DependaFix: A GitHub App for Fixing Breaking Dependency Updates in CI Build for Java Projects

Enock Mecheo
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates
enockmecheo@nyu.edu

May Mahmoud
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates
m.mahmoud@nyu.edu

Sarah Nadi
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates
sarah.nadi@nyu.edu

Abstract

Third-party libraries are essential to modern software development, but updating them can introduce changes that break the project build. Manually diagnosing and repairing such failures in continuous integration (CI) pipelines is time-consuming and delays the adoption of security and feature updates. In this paper, we present DependaFix, a GitHub App that automates the end-to-end repair of Java projects whose builds break after dependency version updates. DependaFix builds on Byam, an automated repair tool based on large language models (LLMs), by integrating the repair process into GitHub's CI/CD workflow for pull requests. DependaFix detects failing dependency version-update pull requests and attempts to repair them. It extracts build context from CI logs and local Maven builds, delegates the repair to Byam, and creates a pull request for the repair if the fix succeeds. We demonstrate, through an example, that DependaFix can automate the repair process, potentially reducing the manual effort required by developers to diagnose and fix dependency-update failures in pull requests.

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

dependency management, breaking dependency update, automated program repair, GitHub App, LLMs

ACM Reference Format:
Enock Mecheo, May Mahmoud, and Sarah Nadi. 2026. DependaFix: A GitHub App for Fixing Breaking Dependency Updates in CI Build for Java Projects. In *Companion Proceedings of the 34th ACM Symposium on the Foundations of Software Engineering (FSE '26), June 5–9, 2026, Montreal, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modern Java projects depend on third-party libraries, often referred to as dependencies, for core functionality and framework support. These dependencies evolve over time, with updates introduced to address security vulnerabilities, improve performance, or incorporate new functionality [4]. However, such updates can introduce breaking changes that cause client projects that depend on these libraries to fail to compile or pass tests [12, 13]. Typical causes of breaking dependency updates include unresolved imports, missing transitive dependencies, incompatible method signatures, or changes in API behavior that invalidate existing code.

Prior work shows that dependency-induced breakages are both common and costly [3]. As a result, developers frequently postpone dependency updates, as diagnosing and repairing the resulting failures requires careful inspection of build logs and manual code modifications, a process that is often tedious and time-consuming [8, 12, 13].

Recent progress in large language models (LLMs) suggests that automated repair of such failure is possible, especially when detailed diagnostic information, such as a build log, is available. Byam, an LLM-based repair system, uses contextual information regarding the build failure to generate fixes to Java source files in response to a breaking dependency update [14]. However, Byam operates as an external command-line tool and assumes a local, user-prepared workspace, limiting its integration into developers' everyday workflows.

To address this gap, we build DependaFix, a GitHub App that brings automated dependency update repair directly into GitHub's pull-request (PR) workflow. DependaFix is a GitHub App that automates the repair of compilation failures caused by dependency updates in pull requests. When a developer opens a pull request that updates a dependency, DependaFix checks whether the build still compiles. If the update causes a compilation failure, DependaFix extracts the build-failure context and invokes Byam to generate a code update that repairs the failure. If the fix is successful, DependaFix opens a separate repair pull request that includes the code changes. It then comments on the developer's original pull request with a summary of the repair process.
We provide the code for DependaFix at https://github.com/sanadlab/dependafix-githubapp and instructions on how developer can install the app in their repo. We also provide a video demonstrating DependaFix workflow at https://youtu.be/BxafbQsmxSw.

2 A Breaking Dependency Update Example

Third-party software libraries offer Application Programming Interfaces (APIs) to allow developers access to their offered functionality. Third-party libraries used in a project are often referred to as *dependencies* [12, 13]. A change in the dependency's API that leads to a break in the client code is often referred to as *breaking dependency update* [12, 13]. When a new version of a dependency has a breaking dependency update, the developers using the library (i.e.,

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
237 ...
238 try {
239     GHCompare.Status status = GitHub.connect().getRepository(ghc.owner
            + '/' + ghc.repo).getCompare(branch, ghc.hash).status;
240     return status == GHCompare.Status.identical || status == GHCompare.
            Status.behind;
241 } catch (FileNotFoundException x) {
242     // For example, that branch does not exist in this repository.
243     return false;
244 }
245 ...
```

```
[ERROR] /incrementals-tools/lib/src/main/java/io/
jenkins/tools/incrementals/lib/UpdateChecker.java:
[239,126] status has private access in
org.kohsuke.github.GHCompare
```

Figure 1: Breaking dependency udpate code example and error from updating `org.kohsuke.github` from version 1.93 to 1.314 in the project `incrementals-tools`.

the client developers) have to figure out how to update their code to fix the errors they face, which is often a tedious task especially since semantic versioning practices are not always adhered to in practice [12, 15].

Figure 1 shows an example of a breaking dependency update in the project incrementals–tools[1], where the org.kohsuke.github dependency was updated from version 1.93 to 1.314 in commit c09896887acf0fe59320e01145a7034cd8d4e326. The org.kohsuke.github library, which is a GitHub API for Java, has a class GHCompare, which has a field named status. Between versions 1.94 and 1.314, access to the field changed from public to private. This change caused the build to break with the compilation error shown on the right side of the figure.

To resolve such failures, client developers must first identify the root cause of the build breakage by inspecting error logs and, when necessary, the updated API documentation. They then need to determine how to use the updated API and modify their code accordingly. This process may require locating and updating multiple usage sites of the affected API element across the codebase, making the repair effort error-prone and time-consuming, especially in large projects.

3 Related Work

Updating dependencies may introduce breaking changes [4], which remove or alter functionality and cause failures. Studies show developers delay updates to avoid incompatibilities [8, 12, 13].. Kula et al. [7, 8] found most projects use outdated dependencies, while Decan et al. [4] note asynchronous evolution increases risks. Reyes et al. [13] introduced BUMP, a benchmark of reproducible breaking updates, and Breaking-Good [12], which analyzed CI builds to identify breakage reason. Other studies show that breaking changes affect up to 20% of updates [12, 13].

Tools integrated into CI and version control platforms automate dependency updates but do not detect or repair breaking changes. Dependabot [6] automatically opens pull requests that bump dependency versions; it does not check whether those updates will break the build. Instead, it relies on CI workflow (when they exist) to surface any failures. Although it provides a merge confidence measure to indicate the possibility of breaking, it doesn't provide any fixes. Renovate [11] offers configurable schedules, grouping strategies, and "merge confidence" signals that estimate the risk of merging an update, but it similarly does not detect nor fix breaking changes. When a CI run fails after a dependency update, maintainers must manually inspect logs and implement patches, which can be tedious and time-consuming. Beyond industrial tools, research has proposed techniques for identifying breaking API changes by comparing versions of libraries, analyzing source or bytecode, and classifying changes according to their impact on clients [1–3]. These approaches are useful for understanding why an update breaks a client, but they typically do not integrate into a fully automated fix-and-validate loop.

Numerous work is done in Automated program repair (APR) research to investigate techniques that automatically generate and validate code changes to fix software bugs without requiring manual edits from developers [9, 10]. Surveys by Monperrus [10] and Le Goues et al. [9] catalog techniques such as generate-and-validate patching, or search-based repair, and highlight both the promise and the limitations of APR in practice. More recent work explores LLMs for code repair, bug fixing, and test generation [16]. LLMs can leverage rich natural-language and code context but are sensitive to prompt design, context length, and hallucination. Frunkte and Krinke [5] propose an automated approach to fix dependency-related breakages using LLM, investigating an agent approach. Similarly, Byam [14] provides an automated repair system that uses LLMs to fix breaking dependency updates. It addresses compilation failures caused by dependency version updates in Java projects using Maven builds. It investigates various prompt designs to use LLMs to generate repairs. Byam demonstrates strong effectiveness in resolving build breaks caused by dependency updates that result in compilation failures. In the BUMP dataset, Byam fully resolved 27% of builds that failed due to compilation errors, meaning it fixed all issues related to the dependency update. For builds where complete resolution was not achieved, Byam still fixed 78% of individual compilation errors. Given Byam's high success rate in resolving individual compilation errors, DependaFix builds on Byam. and integrates it into GitHub's pull request workflow.

4 Approach

In this section, we describe the design and implementation of DependaFix. We first give an overview of the end-to-end workflow, showing how DependaFix detects failing dependency-update pull requests and orchestrates the repair process. We then explain how DependaFix integrates with Byam [14] to run the repair process,

[1]https://github.com/jenkinsci/incrementals-tools

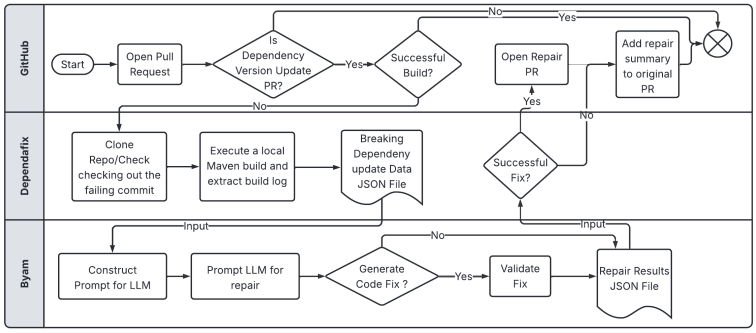

Figure 2: DependaFix Workflow Overview.

and how it reports the results back to developers through GitHub pull requests.

4.1 Workflow Overview

We implement DependaFix as a Node.js GitHub App that runs on a dedicated server. It integrates with Byam (a Java application) through a subprocess protocol and a shared filesystem. We illustrate the workflow in Figure 2. At a high level, the DependaFix workflow is as follows:

(1) When a developer opens a Pull Request (PR) that updates a maven pom.xml file, DependaFix is triggered. Specifically, DependaFix receives GitHub webhook events (e.g., `pull_request` and `workflow_run`) to start its process.
(2) DependaFix analyzes whether the event corresponds to a failed CI run on a pull request that modifies a dependency version. DependaFix only attempts to fix if the PR changes a dependency version that causes a compilation failure.
(3) If that is the case, DependaFix prepares a Byam workspace by cloning the repository and checking out the failing commit.
(4) DependaFix executes a local Maven build to reproduce the failure and generate a full build log.
(5) DependaFix constructs a structured analysis JSON file that summarizes errors, dependency changes, and build context.
(6) DependaFix then runs Byam, while passing to it the JSON file with the required inputs for it to run.
(7) Byam uses the JSON file input to start a repair process, where it constructs a prompt for the LLM with the failure context, and prompts the LLM for code fixes. If an LLM generates code fixes, Byam validates the fixes. Byam then creates an output JSON file containing the repair process results, whether successful or not, and, if successful, provides paths to the patch files.
(8) DependaFix processes Byam's repair result, and if Byam generated successful fixes, it opens a repair pull request updating the code according to the fixes.
(9) Finally, DependaFix adds a summary of the repair process to the original PR as a comment, links it to the repair PR if available, and indicates whether the repair succeeded.

4.2 Integrating with Byam

Before invoking Byam, DependaFix prepares a workspace on the server that mirrors the state of the repository at the failing commit. To do that, DependaFix does the following:

(1) Clones the target repository into a temporary directory.
(2) Checks out the exact commit SHA from the failing pull request.
(3) Runs Maven with the command `mvn clean test-compile -B`, which compiles both main and test sources in batch mode.
(4) Captures the complete build output in a log file.

Once DependaFix reproduces the failure locally, it constructs an analysis JSON that serves as input to Byam. This structured JSON provides information about the build failure, including repository metadata (pull request number, branch, commit SHA), parsed compilation errors from CI and local logs (with file paths and line numbers), dependency changes derived from the `pom.xml` diff (version bumps, additions, removals), and paths to the workspace directory and local build log.

DependaFix spawns Byam as a subprocess, passing it the analysis JSON and workspace paths. Byam's repair logic works as follows:

(1) *Error analysis*: Byam parses the compilation errors from the JSON file, identifying the affected Java files and the specific error messages.
(2) *Prompt construction*: For each affected file, Byam constructs a prompt that includes the error message, the code context, and information about the dependency change that triggered the failure.
(3) *LLM invocation*: Byam prompts the LLM, which proposes code edits to resolve the compilation error. We use the zero-shot basic prompt provided by Byam that includes the code and the error message extracted from the log.
(4) *Patch application*: Byam applies the LLM's suggested edits to the Java source files in the workspace, committing them via its internal Git manager.
(5) *Local verification*: Byam re-runs `mvn clean test-compile -B` in the workspace to check whether the repair fixed the build.

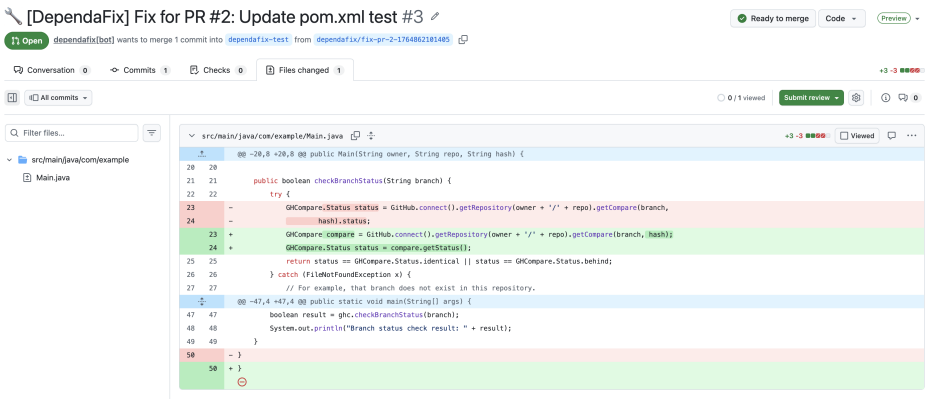

Figure 3: A repair pull request created by DependaFix, showing the code diff for a Java file that Byam edited to fix the compilation error of the breaking dependency update introduced in Figure 1.

4.3 Reporting on the Repair Process

Byam returns a structured *repair result JSON* to DependaFix, reporting whether the build was fixed locally, and the list of files that Byam modified. If Byam was successful in fixing the breaking dependency update, DependaFix creates a new repair pull request on GitHub including all code updates generated by Byam. DependaFix does the following steps to report on a successful repair process:

(1) Creates a new branch in the target repository.
(2) Commits all changes produced by Byam, including Java source and test files.
(3) Opens a *repair pull request* targeting the repository's main branch.
(4) Posts a concise comment on the original failing pull request, summarizing what failed and linking to the repair pull request so the developer can review the fix without leaving GitHub.

4.4 Repairing a Breaking Dependency Update Example

Going back to the breaking dependency update example introduced in Section 2. To fix the compilation failure, the developer needs to change the code to use the function getStatus() instead of using the status field. We simulated this example in a GitHub repository [2]. Figure 3 shows the repair pull request created by DependaFix upon detecting the failing pull request [3]. The pull request title clearly references the original failing pull request (i.e, "[DependaFix] Fix for pull request #2"), and the description lists the modified files, links back to the original pull request, and notes that the fix was verified locally. The diff view shows the exact code changes that Byam made, in this case, adding the required method call in `Main.java`.

Figure 4 shows the comment that DependaFix posts on the original failing pull request. The comment indicates that an automated fix is available and links to the repair pull request. It confirms that the fix was verified locally and that the build passes. To fix the breaking dependency update, the developer only needs to review

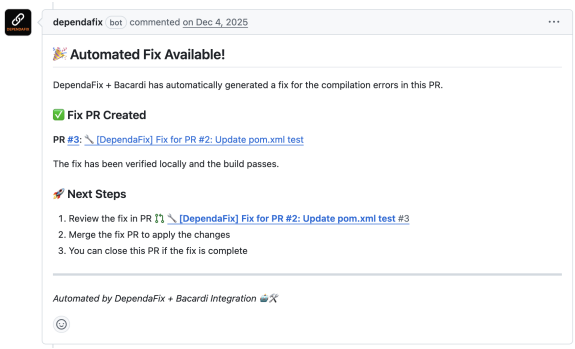

Figure 4: The comment DependaFix posts on the original failing pull request, notifying the developer that an automated fix is available and linking to the repair pull request.

the fix, merge the PR, and delete the fix branch. This process lets developers review and merge the repair directly on GitHub without switching contexts.

5 Conclusion and Future Work

In this paper, we investigate how automated repair of dependency-related build failures can be integrated into GitHub's development workflow. We designed and implemented DependaFix, a GitHub App that integrates Byam, an existing LLM-based automated repair tool [14], into the CI/CD Workflow of GitHub. DependaFix addresses CI failures on dependency version update pull requests. For developers, DependaFix reduces the manual effort of fixing broken builds by automatically generating repair pull requests that they can review and merge. We showcase DependaFix on a controlled Java test repository, demonstrating that DependaFix can successfully repair compilation errors introduced by source-level API changes.

Given the current initial DependaFix prototype, our next steps include testing DependaFix on simulated PRs from large benchmark datasets like BUMP [13], adding support for test failures and semantic errors, and studying how developers use the tool.

[2] https://github.com/sanadlab/dependafix-test/pull/2
[3] https://github.com/sanadlab/dependafix-test/pull/3

References

[1] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung. 2016. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proc. 24th ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE)*. 109–120. https://doi.org/10.1145/2950290.2950325

[2] A. Brito, L. Xavier, A. C. Hora, and M. T. Valente. 2018. APIDiff: Detecting API Breaking Changes. In *Proc. IEEE 25th Int. Conf. Software Analysis, Evolution and Reengineering (SANER)*. 507–511. doi:10.1109/SANER.2018.8330254

[3] A. Brito, L. Xavier, A. C. Hora, and M. T. Valente. 2018. Why and How Java Developers Break APIs. *arXiv preprint arXiv:1801.05198* (2018). https://arxiv.org/abs/1801.05198

[4] A. Decan, T. Mens, and E. Constantinou. 2018. On the evolution of technical lag in the npm package dependency network. In *Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME)*. 404–414. https://doi.org/10.1109/ICSME.2018.00050

[5] L. Frunkte and J. Krinke. 2025. Automatically Fixing Dependency Breaking Changes. In *Proc. ACM Software Engineer.*, Vol. 2.

[6] GitHub. 2024. Dependabot: Automated dependency updates. GitHub Documentation. https://docs.github.com/en/code-security/dependabot

[7] R. G. Kula, D. M. German, T. Ishio, and K. Inoue. 2017. Trusting a library: A study of the latency to adopt the latest version of a library. In *Proc. IEEE 24th Int. Conf. Software Analysis, Evolution and Reengineering (SANER)*. 520–523. https://ieeexplore.ieee.org/document/7081869

[8] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. 2018. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23, 1 (2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[9] C. Le Goues, M. Pradel, and A. Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[10] M. Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1 (2018).

[11] Renovate. 2024. Renovate: Automated dependency update tool. Renovate Documentation. https://docs.renovatebot.com/

[12] F. Reyes, B. Baudry, and M. Monperrus. 2024. Breaking-Good: Explaining breaking dependency updates with build analysis. *arXiv preprint arXiv:2407.03880* (2024). https://arxiv.org/abs/2407.03880

[13] Frank Reyes, Yogya Gamage, Gabriel Skoglund, Benoit Baudry, and Martin Monperrus. 2024. BUMP: A Benchmark of Reproducible Breaking Dependency Updates. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. doi:10.1109/SANER60148.2024.00024

[14] Frank Reyes, May Mahmoud, Federico Bono, Sarah Nadi, Benoit Baudry, and Martin Monperrus. 2025. Byam: Fixing Breaking Dependency Updates with Large Language Models. *arXiv preprint arXiv:2505.07522* (2025). doi:10.48550/arXiv.2505.07522

[15] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2022. Has my release disobeyed semantic versioning? static detection based on semantic differencing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[16] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, and Z. Chen. 2024. A survey on large language models for software engineering. *arXiv preprint arXiv:2312.15223* (2024). https://arxiv.org/abs/2312.15223