutoCodeRover `Run`, each `Agent` logs and locally saves an intermediate `state` upon completing its execution. This `state` is then passed along the agent pipeline (see Figure 4), with each subsequent agent extending it with the results of its own execution.

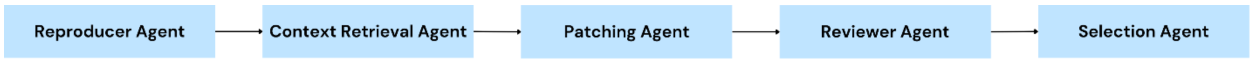


Figure 4: Pipeline of Agents

AutoCodeRover is now extended to support user feedback for individual Agents. Whenever users provide feedback for an Agent, the feedback will be incorporated into that Agent's intermediate state. ACR then "*replay*" the unchanged intermediate states of preceding Agents by simply reloading their previous states without executing the Agents, re-executes the selected Agent with user feedback which will lead to new execution results in `state`, and then proceeds to run the subsequent Agents as usual, based on the new execution results from the selected Agent.

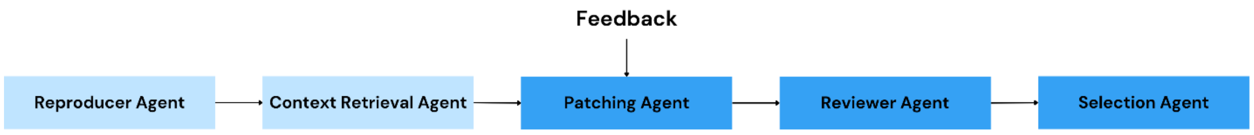


Figure 5: Example of Replay

After an ACR `Run` (see Figure 4), users can provide feedback for the Agents, triggering the ACR's "*replay*" mode. For example, as shown in Figure 5, if users provide feedback for the *Patching Agent*, ACR initiates *replay* by reloading the states from the *Reproducer* and *Context Retrieval Agents* without re-executing them. It then re-runs the *Patching Agent* by integrating the feedback using the `state` after the *Context Retrieval Agent*, and executes the subsequent Agents based on the updated state (highlighted in the dark blue background).

In the interactive ACR, when users provide feedback for an Agent, rather than re-executing the entire `Run` from scratch, the system traverses only those stages that might be improved based on the new input, so subsequent agents operate on the most up-to-date outputs. This approach aligns with the natural feedback logic in a pipeline: user feedback at a specific point implies satisfaction with all preceding stages, indicating that only the targeted stage requires improvement. Once updated, the downstream Agents are re-executed based on the new results from that stage.

4.2.2 Design of Feedback Structure

To make AutoCodeRover interactive, we need a robust way to capture and handle user feedback at any point in the pipeline. The primary challenge is tracking which agent's output the feedback refers to, what content in that output is under scrutiny, and how to incorporate new user feedback into subsequent runs. Hence, the `state`, the system's main log of execution results after each Agent, has been extended to facilitate this interaction.

Internally, the *replay* mechanism checks each recorded agent `state` for the *incoming_feedback* field, which stores the user feedback specific to that Agent. By default, this field is empty, indicating no feedback for the current Agent. If a non-empty *incoming_feedback* is detected in a `state`, it triggers a re-execution of that `Agent`. In *replay* mode, each Agent is extended to take the `state` from prior execution (as in a normal `Run`), along with its own previous output, and a transitional feedback prompt (see Figure 6 for an example), followed by the user feedback. Now this Agent will be re-executed with context of its own previous results and how users would like it to improve. After re-execution, it produces a new state, which is passed to downstream Agents, triggering their re-execution. All newly generated `state` overwrite the previous ones.

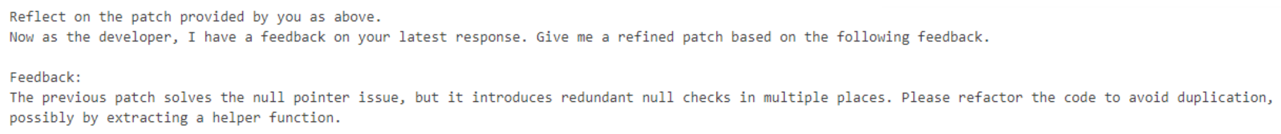

```
Reflect on the patch provided by you as above.
Now as the developer, I have a feedback on your latest response. Give me a refined patch based on the following feedback.

Feedback:
The previous patch solves the null pointer issue, but it introduces redundant null checks in multiple places. Please refactor the code to avoid duplication,
possibly by extracting a helper function.
```

Figure 6: Feedback Prompt for Patching Agent

Since the LLM introduces uncertainty into the ACR's performance, users often provide feedback specifically on the responses from LLM, instead of the embedded analysis by AutoCodeRover. In the Agent `state`, each LLM response is assigned with a unique *Universally Unique Identifier (UUID)* for identification. During the *replay* mode, when user feedback is passed to an Agent, it is accompanied by the *UUID* of the original LLM response to which the feedback refers. Upon receiving feedback, the Agent locates the corresponding LLM response by its *UUID* and discards all subsequent context generated after that point. This ensures the LLM understands exactly which response the user is referring to. And the discarded content is not needed, as it will be overwritten during the re-execution based on the updated `state`. Thus, the integrity of the replay process is maintained without loss of relevant information.

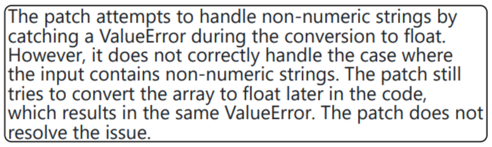
Rather than treat each piece of feedback as an isolated event, interactive ACR Agent retains a continuous conversation of past feedback. Each new feedback entry is appended to the field *feedback_history* in the `state`, including references to the model's prior response and any patch or context that prompted the user's remarks. As a result, Agent can see not only the user's latest instructions but also earlier commentary - making it possible to maintain a coherent "chat-like" flow of clarifications, corrections, and incremental improvements.

4.2.3 Conclusion of Interactive AutoCodeRover

This refined feedback structure ensures that only one Agent is provided with feedback at a time. In scenarios where conflicting instructions arise - such as when both the *Context Retrieval Agent* and the *Patching Agent* receive feedback simultaneously - the *replay* algorithm will ignore the feedback intended for the *Patching Agent*. This behavior occurs because the `state` provided to the *Patching Agent* is not derived from the previously saved `state` but is overwritten by the new `state` generated by the *Context Retrieval Agent*. To facilitate early detection of such conflicts, *replay* is designed to identify multiple feedback entries in the `state` and simply abort the program at the beginning. In practical usage, developers can provide feedback in many small increments, which is a typical scenario in real software development. Because we capture each piece of feedback alongside its relevant LLM response and preserve all commentary in *feedback_history*, interactive AutoCodeRover can run with preserving all prior conversation. This approach yields a more fluid, interactive workflow, further closing the gap between fully automated code repair and the iterative style characteristic of human-in-the-loop software engineering.

4.3 Self-Fix Agent

The *Self-Fix Agent* represents a significant extension to the AutoCodeRover algorithm, designed to enhance the system's autonomy in *patch repairment*. Traditional ACR implementations rely on a linear flow where user input triggers a sequence of Agents, ranging from *Reproducer Agent* to *Selection Agent* (see Figure 4). After a Patch is generated, it will be passed to the *Reviewer Agent* for scrutiny, which will produce a *reviewer feedback* (see Figure 7, source: Ruan et al., 2024) if the `Patch` is deemed incorrect or *inapplicable.*



The patch attempts to handle non-numeric strings by catching a ValueError during the conversion to float. However, it does not correctly handle the case where the input contains non-numeric strings. The patch still tries to convert the array to float later in the code, which results in the same ValueError. The patch does not resolve the issue.

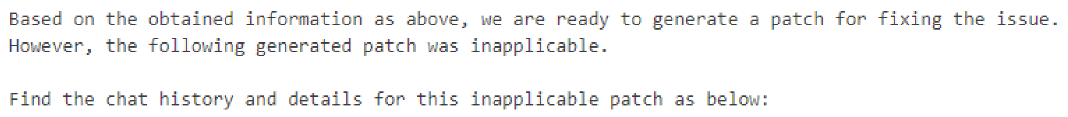
Figure 7: Example of Reviewer Feedback

However, when the `Patch` generated by the pipeline proves *inapplicable* by *Reviewer Agent*, the original design forces users to restart the process manually. The *Self-Fix Agent* addresses this limitation by automatically diagnosing the failure, generating corrective feedback, and replaying the process from the problematic stage. In this way, the system can autonomously "*self-repair*" before necessitating human intervention. The algorithm of *Self-Fix Agent* involves 4 steps:

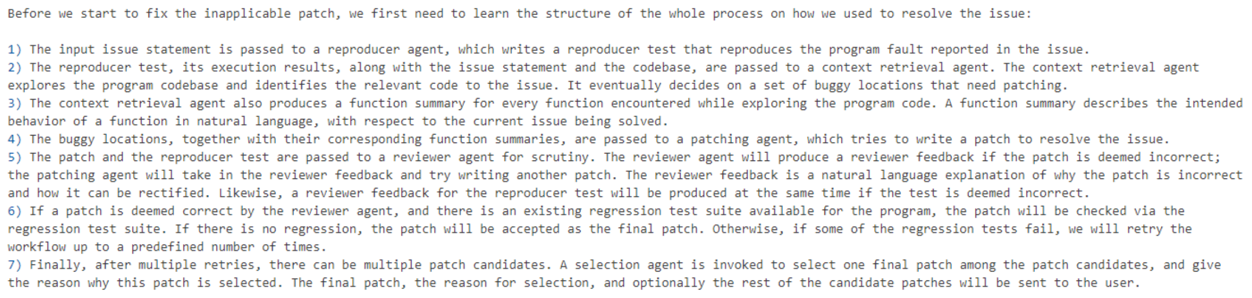
1. **Collect Reasons for why `Patch` is not applicable**: The *Self-Fix Agent* is invoked when the final selected `Patch` is marked as *inapplicable*, as indicated by the *Reviewer Agent*. Upon detection of such a failure, the *Self-Fix Agent* collects detailed error information of why the `Patch` is not applicable. This information, along with a transitional "collect reasons" prompt (see Figure 8a), is put into a newly initialized message thread, which will be extended during the subsequent analysis phases. After collecting the failure messages or reasons of inapplicable `Patch`, *Self-Fix Agent* starts to analyze the reasons.
2. **Identify what `Agent` caused the inapplicable `Patch`**: In the analysis phase, the *Self-Fix Agent* constructs a composite prompt that incorporates not only the error details but also the broader context of the `Agents` in the ACR algorithm. This prompt is designed using a Chain-of-Thought (CoT) approach, where LLM acts as a diagnostic *Judge* or *Evaluator* here to evaluate work done by others (Zheng, et al., 2023). The CoT prompt first guides LLM to learn the role and function of each `Agent` as shown in Figure 8a. Due to the nature of the *Context Retrieval* and *Patch Agent* which can directly affect the final performance of ACR Agent, they may be more likely to cause the problem. Therefore, the *Self-Fix Agent* is designed to place special emphasis on these stages (see Figure 8b). The prompt constructed here will be appended to the message thread created in step 1, which contains the collected error messages from the inapplicable `Patch`, also accumulating the conversation by including the initial issue description from users, intermediate responses from all previous `Agents`. *Self-Fix Agent* will then prompt LLM to identify which

`Agent` most likely contributed to the inapplicable `Patch`. An example shown in Figure 9a demonstrates how LLM identifies the faulty Agent with justifications.

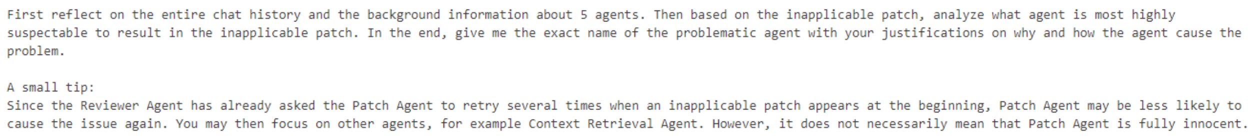
3. **Generate feedback on faulty `Agent` to improve**: Once the faulty `Agent` is identified, the *Self-Fix Agent* then generates instructional feedback targeted at that `Agent`. The feedback prompt as shown in Figure 7d is carefully engineered to let LLM first reflect on the function of that `Agent`, and based on incorporated historical analysis of how it went wrong, provide specific, actionable suggestions for the `Agent` to refine. For example (see Figure 9b), if the *Context Retrieval Agent* is deemed responsible, the feedback might instruct the `Agent` to refine its search heuristics and verify buggy locations carefully.


```
Based on the obtained information as above, we are ready to generate a patch for fixing the issue.
However, the following generated patch was inapplicable.

Find the chat history and details for this inapplicable patch as below:
```

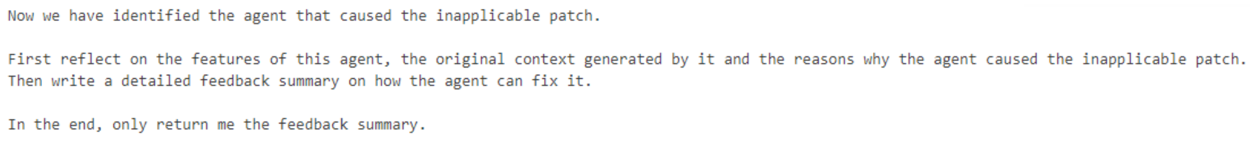
(a) Prompt for 'Collect Reasons'


```
Before we start to fix the inapplicable patch, we first need to learn the structure of the whole process on how we used to resolve the issue:

1) The input issue statement is passed to a reproducer agent, which writes a reproducer test that reproduces the program fault reported in the issue.
2) The reproducer test, its execution results, along with the issue statement and the codebase, are passed to a context retrieval agent. The context retrieval agent
explores the program codebase and identifies the relevant code to the issue. It eventually decides on a set of buggy locations that need patching.
3) The context retrieval agent also produces a function summary for every function encountered while exploring the program code. A function summary describes the intended
behavior of a function in natural language, with respect to the current issue being solved.
4) The buggy locations, together with their corresponding function summaries, are passed to a patching agent, which tries to write a patch to resolve the issue.
5) The patch and the reproducer test are passed to a reviewer agent for scrutiny. The reviewer agent will produce a reviewer feedback if the patch is deemed incorrect;
the patching agent will take in the reviewer feedback and try writing another patch. The reviewer feedback is a natural language explanation of why the patch is incorrect
and how it can be rectified. Likewise, a reviewer feedback for the reproducer test will be produced at the same time if the test is deemed incorrect.
6) If a patch is deemed correct by the reviewer agent, and there is an existing regression test suite available for the program, the patch will be checked via the
regression test suite. If there is no regression, the patch will be accepted as the final patch. Otherwise, if some of the regression tests fail, we will retry the
workflow up to a predefined number of times.
7) Finally, after multiple retries, there can be multiple patch candidates. A selection agent is invoked to select one final patch among the patch candidates, and give
the reason why this patch is selected. The final patch, the reason for selection, and optionally the rest of the candidate patches will be sent to the user.
```

(b) Background of Agents


```
First reflect on the entire chat history and the background information about 5 agents. Then based on the inapplicable patch, analyze what agent is most highly
suspectable to result in the inapplicable patch. In the end, give me the exact name of the problematic agent with your justifications on why and how the agent cause the
problem.

A small tip:
Since the Reviewer Agent has already asked the Patch Agent to retry several times when an inapplicable patch appears at the beginning, Patch Agent may be less likely to
cause the issue again. You may then focus on other agents, for example Context Retrieval Agent. However, it does not necessarily mean that Patch Agent is fully innocent.
```

(c) Chain-of-Thought Prompt for Diagnosing the Faulty Agent


```
Now we have identified the agent that caused the inapplicable patch.

First reflect on the features of this agent, the original context generated by it and the reasons why the agent caused the inapplicable patch.
Then write a detailed feedback summary on how the agent can fix it.

In the end, only return me the feedback summary.
```

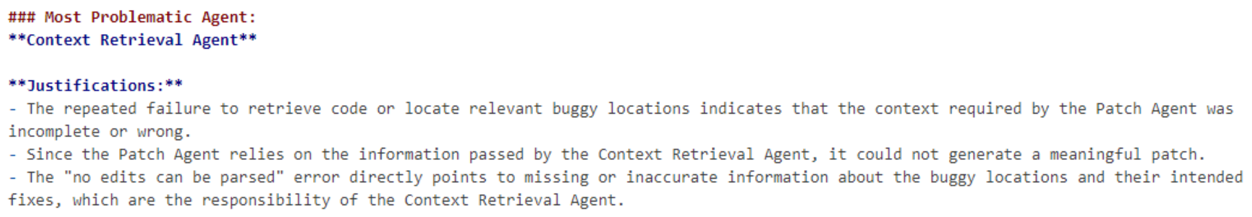
(d) Prompt for 'Generate Feedback'

Figure 8: Template Prompt for Self-Fix Agent

4. ***Replay* from the faulty `Agent` with feedback**: The final step involves the same *replay* mechanism introduced in *Chapter 4.2.1*, which leverages the feedback loop architecture. The *Self-Fix Agent* triggers a partial *replay* starting from the problematic `Agent`'s stage.

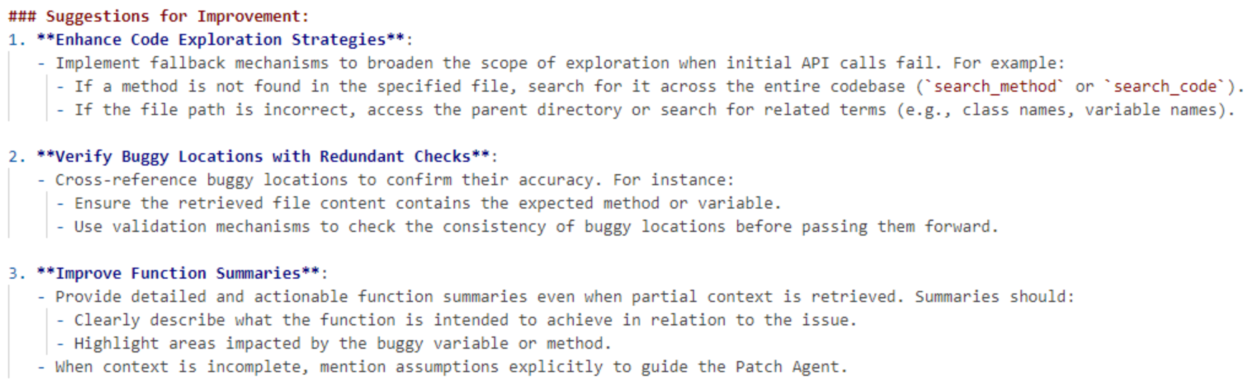
In this *replay*, the new feedback is incorporated into the `Agent`'s input, prompting it to re-run its task with the corrective guidance from step 3. Similarly, the subsequent `Agents` in the pipeline then use the updated `state`, ensuring that the self-fix improvement propagates through the remaining stages.

5. ***Self-Fix* until `Patch` is applicable**: Step 1-4 are iterated up to a predefined retry limit. If the self-fix replay eventually yields an applicable `Patch`, the revised `Patch` is accepted and returned; otherwise, the system still results in an inapplicable `Patch` if hits the limit.



```
### Most Problematic Agent:
**Context Retrieval Agent**

**Justifications:**
- The repeated failure to retrieve code or locate relevant buggy locations indicates that the context required by the Patch Agent was
incomplete or wrong.
- Since the Patch Agent relies on the information passed by the Context Retrieval Agent, it could not generate a meaningful patch.
- The "no edits can be parsed" error directly points to missing or inaccurate information about the buggy locations and their intended
fixes, which are the responsibility of the Context Retrieval Agent.
```

(a) Example of LLM Response from Identifying Faulty Agent



```
### Suggestions for Improvement:
1. **Enhance Code Exploration Strategies**:
   - Implement fallback mechanisms to broaden the scope of exploration when initial API calls fail. For example:
     - If a method is not found in the specified file, search for it across the entire codebase (`search_method` or `search_code`).
     - If the file path is incorrect, access the parent directory or search for related terms (e.g., class names, variable names).

2. **Verify Buggy Locations with Redundant Checks**:
   - Cross-reference buggy locations to confirm their accuracy. For instance:
     - Ensure the retrieved file content contains the expected method or variable.
     - Use validation mechanisms to check the consistency of buggy locations before passing them forward.

3. **Improve Function Summaries**:
   - Provide detailed and actionable function summaries even when partial context is retrieved. Summaries should:
     - Clearly describe what the function is intended to achieve in relation to the issue.
     - Highlight areas impacted by the buggy variable or method.
   - When context is incomplete, mention assumptions explicitly to guide the Patch Agent.
```

(b) Example of LLM Response from Generating Feedback

Figure 9: Example of LLM Responses in Self-Fix Agent

By introducing the Self-Fix Agent, we extend the original ACR algorithm as shown in Figure 10. From a technical standpoint, this design embodies several innovative methodologies. First, it leverages iterative feedback loops to make the repair process self-correcting. By storing comprehensive state information - including conversation history, error logs, and feedback history - the system maintains a rich context that enables subsequent replays to be more informed and targeted. Second, the use of multiple LLM queries in the diagnostic phase allows for a nuanced assessment of the repair process, mimicking a human engineer's reflective evaluation of why a patch may have failed. Finally, by integrating self-fix capabilities directly into the autonomous workflow, the system can reduce dependency on user input, significantly lowering the time and effort required for manual debugging.

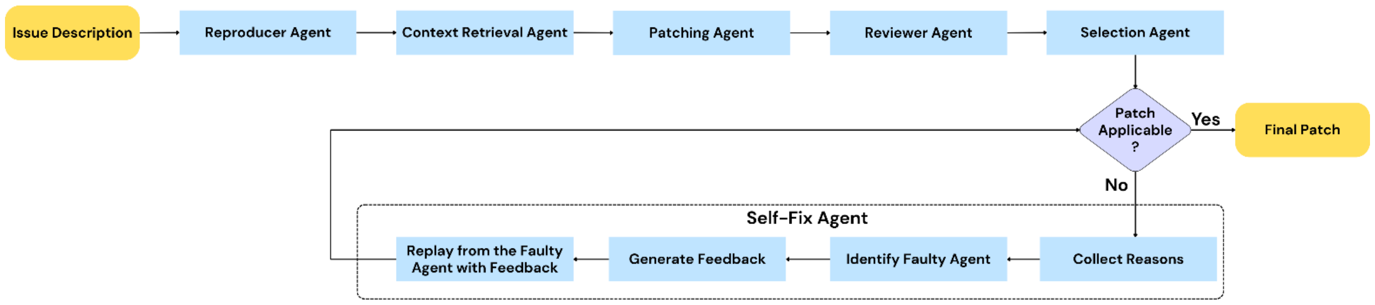


Figure 10: New Pipeline with Self-Fix Agent

In summary, the *Self-Fix Agent* extends the original AutoCodeRover algorithm by introducing a robust self-repair mechanism. This `Agent` not only augments the feedback loop with an automated diagnostic process but also reinforces the system's ability to autonomously correct errors through iterative replays. Such an enhancement is critical in advancing the vision of fully autonomous program repair, bridging the gap between human oversight and machine-led software maintenance.

Chapter 5. IDE Extension

5.1 Overview

This chapter details the architectural design and integration of the ACR Plugin into modern JetBrains-based IDEs, transforming ACR from a cloud-native service into a practical in-IDE development assistant. The ACR Plugin is designed to support the full development lifecycle by embedding context-aware automation directly into the developer's environment. By integrating deeply with IDE APIs and external tools, it empowers developers to improve productivity, code quality, and debugging efficiency - all without leaving the IDE. The Plugin offers three main features, each designed to solve real pain points during software development and tightly integrated with the ACR Agent: prompting ACR to fix bugs or add features, detecting and responding to build/test failures, and integrating static analysis results via SonarLint. These features are not isolated but deeply integrated with IDE event listeners, UI components, and the underlying ACR agent pipeline to ensure responsiveness and traceability.

The chapter begins with configuration and usage of ACR Plugin. Subsequent sections explain how the plugin collects contextual cues (e.g., cursor movement, open files, and code references) to enhance issue descriptions sent to the ACR Agent. It further outlines the lifecycle of an ACR Run, from task initiation, event-stream monitoring, to patch generation and structured feedback handling. A significant portion of the chapter focuses on background listeners that capture build and test failures inside IDE, and how these events are processed and transformed into actionable prompts. Then SonarLint engine is introduced which allows developers to check code quality inside ACR Plugin and prompt ACR to fix code quality issues. Finally, the chapter highlights extended features such as patch application, diff alignment, tooling shortcuts, and UI elements that collectively elevate the usability and feedback mechanisms of the plugin.

Through this integration, the ACR Plugin bridges the gap between local development workflows and remote AI-assisted code generation, enabling a hybrid programming model where automation complements human-in-the-loop software engineering.

5.2 Configuration of AutoCodeRover Plugin

The configuration of the ACR Plugin is designed to provide a seamless, secure, and user-friendly integration with the AutoCodeRover ecosystem. There are two primary ways to run the plugin. In a development environment, developers can launch the plugin in a sandboxed IDE instance using Gradle (via *./gradlew runIde*). Alternatively, end users can install the released version of the plugin packaged as a ZIP file by using the "*Install Plugin from Disk…*" option within their JetBrains IDE. Both approaches ensure that the plugin is readily available to support tasks such as automated bug fixing and feature addition.

To facilitate a smooth user experience, the plugin offers a dedicated *Configuration Page* that is integrated into the IDE's standard Settings dialog. This page consolidates all the necessary credentials and environment parameters required for AutoCodeRover to operate. These fields correspond to the IDs or credentials generated by the ACR WebConsole when users first register an account or initialize a new project there. A key technical feature of this configuration mechanism is the persistent storage of settings, which serializes configuration data into an XML file and is saved locally in developers' projects. This ensures that settings are maintained consistently across sessions and project openings, minimizing the need for repetitive manual input.

By implementing the Configurable interface, the plugin naturally integrates with the IDE's native workflow for applying configuration changes. Since the majority of these configuration fields are vital for the plugin's core functions, the design enforces that all critical fields contain valid data before any tasks are executed.

5.3 Usage and Workflow of AutoCodeRover Plugin

5.3.1 Usage

The ACR Plugin integrates multiple IDE APIs and external tools to deliver a seamless developer experience within the IDE environment. It enables AutoCodeRover to assist developers in various real-world scenarios by offering three core features, each tightly integrated with the ACR Agent:

1. Fix a Bug / Add a Feature: This is the primary function of the ACR Agent. The ACR Plugin sends task requests such as *fix a bug* or *add a feature* to the cloud-based ACR service. Once processed, the ACR Agent returns the execution results to ACR Plugin for application.

2. Detect Build / Test Event Failures: ACR Plugin actively listens for build and test events triggered within the IDE. If a failure occurs, it automatically captures the associated error messages and provides developers with an option to request help from ACR to resolve the issue.

3. Trigger SonarLint Analysis: ACR Plugin integrates the SonarLint engine to perform static code analysis. Developers can initiate code quality checks directly through ACR Plugin. When issues are detected, they are captured by the plugin, and developers are given the option to request fixes from ACR.

5.3.2 Workflow

Figure 11 illustrates the internal workflow of the ACR Plugin. After the plugin starts, it branches based on the type of event - whether it's a user-initiated request (e.g. *fix a bug* or *add a feature*), a failed build / test event, or a SonarLint analysis. In cases where ACR involvement is optional (such as after code analysis), the plugin prompts the user to decide whether to proceed with prompting ACR Agent to resolve the issues.

Once ACR Agent is activated, ACR Plugin builds a connection with the deployed ACR Agent and monitors the connection in the background. After a Patch is returned from ACR Agent, the plugin then checks if the user wants to take further actions. If no action is taken, the plugin simply returns to the initial "*ACR Plugin Start*" state, and users can revisit and take further actions on the returned Patch at any later time. If the user decides to proceed, ACR Plugin support users to apply the Patch, rate it, or provide feedback.

The ACR Plugin enhances the development experience by embedding intelligent automation directly into the IDE workflow. This workflow ensures flexibility and minimizes disruption by allowing users to selectively engage with ACR at different stages of development. Whether fixing bugs, handling test failures, or improving code quality, the Plugin acts as a smart assistant - one that's always in sync with both your local project and the cloud-powered ACR Agent. By handling routine tasks and suggesting high-quality code changes, the Plugin allows developers to focus on what matters most: building great software.

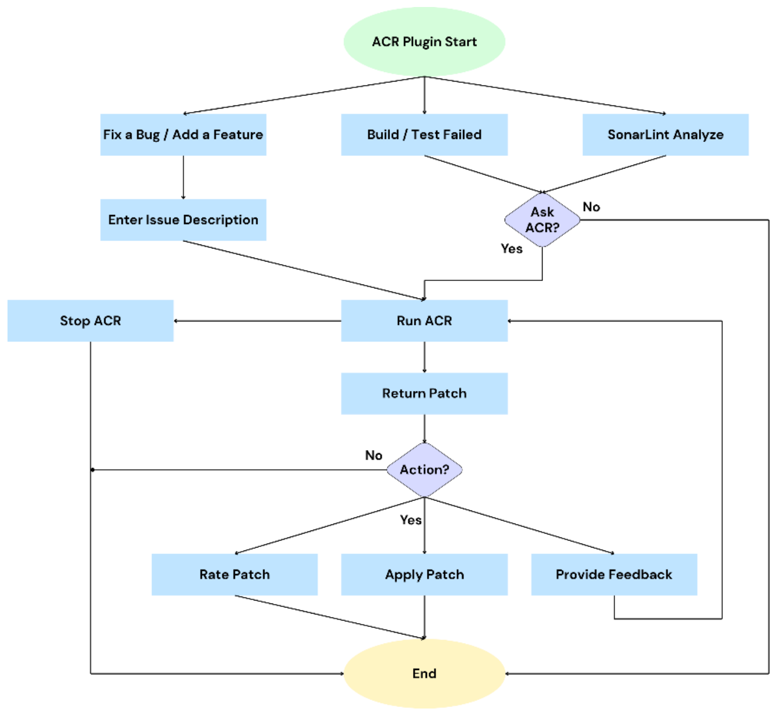


Figure 11: Workflow of ACR Plugin

5.4 Prompt AutoCodeRover

Prompting AutoCodeRover constitutes the primary functionality of the ACR Plugin. All the plugin's additional features - such as background listeners, SonarLint integration - are ultimately designed to either compensate for or augment ACR's activities inside the IDE. The central idea is that, when a user requests ACR to *fix a bug* or *add a feature*, the plugin communicates with ACR Agent and manages the resulting patches within the local IDE environment. This section explores the implementations and workflow for prompting ACR Agent from ACR Plugin.

5.4.1 Preparations

To perform its main tasks – *fix a bug* or *add a feature* - ACR requires the latest snapshot of the relevant repository along with an issue description of what needs to be done. The ACR Plugin is responsible for assembling all necessary contextual information in the background to support each ACR task. It automatically gathers details such as the active IDE name, the programming language in use, the local project path, and the current Git branch. Additionally, it invokes Git utilities to retrieve the latest commit from the remote repository. This contextual data is collected every time a new ACR task is initiated, ensuring that the ACR Agent operates on the most accurate and up-to-date version of the code. Moreover, this contextual information is supplemented by settings retrieved from the configuration page, which serves as an authentication mechanism, allowing the ACR WebConsole to identify the user and associate the task with the correct project. Once these preliminary elements are in place, ACR Plugin packages the issue description input by users together with the automatically gathered metadata and sends the request to ACR's API endpoint, initiating an ACR `Run` task.

5.4.2 Enrichment of Issue Description

The performance of ACR can be greatly affected by vague or incomplete issue description, especially if the underlying codebase lacks well-defined structure. In such cases, ACR's *Context Retrieval Agent* may produce suboptimal searches and experience an obvious delay due to multiple rounds of retrying the search for potential bug locations or relevant modules. To mitigate this, ACR Plugin augments users issue description with three additional sources of context - specifically, a record of recent cursor movements, a catalog of open file paths, and a list of references to objects or methods extracted from the description. By appending these details to the user's original text, the plugin empowers ACR to conduct a more focused exploration of the project, thereby reducing the overhead of scanning large or disorganized repositories.

5.4.2.1 Cursor Tracker

While individual cursor movements may seem trivial, collectively they provide valuable behavioral context. These signals reflect the developer's current attention and may indicate users' recent edits or potential issues being investigated. For instance, if a user submits a task with a vague prompt like "*fix the broken function call in this file*" without specifying the function name or line, the cursor tracker helps narrow down the scope by pointing ACR toward the regions the user has recently explored.

The plugin's cursor tracking feature showcases how subtle user interactions can be leveraged to guide ACR's focus during code analysis. It registers a *CaretListener* to capture changes in the editor's caret position. Each time the developer's caret moves, the plugin retrieves the associated file, line number, and column index, then stores this record in an internal queue, preserving only the most recent ten entries (*e.g., File: ToDos.java, Line: 11, Column: 1, File: Deadlines.java, Line: 32, Column: 32, etc*). Although these movement logs may appear minor, they can serve as powerful signals to ACR about areas in the code that the developer is currently inspecting. ACR

Plugin incorporate the latest 10 moves of cursor in editor into a simple prompt "*The user's most recent 10 cursor movements in the IDE have been tracked. These positions may indicate areas of interest or potential issues related to the task*". When sending a request *fix a bug* or *add a feature*, users' issue description is extended by this information.

5.4.2.2 References Finder

In addition to cursor movements, the plugin consolidates references to classes and methods that it identifies from the user's input description. When a user types a description indicating, for instance, "*the function processData in DataHandler fails on edge cases*", ACR Plugin leverages a smaller local LLM client that attempts to parse out the target element name (*processData*) and class (*DataHandler*). Once extracted the target and class names, the plugin searches all the references of the target in the identified class using PSI (Program Structure Interface) elements to determine whether a reference belongs in a parent function, a class field, or another relevant container, in the end obtaining locations such as line numbers and contextual descriptors. The process of enumerating references is implemented by traversing all PSI elements in the identified class. Each time it finds a token that matches the target function or field name, it retrieves the element's offset in the file, converts that offset to a line number, and infers whether the surrounding code belongs to a method declaration, a property, or a class field. This granular detail provides ACR with valuable "anchors" for *Context Retrieval Agent*, making the context search more efficient.

5.4.2.3 Current Open Files

A third type of enrichment comes from the list of currently open files in the IDE. ACR Plugin obtains this list via the IDE's file manager, extracting the relative paths rather than absolute paths so as not to confuse ACR with overly long or machine-specific directory structures. Merging information on open files with cursor history helps guide the system toward code sections that the user is actively browsing and editing.

The rationale for incorporating these three enrichment steps becomes evident when considering the nature of large projects or issues with incomplete bug reports. Without additional hints, when the issue description is insufficient, ACR might need to perform a heavy structural analysis of the entire repository, checking all code paths or searching for specific names in thousands of files. By layering context about where the user's caret was, which files are open, and which specific elements are repeatedly referenced in the textual description, the plugin conveys a stronger focus or "search corridor" for the `Agents`. This, in turn, can lower network overhead, reduce extraneous search queries, and improve the likelihood of identifying the correct bug location on the first attempt for *Context Retrieval Agent*.

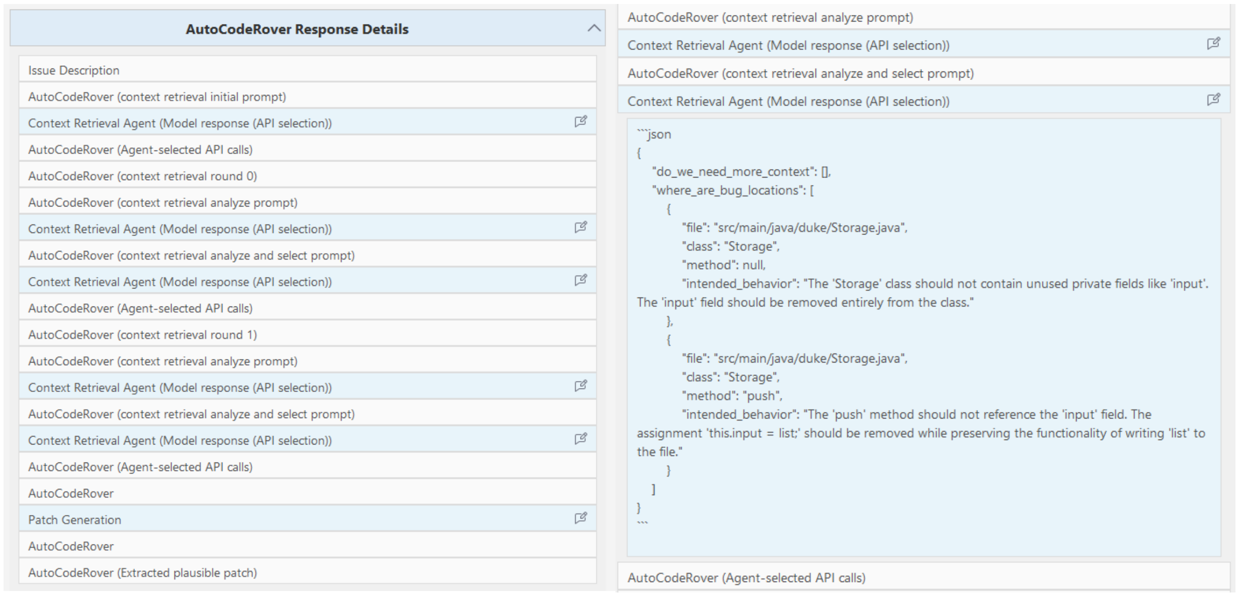
5.5 AutoCodeRover Run

After ACR Plugin prepares the required set-ups and finalizes the issue description, it initiates an ACR task by sending an HTTP request to the deployed ACR API endpoint. Upon successfully transmitting this request, the ACR endpoint responds with a JSON object containing two crucial fields: *subscribe_link* and *runId*. The *subscribe_link* designates a server-sent event (SSE) endpoint used to stream all intermediate information produced by ACR during its reasoning and code manipulation steps. The *runId* is a unique identifier for ACR WebConsole to distinguish this `Run` from any other tasks that might occur on the same project. Maintaining a clear mapping between *runId* and the associated project session allows to differentiate not only concurrent runs but also subsequent feedback loops in which the user might refine or retry a patch.

ACR Plugin then creates an SSE connection using OkHTTP and maintains a subscription to *subscribe_link*, the server's event stream, to capture intermediate progress updates of ACR Agent and the `Patch` in real time. Throughout this connection, the *subscribe_link* sends a sequence of JSON-formatted messages to the plugin, each containing intermediate outputs from `Agents` inside ACR (including the evolving LLM responses, embedded prompts of `Agents`, and other analysis). The plugin closes the SSE connection when it receives the message with status "*Finished*", indicating that ACR `Run` has completed.

After ACR Agent finishes a `Run`, ACR Plugin will parse the intermediate outputs and the final Patch, and display them in the UI. Rather than dumping the text directly into the conversation, the ACR Plugin UI uses a nested panel structure to build a collapsible hierarchy. It balances transparency and clarity by storing these multi-step results in an *expandable menu* labeled "*AutoCodeRover Response Details*", serving as an anchor for the entire sub-step discussion. Beneath it, multiple "*step*" entries appear as clickable rows. When the user clicks a row, an *expand* or *collapse* effect reveals or hides the associated textual details. For example (see Figure

12), an intermediate sub-step might be labeled "*Context Retrieval Agent (Model response (API selection))*". Clicking the row reveals a nested panel that displays the full summary string. This approach keeps the chat history from becoming unwieldy yet allows developers to explore ACR's reasoning flow whenever they desire



AutoCodeRover Response Details

- Issue Description
- AutoCodeRover (context retrieval initial prompt)
- Context Retrieval Agent (Model response (API selection))
- AutoCodeRover (Agent-selected API calls)
- AutoCodeRover (context retrieval round 0)
- AutoCodeRover (context retrieval analyze prompt)
- Context Retrieval Agent (Model response (API selection))
- AutoCodeRover (context retrieval analyze and select prompt)
- Context Retrieval Agent (Model response (API selection))
- AutoCodeRover (Agent-selected API calls)
- AutoCodeRover (context retrieval round 1)
- AutoCodeRover (context retrieval analyze prompt)
- Context Retrieval Agent (Model response (API selection))
- AutoCodeRover (context retrieval analyze and select prompt)
- Context Retrieval Agent (Model response (API selection))
- AutoCodeRover (Agent-selected API calls)
- AutoCodeRover
- Patch Generation
- AutoCodeRover
- AutoCodeRover (Extracted plausible patch)

- AutoCodeRover (context retrieval analyze prompt)
- Context Retrieval Agent (Model response (API selection))
- AutoCodeRover (context retrieval analyze and select prompt)
- Context Retrieval Agent (Model response (API selection))

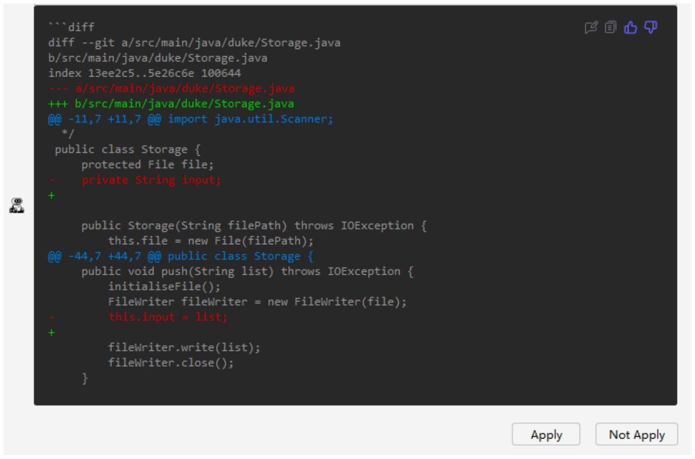
~~~
```json
{
   "do_we_need_more_context": [],
   "where_are_bug_locations": [
      {
         "file": "src/main/java/duke/Storage.java",
         "class": "Storage",
         "method": null,
         "intended_behavior": "The 'Storage' class should not contain unused private fields like 'input'.
The 'input' field should be removed entirely from the class."
      },
      {
         "file": "src/main/java/duke/Storage.java",
         "class": "Storage",
         "method": "push",
         "intended_behavior": "The 'push' method should not reference the 'input' field. The
assignment 'this.input = list;' should be removed while preserving the functionality of writing 'list' to
the file."
      }
   ]
}
```
~~~

- AutoCodeRover (Agent-selected API calls)

Figure 12: Intermediate ACR Responses

To display `Patch`, ACR Plugin uses a distinct color scheme and typeface to signal its special and reflect the structure of a diff file. The plugin uses a dark background with a monospace font (*Consolas*), closely mirroring typical diff displays in Git tools. Within this context, the syntax highlighting is meticulously designed: added lines are rendered in green, deleted lines in red, chunk metadata (for example, @@ -10,7 +10,7 @@) in blue, and unmodified context lines in gray.

On the top-right corner of the patch bubble (see Figure 13), the plugin displays a "*Feedback*" icon, a "*Copy*" icon, *thumbs-up* and *thumbs-down* buttons. The thumbs convey a binary rating for the patch and correspond to ACR's *rating* API, allowing the user to signal satisfaction or dissatisfaction. Because `Patch` is often the most significant artifact in an ACR `Run`, the UI applies a quick feedback button to gather immediate user feedback on `Patch`. Once the Patch is displayed, ACR Plugin prompts 2 buttons "*Apply*" or "*Not apply*". If "*Apply*" is clicked, the plugin calls the backend logic to apply the `Patch`, highlight changed lines, and open the modified files in the editor. Clicking "Not apply" removes both options from view, effectively dismissing this version of the patch without permanently altering the user's workspace.



~~~~
```diff
diff --git a/src/main/java/duke/Storage.java
b/src/main/java/duke/Storage.java
index 13ee2c5..5e26c6e 100644
--- a/src/main/java/duke/Storage.java
+++ b/src/main/java/duke/Storage.java
@@ -11,7 +11,7 @@ import java.util.Scanner;
  */
 public class Storage {
     protected File file;
-    private String input;
+

     public Storage(String filePath) throws IOException {
         this.file = new File(filePath);
@@ -44,7 +44,7 @@ public class Storage {
     public void push(String list) throws IOException {
         initialiseFile();
         FileWriter fileWriter = new FileWriter(file);
-        this.input = list;
+
         fileWriter.write(list);
         fileWriter.close();
     }
~~~~

Figure 13: Display of Patch

Overall, this event-driven mechanism is motivated by the fact that ACR's internal workflows span multiple reasoning steps in the `Agents`. Streaming these steps to the plugin allows the user to monitor progress and provides a transparent view of ACR's underlying logic - particularly useful in scenarios where developers want to validate the patch's correctness or trace how certain lines of code were identified to be relevant to a task.

5.6 AutoCodeRover APIs

The ACR plugin communicates with the ACR WebConsole through a series of HTTPS-based API endpoints. Besides the normal ACR request which is the primary channel for initiating a new `Run`, ACR plugin also supports additional requests: Rating a `Patch`; Sending user feedback; and Stopping ongoing ACR runs. These requests enhance the interactions between developers and AutoCodeRover across the entire lifecycle of ACR sessions.

5.1.4.1 Request an ACR Run

The first part of this workflow consolidates all relevant data points - such as the user's issue description, the references extracted from local code analysis, cursor history, and open file paths - into a single JSON payload. The plugin's settings, including the ACR authentication token and the chosen LLM model, are also incorporated into this payload, which are necessary for starting an ACR `Run`. Right before finalizing that payload, the issue description is processed to ensure that any special characters like embedded quotes, backslashes, or control characters do not invalidate the JSON body. The final JSON remains syntactically valid even if the user typed multi-line statements or special formatting symbols. ACR Plugin then initiates an HTTPS POST request to the ACR API endpoint. The plugin verifies that the protocol is secure (HTTPS) before proceeding, thereby safeguarding any sensitive data like tokens or user credentials.

5.1.4.2 Rate a Patch

Further interactions flow from the *runId* assigned to each session. If a developer chooses to rate a resulting `Patch`, ACR plugin calls the *rating* API endpoint, simply supplying the *runId* and a numerical rating (0 or 1, indicating "dissatisfied" or "satisfied") in the JSON payload.

5.1.4.3 Stop an ACR Run

In some situations, developers may want to stop an ACR `Run`. The *stop* API endpoint accomplishes this by taking only *runId* as input. ACR Plugin simultaneously terminates its local SSE subscription to *subscribe_link*, which eliminates unnecessary delay of ACR Plugin workflow. This dual action keeps the plugin UI from blocking while the server finalizes its shutdown routine in the background.

5.1.4.4 Feedback on Model Response

As discussed in *Chapter 4.2*, ACR Agent was enhanced to be interactive such that it supports user feedback on each LLM response. After the intermediate steps of ACR `Run` are displayed, developers can provide feedback on a model response through the UI. ACR Plugin differentiates two main categories of intermediate data: *Raw LLM responses* and *Analysis by ACR*. This color-coding helps developers quickly identify which messages originated directly from the LLM versus those that ACR has already refined. Furthermore, any "model response" items carry a dedicated *feedback icon* so that users can submit feedback on a targeted model response.

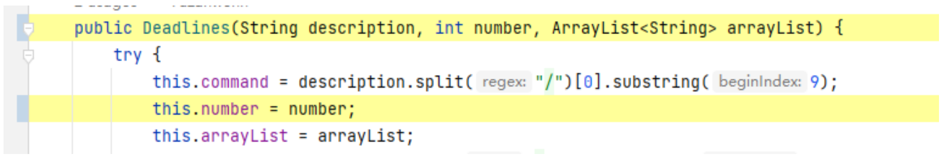
Once users click on a *feedback* icon and input feedback description, ACR Plugin sends a request to *feedback* API endpoint. Besides including the *runId*, this request also specifies three additional fields in the JSON payload: *stepName* (identifying which `Agent` is being addressed), *feedback* (the textual content entered by the developer), and *model_resp_id* (an identifier referencing the relevant step's LLM response). By linking user feedback to a particular LLM response, ACR Agent can reconstruct the program workflow and incorporate the user's remarks in a new iteration of code generation.

5.7 Patch Apply

An essential step in making AutoCodeRover practical for real development tasks is the quick application of `Patch` inside the IDE. Although ACR returns diffs in standard git-style format, developers should not have to manually transfer these changes into their local environment. The ACR plugin therefore streamlines this final step by parsing, merging, and highlighting the change in a manner consistent with typical IDE workflows.

Since a `Patch` can contain multiple `diff` chunks (see Figure 2), the process begins with orchestrating how each `diff` chunk is split and subsequently reflected in local files. Upon receiving a `Patch`, the plugin divides it into discrete file-level `diffs`, each tagged by a relative path. For each file-level `diff`, the plugin decides whether the corresponding `diff` can be safely applied via standard Git commands. There may be situations where developers modify the project locally while ACR is running, which can cause a mismatch between the returned `Patch` and the latest modified project, and `diff` cannot be directly applied due to the mismatch. ACR Plugin introduces a *Patch Alignment* strategy (*See Chapter 6*) to still apply the `Patch` by aligning it with user modification, accommodating potential changes that users may have introduced while ACR was running.

Highlighting changed lines is an optional but valuable step for developer awareness. After the `Patch` is applied, the plugin locates the lines of code introduced or modified, by either parsing the raw `diff` directly (for unmodified code) or relies on a specialized approach to match lines inside the "merged" version (for files that require *Patch Alignment*). Each changed line is assigned a highlight region in the editor (see Figure 14). This approach leverages IDE's internal event model, hooking a mouse listener to clear highlights once the user clicks in the editor. In doing so, developers get immediate visual feedback on how ACR's fix altered their code, but they also retain the freedom to dismiss these highlights easily.



```
public Deadlines(String description, int number, ArrayList<String> arrayList) {
    try {
        this.command = description.split( regex: "/")[0].substring( beginIndex: 9);
        this.number = number;
        this.arrayList = arrayList;
```

Figure 14: Example of Highlight Lines

Once each file's `diff` application concludes, the plugin opens all the updated files in editor tabs and relocates the cursor to the final updated file. This user-centric design ensures that developers immediately see ACR's changes, can inspect the highlighted diff lines for correctness, and then resume coding or testing without extra manual steps. By managing everything from splitting the `diff` to rendering post-merge highlights, ACR Plugin not only expedites the acceptance of `Patch`, but also fosters greater clarity and trust in automatically generated code.

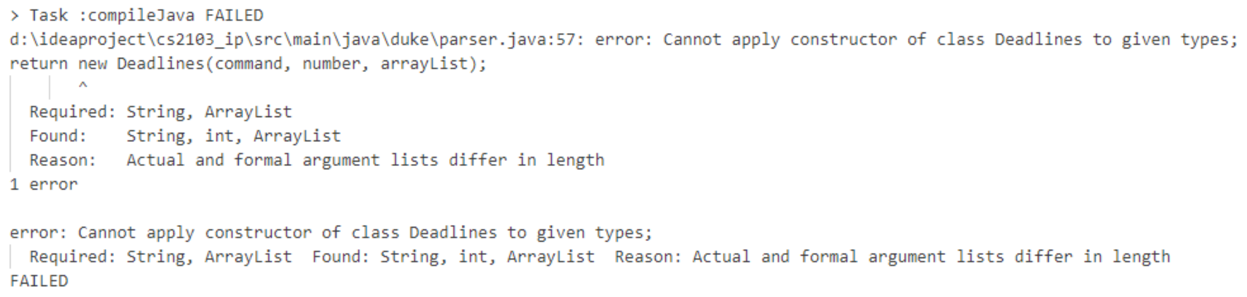
5.8 Background Listeners

In many development workflows, problems only come to light when a build or test suite fails. ACR Plugin addresses these scenarios by maintaining an *event listener* that silently monitors IDE events and classifies them as either *build event* or *test event*. Whenever there is an *event failure,* the plugin gathers relevant context and offers the developer an opportunity to let ACR generate a fix.

5.8.1 General Build Failures

As IDE (or Gradle) builds the project, the listener intercepts every build event. When the system flags a *build* as finished with a failure, the listener runs its filtering and cleaning logic. These remove extraneous entries, such as progress notes or HTML tags. Any "unique" messages - meaning those not already captured - are saved in a String buffer until the *build* has definitively ended. Once the listener concludes that no more events are incoming, it calls its failure callback. This callback in turn displays the collected error messages followed by buttons for prompting ACR Agent in the UI. By centralizing the *build* failure logic, the plugin ensures that ephemeral logs do not vanish, but instead become a simple one-click bug fix request which helps ACR Agent analyze more efficiently.

As shown in Figure 14, the function *Deadlines* is a class constructor. In this case, we remove the variable *int num* from the constructor declaration. When we build the project, this change causes a build error because the constructor of the *Deadlines* class is still being called in *Parser.java*, where the *int num* variable is still correctly included during the initialization of *Deadlines*. Now due to the mismatch in parameters, the build fails, and this failure is correctly captured by the build listener, as shown in Figure 15.



```
> Task :compileJava FAILED
d:\ideaproject\cs2103_ip\src\main\java\duke\parser.java:57: error: Cannot apply constructor of class Deadlines to given types;
return new Deadlines(command, number, arrayList);
          ^
  Required: String, ArrayList
  Found:    String, int, ArrayList
  Reason:   Actual and formal argument lists differ in length
1 error

error: Cannot apply constructor of class Deadlines to given types;
  Required: String, ArrayList  Found: String, int, ArrayList  Reason: Actual and formal argument lists differ in length
FAILED
```

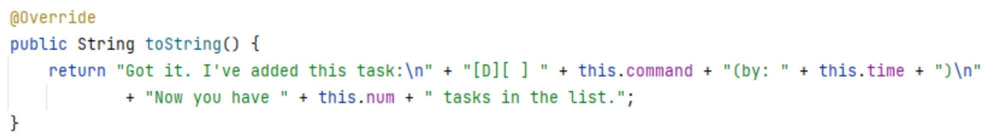
Figure 15: Build Event Failure

5.8.2 Test Failures

When the IDE detects a *test failure* event, the listener switches to a more specialized parsing procedure. Instead of relying solely on the trace stacks from ephemeral events, it looks for a hyperlink pointing to a test report stored locally in the project (e.g., "file://.../index.html"). This local file path is extracted from the messages collected by *event listener* and is passed to a lightweight HTML parser (*JSoup*) to sift through the test failure content - identifying which methods failed, the class they belong to, and a truncated version of the associated stack trace. Armed with these more granular details, ACR Plugin can then package a specialized bug description for failures of tests, highlighting not only the symptom (“test fails”) but also the specific test name, error lines and reasons of error.

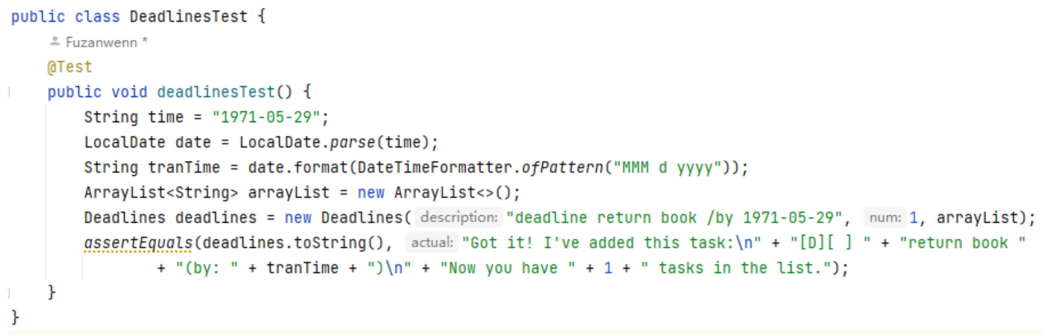
For example, we created a unit test named *deadlinesTest* (see Figure 16b) for the *Deadlines* class. This test verifies whether the *toString* method (see Figure 16a) correctly formats the output by capturing the internal fields of a *Deadlines* object during initialization. Initially, the unit test was correctly implemented and passed as expected. However, to demonstrate how the test failure listener in the ACR Plugin functions, we introduced a subtle modification: changing the character `"."` to `"!"` in the expected string after *"Got it"*. This intentional change will cause an assertion failure during test execution. As shown in Figure 16c, the test failure listener first locates a test failure report which is generated and saved locally by IDE when a test failure event occurs. It then parses the report and outputs a structured failure message that accurately identifies the failing test class and method, along with the error type and a detailed comparison of the expected versus actual values.

In essence, the background listener approach underscores how the ACR plugin weaves automated bug-fixing capabilities into the developer's normal cycle of building and testing inside the IDE. A single piece of logic differentiates general build errors from test errors by scanning the output for certain markers - once it finds them, it tailors the captured logs or test details into a more meaningful bug or test-fix prompt. This creates a responsive environment, where recurring or surprising failures are quickly funneled into ACR's automated solutions.



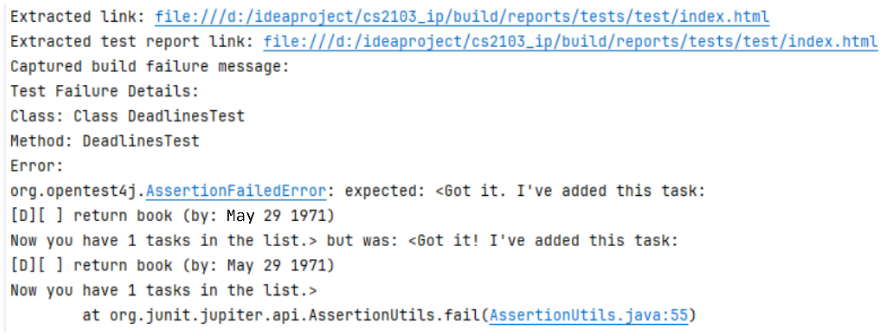
```
@Override
public String toString() {
    return "Got it. I've added this task:\n" + "[D][ ] " + this.command + "(by: " + this.time + ")\n"
            + "Now you have " + this.num + " tasks in the list.";
}
```

(a) Tested Function: toString



```
public class DeadlinesTest {
    Fuzanwenn *
    @Test
    public void deadlinesTest() {
        String time = "1971-05-29";
        LocalDate date = LocalDate.parse(time);
        String tranTime = date.format(DateTimeFormatter.ofPattern("MMM d yyyy"));
        ArrayList<String> arrayList = new ArrayList<>();
        Deadlines deadlines = new Deadlines( description: "deadline return book /by 1971-05-29",  num: 1, arrayList);
        assertEquals(deadlines.toString(),  actual: "Got it! I've added this task:\n" + "[D][ ] " + "return book "
                + "(by: " + tranTime + ")\n" + "Now you have " + 1 + " tasks in the list.");
    }
}
```

(b) Unit Test: DeadlinesTest



```
Extracted link: file:///d:/ideaproject/cs2103_ip/build/reports/tests/test/index.html
Extracted test report link: file:///d:/ideaproject/cs2103_ip/build/reports/tests/test/index.html
Captured build failure message:
Test Failure Details:
Class: Class DeadlinesTest
Method: DeadlinesTest
Error:
org.opentest4j.AssertionFailedError: expected: <Got it. I've added this task:
[D][ ] return book (by: May 29 1971)
Now you have 1 tasks in the list.> but was: <Got it! I've added this task:
[D][ ] return book (by: May 29 1971)
Now you have 1 tasks in the list.>
        at org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:55)
```

(c) Test Failure Messages Captured by ACR Plugin

Figure 16: Test Failure Example

5.9 SonarLint Integration

5.9.1 Implementation

SonarLint (now SonarQube) is widely used in many IDEs to detect common coding pitfalls and provide immediate feedback on potential improvements. While it largely acts as a contextual guide for developers - flagging style violations, potential bugs, or security vulnerabilities - it remains disconnected from deeper automated fixes if used on its own. ACR Plugin bridges this gap by invoking SonarLint's analysis engine and feeding the resulting findings directly into ACR Agent, so that developers can decide whether to act on SonarLint's advice manually or to prompt ACR for an immediate fix.

In the current implementation, the integration is targeted at Java code. The ACR Plugin thus leverages SonarLint purely for Java analysis. Once configured via environment variables such as *SONARLINT_HOME* (pointing to a local JAR of SonarLint enginee) and *JDK_HOME*, the plugin loads the SonarLint engine and scans the local Java source set. The scanning process iterates over discovered source files in src/main/java, src/generated/java, or any other directories recognized that are relevant to Java. Each file is converted to a *ClientInputFile* object that provides a standard interface for SonarLint. By coupling these input files with user-supplied environment data, the plugin can run the SonarLint engine transparently in the background.

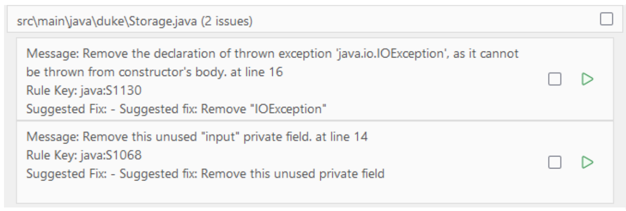
A typical analysis run begins with setting up an analysis configuration that includes rules (Java code format offered by SonarLint), project library paths, and Java language settings. Internally, SonarLint creates a sequence of *Issue* objects whenever it detects a violation of rules or a potential bug. The plugin accumulates these issues in memory, along with any quick fixes SonarLint may propose. These might be simple textual transformations (e.g., renaming a variable, removing unused imports) or more involved changes that touch multiple lines of code. Once the analysis is completed, the plugin collects all identified issues and nicely displays them in the UI. If SonarLint fails to find any problems, the log output simply shows no issues found, which can be an indicator that the user's code meets SonarLint's style guidelines or that further configuration is needed.

5.9.2 Display of SonarLint Issues

ACR plugin visualizes the SonarLint analysis results in a way that resembles the display of ACR intermediate steps, with a few design-specific adjustments for SonarLint. To display SonarLint issues, ACR Plugin populates a tree-like menu that is expandable. The top-level view shows a clickable entry for each file in which SonarLint flagged issues, accompanied by the total number of issues in parentheses (see Figure 17c). Clicking on a file entry reveals a sub-panel enumerating individual issues in that file, each with a short textual summary: the violation message, violated rule key, and a suggested fix provided by SonarLint.

Each SonarLint issue also includes interactive controls on the right side. As shown in Figure 17a, there is a "*run*" button for each single issue, click on which users can ask ACR to only resolve that specific issue. There are four types of UI indicators for an issue: *run*, *stop*, *success* and *failure*. During ACR `Run`, ACR Plugin toggles that "*run*" button into a "*stop*" button, and creates an obvious visualization that outlines the current running issue with a shining multi-colored border around the issue panel.

There is also a "*select box*" for each issue, each file, and the whole project, indicating whether developers want to queue a particular issue or a combination of issues for fix via ACR Agent. Once ready, users can click "*run all*" button to process all queued issues one by one with ACR Agent. For bulk runs, the plugin cycles through all selected items sequentially, giving users an option to stop the current running process at any point; any waiting issues continue after a cancellation or completion of the current issue (see Figure 17b). For the issues that are queuing, the "*run*" button will be replaced with a spinner icon to indicate they are waiting to run. If ACR Agent manages to generate a `Patch` for a particular issue, the status updates the interface with a small *success* icon next to that issue's label, a *failure* icon otherwise.

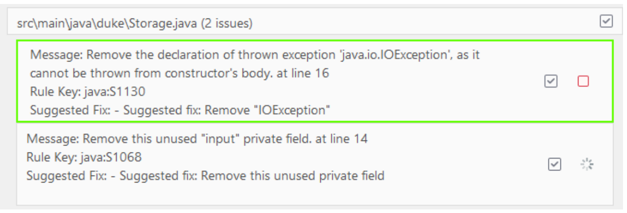


src\main\java\duke\Storage.java (2 issues)

Message: Remove the declaration of thrown exception 'java.io.IOException', as it cannot be thrown from constructor's body. at line 16
Rule Key: java:S1130
Suggested Fix: - Suggested fix: Remove "IOException"

Message: Remove this unused "input" private field. at line 14
Rule Key: java:S1068
Suggested Fix: - Suggested fix: Remove this unused private field

(a) Expanded Issues

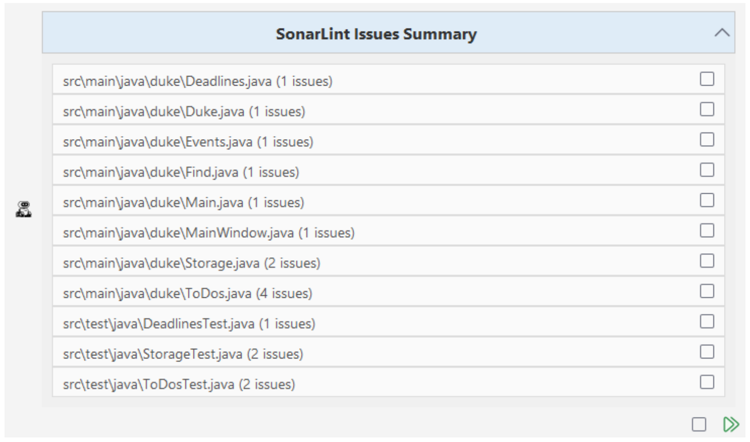


src\main\java\duke\Storage.java (2 issues)

Message: Remove the declaration of thrown exception 'java.io.IOException', as it cannot be thrown from constructor's body. at line 16
Rule Key: java:S1130
Suggested Fix: - Suggested fix: Remove "IOException"

Message: Remove this unused "input" private field. at line 14
Rule Key: java:S1068
Suggested Fix: - Suggested fix: Remove this unused private field

(b) Run ACR on Issues



SonarLint Issues Summary

src\main\java\duke\Deadlines.java (1 issues)
src\main\java\duke\Duke.java (1 issues)
src\main\java\duke\Events.java (1 issues)
src\main\java\duke\Find.java (1 issues)
src\main\java\duke\Main.java (1 issues)
src\main\java\duke\MainWindow.java (1 issues)
src\main\java\duke\Storage.java (2 issues)
src\main\java\duke\ToDos.java (4 issues)
src\test\java\DeadlinesTest.java (1 issues)
src\test\java\StorageTest.java (2 issues)
src\test\java\ToDosTest.java (2 issues)

(c) Overall Display of SonarLint Issues

Figure 17: UI of SonarLint Issues

As a result, developers can focus on higher-level problem-solving while the system handles both the detection and (potentially) the remediation of code-quality pitfalls. Although at present the plugin focuses on Java, the code structure implies that other languages can be introduced in the future by swapping in different SonarLint modules or third-party analyzers - mirroring the plugin's broader design philosophy of hooking into multiple static-analysis or runtime tools and channeling them seamlessly into ACR's automated fix pipeline.

5.10 Other Functionalities

Beyond its primary support for prompting ACR Agent, `Patch` application, background listening, and SonarLint analysis, ACR plugin also provides several supplementary features designed to enhance user workflow and maintain a clean execution environment. These include shortcuts for zipping projects and applying external diffs, as well as a `Reset` function for clearing the chat history in the UI and restore ACR plugin to its initial state. Both features reflect the plugin's overarching emphasis on flexibility: users can choose to handle tasks with or without the presence of a continuous ACR session, and they can seamlessly revert or reinitialize their environment when necessary.

5.10.1 Tool Menu

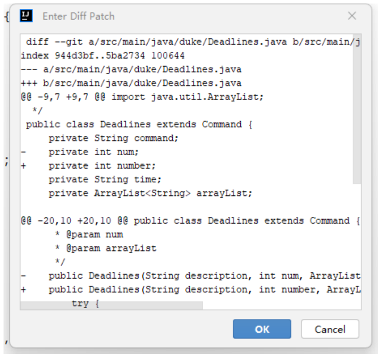
ACR Plugin puts the quick actions at *Tools* menu inside the IDE, addressing two common scenarios. The first is the *Zip Project* function, which packages the current project's source into a single archive in .zip format. The second is a shortcut for *Patch Application* that extends the plugin's ability to incorporate diffs or `Patch` without requiring an ACR `Run`.

5.10.1.1 Zip Project Action

This feature was implemented and kept for future request of ACR Agent. Although ACR Agent currently grabs the latest commit from remote on GitHub, developers sometimes want to request an ACR fix on uncommitted changes. A trivial way to enable ACR to work directly on local changes is to zip the project and send it to the ACR Agent. By selecting "Zip Project" from the Tools menu, the plugin compresses all files in the base directory and saves the resulting archive alongside the project folder.

5.10.1.2 Quick Apply Patch

Ordinarily, ACR plugin displays a newly returned patch in the UI immediately after an ACR session finishes. However, developers may sometimes retrieve `Patch` at a later time - possibly from the "*task history*" in the ACR WebConsole. In these cases, the "*Quick Apply Patch*" action in the *Tools* menu provides a straightforward path to apply a `Patch` (see Figure 18). The plugin prompts users with a text area to input the `diff`, then calls the same patch application and line-highlighting logic. This effectively lets users "pull in" a fix from anywhere - past or present - so long as they have the raw patch data.



```
 diff --git a/src/main/java/duke/Deadlines.java b/src/main/j
index 944d3bf..5ba2734 100644
--- a/src/main/java/duke/Deadlines.java
+++ b/src/main/java/duke/Deadlines.java
@@ -9,7 +9,7 @@ import java.util.ArrayList;
  */
 public class Deadlines extends Command {
     private String command;
-    private int num;
+    private int number;
     private String time;
     private ArrayList<String> arrayList;

@@ -20,10 +20,10 @@ public class Deadlines extends Command {
      * @param num
      * @param arrayList
      */
-    public Deadlines(String description, int num, ArrayList
+    public Deadlines(String description, int number, ArrayL
         try {
```

Figure 18: Quick Apply Patch

5.10.2 Reset

Real-world development can be fluid and unpredictable. Sometimes the user may want to halt all currently running tasks, discard any partially completed changes, and revert the plugin to a clean slate. To handle this, ACR plugin defines a `Reset` feature that simultaneously: clears UI state and kills running threads. The plugin eliminates all the chat history in the UI, thereby returning the interface to its initial "*start*" messages. Concurenlty, it sends a "stop run" request to ACR if a session is ongoing, ensuring that the remote side does not continue processing. In parallel, it cancels local `coroutines` or `jobs` so that no further background tasks remain.

By performing these steps in one coordinated operation, `Reset` prevents orphaned connections and syncs with ACR WebConsole, kills all running processes that could otherwise cause confusion or unexpected behavior. This design also aligns with how ACR plugin organizes concurrency: each significant operation (e.g., SonarLint analysis, background scanning) runs under a dedicated job hierarchy. Each job is canceled in an orderly manner, ensuring that any blocked or pending connections (like SSE subscriptions described in *Chapter 5.1.3*) are released.

The `Reset` functionality thus completes the lifecycle management around ACR Plugin usage. Whenever developers decide that a session is no longer needed - or if something has gone amiss in the environment - resetting transitions the plugin back to a known-good baseline. From there, users can initiate new tasks or reapply previously saved patches without incurring side-effects from an older run. This complements other plugin modules by guaranteeing that each new cycle starts fresh, consistent, and free of partial state.

5.10.3 User Experience Management

System messages - encompassing the plugin's normal outputs, ACR's intermediate steps, or `Patch` - are shown in the chat panel with an ACR logo on the left and tinted background colors to distinguish them from user inputs. The plugin adds a "*typewriter effect*" for these messages so that text reveals itself gradually, as though typed in real time. The effect is largely managed by a timer logic which appends characters one by one to a `JTextArea` or `JTextPane`, continuously adjusting the component's height. This approach lends a more dynamic feel to the conversation.

To prevent excessively long messages from rendering too slowly, the plugin switches to a "*fast display*" mode after five seconds, immediately revealing any remaining text. This behavior benefits users who might otherwise wait for large blocks of output to appear one character at a time, which improves the overall user experience while preserving the immersive, step-by-step feel for shorter messages.

ACR plugin also uses a *scroll-to-bottom* button to aid navigation whenever new content is added (or the user scrolls up). As soon as the user is no longer at the bottom of the chat, an arrow button appears to quickly jump them down to the latest message. This is especially helpful during an extended ACR `Run`, where multiple intermediate updates may cause the chat panel to overflow.

Another addition is a *pausing line* indicator positioned above the text input area. When ACR Plugin is running an ACR query, this thin animated bar appears, sliding from side to side to reflect that the plugin is waiting for response from ACR Agent. The plugin's code drives this animation with a timer, incrementally shifting a small gradient-filled rectangle across the bar. As soon as the ACR `Run` completes (either finishing successfully or stopped by the user), the plugin hides the bar by toggling its visibility.

Chapter 6. User Experience Improvement

6.1 Overview

Modern development workflows often integrate automated patch suggestions, which risk becoming stale if the developer makes changes to the codebase after the patch was generated. In the context of AutoCodeRover, when developers modify their projects locally before ACR returns a Patch, the local code may have diverged from the patch's baseline (i.e., the latest pushed commit on the remote). As a result, directly applying the patch using the Git API may fail due to merge conflicts.

To address this problem, this Chapter introduces a novel solution we proposed - *Patch Alignment* - for resolving the mismatch between a `Patch` returned from ACR, and the current uncommitted local user modifications. From a user's perspective, *Patch Alignment* begins when a developer clicks on "Apply" button to apply a returned `Patch` from ACR Agent and conflicts are detected between local code and the Patch. This *Patch Alignment* solution fills up the gap and limitation of time required for ACR running, which prevents developers from being blocked while waiting for ACR response, and it greatly improves and smooths the development workflow.

6.1 GumTree

6.2.1 Introduction of GumTree

GumTree is a widely adopted Abstract Syntax Tree (AST) differencing tool that forms the backbone of the Patch Alignment algorithm (Falleri, Morandat, Blanc, Martinez, & Monperrus, 2014). Unlike traditional diff tools that compare source code line-by-line, *GumTree* parses code into ASTs to detect semantic changes—such as insertions, deletions, updates, and moves—thereby capturing the true structural transformations in the code.

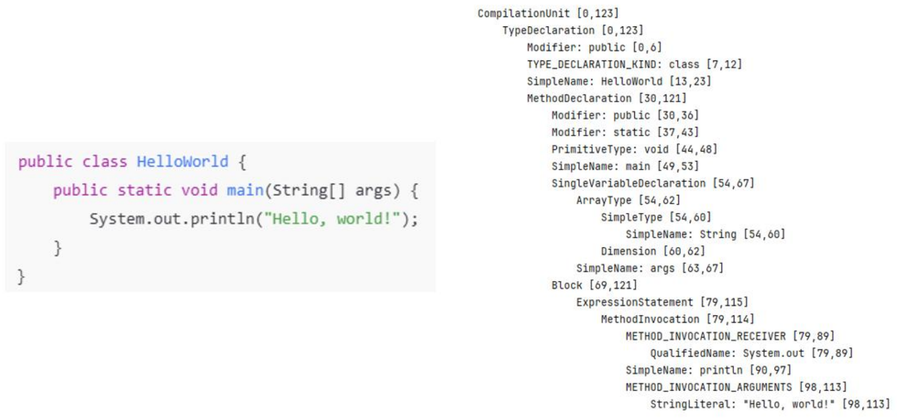
At its core, *GumTree* employs a two-phase matching algorithm: a top-down phase identifies large isomorphic subtrees, and a bottom-up phase refines the alignment by mapping remaining nodes based on structural similarity. This dual approach not only enhances the accuracy of the diff but also supports multiple AST meta-models (e.g., JDT and Spoon), ensuring flexibility across different code representations.

In the Patch Alignment solution, *GumTree*'s capabilities are leveraged to perform a three-way merge between the baseline version, the user-modified code, and the auto-generated patch. By precisely mapping corresponding nodes across these versions, *GumTree* minimizes false conflicts and preserves developer intent, thereby smoothing the integration of automated patches into evolving codebases.

6.2.2 Demo of GumTree

In the *Patch Alignment* strategy, we mainly use the *Tree Generator* and *Matcher* offered by *GumTree*. Here, we demonstrate a simple demo to illustrate how these embedded tools work.

1. *Tree Generator*: We begin by feeding a simple Java program into *GumTree*'s Tree Generator, which parses the code and produces an Abstract Syntax Tree (AST). As shown in Figure 19, the structure of the Java HelloWorld program is transformed into a hierarchical tree where each syntactic element - such as method declarations, variable types, and statements - is mapped as an individual node.



```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

```
CompilationUnit [0,123]
    TypeDeclaration [0,123]
        Modifier: public [0,6]
        TYPE_DECLARATION_KIND: class [7,12]
        SimpleName: HelloWorld [13,23]
        MethodDeclaration [30,121]
            Modifier: public [30,36]
            Modifier: static [37,43]
            PrimitiveType: void [44,48]
            SimpleName: main [49,53]
            SingleVariableDeclaration [54,67]
                ArrayType [54,62]
                    SimpleType [54,60]
                        SimpleName: String [54,60]
                    Dimension [60,62]
                SimpleName: args [63,67]
            Block [69,121]
                ExpressionStatement [79,115]
                    MethodInvocation [79,114]
                        METHOD_INVOCATION_RECEIVER [79,89]
                            QualifiedName: System.out [79,89]
                        SimpleName: println [90,97]
                        METHOD_INVOCATION_ARGUMENTS [98,113]
                            StringLiteral: "Hello, world!" [98,113]
```

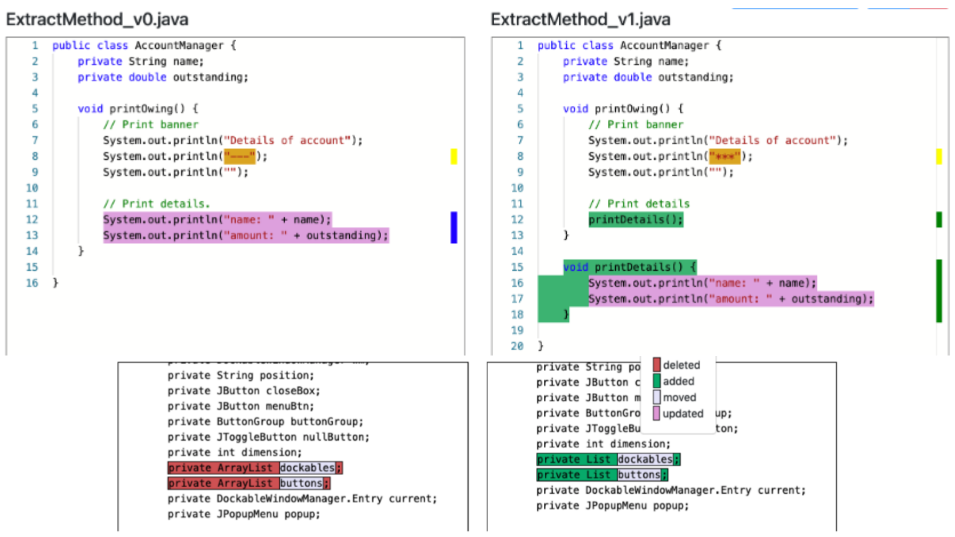
Figure 19: Example of Tree Generator

2. *Matcher*: *GumTree* provides an internal *Matcher* that is a core component responsible for identifying how two ASTs align and differ. Rather than relying solely on textual similarity, it proceeds by comparing subtrees in a *bottom-up* manner, mapping each node in the source tree to its closest counterpart in the destination tree. This matching algorithm tracks insertions, deletions, moves, and updates, producing both a *mapping store* - indicating which nodes match across the two versions - and an *edit script* that pinpoints transformations needed to convert one tree into the other. By incorporating

structural and labeling information, *GumTree*'s Matcher is able to capture subtle tree-based changes more reliably than naïve string comparisons, making it particularly suitable for building refactoring tools, code differencing utilities, and merge operations that depend on AST-level intelligence.

Next, we demonstrate the *GumTree Matcher* using two similar versions of a Java class with minor edits. As shown in Figure 20 (Falleri et al., 2014), the left and right images represent the *before* and *after* versions of the code. *GumTree* identifies semantic-level changes, such as inserted or modified print statements and variable initializations. The *Matcher* highlights these transformations using color-coded bars and generates a mapping between corresponding nodes in the ASTs of both versions.

This fine-grained differencing enables our *Patch Alignment* algorithm to understand not just what changed, but how it changed structurally by providing the precision needed to merge user-edited and patched code with minimal conflict.



```
public class AccountManager {
    private String name;
    private double outstanding;

    void printOwing() {
        // Print banner
        System.out.println("Details of account");
        System.out.println("---");
        System.out.println("");

        // Print details.
        System.out.println("name: " + name);
        System.out.println("amount: " + outstanding);
    }

}
```

```
public class AccountManager {
    private String name;
    private double outstanding;

    void printOwing() {
        // Print banner
        System.out.println("Details of account");
        System.out.println("***");
        System.out.println("");

        // Print details
        printDetails();
    }

    void printDetails() {
        System.out.println("name: " + name);
        System.out.println("amount: " + outstanding);
    }

}
```

```
private String position;
private JButton closeBox;
private JButton menuBtn;
private ButtonGroup buttonGroup;
private JToggleButton nullButton;
private int dimension;
private ArrayList dockables;
private ArrayList buttons;
private DockableWindowManager.Entry current;
private JPopupMenu popup;
```

```
private String po
private JButton c
private JButton m
private ButtonGro                up;
private JToggleBu                ton;
private int dimension;
private List dockables;
private List buttons;
private DockableWindowManager.Entry current;
private JPopupMenu popup;
```

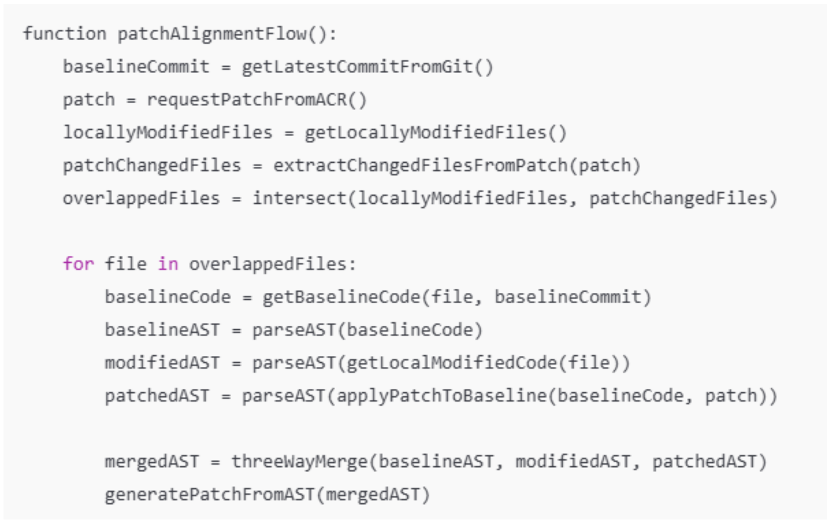
Figure 20: Example of GumTree Matcher (source: Falleri, Morandat, Blanc, Martinez, & Monperrus, 2014)

6.2 Algorithm of Patch Alignment

6.3.1 Overview of the Algorithm

The whole algorithm can be divided into 3 parts as shown in Figure 21: Prepare 3 versions of AST (*parseAST*); Merge 3 versions of ASTs into a merged AST (*threeWayMerge*); Convert merged AST back to source code and apply the merged code in the editor (*generatePatchFromAST)*. The algorithm starts by comparing the `Patch`'s affected paths (file paths in *diffs*) to the user-modified files, the system detects "*intersection files*", which have both local modifications and lines changed by the `Patch`. Everything else is treated more simply, since either the patch or the user did not touch it. For each intersection file, the system retrieves: (a) *baseline code / AST*: the baseline version from the latest remote commit in the branch, (b) *modified code / AST*: the user's current local file, and (c) *patched code / AST*: a Patched file obtained by applying the `Patch` text to the baseline. Each version is then parsed into a *GumTree* AST.

Once the three versions of ASTs are constructed, ACR Plugin systematically merges them. The procedure first identifies matching nodes across *baseline*, *modified*, and *patched* ASTs and returns a triple of matching nodes (3 versions of AST). Iterating through all the matching nodes, it either merges changes if they can be cleanly combined, or flags merge conflicts otherwise. Finally, after the merge, a finalized AST is converted back to source code. That code is written to the user's editor and highlighted to show changed or newly introduced lines.



```
function patchAlignmentFlow():
    baselineCommit = getLatestCommitFromGit()
    patch = requestPatchFromACR()
    locallyModifiedFiles = getLocallyModifiedFiles()
    patchChangedFiles = extractChangedFilesFromPatch(patch)
    overlappedFiles = intersect(locallyModifiedFiles, patchChangedFiles)

    for file in overlappedFiles:
        baselineCode = getBaselineCode(file, baselineCommit)
        baselineAST = parseAST(baselineCode)
        modifiedAST = parseAST(getLocalModifiedCode(file))
        patchedAST = parseAST(applyPatchToBaseline(baselineCode, patch))

        mergedAST = threeWayMerge(baselineAST, modifiedAST, patchedAST)
        generatePatchFromAST(mergedAST)
```

Figure 21: Main Workflow of Patch Alignment

6.3.2 Implementations

6.3.2.1 Prepare 3 versions of ASTs

To prepare 3 versions of ASTs: *baseline AST, modified AST, and patched AST*, the code first repeatedly references Git-based helper functions and enumerates local modifications to see which files overlap with the files modified by the Patch. Only these files can cause potential mismatches, so we only need *Patch Alignment* for these files. Using *GumTree*'s JDT parser on large files can be computationally significant, but the code mitigates overhead by focusing only on changed files that are necessary – they are represented as *overlappedFiles* shown in Figure 21. In projects with thousands of files, the intersection step ensures that only relevant files are parsed into ASTs. Caching or partial updates could reduce overhead further, but the existing approach is likely sufficient for moderate-size codebases. These steps effectively limit expensive AST merges only to files that are truly changed, which is crucial for performance in larger projects.

The function *parseAST* simply utilizes *Tree Generator* in *GumTree* to parse either file contents or raw strings into Java ASTs. Specifically, the parser relies on the *JdtTreeGenerator* to build a *TreeContext*, whose root node is the base of the AST. Because Java code is statically typed, using GumTree's JDT-based tree generator helps preserve structure. It is sufficiently robust to detect classes, methods, variable declarations, and even certain AST-level constructs like method calls and conditionals. With all three versions of the code now in AST form, node-by-node matching and conflict detection become significantly more precise than a naïve line-based diff.

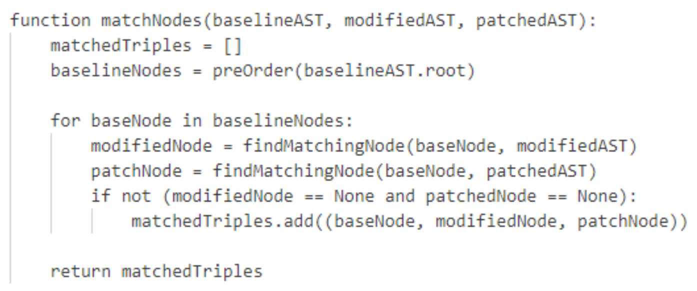
6.3.2.2 Three-Way Merge of ASTs

In the *patch alignment* algorithm, we leverage the *Matcher* to produce *mapping stores* for both the *baseline* versus *modified* code and the *baseline* versus *patched* code. These *mapping stores* reveal which nodes remain the same, which have changed, and which are entirely new, allowing us to build a triple-based merge that integrates both user changes and automatically generated patches. The *Matcher*'s edit script and mapping data thereby guide how we detect unmatched nodes, confirm whether a particular *baseline* node exists in both *modified* and *patched* versions, and ensure that newly added code is correctly anchored to an established parent in the final *merged* AST.

Now we move on to the heart of *patch alignment* lies in how three ASTs - *baseline*, *modified*, and *patched* ASTs - are merged. We will use pseudo code first to explain the idea of *Three-Way Merge*:

1. **Mapping Baseline to Modified/Patch**: In function *matchNodes* (see Figure 22), *GumTree Matcher* is invoked twice (*findMatchingNode)* - first matching *baseline* to *modified* AST, then *baseline* to *patched* AST. The resulting `Map` let the program know, for

each *baseline* node, which node in the *modified* or *patched* AST corresponds to the "same" code element. For every *baseline* node in a pre-order traversal, the program creates a `NodeTriple` that includes the *baseline* node, its matched counterpart in the *modified* AST (if any), and its matched counterpart in the *patched* AST (if any). If both users and the `Patch` delete a batch of code, reflecting in AST level by removing a node, no triple of that node is formed because the node is effectively agreed upon to be deleted by both users and ACR Agent. Simply not returning this triple will guarantee that the node is not added to the final merged AST.



```
function matchNodes(baselineAST, modifiedAST, patchedAST):
    matchedTriples = []
    baselineNodes = preOrder(baselineAST.root)

    for baseNode in baselineNodes:
        modifiedNode = findMatchingNode(baseNode, modifiedAST)
        patchNode = findMatchingNode(baseNode, patchedAST)
        if not (modifiedNode == None and patchedNode == None):
            matchedTriples.add((baseNode, modifiedNode, patchNode))

    return matchedTriples
```

Figure 22: Pseudo Code for `matchNodes`

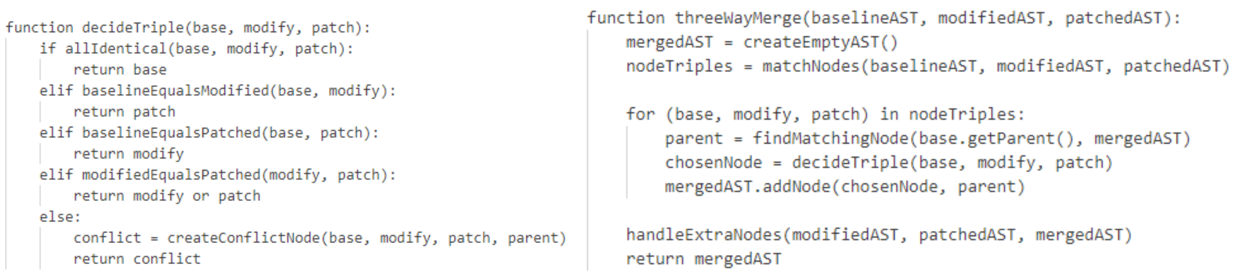
2. **Node-Level Decision**: In *decideTriple* (see Figure 23), the system tests whether the *baseline* node is *isomorphic* (identical in structure) to the *modified* node, or to the *patched* node, or whether the *modified* and *patched* nodes match each other. A naive preference is used:

    - If all three match, the *baseline* node is reused.
    - If *baseline* matches only *modified*, the *patch* node's changes take precedence.
    - If *baseline* matches only *patched*, the *modified* node's changes take precedence.
    - If *patched* and *modified* match each other but not *baseline*, that new version is used.
    - Otherwise, it defaults to the *baseline* and flagged with a *conflict marker*.

    This logic can be extended to handle more nuanced or domain-specific tie-breakers, but for general merges, it provides a straightforward resolution approach.

3. **Rebuilding the *Baseline* Hierarchy**: After deciding which node to keep, the system copies the chosen node into the *merged* AST. Children are reattached in an order consistent with the *baseline* to preserve a stable structure. This ensures that even if the `Patch` modifies a later sibling in the code, the node order remains consistent with the user's original layout unless explicitly changed. After this step, all the nodes in *baseline*

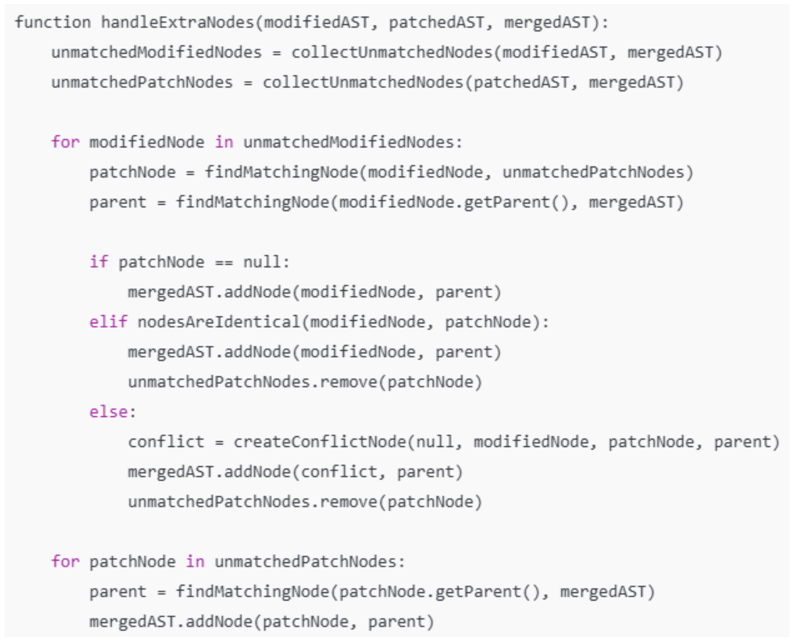
AST will either be added to the preliminary *merged* AST, or they are suggested to be deleted by both users and `Patch`, which will simply not be added. Therefore, we preserve all the information in the *baseline* code as much as needed.



```
function decideTriple(base, modify, patch):
    if allIdentical(base, modify, patch):
        return base
    elif baselineEqualsModified(base, modify):
        return patch
    elif baselineEqualsPatched(base, patch):
        return modify
    elif modifiedEqualsPatched(modify, patch):
        return modify or patch
    else:
        conflict = createConflictNode(base, modify, patch, parent)
        return conflict
```

```
function threeWayMerge(baselineAST, modifiedAST, patchedAST):
    mergedAST = createEmptyAST()
    nodeTriples = matchNodes(baselineAST, modifiedAST, patchedAST)

    for (base, modify, patch) in nodeTriples:
        parent = findMatchingNode(base.getParent(), mergedAST)
        chosenNode = decideTriple(base, modify, patch)
        mergedAST.addNode(chosenNode, parent)

    handleExtraNodes(modifiedAST, patchedAST, mergedAST)
    return mergedAST
```

Figure 23: Pseudo Code for Three-Way Merge



```
function handleExtraNodes(modifiedAST, patchedAST, mergedAST):
    unmatchedModifiedNodes = collectUnmatchedNodes(modifiedAST, mergedAST)
    unmatchedPatchNodes = collectUnmatchedNodes(patchedAST, mergedAST)

    for modifiedNode in unmatchedModifiedNodes:
        patchNode = findMatchingNode(modifiedNode, unmatchedPatchNodes)
        parent = findMatchingNode(modifiedNode.getParent(), mergedAST)

        if patchNode == null:
            mergedAST.addNode(modifiedNode, parent)
        elif nodesAreIdentical(modifiedNode, patchNode):
            mergedAST.addNode(modifiedNode, parent)
            unmatchedPatchNodes.remove(patchNode)
        else:
            conflict = createConflictNode(null, modifiedNode, patchNode, parent)
            mergedAST.addNode(conflict, parent)
            unmatchedPatchNodes.remove(patchNode)

    for patchNode in unmatchedPatchNodes:
        parent = findMatchingNode(patchNode.getParent(), mergedAST)
        mergedAST.addNode(patchNode, parent)
```

Figure 24: Pseudo Code for Handle Extra Nodes

4. **Handling Newly Added Nodes**: Once *baseline*-based merges are processed, the system calls *handleExtraNodes* (see Figure 24). This step collects all nodes in both the *modified* and *patched* ASTs that had no corresponding *baseline* node, meaning users or the `Patch` introduced something entirely new. For each "*top-most unmatched*" node, the system climbs up the parent chain looking for an ancestor that existed in the *baseline*. Once found, the new node is added under the ancestor's merged counterpart in the final *merged* AST. If users and the `Patch` both introduce the same new node without conflicts (based on isomorphism checks), it is added to the *merged* AST. Otherwise, it can trigger conflict

insertion or fallback logic. After this step, all the *baseline* nodes that should be kept, and all the newly introduced nodes from either users or `Patch` that do not have conflicts, will appear in the final *merged* AST in the right order.

6.3.2.3 Applying the Merged AST

When the *Three-Way Merge* is complete, the program performs a *pretty-print* method which coverts AST back to the source code with correct format and indentations. Currently it calls out to an LLM to read and parse the AST structure to Java source code string. Although a language model is invoked here, the main point is that the AST is systematically flattened back into code, preserving changes. The plugin then overwrites the corresponding files with the source code in the user's editor. To aid comprehension, the plugin also highlights lines that are changed or added.

However, the current implementation of applying *merged* AST is going to be replaced soon by leveraging a function that iterates through the AST structure and uniformly parses it to source code, followed by running a code-formatter to handle the indentations and other formats. When *modified* and *patched* nodes encounter merge conflicts, the function *createConflictNode* will select *baseline* node by default, and flag it with a *conflict marker*. This *conflict marker* will be implemented to store information about how *modified* and *patched* nodes conflict with each other. While traversing the AST to parse it back to source code, if encountered a *conflict marker*, the plugin will display a merge conflict UI to highlight the conflict to users and require user intervention to resolve it.

6.4 Case Study of Patch Alignment

We will use a case study to better illustrate how *Patch Alignment* works.

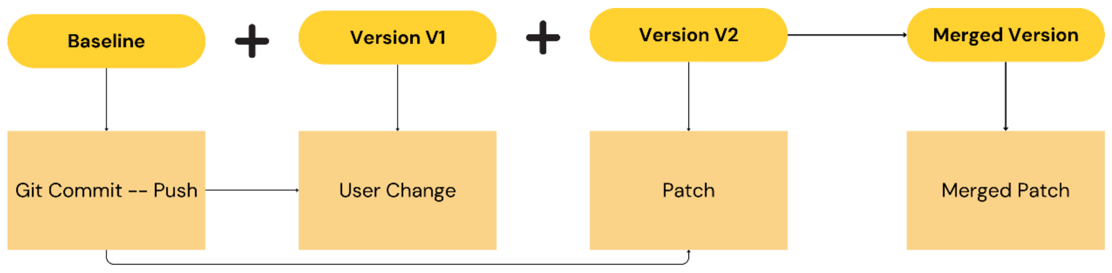
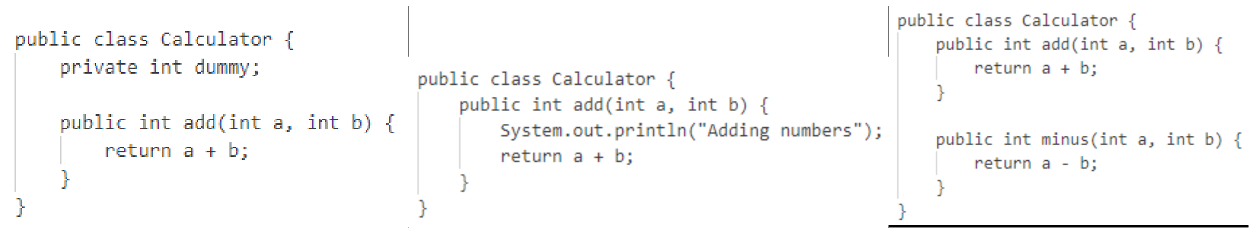


Figure 25: Patch Alignment Workflow

Suppose a user commits and pushes their code - this latest pushed commit will be marked as *baseline.* The user then continues modifying the project locally without pushing the new changes. At this point, the user triggers AutoCodeRover to fix a bug. ACR will run and generate a Patch based on the baseline version, unaware of the user's ongoing local changes. When the Patch is returned, the ACR Plugin detects a mismatch between the current local code and the Patch. When users click on "Apply" button to apply the Patch, ACR Plugin automatically invokes **Patch Alignment** to reconcile the differences and safely apply the Patch.

6.4.1 No Conflict

Suppose we have the 3 versions of code as shown in Figure 25: *baseline*, *modified*, *patched* code, with *Baseline* code corresponding to the *Baseline* in Figure 26; *Modified* code corresponding to *Version V1* in Figure 26; and *Patched* code corresponding to *Version V2* in Figure 26, where ACR Plugin has already applied the Patch to the *baseline*. Now *Patch Alignment* starts by parsing three versions of codes to three Abstract Syntax Trees, and triggers *Three-Way Merge* of the three ASTs.



```
public class Calculator {
    private int dummy;

    public int add(int a, int b) {
        return a + b;
    }
}
```

```
public class Calculator {
    public int add(int a, int b) {
        System.out.println("Adding numbers");
        return a + b;
    }
}
```

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int minus(int a, int b) {
        return a - b;
    }
}
```

(a) Baseline Code *(b) Modified Code* *(c) Patched Code*

Figure 26: Case Study of Patch Alignment - No Conflict

6.4.1.1 matchNodes

During the *Three-Way Merge*, the algorithm (*matchNodes*) begins by iterating through each node in the *Baseline* AST using pre-order tree traversal. For every node, it attempts to find corresponding matches in both the *Modified* and *Patched* ASTs using the *Matcher* (*findMatchingNode*). These matched nodes form a *triple* consisting of the *baseline* node, its counterpart in the *modified* version, and its counterpart in the *patched* version.

For example, as illustrated in Figure 26, the first node in baseline AST is a Class declaration. The *Matcher* successfully identifies the corresponding nodes in both the *Modified* and *Patched* ASTs, and a triple is created with these three nodes.

Next, the traversal encounters a variable declaration in *Baseline* AST. The algorithm fails to find matching nodes in either the *Modified* or *Patched* AST - both return null. A *double-null* indicates that both the user and ACR agree that this node should be deleted, so no triple is returned for this node.

For the rest of the code, such as the nodes within the add function, the algorithm successfully finds matches in both the *Modified* and *Patched* ASTs. As a result, these are returned as valid triples for further merging.

6.4.1.2 decideTriple

After collecting all the *triples*, the algorithm proceeds to determine which node from each *triple* should be included in the *merged* AST. Starting with the class declaration, all three nodes - *baseline*, *modified*, and *patched* - are identical. In this case, the algorithm simply reuses the *baseline* node. Next, it evaluates the *add* function. Since there are no changes in either the *Modified* or *Patched* ASTs, the nodes remain the same as the *baseline*. As a result, the *baseline* nodes are again selected and added directly to the *merged* AST.

6.4.1.3 handleExtraNodes

After the algorithm finishes adding all nodes in *baseline* to the *merged* AST, the *handleExtraNodes* function begins traversing the *Modified* and *Patched* ASTs to identify any unmatched nodes - that is, nodes that do not exist in the *Baseline* AST. These represent newly added code elements introduced by either the user or ACR.

The algorithm first processes unmatched nodes in the *Modified* AST. In our example, it encounters a new print statement. It checks whether this node exists in the *Patched* AST by attempting to find a matching node. If none is found (i.e., it returns null), this means the node is unique to the *Modified* version and does not conflict with the *Patch*. As such, it can be safely added to the *merged* AST. After completing the traversal of the *Modified* AST, the algorithm moves on to the unmatched nodes in the *Patched* AST. If, for instance, the same print statement is also found in the *Patched* AST and is structurally identical, the system recognizes it as already added via the *Modified* AST and removes this node in the unmatched nodes of *Patched* AST so that when iterating the *Patched* AST, this node will be skipped to avoid duplication. This deduplication step improves efficiency by ensuring the node isn’t processed twice.

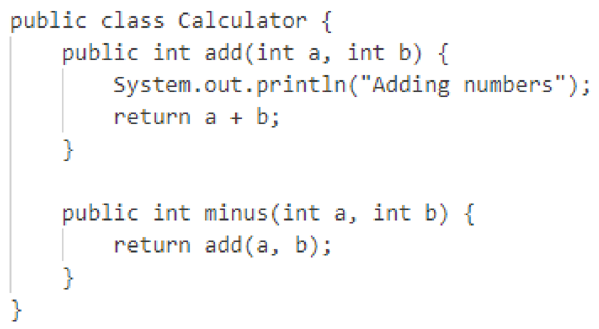
The algorithm then continues processing the remaining unmatched nodes in the *Patched* AST. In the example, a new function *minus* is introduced only in the Patch, and it does not exist in either the *Baseline* or *Modified* ASTs, so it is considered conflict-free. Since it's entirely new and independent, the node can be safely added to the merged AST.

6.4.2 Merge Conflict

Now we demonstrate a case where a merge conflict is detected. Suppose the *Baseline* and *Patched* code remain the same as shown in Figure 26. However, we update the *Modified* code as shown in Figure 27, where the user also adds a new *minus* function. In this modified version, the return statement within *minus* is implemented as a function call to *add*.

Everything proceeds same as described in *Chapter 6.4.1* until the *handleExtraNodes* phase. While traversing the *Modified* AST, the algorithm encounters the newly added *minus* function. It then checks for a corresponding node in the *Patched* AST and successfully finds a match, since the Patch also introduces a function named minus. However, when comparing the internal structure of the return statements, a difference is detected: in the *Modified* AST, the return value is a function call, while in the *Patched* AST, it is a direct addition of two integers. The *GumTree Matcher* identifies these nodes as not structurally identical.

According to the algorithm, when a newly added node exists in both the *Modified* and *Patched* ASTs but differs in content or structure, neither version is merged. Instead, the conflict is flagged. This flagged conflict is later processed during the application phase, where the plugin will highlight the conflict in the editor and prompt the user to manually resolve it.



```
public class Calculator {
    public int add(int a, int b) {
        System.out.println("Adding numbers");
        return a + b;
    }

    public int minus(int a, int b) {
        return add(a, b);
    }
}
```

Figure 27: Updated Example of User Modified Code – Conflict

6.5 Reflection of Patch Alignment

Patch Alignment in the ACR Plugin represents a robust, AST-based solution to a familiar development challenge: reconciling locally changed code with automated patches generated against a stale *baseline*. By mapping each *baseline* node to its counterparts in the user's version and the `Patch`, the plugin thoroughly determines which lines have truly changed, which have been removed, and which are newly added. A final pass merging unmatched nodes ensures that wholly new functions or classes are not lost.

On the implementation side, *GumTree*'s structural matching avoids spurious text-based conflicts. The result is a streamlined merging experience that consistently preserves the user's code while incorporating patch improvements. Although there are ways to refine the conflict-resolution logic or incorporate direct machine learning models for improved merging suggestions, the current approach is both transparent and extensible. It provides immediate value to developers who rely on automated *patch generation* and ensures that the overhead of stale patches is minimized. By bridging the gap between a developer's ongoing edits and `Patch` output, this *Patch Alignment* strategy stands as an effective blueprint for managing concurrent code changes in modern continuous-integration and code-suggestion pipelines.

Chapter 7. Maintainability of the Extension

7.1 Overview

This chapter presents the comprehensive evaluation of the features and program developed in this project, detailing the testing strategies and results obtained for both the ACR Plugin and the ACR Agent backend. The real evaluation process encompassed unit testing, integration testing, and manual testing of user interactions within the IDE. Even though the chapter does not demonstrate everything, **all** the features introduced in the project were continuously tested throughout the implementation phase, which ensured that each component functions correctly both in isolation and as part of the overall workflow. Furthermore, the robustness of the backend enhancements, particularly the self-fix and feedback mechanisms, was rigorously validated through controlled experiments and real-world task simulations. Regarding code quality, both the ACR Plugin and ACR Agent projects have GitHub Actions set up for continuous integration (CI) and are integrated with SonarQube for ongoing code quality checks.

7.2 Unit and Integration Testing for the ACR Plugin

The ACR Plugin was subjected to an extensive suite of unit tests targeting its core functionalities. These tests verified individual functions such as configuration persistence, state serialization/deserialization, and UI component behavior. For instance, unit tests confirmed that the integration of *GumTree Parser* can parse either a *File* object or a source code in *String* to an AST object. Subsequently, given two ASTs, the *Matcher* is able to return a `Map` of matched nodes, including the information about how two ASTs are different.

Integration tests were then performed to validate the communication between the plugin and the ACR WebConsole. We simulated full API interactions by invoking HTTPS requests and verifying that the plugin successfully received JSON payloads containing the *subscribe_link* and *runId*. These tests ensured that the plugin could reliably establish an SSE connection to receive real-time updates from the ACR WebConsole. In addition, the plugin's background listeners were tested to ascertain that they correctly capture and process build and test failure events. The file path of test failure report can be successfully identified from the error messages collected by *test failure listener*, and it can correctly parse the test failure details from the report.

7.3 UI and Workflow Testing for the ACR Plugin

A critical aspect of the project was ensuring that the plugin's user interface provided a smooth and intuitive experience. Manual testing was carried out by launching the ACR Plugin within the developer mode. During these sessions, we verified the following, though our testing was not limited to them:

1. **Interactive Elements**: The chat window dynamically updates with both user and system messages, and all buttons (e.g., Send, Stop, Reset, Apply) are responsive. Special attention was paid to the typewriter effect for system messages and the dynamic card-panel swapping between Send and Stop buttons, ensuring that the interactive experience remained fluid throughout extended ACR runs.
2. **Feedback Features**: Feedback buttons associated with model responses were tested for proper operation. When a user clicks on a feedback button, the corresponding model response identifier (UUID) is correctly attached and transmitted to the ACR endpoint. The system enforces that only one response can be actively critiqued at any time, and that subsequent selections properly overwrite previous feedback, thereby avoiding conflicts. And we provided two consecutive feedback from ACR Plugin to verify whether the interactive ACR Agent had knowledge of the previous feedback history in the feedback mechanism.
3. **Stop and Rate APIs:** We tested that when users *thumb-up* or *thumb-down* a patch from ACR Plugin, ACR WebConsole can correctly display the rating results, which reflects that the integration of rating API works correctly in real-time. And we also tested that when users stop an ACR `Run`, the status of a task is correctly reflected to be "*Canceled"* on ACR WebConsole.
4. **Patch Alignment and Application**: The *patch alignment* functionality was tested under scenarios where the local code had diverged from the *baseline* used to generate the patch. We confirmed that the alignment algorithm successfully merged user modifications with the patch diff and highlighted the modified lines within the IDE editor.
5. **Error Handling**: We verified that when a task fails or is interrupted unexpectedly, ACR Plugin removes visual cues that had indicated an ongoing process, and displays the error

message in the chat history. This keeps the interface clear and immediately reflects that the process has been halted, eliminating any potential confusion for the user.

7.4 Evaluation of ACR Agent Enhancements

On the ACR Agent backend side, testing focused on the enhanced features of AutoCodeRover that underpin the interactive capabilities of the system. The following aspects were evaluated:

1. **Feedback Loop Functionality:** We manually injected JSON feedback into the `state` and initiated *replay* runs to verify that the *Context Retrieval Agent* and *Patch Generation Agent* correctly incorporate and respond to user feedback. The experiments confirmed that when feedback is provided, the corresponding `Agent` reloads its `state` and re-executes its logic, updating downstream outputs accordingly.

2. **Self-Fix Agent Behavior:** The self-fix mechanism was tested using a series of complex, real-world tasks designed to produce non-applicable patches. By analyzing the error messages captured during these runs, the *Self-Fix Agent* was able to diagnose the source of the failure, generate targeted corrective feedback, and trigger a *replay* starting from the problematic `Agent`. The iterative self-fix process was observed to progressively refine the `Patch` until an applicable solution was produced.

7.5 Maintainability

A core consideration throughout the design and implementation of this project was ensuring long-term maintainability and extensibility. As the ACR Plugin and ACR Agent continue to evolve as part of a larger ecosystem - integrating with LLMs, IDE APIs, and backend services - clear code structure and developer-oriented documentation are essential for enabling seamless future development by others. For both ACR Agent backend and ACR Plugin, maintainability is reinforced by a solid CI pipeline powered by GitHub Actions and SonarQube, which provides continuous feedback on code quality and potential regressions, encouraging disciplined development practices.

7.5.1 ACR Agent Backend

Each new entry points of Agent introduced in interactive ACR is implemented as an independent class with clearly defined responsibilities, which employs standardized interfaces and clearly defined contracts for each agent. The core logic of feedback handling is encapsulated within distinct agents, avoiding any tight coupling with the primary execution pipeline. This means that future enhancements - such as support for new agent types or advanced feedback classification - can be introduced without altering the internal logic of unrelated components.

Similarly, the extended algorithm – *Self-Fix Agent* that can autonomously detect, analyze, and recover from errors significantly enhances system resilience - follows a modular architectural pattern, encapsulating individual functionalities into separate agents and clearly defining their scope, allowing developers to enhance or modify each module independently without unintended side effects. A consistent state-transition model is used throughout the agents, ensuring that each modification to the system state is deliberate, traceable, and reversible. By systematically producing new states without mutating existing ones, the design allows easier debugging and facilitates replayability. The structured logging mechanisms incorporated across components further support traceability, making diagnostics straightforward in complex workflows.

7.5.2 ACR Plugin

The project demonstrates strong maintainability through a well-structured and modular architecture. Core components such as actions, listeners, parsers, settings, and UI elements are cleanly separated into logically defined packages, making the codebase easy to navigate and scale. The integration layer is equally well-organized, with key responsibilities like API communication, SonarLint analysis, and build/test failure handling encapsulated within clear, single-purpose methods. This promotes both readability and ease of extension.

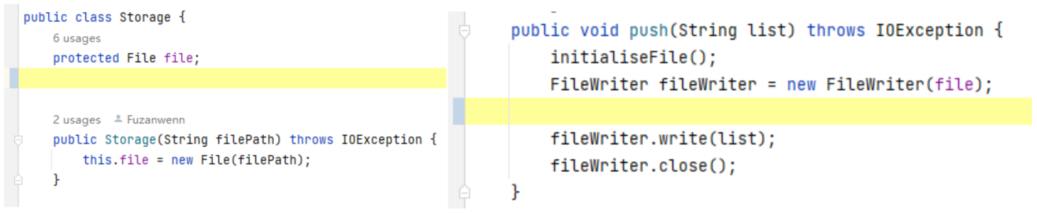
Resource management is handled robustly through a centralized utility that loads external dependencies, such as JAR files, ensuring consistency and reducing redundancy across the codebase. Error handling and logging follow consistent, pragmatic patterns, offering useful insight during development and debugging. Notably, coroutine lifecycles are managed using structured concurrency, allowing for controlled cancellation of background tasks, which is an essential consideration for preserving IDE responsiveness and stability. In summary, the ACR plugin’s maintainability is underpinned by modular design, thoughtful resource and lifecycle management, and strong CI integration. Moving forward, the project would benefit from adopting structured logging, expanding test coverage, and formalizing coding standards to further support its long-term scalability and robustness.

Chapter 8. Real-World Example

In this chapter, we will demonstrate how the ACR Plugin seamlessly integrates the automatic solution AutoCodeRover into a real-world IDE scenario. Consider a simple Java project named after Java's iconic mascot, *Duke*. At this point, we have completed the initial implementation of the project, and now we intend to leverage SonarLint to analyze our codebase for potential quality issues.

8.1 SonarLint Analysis

Upon executing the SonarLint analysis, the ACR Plugin identifies several issues, as illustrated in Figure 17. We specifically focus on the problematic file: *Storage.java*. To examine the issues, we click on each file's label panel and expand to reveal the respective details. We then click the "*Run*" button to initiate the automatic resolution for the issue depicted in Figure 17b, "*Remove this unused 'input' private field*", the ACR Plugin successfully resolves the problem in *Storage.java* by removing the unused private variable input in the returned Patch (see Figure 13). We can effortlessly incorporate this improvement by applying the provided Patch directly to the editor (see Figure 28).



```
public class Storage {
    6 usages
    protected File file;


    2 usages  Fuzanwenn
    public Storage(String filePath) throws IOException {
        this.file = new File(filePath);
    }
```

```
public void push(String list) throws IOException {
    initialiseFile();
    FileWriter fileWriter = new FileWriter(file);

    fileWriter.write(list);
    fileWriter.close();
}
```

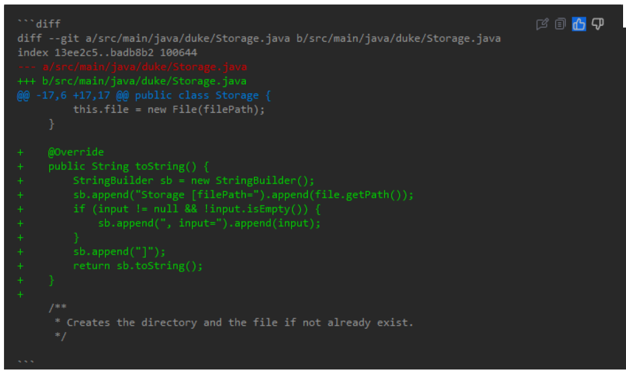
Figure 28: Apply Patch for Storage.java

8.2 Add a Feature

Now we realize that *Storage.java* is missing a *toString* method, so we prompt ACR to automatically implement it for us. Figure 29a shows the Patch successfully returned from ACR. Since the *toString* method compiles and runs without errors, we give the Patch a *thumb-up*.

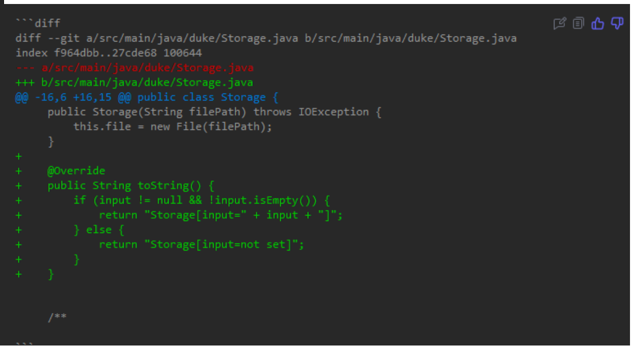
8.3 Provide Feedback

However, there are some issues with its current implementation. Specifically, the *toString* method in the *Storage* class does not fully align with the intended behavior. It is supposed to return a string representation of the *Storage* object in the format *Storage[input=<value>]* when the input field is set, and *Storage[input=not set]* when it is null or empty. Currently, the implementation has two main issues: It includes the *filePath* in the output, which is not part of the intended format; It does not properly handle the case when the input field is unset, failing to return *Storage[input=not set]* as expected. Without pushing a commit of the latest change, we can directly give feedback to Patch in Figure 29a to let ACR refine its answer. And ACR gives an improved Patch based on user feedback (see Figure 29b).



~~~
```diff
diff --git a/src/main/java/duke/Storage.java b/src/main/java/duke/Storage.java
index 13ee2c5..badb8b2 100644
--- a/src/main/java/duke/Storage.java
+++ b/src/main/java/duke/Storage.java
@@ -17,6 +17,17 @@ public class Storage {
         this.file = new File(filePath);
     }

+    @Override
+    public String toString() {
+        StringBuilder sb = new StringBuilder();
+        sb.append("Storage [filePath=").append(file.getPath());
+        if (input != null && !input.isEmpty()) {
+            sb.append(", input=").append(input);
+        }
+        sb.append("]");
+        return sb.toString();
+    }
+
     /**
      * Creates the directory and the file if not already exist.
      */

```
~~~

(a) Patch: First Version



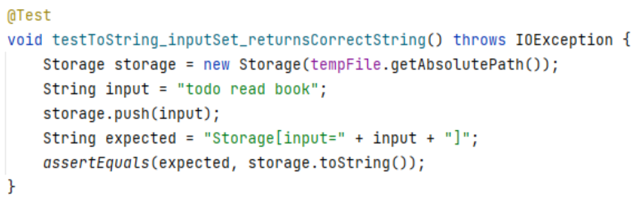
~~~
```diff
diff --git a/src/main/java/duke/Storage.java b/src/main/java/duke/Storage.java
index f964dbb..27cde68 100644
--- a/src/main/java/duke/Storage.java
+++ b/src/main/java/duke/Storage.java
@@ -16,6 +16,15 @@ public class Storage {
     public Storage(String filePath) throws IOException {
         this.file = new File(filePath);
     }
+
+    @Override
+    public String toString() {
+        if (input != null && !input.isEmpty()) {
+            return "Storage[input=" + input + "]";
+        } else {
+            return "Storage[input=not set]";
+        }
+    }


     /**

```
~~~

(b) Patch: Refined Version

Figure 29: Patch For toString Method

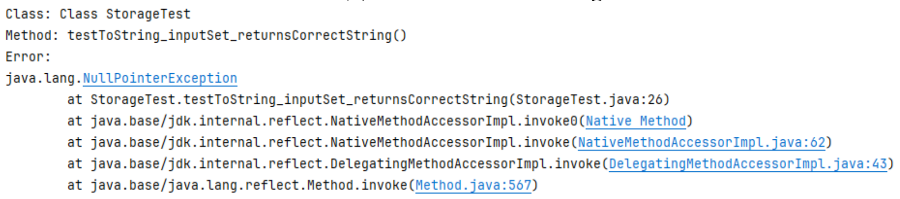
8.4 Test Failure Issue

We applied the enhanced *toString* patch and wrote a unit test to verify its correctness, as shown in Figure 30a. The test failed upon execution, and the ACR Plugin automatically captured the failure message (Figure 30b). We then triggered ACR to help fix the failed test in StorageTest.java. In the end, ACR suggests creating a dummy initialization of the object *tempFile*, which solves the issue.



```
@Test
void testToString_inputSet_returnsCorrectString() throws IOException {
    Storage storage = new Storage(tempFile.getAbsolutePath());
    String input = "todo read book";
    storage.push(input);
    String expected = "Storage[input=" + input + "]";
    assertEquals(expected, storage.toString());
}
```

(a) Unit Test For toString



```
Class: Class StorageTest
Method: testToString_inputSet_returnsCorrectString()
Error:
java.lang.NullPointerException
	at StorageTest.testToString_inputSet_returnsCorrectString(StorageTest.java:26)
	at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
	at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
	at java.base/java.lang.reflect.Method.invoke(Method.java:567)
```

(b) Test Failure Messages

Figure 30: Test Failure For toString

Chapter 9. Conclusion

This project marks a significant step toward transforming traditional software maintenance workflows by embedding autonomous program improvement directly into the development environment. The work presented herein extends the AutoCodeRover framework - hitherto a promising prototype of automated bug fixing and feature addition - into a fully interactive tool that integrates seamlessly with modern IDEs. By coupling advanced language models with program analysis techniques, ACR not only generates code fixes but also iteratively refines them based on user feedback, thereby reducing the manual overhead in debugging and patch review.

A key contribution of this work is the design and implementation of the ACR Plugin, which acts as the local interface between developers and the ACR Agent. The plugin features a modern, chat-based user interface that displays system messages, intermediate analysis, and patch suggestions from ACR Agent. By leveraging IDE internal APIs to detect common development events like test failures and build failures, with the integration of SonarLint static analysis, the plugin brings AutoCodeRover solution to benefit developers in multiple scenarios in the IDE context. An advanced *Patch Alignment* strategy is engineered to apply a `Patch` even with mismatch or conflicts of context, which better enhance efficiency and productivity for developers when using ACR service at the IDE.

ACR backend is improved by integrating multiple layers of feedback - both at the *context retrieval* and *patch generation* stages - ensuring that the `Agent` is responsive to developer input. In addition, the introduction of a *Self-Fix Agent* further enhances the system's autonomy. When a generated `Patch` is inapplicable, the *Self-Fix Agent* autonomously analyzes the failure, determines the likely source of error among the participating agents, and initiates a *replay* mechanism to correct the issue without requiring full manual intervention.

The integration of the ACR Plugin with ACR WebConsole and ACR Cloud further illustrates the commitment to maintaining a persistent, traceable history of interactions and system states. Each model response is uniquely identified and stored, enabling a transparent feedback loop that not only aids in patch refinement but also supports post-run analysis and quality assurance.

While the current implementation has demonstrated promising efficacy in resolving real-world GitHub issues and improving developer productivity, several limitations remain. The system's reliance on iterative feedback loops, though effective, may be further optimized through more sophisticated prompt engineering and enhanced state management. Future enhancements could include expanding support for additional programming languages, integrating more robust conflict-resolution techniques in patch alignment, and leveraging reinforcement learning to adapt feedback mechanisms dynamically.

Looking ahead, this work lays the groundwork for an AI-native software engineering paradigm. By evolving from traditional, one-shot automation toward an interactive, self-correcting framework, AutoCodeRover embodies the principles of Software Engineering 3.0 - where intent-first, conversation-oriented development becomes the norm. This vision, supported by robust integration into modern development environments, promises to significantly reduce the cognitive load on developers and improve code quality, ultimately leading to more efficient and reliable software maintenance practices.

In conclusion, the advancements detailed in this report demonstrate a viable pathway to fully autonomous program improvement. The convergence of interactive IDE plugins, iterative agent feedback, and self-fix capabilities represents a meaningful stride in the evolution of AI-assisted software engineering, setting the stage for future research and practical adoption in industry.

References

Bird, J., Zhang, B., Liang, P., Zhou, X., Ahmad, A., & Waseem, M. (2023). Practices and challenges of using GitHub Copilot: An empirical study. *Empirical Software Engineering*, 1-22.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., & Monperrus, M. (2014). Fine-grained and accurate source code differencing. *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE '14)* (pp. 313–324). Västerås: ACM/IEEE. Retrieved from https://doi.org/10.1145/2642937.2642982

Ruan, H., Zhang, Y., & Roychoudhury, A. (2024). *SpecRover: Code Intent Extraction via LLMs.* arXiv, Ithaca, NY. Retrieved from https://doi.org/10.48550/arXiv.2408.02232

Sergeyuk, A., Koshchenko, E., Zakharov, I., Bryksin, T., & Izadi, M. (2024). *The Design Space of in-IDE Human-AI Experience.* Delft, Netherlands: JetBrains Research and Delft University of Technology.

Yuan, Q., Nadi, S., Nguyen, T. L., & Sandoval, L. (2024). A study on JetBrains AI Assistant and its usability for software development. *Journal of Software Engineering Research*, 12-35.

Zhang, B., Liang, P., Zhou, X., Ahmad, A., & Waseem, M. (2023). Study reveals common issues faced by GitHub Copilot users. *Software Engineering Notes*, 39-47.

Zhang, B., Liang, P., Zhou, X., Waseem, M., & Ahmad, A. (2023). Demystifying Practices, Challenges and Expected Features of Using GitHub Copilot. *International Journal of Software Engineering and Knowledge Engineering*, 1653–1672.

Zhang, Y., Ruan, H., Fan, Z., & Roychoudhury, A. (2024). AutoCodeRover: Autonomous Program Improvement. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)* (pp. 1–13). New York, NY: ACM. Retrieved September 2024, from https://doi.org/10.1145/3650212.3680384

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., . . . Stoica, I. (2023). *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena.* arXiv. Retrieved from https://doi.org/10.48550/arXiv.2306.05685