AutoCodeRover: Agentic Program Repair for SonarQube Issues

Martin Mirchev*
martin.mirchev@u.nus.edu
National University of Singapore
Singapore

Ridwan Shariffdeen
shariffdeenr@acm.org
SonarSource Singapore
Singapore

Haifeng Ruan
haifeng.ruan@u.nus.edu
National University of Singapore
Singapore

Yuntong Zhang
zhang.yuntong@u.nus.edu
National University of Singapore
Singapore

Abhik Roychoudhury
abhik@nus.edu.sg
National University of Singapore
Singapore

Abstract

Agentic systems have been gaining traction in solving software engineering tasks. These tasks span from writing documentation, fixing faults in the codebase, and developing new features. A promising application of LLM agents is addressing software "issues", with an issue capturing a unit of improvement needed in a software project. Issues can be detected and constructed by static analysis tools, such as SonarQube. Static analysis tools frequently generate a substantial number of reports related to security vulnerabilities and code quality, imposing a significant manual workload on developers. With the advances in agentic AI, there is potential to automatically remediate these issues, thereby reducing developer effort.

In this paper, we present our experience and lessons learned in adapting the AutoCodeRover program improvement agent to automatically propose patches for issues reported by SonarQube. We name this new agent SonarQube Remediation Agent, specialized for fixing SonarQube issues. SonarQube Remediation Agent is designed to be capable of interacting with mission-critical codebases in a secure and trustworthy manner. We discuss our approach in tackling practical challenges such as handling large volumes of issues and designing seamless user interactions. SonarQube Remediation Agent is integrated into the software development lifecycle by suggesting patches during the pull request review workflow, enabling developers to efficiently improve software quality and security with SonarQube.

CCS Concepts

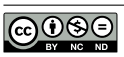
• **Software and its engineering → Software evolution**; **Automatic programming**; *Automated static analysis.*

Keywords

Static Analysis, Program Repair, AI Agents

ACM Reference Format:
Martin Mirchev, Ridwan Shariffdeen, Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2026. AutoCodeRover: Agentic Program Repair for SonarQube Issues. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 5–9, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3803437.3805209

*First author was a full-time employee at SonarSource when this work was conducted.



FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805209

1 Introduction

Software development practitioners have sought high velocity. To ensure that velocity does not come at the cost of correctness, validation methods are applied throughout the Software Development Lifecyle (SDLC), from the Integrated Development Environment (IDE) all the way to code review. These validation methods often rely on automated tools to detect issues. One widely used tool for code security and quality validation is SonarQube. SonarQube is positioned in the Quality Assurance step of the SDLC, before the Deployment stage, to ensure the product adheres to a pre-defined development standard. The development standard embodied by SonarQube captures rules for style consistency, maintainability, reliability, and security. SonarQube imposes these rules on the software project through *Quality Gates*. Quality Gates can be applied to existing or new code in the project, and report issues detected by SonarQube static analysis to developers. These issues are defined to be actionable and unambiguous, enabling developers to take immediate action. Once issues are reported by the Quality Gate, developers usually manually fix them by introducing additional code changes. This hinders product development velocity and requires developers to spend time iterating over the existing code. However, it is essential to address these issues during the development phase to prevent security vulnerabilities from reaching production and to avoid the accumulation of code quality-related technical debt.

Automating the remediation of issues detected by static analysis tools is therefore highly attractive. Recent advances in LLM-based agents have demonstrated promising results in automating a range of software engineering tasks, including program repair [6, 23, 25], test generation [1, 2, 21], and other general development tasks [3, 4, 16, 24]. However, automatically resolving SonarQube-reported issues in enterprise settings presents new challenges. First, the number of issues reported by Quality Gate can vary substantially across projects and code changes. In practice, a single Quality Gate scan may report hundreds of issues, making it time- and resource-intensive to invoke an agent independently for each issue. The issues vary widely in complexity, ranging from trivial issues to intricate problems that may require several hours of developer effort to resolve. Secondly, SonarQube is widely adopted by large enterprises in finance, healthcare, and many other domains. For

an agent to be trusted by developers in enterprise environments, it must uphold high security standards, integrate seamlessly with existing workflows, and present a low barrier to adoption.

Given these challenges or requirements, an agent with controlled autonomy and a strong emphasis on explainability and program analysis is preferable. To construct such an agent, we build upon the existing AutoCodeRover agent [25] and extend it to the domain of static analysis issue remediation. AutoCodeRover follows predefined workflow rather than relying on highly permissive terminal command execution, as seen in agents such as Claude Code [3] and Codex [16]. It integrates program analysis into its workflow, which provides more grounded reasoning in conjunction with the LLM. Building on AutoCodeRover, we develop the SonarQube Remediation Agent to address issues reported by SonarQube by introducing components to handle large volumes of issues and enforce security protections. SonarSource acquired the AutoCodeRover technology as a spinoff acquisition in early 2025, after which we evolved it into the SonarQube Remediation Agent, which has been in beta testing since October 2025.

The SonarQube Remediation Agent takes in a codebase and a collection of issues reported by SonarQube. It distributes the issues into separate tasks, each of which is addressed by an agent in an isolated environment. To build trust in the agent-generated patches, we employ an iterative loop that leverages SonarQube to verify that no regressions are introduced during remediation. Once all tasks are completed, the SonarQube Remediation Agent generates explanations for each patch fragment (or "hunk"), which help developers understand the rationale behind the proposed changes. These patches, along with their explanations, are provided to developers for review before committing to the codebase. These patches can be provided as suggestions or a new pull request.

The remaining sections of this paper present background on AutoCodeRover and SonarQube, the design of the SonarQube Remediation Agent, and the lessons learnt throughout its development.

2 Background

2.1 AutoCodeRover Agent

AutoCodeRover [25] is an LLM agent designed for software engineering tasks like bug fixing and feature addition. It aims to resolve software engineering issues in a realistic setup, where only a natural-language description of the issue/requirement is available. One such setup is GitHub issues, where users submit bug reports or feature requests for a software project.

Figure 1 shows the workflow of AutoCodeRover. Given a codebase C and a natural-language issue report NL, AutoCodeRover autonomously produces a patch R which aims to resolve the issue described in NL. From the issue report, AutoCodeRover begins the main loop by gathering context and generating a reproducer test case that aims to exercise the fault. Typically, an issue contains only a natural-language description and no reproducer test. This generated test can act as an additional specification for the patch generation stage.

The goal of the context retrieval stage is to extract code snippets relevant to the issue NL. This allows the LLM to better understand the issue in relation to the code. AutoCodeRover performs context retrieval using a set of structure-aware search tools, enabling the LLM to search as a developer would. These tools mirror everyday search actions a developer might use when exploring the codebase, e.g., `search_function(...)` and `search_method_in_class(...)`. Upon receiving such an API request, the backend of AutoCodeRover searches for the actual code/signature in the Abstract Syntax Tree (AST) representation of the codebase and returns the code/signature to the LLM. This process of invoking search APIs and enriching the code context occurs iteratively until the LLM deems the current context sufficient to understand the issue.

At the end of the context retrieval stage, the agent selects a set of buggy locations from the accumulated code context. These locations are provided to a patch generation agent, which crafts candidate patches to resolve the issue. After a candidate patch is generated, an LLM-as-a-Judge reviewer is used to determine whether it resolves the issue. If the patch is deemed sufficient to resolve the issue, the workflow concludes with that patch as the output artifact. Otherwise, a natural-language "suggestion" from the reviewer is presented to the patch-generation agent to iteratively improve the patch. A natural way to evaluate whether the patch resolves the issue is to execute the reproducer test on the patched program. The reviewer agent takes in both the candidate patch and the reproducer test, executes the test on the patched code base, and decides on (1) whether the test successfully reproduces the issue, and (2) whether the patch successfully resolves the issue. Both artifacts are subject to iterative refinement because an LLM generates them and may be incorrect.

If no acceptable patches are generated after several rounds of review, AutoCodeRover returns to the context retrieval stage to rediscover buggy locations and generate a new set of patches. This process continues for a predefined number of rounds.

2.2 SonarQube Static Analyser

SonarQube [20] is a static analysis tool that supports 30+ programming languages, including Java, Python, C#, C/C++, JavaScript, TypeScript, ABAP, and more. SonarQube can find issues ranging from simple syntactic issues based on abstract syntax tree (AST) patterns to security vulnerabilities, such as cross-site scripting (XSS). To run SonarQube, a scanner is executed on a long-lived branch or on a pull request. Futhermore, the scanner can be provided a code coverage map of the project. In the following, we focus on the pull request analysis.

The scanner communicates with an instance of SonarQube Server or SonarQube Cloud to collect a list of analyzers that can be executed on the project. An analyzer contains the logic for checking for rules in a language or a class of languages. After the analyzers complete, the scanner sends a snapshot of all issues, the current coverage map, and the changes in the pull request to the SonarQube Server. Using this information, the SonarQube Server can identify new issues in the pull request, report coverage for the latest code, and detect any introduced code duplication. The values of these metrics are then compared against a set of constraints stored on the SonarQube Server instance. These constraints are known as a quality gate. Examples of such constraints are: (1) the number of security issues introduced by the PR is more than five; (2) the

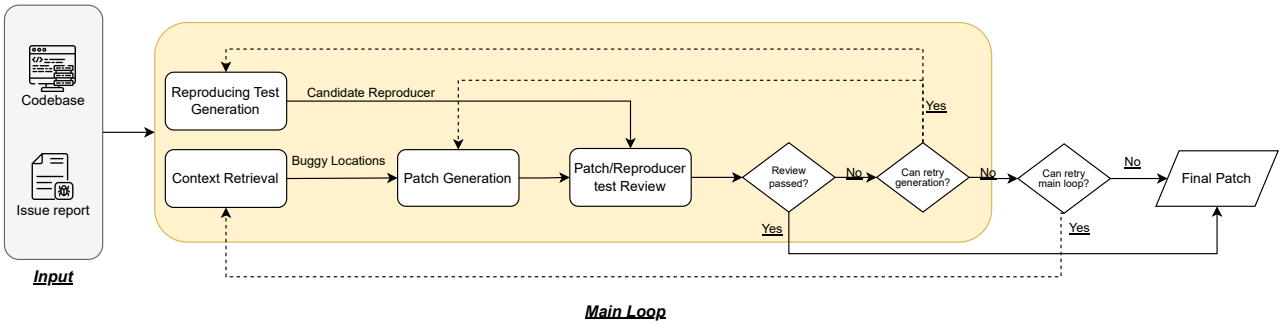


Figure 1: AutoCodeRover workflow for resolving GitHub issues. Dotted lines are back edges in the workflow.

percentage of code duplication the PR adds is more than 10%; (3) the number of issues that are classified as blockers is non-zero.

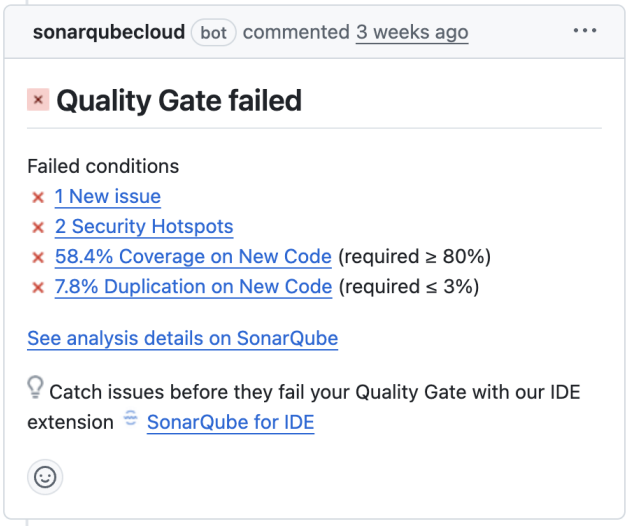


Figure 2: SonarQube Cloud Quality Gate failure comment in GitHub

An example quality gate failure is shown in Figure 2. In this failure, we can notice the following.

- the developer has introduced a new issue;
- the developer has introduced two security hotspots; a security hotspot is a security issue that has a high probability of being a false positive and requires manual investigation;
- there is insufficient coverage (the quality gate requires ≥ 80% coverage), and
- the pull request introduces a significant amount of code duplication.

Having an automated solution for resolving quality gate failures can save developers' time. Developers can then shift their focus from fixing issues to developing new features and reviewing proposed fixes.

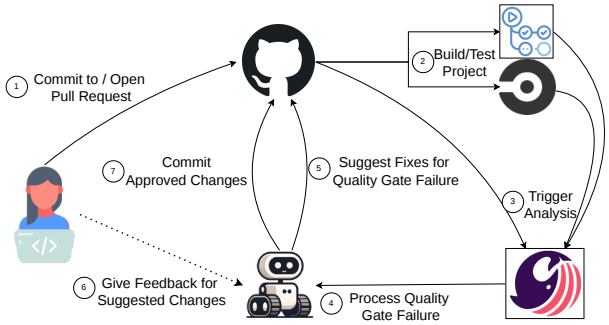


Figure 3: AutoCodeRover and SonarQube in the Software Development Life Cycle.

3 SonarQube Remediation Agent

Figure 3 presents the location of the SonarQube Remediation Agent in the SDLC. When a developer creates a pull request (PR) or commits to an already existing PR (1), the version control system can trigger the Continuous Integration systems (2). These systems will build the project, execute the test suite, and subsequently trigger the SonarQube analysis with artifacts from the previous steps (3). In the event of a Quality Gate failure, the SonarQube Remediation Agent is triggered to resolve the issues (4). Upon completion of the agent, the resulting artifacts are submitted to GitHub (5), allowing the developer to review them and provide feedback (6). If a developer accepts a patch, the agent infrastructure will commit the changes to the branch representing the Pull Request (7).

Figure 4 illustrates the internal workflow of the SonarQube Remediation Agent. Given a codebase C and a collection of SonarQube issues I, the SonarQube Remediation Agent autonomously produces a patch R which aims to resolve the issues in I and a collection of explanations E_R for each "hunk" in the patch R.

The agent first splits the issues into a collection of *partitions* where each partition contains a subset of the issues. How the partitioning is done is described in subsection 3.1. For each partition, the agent executes a context-retrieval stage, and the discovered bug locations are passed to a patch-generation stage. Once patch generation is complete, the agent validates that the generated patch resolves the issue using SonarQube analysis as an oracle. Once all

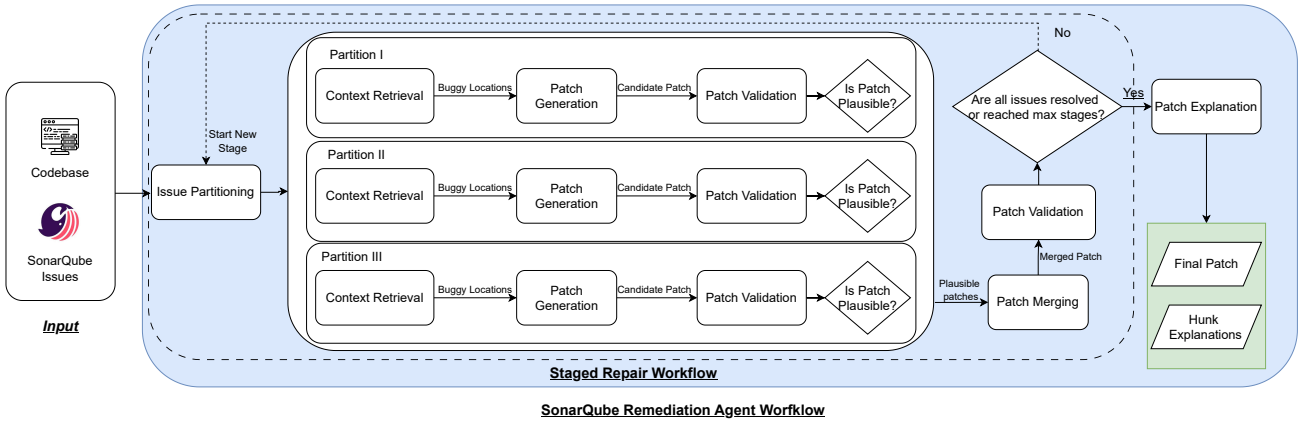


Figure 4: SonarQube Remediation Agent workflow for resolving SonarQube issues. Dotted lines are back edges in the workflow.

partitions have produced a candidate patch, all plausible patches are merged, as described in subsection 3.2. We define a plausible patch as one that does not break the file's syntax, does not introduce regressions, and resolves at least one issue for the partition. The merged patch is validated as described in subsection 3.3 to check whether any regressions were introduced during the merge or if any issues remain. If the merged patch validation identifies a regression or unresolved issues, the agent will again partition the remaining issues and start new agent instances on them. We refer to this retry loop as the *staged repair loop*, which can run for up to a predefined number of rounds. Otherwise, the agent extracts the accumulated patch, generates explanations for each hunk, and presents them to the user as a per-issue suggestion comment, as described in subsection 3.4.

3.1 Issue partitioning

The number of issues reported by a SonarQube Quality Gate can range from a few to several hundred. These issues also vary in severity and complexity. Assigning an agent to each issue is ineffective because each agent may need to gather the same context for issues that involve nearby code locations, resulting in wasted compute cycles and LLM calls. On the other hand, invoking an agent once to resolve all the issues in one run forces the agent to tackle multiple independent goals at the same time, and can make it lose track of its work.

To address this problem, we implemented an issue-partitioning scheme based on issue complexity and abstract syntax tree (AST) locality. All SonarQube issues have a primary location that identifies the file and the specific line where it is introduced. In addition, security vulnerabilities such as taint-based security issues can include an auxiliary set of "flow" locations that are critical control-flow points: the source, call sites that propagate the taint, and the sink.

During issue partitioning, the agent first classifies issues as single-location (no flows) or multi-location. Each multi-location issue will be processed as a standalone partition, since each of them may require a distinct code context. For single-location issues, we split them into different partitions based on AST locality. Specifically, issues within the same AST component are grouped into the same partition, where the granularity of the AST component can be adjusted (e.g., at the file, class, or function level). From an implementation standpoint, since the agent has already indexed the entire codebase for context retrieval, the issue partitioning can reuse the same index for AST locality-based grouping.

3.2 Patch Merging

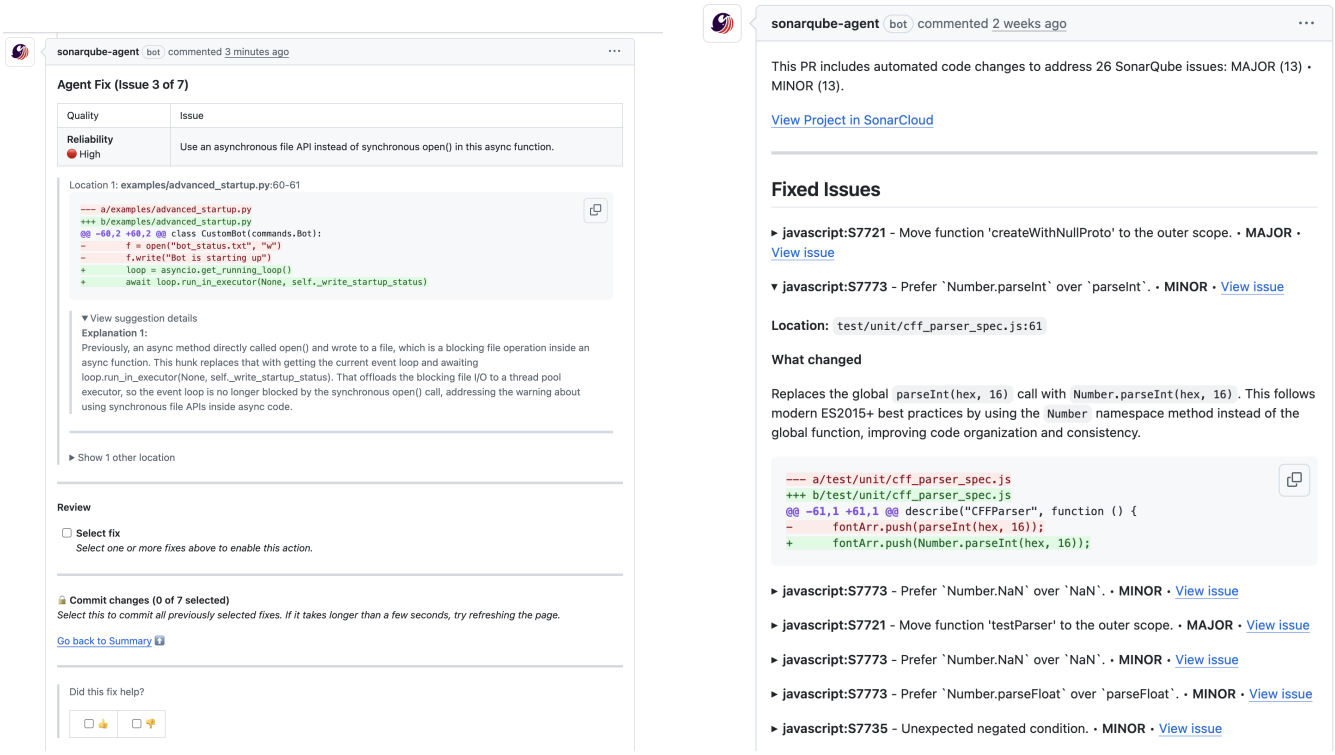
Recall that issue resolution is handled in partitions, and once all issue partitions are resolved, the staged program repair iteration is completed. Upon completion of a staged repair iteration, the agent must merge the final plausible patches for each partition of that stage. Without merging patches, developers cannot apply multiple patches targeting the same location. This also allows the agent to build on previous work to address remaining issues and to continuously improve the codebase, thereby resolving all issues that block the quality gate. A simple first approach for patch merging is to apply the patches using the version control system. If the patch application fails, then there is a case of the agent instances modifying the same location. To solve this problem, we have developed an agent to rewrite the patches that failed to apply in the new context.

3.3 Patch Validation

With the SonarQube analyzers at our disposal, we can create an oracle to validate that the agent has fixed the issue and whether any regressions have been introduced. This validation ensures users that a patch will resolve issues without increasing the developers' workload. This validation oracle is invoked at two different places in the SonarQube Remediation Agent: (1) in each partition's workflow to prevent patches that introduce regressions from being merged; (2) to evaluate whether regressions were introduced during the merge process and whether any issues remain for the next staged repair iteration.

3.4 Presenting the Artifacts

Noller et al. [15] found that patch explanations help developers when reviewing code generated by an APR tool. As previously mentioned, a quality gate failure can encompass multiple issues.



(a) As a comment suggesting a fix in the Pull Request. **(b) As a separate Pull Request.**

Figure 5: Example patches generated by the SonarQube Remediation Agent.

Therefore, having a single explanation per agent patch is not effective. A better approach is to include an explanation for each patch hunk. At the end of agent execution, we employ a patch-explanation agent that provides an explanation for each hunk and how it helped resolve specific issues. The SonarQube Remediation Agent currently supports two ways of presenting patches to the developers: (1) either as suggestions in the current pull request or (2) as a pull request containing all the fixes.

Fixes as Pull Request Comments. Figure 5a illustrates the patch presentation interface utilizing pull request comments for the SonarQube Remediation Agent. The figure contains the target issue, the proposed patch, its location, and an explanation. In Figure 5a, the developer is provided with a patch for a reliability issue of high severity for using a synchronous I/O API in an asynchronous context. They can select the fix to accumulate all suggestions the agent has presented into a single commit, which can be applied to the codebase through the "Commit changes" button, which appears once at least one fix is selected. Furthermore, agent users can provide feedback for patches via a survey or a simple thumbs-up/thumbs-down.

Fixes as Pull Requests. Figure 5b illustrates the patch presentation interface utilizing pull requests for the SonarQube Remediation Agent. The figure describes the issues the agent has fixed, their severity, a short summary of the issue, and a link to the SonarQube Analysis. Furthermore, for each issue, a dropdown can be expanded to present the changes created to resolve it and explanations. In Figure 5b, the developer is presented with a Pull Request in which SonarQube Remediation Agent has fixed 26 issues. Not shown in the image is that the Quality Gate analysis indicates the agent has not introduced any new issues in their code. Furthermore, the developer who had created the pull request is assigned as a reviewer of the agent-generated pull request.

Compared to the Pull Request comments, the Pull Request is presented as a way to: (1) evaluate the agent's changes in a new pull request; (2) allow addition of user changes before merging the fixes; (3) provide commits, which can be "cherry-picked" to other branches.

4 Design Considerations in Enterprise Agent Deployment

While developing the agent, we identified several critical factors for successful adoption. We expand on the more notable ones.

4.1 Agent Autonomy

Most widely used agents for software development are designed to be fully autonomous systems with near-unlimited capabilities. The agents can execute any command and have full access to the file system. This can have detrimental effects when users are not careful, as seen in the past year [1]. Common approaches to reducing the

[1] https://www.pcmag.com/news/vibe-coding-fiasco-replite-ai-agent-goes-rogue-deletes-company-database

impact of the agent include sandboxing and interactive guardrails, which prompt the user before executing a command. The SonarQube Remediation Agent operates autonomously on sensitive code, preventing the creation of interactive guardrails. Therefore, the agent runs in a sandbox and is made secure by limiting the capabilities of the stages and tools it uses. Furthermore, we have worked with the security team to audit the agent.

4.2 Protection against package hallucinations

One recent issue observed with agents is that they can hallucinate package names in generated code, potentially enabling a supply chain attack. This is colloquially known among developers as slop squatting [2]. To address this, we have implemented an import guard that discards any patches that use a non-standard library or an unknown import. This is a conservative approach that reduces the risk of breaking the compilation pipeline. Most SonarQube issues do not require additional imports to be resolved. Issues in which external imports may be needed include some security rules, deprecation warnings, and attempts by the model to import a helper function or class.

4.3 Agent Observability

Where the SonarQube Remediation Agent sits in the SDLC is critical, as it has access to clients' source code. This requires that the agent can be debugged and its behavior be explainable. To prevent the propagation of any anomalous agent behavior into the codebase, we have implemented robust observability and agent behavior tracking. Following standard security practices, we cannot log the contents of the codebase and how they influence the agent. To protect intellectual property (IP), we had to take a different approach. Given the exact actions the agent can take, we created custom rules that alert our engineers to anomalous behavior. These monitoring mechanisms function as security controls to mitigate the risk of agent misbehavior and enable proactive management of potential issues that may arise in production.

4.4 Protections against Prompt Injection

The SonarQube static analysis tool is used by many Fortune 500 companies [3], indicating that the analyzers are widely trusted and valued. During agent development, we implemented multiple security enforcement mechanisms within the agent. One notable measure is the prevention of prompt injection. Prompt injection attacks are becoming popular in LLM-based systems.[4] To carry out prompt injection attacks, attackers insert malicious instructions into either the user input or the environment (e.g., webpages) accessed by the LLM-based system. When analyzing potential sources of such attacks, we found that the codebase is a low-trust information source, with comments being a potential attack vector. To prevent comment-based injection, we hardened the prompts across the agents and applied comment obfuscation to protect them against such attacks.

[2] https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/slopsquatting-when-ai-agents-hallucinate-malicious-packages
[3] https://www.sonarsource.com/company/press-releases/sonar-record-growth-2022/
[4] https://brave.com/blog/comet-prompt-injection/

An example of such an obfuscation is provided in Figure 6. Before the Context Retrieval round starts, the SonarQube Remediation Agent code traverses the client codebase and locates all AST constructs that can be considered comments. The contents of the comments are replaced with a unique value, which can subsequently be safely reverted before presenting the patch to the developers or invoking the validation loop.

```
@@ -85,8 +85,8 @@
-    # Example Python Comment
+    # ACR-7c482371-cdd6-4907-...
```

Figure 6: Comment obfuscation example.

Although we have obfuscated the comments, the quality gate may be failing due to issues that depend on the contents of the comments. The SonarQube analyzers have rules that flag comments indicating future or incomplete work using keywords such as `TODO` (`S1135`) and `FIXME` (`S1134`). Comments containing such keywords indicate technical debt or partial implementation. Since the agent cannot see the comments, the agent is highly unlikely to resolve such issues. Therefore, we have disabled the agent from ingesting such issues.

4.5 Regressions v.s. Follow-On Issues

When an agent-generated patch is applied to the codebase and validated, SonarQube may report new issues. During development, we learnt that not all issues are *regressions* caused by an incorrect patch proposed by the agent – they can also be *follow-on* issues. Follow-on issues are those newly introduced by patches that implement new requirements or features, rather than those arising from changes to existing functionality.

We present an example of a follow-on issue in Figure 7. The developer has a statement that calls a computationally expensive function and prints the result to the standard output stream `System.out`. For this statement, SonarQube reported an issue for the rule `java:S106` - "Standard outputs should not be used directly to log anything". When the agent tries to resolve an issue from this rule, it must replace the usage of the standard stream with a logging library.

Concretely, the agent proposes the patch denoted by `+`, highlighted in green. The patch replaces usage of `System.out.println` with `LOG.debug`. For brevity, we assume a logger is available in this example. We note that the agent implemented a new logging mechanism to resolve the issue. When running the SonarQube validation on the proposed patch, a follow-on issue for rule `java:S2629` - "Logging arguments should not require evaluation" is raised. The agent prepares a new patch denoted by `++`, which is colored blue. The patch inserts a conditional statement that serves as a guard for the logging statement, disabling it when the target function is executed without requiring debug logging. The follow-on issues and regressions are handled automatically by our staged repair loop, as described in section 3.

```
@@ -85,8 +85,8 @@
     // java:S106 "Standard outputs should
           not be used directly to log anything"
-    System.out.println(
-       "This operation will be expensive: \n" +
-       resource_intensive_function() );
     // java:S2629 - "Logging arguments
           should not require evaluation"
+    LOG.debug(
+       "This operation will be expensive: \n{}",
+       resource_intensive_function() );
     // No issues raised
++   if(Logger.isLoggable(DEBUG))
++      LOG.debug(
++      "This operation will be expensive: \n{}",
++      resource_intensive_function() );
```

Figure 7: Example of an additional patch to a follow-on issue.

5 System Walkthrough

A full execution of the agent can demonstrate the benefits it provides. We will use the open-source PDF.js [5] reader by the Mozilla Foundation to demonstrate the agent's workflow. For rendering PDF documents, some files use the Compact Font Format (CFF) to store the document's fonts. A developer wants to add a CFF parser to the project. They create the source file and a test file that define the specifications, and then open a Pull Request. Upon the completion of the Pull Request analysis, we see the quality gate in Figure 8.

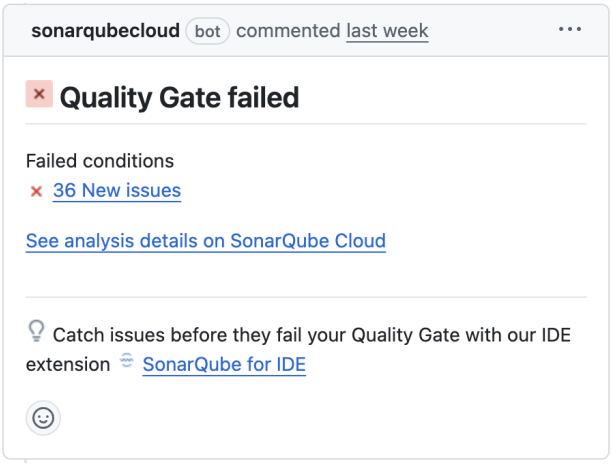


Figure 8: Quality Gate Failure for addition of CFF parser to PDF.js

In the pull request, the developer has issues in both the source and test files. These issues make the code harder to maintain:

- They are not following the ECMAScript 2015 format and are using global constants such as NaN;
- Incorrectly defining variables in blocks, leading to pollution of the method;
- Declaring variables that are not used;
- Classes with only a constructor;
- Inappropriate class field declaration;
- Functions in incorrect scopes;
- TODO comments and Cognitive complexity of the functions (not tackled by agent currently, further explained in subsection 6.3);

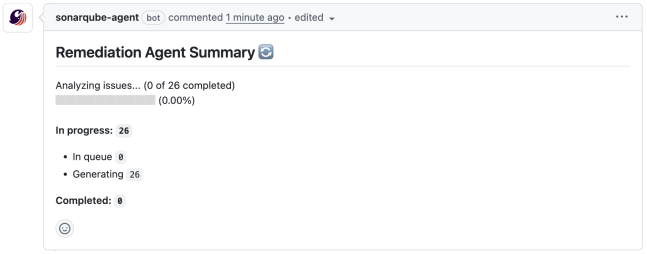


Figure 9: Initial Notification from SonarQube Remediation Agent

In the example provided, there will be a separate partition for each class and function in files that contain issues. The agent runs in this case with 15 partitions. The developers see the initial message in Figure 9 describing the number of issues the agent will handle. This comment is continuously updated while the agent runs, allowing developers to track its current state.

Upon completion of the agent, all the comments are submitted, and the initial comment is updated. We present the result in Figure 10. We observe that the agent took 10 minutes to solve issues. According to SonarQube data, manual repair is expected to take 91 minutes of a developer's time. The agent has saved an hour and a half of work by compressing it down to a quick, couple-minute-long review! After sending the patches in the pull request window, developers can quickly review each suggestion. Furthermore, if they wish, they can read its explanation and have the remediation agent infrastructure apply it if they deem it appropriate. An example patch is presented in Figure 11. It modifies two locations to ensure that any variable defined in the case block is properly scoped.

If developers do not want to commit the patch directly, they can make a local commit with the changes, and SonarQube will rerun

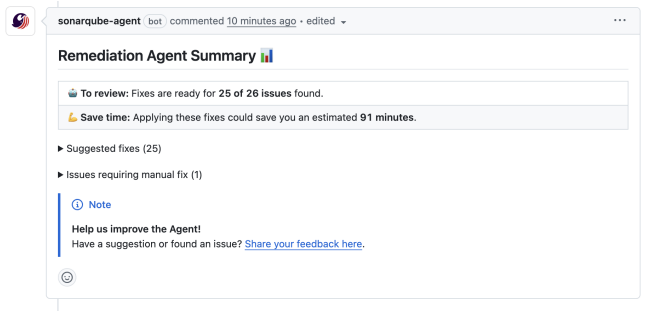


Figure 10: Summary comment sent by SonarQube Remediation Agent upon run completion

[5] https://github.com/mozilla/pdf.js

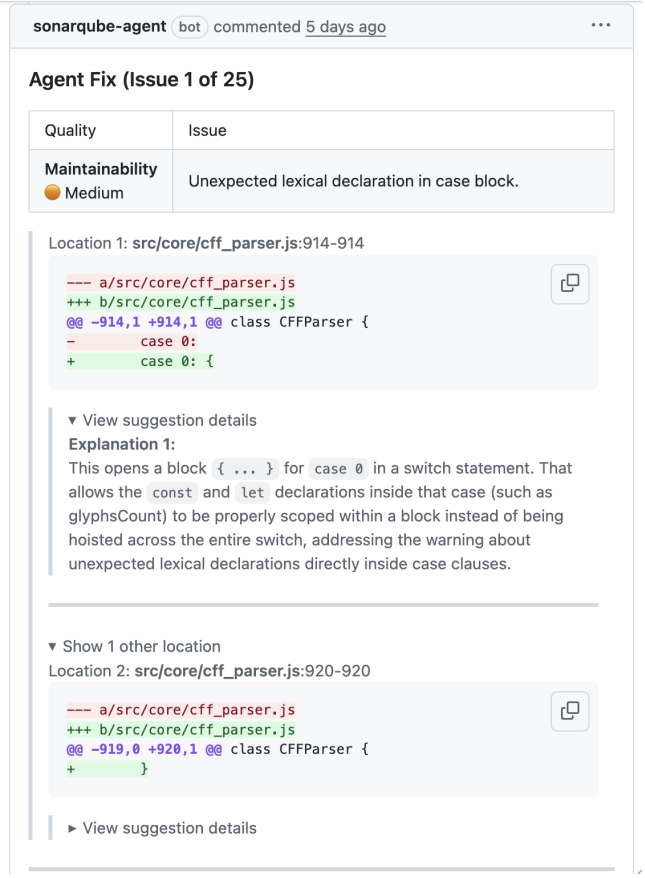


Figure 11: Multiple Location Patch proposed by the SonarQube Remediation Agent

the analysis with the new commits. When the analysis runs again, and a quality gate failure occurs, all unapplied agent comments will be deleted to reduce clutter.

The above shows the benefits of agentic repair while using the SonarQube analysis for code quality and security. Later in Section 6.3, we also discuss some difficulties we faced in our agentic repair implementation, vis-a-vis SonarQube issues, which were subsequently resolved.

6 Lessons Learnt

We now distill the key lessons learnt in our year-long effort to repurpose AutoCodeRover into the SonarQube Remediation Agent.

6.1 User Experience is Key

One of the most important lessons learned from the agent deployment is that user experience is key to successful adoption. We experimented with different approaches and patch presentations before settling on grouping patches by issue. This allows developers to reason about a specific issue using the artifacts. Furthermore, for quality gate failures with more than ten issues, the developer may be overwhelmed by the number of presented patches. Therefore, a minimal initial view of the issues has been proposed. This allows detailed explanations to be hidden until the developer chooses to examine the suggestion. Compared to a patch-centric view, which presents the issues for a specific hunk, this approach consolidates the repair intent into a single suggestion.

One limitation of this issue-based approach is the order in which patches are presented to the user. SonarQube has multiple rules for magic constants and duplicated literals, such as `S1192` - "String literals should not be duplicated", `S109` - "Magic numbers should not be used". Although such issues can occur in different partitions and address different problems, it is likely that the model will ultimately modify the same section of the file to define additional constants. These patches will be merged, and the hunk containing the constant definition will appear in both suggestions. When the developer accepts the patches, the hunk will be applied once, but on first inspection, they may notice the same content in different suggestions, which can be confusing.

The specific method to extract patches also plays an important role. Let us return to Figure 11. This suggestion visualizes the exact changes to the file, and we have split them into smaller modifications to make them easier to digest. Another way this could have been visualized is by presenting a "semantic patch" that shows how it affects the function's structure.

6.2 Security Considerations

Another important aspect is that there are many security considerations for deployment. The SonarQube Remediation Agent has access to mission-critical source code. Therefore, a workflow with clear attribution is necessary to ensure that responsibility and ownership are clear. Furthermore, we have made the agent workflow so that the PR author is the only person who can commit the agent changes. Having a proper audit trail for the agent has been crucial to confidence in our approach. Another important aspect of the SonarQube Remediation Agent is that it does not use a memory mechanism for repository state. This removes the risk of memory-poisoning attacks. Every agent instance starts with no previous knowledge of the repository. Furthermore, by not using developer feedback via comments, the SonarQube Remediation Agent is protected from another possible attack vector for prompt injections.

6.3 Adapting the Agent to Complex Issues

SonarQube reports issues of various complexities, and resolving the complex issues can be challenging for older or less capable models. An example of complex issues is on cognitive complexity of functions. Cognitive complexity is a SonarQube metric similar to cyclomatic complexity but more general in scope. Functions with high cognitive complexity reduce code maintainability; thus, it is important to fix these issues during initial development. In our initial experiments with older/less-capable LLMs, we observed that they had difficulty resolving such issues. We present an example of such an issue, as well as patches generated by the agent with different LLMs, in Figure 12. Function `f` is flagged by SonarQube as having high cognitive complexity due to extensive nesting of conditionals, exception handling, and loops. The original implementation of `f` is prefixed with `-` and colored in red.

In our initial experiments with an older LLM, the agent proposes the patch prefixed with `+` and colored in green. The proposed patch

extracts the heavy workloads into separate subfunctions as self-contained units. However, the patched function is still considered as having high cognitive complexity by SonarQube. This is because although code components are extracted as separate functions, the inner functions still incur deep code nesting. To improve patch quality for complex issues, we employ a combination of (1) incorporating more specific requirements in the issue description, (2) invoking the validation to iteratively improve the patch, and (3) switching to different LLM backends. In our subsequent experiments, the agent generated the patch that resolves the issue, which is prefixed with ++ and colored in blue in Figure 12. This new patch further reduces nesting by moving helper functions out of the scope of `f`. As this example demonstrates, equipping the agent with targeted enhancements enables it to resolve complex issues reported by static analysis.

```
@@ -20,40 +20,40 @@
def f(a):
-    if a:
-       while:
-         # Heavy Workload 1
-         # Heavy Workload 2
-    else:
-       # Light branch
+    def on_a(...):
+       def heavy_workfload_1(): # Heavy Workload 1
+       def heavy_workfload_2(): # Heavy Workload 2
+       while:
+           heavy_workload_1(...)
+           heavy_workload_2(...)
+    if a:
+       on_a(...)
+    else:
+       # Light branch
++   if a:
++      on_a(...)
++   else:
++      # Light branch
++ def heavy_workfload_1(...): # Heavy Workload 1
++ def heavy_workfload_2(...): # Heavy Workload 2
++ def on_a(...):
++     while:
++        heavy_workload_1(...)
++        heavy_workload_2(...)
```

Figure 12: Example of an issue that is difficult for the SonarQube Remediation Agent with older models

6.4 Issue tracking

For proper evaluation and analysis of agent performance, we need a clear view of the state of an issue. Throughout our development, we have learnt that this is a difficult problem. We discussed this matter with the SonarQube developers to better understand how they have solved it. SonarQube's goal is to detect code quality and security issues, and classifying an issue as new may be permitted. For our remediation agent, this is a more serious issue, because we must clearly determine whether a regression occurred or the agent failed to perform. Issue tracking is more critical for us, as we are not detecting issues but repairing detected issues. Furthermore, presenting a clear view of the progress in resolving issues to developers helps them better understand the agent's current state. The issue tracking helps in this regard.

6.5 Experimenting with different models

During our internal evaluations of the agent, we experimented with different models to achieve an optimal balance among cost, efficiency, and quality. An important difference we observed is that agent trajectories vary substantially across underlying models. For example, using a model such as Anthropic Claude has led the agent to perform more context-retrieval rounds than OpenAI's GPT in some cases. We attribute this difference in the agent to the alignment performed on the Claude models, which is designed to produce responses with higher confidence. Although this yields a performance improvement, it has not been significant for our use case and carries the risk of substantially increasing the agent cost. We have been sensitive about keeping agent costs low to enable wider use of the agent.

7 Related Work

Automated program repair [11] has been an active area of research in the software research community. Prior to the arrival of Large Language Models (LLMs), repair techniques have relied in meta-heuristic search [8], constraint based specification inference [14], machine learning [12], or its combinations. The work on AutoCodeRover [25] that we have repurposed, draws upon these works by seeking to infer simple specifications of intended behavior from the software project structure.

The AutoCodeRover [25] agent has shown success for various software engineering tasks, specifically error fixing and feature addition in software. The original work of AutoCodeRover and subsequent work on SpecRover [19] were evaluated on the SWE-Bench [9] benchmark, which comprises of issues in open source Python-based projects. CodeRover-S [26] is an enhancement of the AutoCodeRover agent that resolves issues found by the OSS-Fuzz project for C and C++-based projects. The SonarQube Remediation Agent has been enhanced to support languages such as Java, JavaScript, Python, and TypeScript.

Agentic systems have demonstrated the ability to assist developers in an interactive setting. Examples of such systems are GitHub Copilot [7], Claude Code [3], OpenAI Codex [16], and Devin [10]. Compared to them, the SonarQube Remediation Agent operates fully autonomously to address issues identified by SonarQube within the Continuous Integration Pipeline. The Passerine [17] agent, similar to the Remediation Agent, demonstrates that a minimal agent is sufficient to resolve many real-world issues in Google's codebases.

Non-LLM-based Automated Program Repair techniques have previously been applied in the Continuous Integration pipeline, in the context of large companies such as Meta [13] and Bloomberg [22], demonstrating promise in automatically resolving issues identified by analysis tools. Such works have relied on pattern-based fixes and

*[...] a significant one in terms of **technical debt time**: [...] This represents a reduction of ∼ 1.25K hours (∼ 4%), which is quite meaningful.* - *Company X*

Figure 13: Client feedback for clearing technical debt using the SonarQube Remediation Agent

- ***Overall impression**: "Really good, it helps a lot to keep the code clean and improve the quality of our deliveries."*
- ***Did it help?**: "Yes, it suggested logic adjustments and even improvements in function names and flows."*
- ***Points to improve**: "For code analysis on modified code, it's already good with improvement suggestions and so on."*

- Developer in Company X

Figure 14: Developer Feedback for the SonarQube Remediation Agent

are tailored to the company's ecosystem. The SonarQube Remediation Agent is a specialized agent that leverages LLM capabilities to repair SonarQube issues, which balances the need for both capability and trust in coding agents. Recent proposals such as the Unified Software Engineering agent (USEagent) [5] also seek to solve this tension by providing a rich set of actions to the agent, where each action is a meaningful software engineering activity.

8 Deployment and Initial Feedback

The SonarQube Remediation Agent has progressed through multiple deployment phases to better understand what would bring value to clients and to improve based on user feedback. Furthermore, we have been using the SonarQube Remediation Agent internally on SonarSource repositories to evaluate our proposed workflow. We launched an initial closed alpha with a small set of clients in July 2025 and expanded to a broader beta rollout in October 2025. Subsequently, starting in April 2026, the agent entered an open beta phase for SonarQube Cloud Enterprise customers. Across these phases, we maintained communication with participating clients to provide support, gather feedback, and iterate. In parallel, we increased the number of clients the agent is enabled for.

The staged rollouts have delivered positive results, and the SonarQube Remediation Agent is currently in Open Beta Access[6]. Clients and other teams within the SonarSource organization have reported that the agent has been crucial in saving time resolving quality-gate failures and enabling them to focus on developing new features. A sample feedback from a client is shown in Figure 13. It highlights the hours of technical debt saved by the SonarQube Remediation Agent when resolving the highest-severity issues, known as `BLOCKER`s. Furthermore, one of the developers in the same organization provided feedback, shown in Figure 14. These deployments and feedback have strengthened our confidence in the agent's capabilities and helped validate that it is safe for use in large enterprise codebases.

9 Disclaimer

Following standard confidentiality practices, we do not refer to any specific client organizations or proprietary code. The paper focuses on conveying key insights from the challenges of transferring AutoCodeRover technology to SonarQube Remediation Agent.

10 Perspectives

The work on AutoCodeRover [25] that we have repurposed for resolving SonarQube issues is an LLM-agent. It is however, a different kind of LLM agent, more grounded in software engineering. Instead of giving an agent access to a terminal where it can type in any command, we give the agent access to meaningful actions from a software engineering point of view - such as searching for code in a class. This leads to an agent with more coarse-grained autonomy, and yet engenders higher trust. This is the key aspect of the agent which enabled us to deploy it in production.

The reader can refer to a recent opinion piece [18] exploring the various dimensions of trust in AI software engineers, including technical trust and human trust. While the market size for coding agents continues to grow (∼25B by 2030 as per Mordor Intelligence[7]), adoption remains low as reported in several studies (such as only 14% of developers using agents daily as per the 2025 StackOverflow developer survey). We thus feel that there is significant disquiet in the software engineering (SE) community and among developers in general. This is somewhat contrary to the excitement around coding agent capability in the artificial intelligence (AI) community and among the broader society. Our work and experience can be seen in exactly this light - it deftly navigates the tension between capability and trust in coding agents. Instead of arguing about the importance of trust via empirical studies, we take a different approach. We seek to demonstrate the possible co-existence of both capability and trust in coding agents in a constructive fashion by building a widely used remediation agent technology.

The SonarQube Remediation Agent reported in this paper is capable of resolving a wide variety of code quality and security issues, and yet engenders trust by enabling acceptance of the automatic patches by client organisations. This provides a valuable positive experience in using agentic AI in production-quality code.

Acknowledgements

The authors would like to acknowledge the product, marketing, security, legal teams and all involved at Sonar for their invaluable contributions to the development and successful deployment of the system described in this work.

This work was achieved via a National University of Singapore (NUS) spinoff acquisition by Sonar, with AutoCodeRover IP currently being owned by Sonar. The work was also partially supported by a Singapore Ministry of Education (MoE) Tier3 research grant "Automated Program Repair" MOE-MOET32021-0001.

[6] https://www.sonarsource.com/blog/join-the-sonarqube-remediation-agent-beta

[7] https://www.mordorintelligence.com/industry-reports/artificial-intelligence-code-tools-market

References

[1] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 185–196.

[2] Juan Altmayer Pizzorno and Emery D Berger. 2025. CoverUp: Effective High Coverage Test Generation for Python. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2897–2919.

[3] Anthropic. 2025. Claude Code. https://claude.com/product/claude-code

[4] Leonhard Applis, Yuntong Zhang, Shanchao Liang, Nan Jiang, Lin Tan, and Abhik Roychoudhury. 2025. Unified Software Engineering agent as AI Software Engineer. *arXiv preprint arXiv:2506.14683* (2025).

[5] Leonhard Applis, Yuntong Zhang, Shanchao Liang, Nan Jiang, Lin Tan, and Abhik Roychoudhury. 2026. Unified Software Engineering agent as AI Software Engineer. In *ACM/IEEE International Conference on Software Engineering (ICSE)*.

[6] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 694–694.

[7] Github. 2025. Github Copilot Agents. https://github.com/features/copilot/agents

[8] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2011).

[9] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL] https://arxiv.org/abs/2310.06770

[10] Cognition Labs. 2025. Devin, the AI Software Engineer. https://devin.ai/

[11] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (2019).

[12] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

[13] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 269–278.

[14] HDT Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *International Conference on Software Engineering (ICSE)*.

[15] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. arXiv:2108.13064 [cs.SE] https://arxiv.org/abs/2108.13064

[16] OpenAI. 2025. Codex. https://openai.com/codex/

[17] Pat Rondon, Renyao Wei, José Cambronero, Jürgen Cito, Aaron Sun, Siddhant Sanyam, Michele Tufano, and Satish Chandra. 2025. Evaluating Agent-based Program Repair at Google. arXiv:2501.07531 [cs.SE] https://arxiv.org/abs/2501.07531

[18] Abhik Roychoudhury, Corina Pasareanu, Michael Pradel, and Baishakhi Ray. 2026. Agentic AI Software Engineers: Programming with Trust. *Commun. ACM* (2026).

[19] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *International Conference on Software Engineering (ICSE)*. https://arxiv.org/abs/2408.02232

[20] SonarSource Sarl. 2025. SonarQube. https://www.sonarsource.com/products/sonarqube/

[21] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. Hits: High-coverage llm-based unit test generation via method slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268.

[22] David Williams, James Callan, Serkan Kirbas, Sergey Mechtaev, Justyna Petke, Thomas Prideaux-Ghee, and Federica Sarro. 2023. User-Centric Deployment of Automated Program Repair at Bloomberg. arXiv:2311.10516 [cs.SE] https://arxiv.org/abs/2311.10516

[23] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 801–824.

[24] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[25] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://arxiv.org/abs/2404.05427

[26] Yuntong Zhang, Jiawei Wang, Dominic Berzin, Martin Mirchev, and Abhik Roychoudhury. 2026. Fixing Security Vulnerabilities with Agentic AI in OSS-Fuzz. In *International Conference on Software Engineering (ICSE), SEIP track*.