Byam: Fixing Breaking Dependency Updates with Large Language Models

Frank Reyes · May Mahmoud · Federico Bono · Sarah Nadi · Benoit Baudry · Martin Monperrus

Abstract Application Programming Interfaces (APIs) facilitate the integration of third-party dependencies within the code of client applications. However, changes to an API, such as deprecation, modification of parameter names or types, or complete replacement with a new API, can break existing client code. These changes are called *breaking dependency updates*; It is often tedious for API users to identify the cause of these breaks and update their code accordingly. In this paper, we explore the use of Large Language Models (LLMs) to automate client code updates in response to breaking dependency updates. We evaluate our approach on the BUMP dataset, a benchmark for breaking dependency updates in Java projects. Our approach leverages LLMs with advanced prompts, including information from the build process and from the breaking dependency analysis. We assess effectiveness at three granularity levels: at the build level, the file level, and the individual compilation error level. We experiment with five LLMs: Google Gemini-2.0 Flash, OpenAI GPT4o-mini, OpenAI o3-mini, Alibaba Qwen2.5-32b-instruct, and DeepSeek V3. Our results show that LLMs can automatically repair breaking updates. Among the considered models, OpenAI's o3-mini is the best, able to completely fix 27% of the builds when using prompts that include contextual information such as the erroneous line, API differences, error messages, and step-by-step reasoning instructions. Also, it fixes 78% of the individual compilation errors. Overall, our findings demonstrate the potential for LLMs to fix compilation

F. Reyes, KTH Royal Institute of Technology, Stockholm, Sweden
E-mail: frankrg@kth.se

M. Mahmoud, New York University Abu Dhabi, United Arab Emirates
E-mail: m.mamhoud@nyu.edu

F. Bono, KTH Royal Institute of Technology, Stockholm, Sweden
E-mail: fbono@kth.se

S. Nadi, New York University Abu Dhabi, United Arab Emirates
E-mail: sarah.nadi@nyu.edu

B. Baudry, Université de Montréal, Canada
E-mail: benoit.baudry@umontreal.ca

M. Monperrus, KTH Royal Institute of Technology, Stockholm, Sweden
E-mail: monperrus@kth.se

errors due to breaking dependency updates, supporting developers in their efforts to stay up-to-date with changes in their dependencies.

Keywords Breaking Dependency Update · Breaking Changes · Software Evolution · LLMs

1 Introduction

Modern software development heavily relies on third-party software libraries to improve developers' productivity and time-to-market. The set of libraries used by a client project is referred to as the *dependencies* of that project. Developers access a library's functionality through its provided *Application Programming Interfaces* (APIs). These APIs evolve over time to introduce new functionality, fix bugs, or address security vulnerabilities (Lamothe et al. 2021). While such evolution is necessary, it often comes at a cost: updates may introduce breaking changes that prevent client projects from compiling or running (Lamothe et al. 2021; Reyes et al. 2024a,b). Failures resulting from API updates are referred to as *breaking dependency updates.* When a new version of a dependency results in a breaking dependency update, the developers using the library (i.e., *the client developers*) have to figure out how to update their code to fix the errors they face. This is known to be a tedious and time consuming maintenance task (Ochoa et al. 2022; Zhang et al. 2022).

To address this problem, we propose Byam, a pipeline that leverages Large Language Models (LLMs) to update client code in response to breaking dependency updates (Chang et al. 2024; Vaswani 2017). We focus on breaking dependency updates that lead to compilation errors, which are the most common types of causes for breakage (Reyes et al. 2024a,b). Byam analyzes compilation errors caused by a dependency update, formulates prompts that incorporate relevant build and dependency information, and queries an LLM to generate updated code. We assess the effectiveness of Byam using the BUMP dataset (Reyes et al. 2024b), which provides reproducible cases of breaking dependency updates in Java projects. Our experiments systematically evaluate different LLMs, prompt design strategies, and information granularity levels.

We evaluate Byam with five different LLMs: Google Gemini-2.0 Flash, OpenAI GPT4o-mini, OpenAI o3-mini, Alibaba Qwen2.5-32b-instruct, and DeepSeek V3. We assess Byam's effectiveness at three levels of granularity: build, file, and individual error. At the build level, we measure the number of fully fixed builds, those still failing due to compilation errors, and cases where compilation was resolved but test failures emerged. At the file level, we track the number of files with compilation errors before and after the fixes, identifying which files were successfully fixed and which ones still include compilation errors. Finally, at the individual error level, we analyze the number of original compilation errors per file, comparing them before and after the LLM-generated fixes to determine fixed, unresolved, and newly introduced errors. To quantify the impact of the corrections generated by LLMs, we define a set of metrics to evaluate the performance of LLMs. We analyze the capability of the models to fully resolve the build process, file-level success rate, and performance in resolving individual errors. Since LLMs can also introduce new errors in the process, we measure the relative error fixed as the proportion of previously existing errors that were successfully resolved, while accounting for any new errors introduced after the code update.

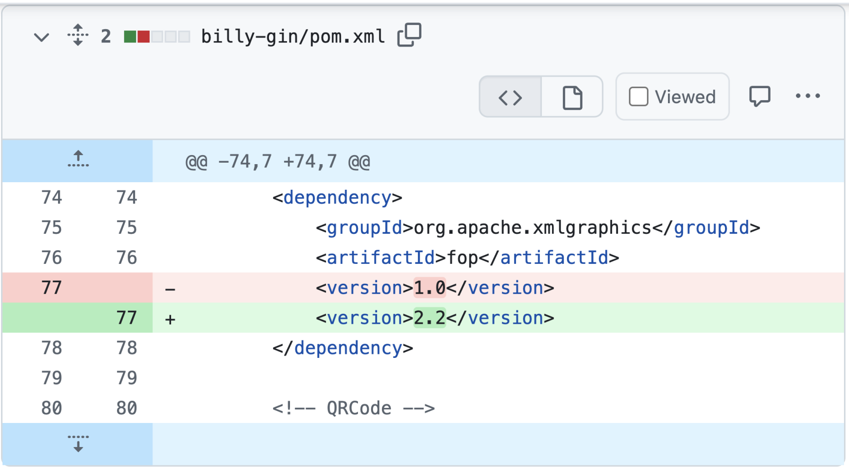
Our results show that LLM can automatically repair breaking updates. OpenAI's o3-mini model achieves the best performance with a build success rate of 27% when using the best prompt that integrates erroneous line, information regarding the differences between the dependency versions (APIDiff), and structured reasoning techniques (CoT). At the file level o3-mini fixes 41% of the originally erroneous files in unsuccessful builds and at the error level, o3-mini fixes 79% of compilation failures. Overall, our experimental results demonstrate the potential for LLMs to reduce developer effort and time in managing breaking dependency updates, helping developers more effectively stay up-to-date with changes in their dependencies.

To summarize, our contributions in this paper are as follows:

1. An approach, called Byam, for fixing compilation errors from breaking dependency updates with LLMs. Its advanced prompting strategies relies on a comprehensive analysis of the breaking update problem space. In particular, we use API differencing of the breaking library and a Chains-of-thought reasoning command dedicated to breaking updates.
2. A large scale experiment with five different LLMs and eight prompts, over 103 breaking dependency updates. Our results show that o3-mini achieves the best outcomes when using a prompt that highlights the erroneous line causing the compilation error(s), that provides information about the API differences between the old and updated dependency versions, and that includes Chains-of-thought instructions.
3. We provide a full replication package that ensures the reproducibility of our study and fosters future research on this topic. The replication package is available at https://github.com/chains-project /bacardi

2 Background

Breaking dependency updates are common in software projects (Brito et al. 2018b), and often discourage developers from upgrading their dependencies (Dietrich et al. 2019; Ochoa et al. 2022; Venturini et al. 2023). However, keeping dependencies up to date is crucial for security, as outdated libraries may contain vulnerabilities (Brito et al. 2020, 2018b; Larios Vargas et al. 2020; Salza et al. 2018; Xavier et al. 2017). A breaking dependency update occurs when a library update introduces incompatibilities that cause client code to fail. This can occur either when developers manually update dependencies or when they use automated tools such as Dependabot or Renovate, which submit pull requests for updates. Such breakages are often changes in method signatures, deprecations, or class removals can cascade into compilation errors across the client project. While breaking updates can also trigger



```
@@ -74,7 +74,7 @@
74  74          <dependency>
75  75              <groupId>org.apache.xmlgraphics</groupId>
76  76              <artifactId>fop</artifactId>
77     -            <version>1.0</version>
    77 +            <version>2.2</version>
78  78          </dependency>
79  79
80  80          <!-- QRCode -->
```

(a) Difference in Maven build file, causing a breaking update.

```
113 ...
114 // create an instance of fop factory
115 FopFactory fopFactory = FopFactory.newInstance();
116 // a user agent is needed for transformation
117 FOUserAgent foUserAgent = fopFactory.newFOUserAgent();
118 ...
```

(b) The broken code with compilation failure after breaking update.

```
[ERROR] /billy/billy-gin/src/main/java/com/premiumminds/billy/gin/services/
impl/pdf/FOPPDFTransformer.java:[115,43] no suitable method found for
newInstance(no arguments)
  method org.apache.fop.apps.FopFactory.newInstance(org.apache.fop.apps
  .FopFactoryConfig) is not applicable
   (actual and formal argument lists differ in length)
  method org.apache.fop.apps.FopFactory.newInstance(java.io.File) is
   not applicable (actual and formal argument lists differ in length)
  method org.apache.fop.apps.FopFactory.newInstance(java.net.URI) is
   not applicable (actual and formal argument lists differ in length)
  method org.apache.fop.apps.FopFactory.newInstance(java.net.URI,java.
  io.InputStream) is not applicable (actual and formal argument lists
  differ in length)
```

(c) Compilation error information from the logs.

Fig. 1: Example of a breaking dependency update when updating `org.apache.xmlgraphics` (dependency) from version 1.0 (old version) to 2.2 (new version) in the project `billy`.

test failures, dependency resolution issues or lock conflicts (Reyes et al. 2024b), in this paper we focus specifically on compilation errors, which are the most prevalent form (Reyes et al. 2024a).

Figure 1 shows an example of a breaking dependency update in project billy [1], where the org.apache.xmlgraphics dependency was updated from version 1.0 to 2.2 in commit 36859167815292f279e570d39dd2ddbcf1622dc6. The org.apache.xmlgraphics library has a method newInstance(), which has zero parameters in version 1.0 and was changed in the new version 2.0. This led to a compilation error in billy, as shown at the bottom of the figure.

Prior work has extensively studied the prevalence and causes of breaking changes and methods to identify them in library code (Brito et al. 2020, 2018a,b; Mujahid et al. 2020; Reyes et al. 2024a; Xavier et al. 2017). Migration support has also been explored through rule-based transformations and recommendation systems (Brito et al. 2018a; Dagenais and Robillard 2009; Xing and Stroulia 2007). However, such approaches typically rely on predefined rules, mined examples, or static analysis heuristics, which limit their ability to handle the diversity of real-world breaking updates and the project-specific context required to repair client code. In contrast, we investigate whether large language models (LLMs) can directly generate client-side repairs when provided with appropriate context.

To formalize our scope, we adopt standard terminology.

Definition 1 A **dependency update** is a change made in a build specification file where the version of a specific dependency is updated to a new version. In the experimentation of this paper, we focus on the Maven build file (`pom.xml`) in Java.

Definition 2 A **breaking dependency update** occurs when updating a dependency version introduces incompatibilities that cause the build to fail.

For systematic evaluation in this study, we examine breaking dependency updates as represented in the BUMP dataset Reyes et al. (2024b): a pair of commits for a project, consisting of a pre-breaking commit with a passing build and a breaking commit that updates the version of a single dependency, causing the build to fail.

Definition 3 A **pre-breaking commit** is the commit before the dependency update, where the project is built successfully.

Definition 4 A **breaking commit** is the project commit where the only change is an update to a dependency version that leads to the build failing.

3 Byam: An Approach to Repairing Breaking Updates

Existing solutions for breaking dependency updates have limited scalability and adaptability. They require predefined rules or abundant historical data, which makes them weak in the face of diverse or novel API changes. Large language models (LLMs) offer a promising alternative due to their ability to

[1] https://github.com/premium-minds/billy/pull/300

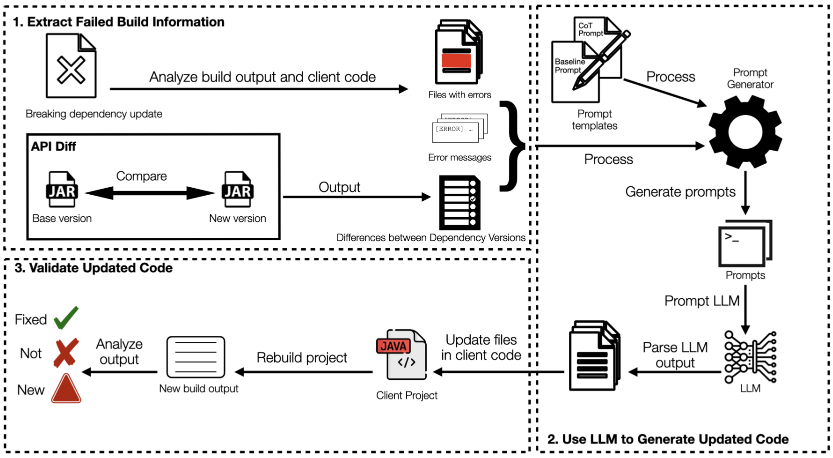


Fig. 2: Overview of our the Byam pipeline using LLMs to fix breaking dependency updates.

generalize across different contexts and reason about unknown patterns. To be effective in automating the repair of breaking dependency updates, LLMs must be guided with rich prompts.

We propose a novel approach, Byam, based on generating a code patch with LLMs to fix breaking dependency updates. Byam examines error messages, locates the erroneous line, and consults API differences before querying an LLM and applying a fix. Byam enables LLMs to operate as context-aware repair assistants, making them more efficient and accurate at addressing breaking changes than prior approaches.

3.1 Overview

Figure 2 shows an overview of our approach, which we refer to as *Byam*. The pipeline consists of three steps. The first step extracts build information regarding the breaking dependency update, including the error-causing code and the compiler error message(s). It also provides information regarding the differences between the two versions of the dependency. The second step focuses on generating a fixed version of the client code using an LLM. We engineer different prompts, which include different types of contextual information, such as API differences, error messages, and affected code lines; we discuss this further below. Byam prompts the LLM, augmented with the extracted build and the previously mentioned contextual details, to generate a fix to address the compilation failure. The third and final step focuses on validating the generated fix by integrating it into the client code and re-running the build.

We detail each stage of the pipeline in the following subsections.

```
---! REMOVED METHOD: PUBLIC(-) STATIC(-) org.apache.fop.apps.FopFactory
     newInstance()
  +++  NEW METHOD: PUBLIC(+) STATIC(+) org.apache.fop.apps.FopFactory
     newInstance(org.apache.fop.apps.FopFactoryConfig)
  +++  NEW METHOD: PUBLIC(+) STATIC(+) org.apache.fop.apps.FopFactory
     newInstance(java.io.File)
    +++  NEW EXCEPTION: org.xml.sax.SAXException
    +++  NEW EXCEPTION: java.io.IOException
  +++  NEW METHOD: PUBLIC(+) STATIC(+) org.apache.fop.apps.FopFactory
     newInstance(java.net.URI)
  +++  NEW METHOD: PUBLIC(+) STATIC(+) org.apache.fop.apps.FopFactory
     newInstance(java.net.URI, java.io.InputStream)
    +++  NEW EXCEPTION: org.xml.sax.SAXException
    +++  NEW EXCEPTION: java.io.IOException
```

Fig. 3: API Difference used in Byam. It captures the relation between the construct that triggers the error and the changes in the new version of the dependency in Figure 1

3.2 Step 1: Extract Failed Build Information

This step isolates the parts of the code that are causing a compilation failure after the dependency update and identifies the API differences involved in the failure. Byam uses this information to provide the LLM with various contextual data. To collect this information, Byam analyzes the build output log after a breaking update. The pipeline automatically extracts a list of files causing the compilation error(s) and the list of error messages for each file. Additionally, Byam analyzes the changes between both versions of the dependency.

Figure 1c shows an excerpt of a build output. Analyzing this output, Byam determines that the file /billy/billy–gin/src/main/java/com/premiumminds/billy/gin/services/impl/pdf/FOPPDFTransformer.java is causing a compilation error. Byam also extracts the error message, *"no suitable method found for newInstance(no arguments)"*, as well as the error location, at line 115, column 43. From the error location, Byam identifies the specific constructs responsible for the compilation failure. In the previous example, the error is caused by a method `newInstanc()`. Next, Byam extracts the API differences between the dependency versions to determine whether any changes in the new version are related to the construct that triggers the error. We term this output the API Difference (APIDiff). For this example, the APIDiff shows that the signature of the method `newInstance()`has been changed in the new version 2.0 of the dependency. Figure 3 shows an excerpt of the APIDiff corresponding to the example in Figure 1, highlighting the change associated with the construct identified as the root cause of the breaking dependency update. This APIDiff information is later incorporated into the LLM prompts, as discussed in Subsection 3.3.

At the end of this first step, Byam has detailed information about: the list of files causing compilation errors; the list of compilation errors for each file, and APIDiff that triggers the error.

Table 1: Prompt Design Space for Fixing Breaking Updates

Variation	Description
Baseline Prompt	Includes client code and error message but excludes additional context.
Erroneous Line Inclusion	Adds the specific line of code causing the compilation error.
API Differences (API Diff)	Includes details of API differences between dependency versions.
Chain of Thought (CoT) Prompting	Guides LLM reasoning by incorporating structured reasoning steps.

In the next step, we use this output to engineer several prompts for an LLM to generate updated code.

3.3 Step 2: Use LLM to Generate Repairs

In this step, we use an LLM to attempt to fix the files with compilation errors. For each file causing compilation errors, we construct a prompt based on a prompt template. Once a prompt is created, we use it to query the LLM to generate fixed code for one buggy file. We iterate over each file in the list to completely fix the project. To improve LLM performance, we explore different prompt design variations that enrich the context provided to the model. Table 1 provides a list of the different prompt design decisions we consider.

Baseline prompt. Our baseline prompt includes the client code causing the error and the compilation error message. The client code provides the Java class where the failure occurs. The error message, extracted from the build log file, helps pinpoint the exact issue.

Figure 4 shows our baseline prompt template, where the placeholders for the different information, with the entire Java class that triggers the compilation error (*client_code*), the error message extracted from the build log file (*error_message*).

Based on the baseline, we evaluate specific combinations of prompt design choices rather than generating all possible variations, focusing on erroneous lines, API version differences (*APIDiff*), and Chain of Thought (CoT) reasoning. Instead of an exhaustive exploration, we systematically test selected configurations that provide meaningful information on their impact on LLM performance. Our approach considers the incremental addition of each factor to a baseline prompt, ensuring a balanced assessment of each factor's contributions. This approach examines two key dimensions: the prompting strategy (zero-shot vs. CoT) and the type of contextual information included in the prompt (error information, erroneous line, APIDiff). This structured evaluation allows us to isolate the effects of each component while maintaining a manageable number of experimental setups.

> Act as an Automatic Program Repair (APR) tool, reply only with code, without explanation.
> You are specialized in breaking dependency updates, in which the failure is caused by an external dependency.
> To solve the failure you can only work on the client code.
>
> the following client code fails:
> ‘‘‘java
> <client_code>
> ‘‘‘
>
> with the following error message:
> <error_message>
>
> - Propose a patch that can be applied to the code to fix the issue.
> - Return only a complete and compilable class in a fenced code block.
> - You CANNOT change the function signature of any method but may create variables if it simplifies the code.
> - You CAN remove the @Override annotation IF AND ONLY IF the method no longer overrides a method in the updated dependency version.
> - If fixing the issue requires addressing missing imports, ensure the correct package or class is used in accordance with the newer dependency version.
> - Avoid removing any existing code unless it directly causes a compilation or functionality error.
> - Return only the fixed class, ensuring it fully compiles and adheres to these constraints.

Fig. 4: Baseline Prompt

> the error is triggered in the following specific lines in the previous code:
> <erroneous_line>

Fig. 5: Adding Erroneous Line to the Prompt. This helps the LLM to identify and focus on the problem to be fixed.

Providing Erroneous Line Information. Developers, when debugging, typically start by identifying the exact line of code causing the issue, as this helps narrow down the scope of the problem and focus efforts on the relevant section of the code. Including the erroneous line in the prompt replicates this natural debugging process, potentially improving code understanding for the LLM (the compiler error message does not include the actual line by default). We include this information by extending the base prompt with the actual line of the client code responsible for the compilation error, as shown in Figure 5, after the error message.

Providing API Difference Information (*APIDiff*) When a dependency update breaks a project, developers often compare the previous and new versions of the API to identify relevant changes, such as method renaming or

> The error is caused by a change in the API of the dependency. The new library version includes the following changes: <api_diff>

Fig. 6: Adding API Differences to the Prompt. It helps to the LLM to identify possible replacements to missing constructs.

```
The error is caused by a change in the API of the dependency.
The new library version includes the following changes:
- Method net.sf.jasperreports.engine.JRPen.setLineWidth(float)
  has been removed in the new version of the dependency.
- Method net.sf.jasperreports.engine.base.JRBasePen.setLineWidth(float)
  has been removed in the new version of the dependency.
```

Fig. 7: APIDiff example included in the prompt to repair the breaking update of the dependency `jasperreports` from version `1.18.1` to version `1.19.1` in project biapi

modifications to the parameters (Brito et al. 2018a). Such changes help developers determine how to update their client code to fix the problem. Following this practice, our intuition is that by explicitly providing these API differences (APIDiff) to the LLM, we can allow it to reason about the likely source of the error and suggest a more accurate solution.

To extract these differences, we rely on *japicmp* (Maven 2022), a widely used tool to detect API changes in Java libraries. *japicmp* performs a comprehensive structural comparison, allowing it to detect modifications across public, internal, and even deprecated parts of an API. This tool identifies modifications such as changes in method signatures, parameter types, deprecated methods, or new functionality. For example, when we use `japicmp` to analyze the changes between versions `6.18.1` and `6.19.1` of the `jasperreports` dependency, it detects that the `setLinewith` method was removed from the new version of the dependency. We augment the prompt with that information, as in the example shown in Figure 6. Figure 7 shows the concrete example of APIDiff for `jasperreports`.

Applying Chain of Thought (CoT) Prompting. Developers often solve complex errors by following a structured step-by-step reasoning process. This structured approach helps them to analyze the problem, identify possible causes and systematically find a solution. This structured approach is replicated in prompting in what is referred to as *Chain of Thought (CoT)* prompting (Wei et al. 2023). Previous work has shown that CoT instructions improve performance in code generation and automated repair tasks (Le Goues et al. 2019; Monperrus 2018). Consequently, we consider CoT prompting for fixing breaking dependency updates. In our approach, CoT prompting explicitly guides the LLM to analyze the error message, identify the affected code, infer the underlying issue, and reason about possible fixes before proposing a corrected version. This approach guides the LLM to produce a more structured and coherent response (Wei et al. 2023). We incorporate the CoT technique in the prompt as shown in Figure 8.

Table 2: The 8 Studied Prompt Configurations in our Experiments.

Prompt ID	Prompt Name	Client Code	Error Message	Erroneous Line	APIDiff	CoT Prompting
P_1	Baseline Prompt	✓	✓			
P_2	Erroneous Line	✓	✓	✓		
P_3	APIDiff	✓	✓		✓	
P_4	Erroneous Line + APIDiff	✓	✓	✓	✓	
P_5	CoT Prompt	✓	✓			✓
P_6	CoT + Errouneous Line	✓	✓	✓		✓
P_7	CoT + API Diff	✓	✓		✓	✓
P_8	CoT + Erroneous Line + APIDiff	✓	✓	✓	✓	✓

To summarize, Table 2 provides an overview of all the prompt configurations we experiment with. The table outlines eight distinct prompt variations, detailing the information included in each prompt and whether we utilize the Chain of Thought (CoT) prompting technique. Our experiments will study the performance of each prompt (see section 4).

3.4 Step 3: Update Code, Rebuild Project and Analyze the Build Outcome

In this stage, we replace the original files in the client project, with the LLM-generated files. Then, we rebuild the project. We use the default build command of the project, typically `mvn test` command to build the project and run all existing tests. If the build is successful, the updates to the code successfully fixed the breaking dependency update. If not, we identify which files/errors are fixed, which are not fixed, and which new errors might have appeared. We note the build where the compilation failure is fixed but results in a different failure category, such as a test failure.

3.5 Example of a Repair

Byam is able to fully fix the breaking dependency update we introduced in Figure 1, where the code on line 3 causes a build breakage. When we process this case using Byam with o3-mini, the LLM generates the code in Figure 9. We can see that Byam successfully updated the method signature, fixing the break in the project build.

4 Experimental Methodology

In this section, we describe our research questions, the data and LLMs we used in our study.

Act as an Automatic Program Repair (APR) tool, reply only with code, without explanation.
You are specialized in breaking dependency updates, in which the failure is caused by an external dependency.
To solve the failure you can only work on the client code.

the following client code fails:
‘‘‘java
<`client_code`>
‘‘‘
the error is triggered in the following specific lines in the previous code:
<`erroneous_line`>

with the following error message:
<`error_message`>

The error is caused by a change in the API of the dependency. The new library version includes the following changes:
<`api_diff`>

Before proposing a fix, please analyze the situation and plan your approach within <`repair_strategy` > tags:

- Identify the specific API changes that are causing the failure in the client code.
- Compare the old and new API versions, noting any changes in method signatures, return types, or parameter lists.
- Determine which parts of the client code need to be updated to accommodate these API changes.
- Consider any constraints or requirements for the fix (e.g., not changing function signatures, potential import adjustments).
- Plan the minimal set of changes needed to fix the issue while keeping the code functional and compliant with the new API.
- Consider potential side effects of the proposed changes on other parts of the code.
- Ensure that the planned changes will result in a complete and compilable class.
- If applicable, note any additional imports that may be needed due to the API changes.

- Propose a patch that can be applied to the code to fix the issue.
- Return only a complete and compilable class in a fenced code block.
- You CANNOT change the function signature of any method but may create variables if it simplifies the code.
- You CAN remove the @Override annotation IF AND ONLY IF the method no longer overrides a method in the updated dependency version.
- If fixing the issue requires addressing missing imports, ensure the correct package or class is used in accordance with the newer dependency version.
- Avoid removing any existing code unless it directly causes a compilation or functionality error.
- Return only the fixed class, ensuring it fully compiles and adheres to these constraints.

Fig. 8: Chain of Thought Prompt for Breaking Update. LLMs benefit from detailed Chain of Thought instructions, esp. for reasoning tasks, such as bug fixing.

```
-      // create an instance of fop factory
-      FopFactory fopFactory = FopFactory.newInstance();
+      // create an instance of fop factory with a base URI
+      FopFactory fopFactory = FopFactory.newInstance(new File(".").toURI());
      // a user agent is needed for transformation
      FOUserAgent foUserAgent = fopFactory.newFOUserAgent();
```

Fig. 9: LLM-generated code for fixing the code example from Figure 1
.

4.1 Research Questions

In this section, we present our three research questions designed to evaluate complementary aspects of Byam's effectiveness to repair compilation errors caused by breaking dependency updates.

RQ1 (Repair Success): How effective is Byam in fixing compilation errors due to breaking dependency updates?
This question aims to measure Byam's capability to fully repair projects with breaking dependency updates by fixing all files and errors to reach a build success. This metric indicates how effective Byam is as a fully automated solution without developer intervention.

RQ2 (Partial Repair): To what extent does Byam partially fix compilation errors at the file and error level?
This question evaluates the extent to which Byam can repair projects partially, even when the build is not fully fixed. While RQ1 focuses on complete build repair, there are cases where Byam cannot resolve all errors but still fixes a substantial portion of them. Such partial repair is valuable in practice, as it reduces the number of remaining errors and allows developers to concentrate their effort on a smaller scope. In this RQ, we therefore measure partial repair at both the file level and the individual error level.

RQ3 (New Error Introduction): To what extent do language models tend to introduce new errors during the breaking update repair process?
In this question, we address the balance between fixing existing errors caused by the breaking dependency update and introducing new ones when applying patches generated by Byam. Minimizing the introduction of new errors with generated fixes is essential for the practical adoption of any automated repair tool, as newly introduced errors will increase debugging effort and affect the developers' confidence in the automated system.

4.2 Study Subjects

Our experiments are based on the real-world dependency updates collected in previous research Reyes et al. (2024b). We use BUMP, a benchmark for reproducible breaking dependency updates. BUMP includes 571 breaking dependency updates, of which 243 (43%), are broken builds due to compilation failure.

BUMP classifies compilation failures into four distinct categories: *Direct compilation errors*, *Indirect compilation errors*, *Java version incompatibility* and *Werror failure* (Reyes et al. 2024a). We discard the 78 failures classified under the *Java version incompatibility* category. Java versioning failures are failures that require a different Java version than the one used in the project. Resolving Java versioning failures requires changing the Java version, rather than updating the code. BUMP also contains 8 compilation errors of type *Werror failure*, which is a failure due to activating the failOnWarning option in the configuration file after the dependency update. Since this type of failure relates to the linter configuration rather than the code, we disregard these data points. We discard cases in which, when resolving dependencies, the new version of the dependency conflicts with the dependencies declared in the Maven configuration file, causing errors when invoking API calls (Bono et al. 2024). These failures appear in the *Direct compilation errors* and *Indirect compilation errors* categories. Such cases are related to classpath issues, they are impossible to solve by updating the Java code. This leaves us with a total of 103 breaking dependency updates for our evaluation.

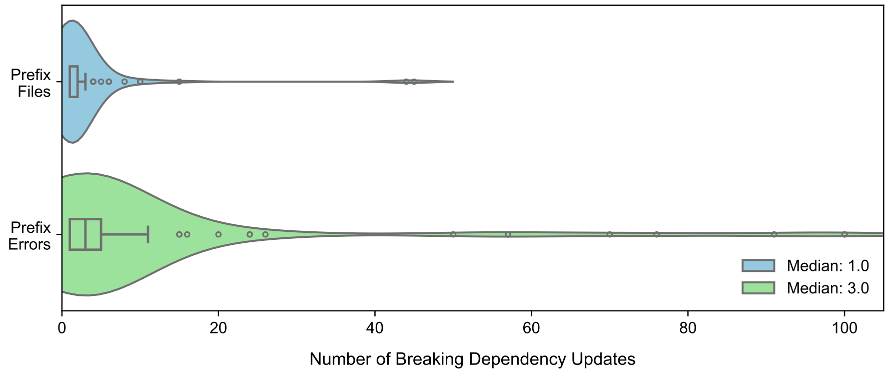


Fig. 10: Distribution of initial error files and initial compilation errors on the 103 breaking dependency updates. Most breakages consist of one file, and there are 3 errors to fix on average.

Figure 10 shows the distribution of the initial files with errors and the initial errors over the 103 breaking updates we consider. The median number of initial files with errors per breaking updates is 1 and the median number of initial errors is 3. The figure shows that the distribution is right-skewed. Specifically, 70% of the updates affect only 1 file. The highest number of initial files with errors is found in the project `billy` when updating the dependency `jaxb2-basics-runtime` from version `0.13.1` to version `1.11.1`, affects 45 files. The 75% of the 103 breaking updates result in between 1 and 5 compiler errors. For example, in project `ChangeSkin`, where the update of dependency `spongeapi` from version `7.4.0` to `8.0.0` causes a compilation failure, resulting in errors in 15 different files and a total of 91 compilation errors. In general, the more the errors, the harder it is to fix a breaking update.

4.3 Experimental Protocol

4.3.1 Protocol for RQ1

This research question aims to investigate how effective is Byam in fixing build failures that are caused by breaking dependency updates. We run Byam with the prompt configurations of Table 2 on the .103 breaking dependency updates of our experiments.

As success metric for answering RQ1, we define the **Build Success Rate (BSR)** as the proportion of initially failing projects that successfully build after applying Byam fixes:

$$BSR = \frac{N_{\text{fixed\_builds}}}{N_{\text{initially\_failing\_builds}}} \tag{1}$$

where $N_{\text{fixed\_builds}}$ is the number of projects that successfully build after applying fixes, and $N_{\text{initially\_failing\_builds}}$ is the total number of projects that originally failed to build due to a dependency update.

4.3.2 Protocol for RQ2

Byam may partially fix the build, while leaving some errors unfixed. We want to know how well it performs when it does not succeed to completely repair the build. Hence, we consider two additional levels of granularity that are finer than the build success level, and are specifically measured on builds that remain failing after applying the generated fixes: (1) the *file level*, which examines the number of source files that were fixed and no longer contain compilation errors, and (2) the *compilation error level*, which evaluates the fixed compilation errors across the project. We define the following success metrics to answer RQ2:

File Fix Success Rate (FFSR) – The percentage of files originally containing compilation errors that were fixed, from failed repairs:

$$FFSR = \frac{N_{\text{fixed\_files}}}{N_{\text{initially\_erroneous\_files}}} \tag{2}$$

where $N_{\text{fixed\_files}}$ is the number of Java files that no longer have compilation errors, and $N_{\text{initially\_erroneous\_files}}$ is the number of Java files that contained compilation errors before applying fixes.

Compilation Error Fix Rate (CEFR) – The percentage of fixed compilation errors, from failed repairs:

$$CEFR = \frac{N_{\text{fixed\_errors}}}{N_{\text{initial\_errors}}} \tag{3}$$

where $N_{\text{fixed\_errors}}$ is the total number of compilation errors successfully fixed, and $N_{\text{initial\_errors}}$ is the total number of compilation errors originally present.

We experiment with the different configurations as detailed in Section 3.3.

4.3.3 Protocol for RQ3

An LLM may be able to fix some errors at the cost of introducing many additional errors. Those errors are bad for developers: they have to fix them manually, potentially needing more work than the initial breaking update errors. If the number of introduced errors is too high, automated correction may impose an additional debugging effort, making manual updates more preferable. To quantify this trade-off, we define a metric that focuses on the extent to which new errors arise due to LLM-generated patches. We define the **Relative Error Fixed Ratio (REF)** metric in Equation 4. The metric is a percentage that ranges from 100% to infinite negative percentages since the best the LLM can do is fix all errors leading to 100% relative error fixes efforts, but it can introduce any number of new errors leading to an infinite number of negative percentages of relative error fixes effort (i.e., added effort).

$$REF = \frac{N_{\text{fixed\_errors}} - N_{\text{new\_errors}}}{N_{\text{initial\_errors}}} \tag{4}$$

where $N_{\text{fixed\_errors}}$ is the number of errors successfully fixed by Byam, $N_{\text{new\_errors}}$ is the number of new errors introduced after the code updates, and $N_{\text{initial\_errors}}$ is the number of errors originally caused by the breaking dependency update.

4.4 Language Models for Experimentation

We select the models we experiment with based on the results of RepairBench (Silva and Monperrus 2024) and LiveCodeBench (Jain et al. 2024), which are established evaluation frameworks for LLMs in program repair. Based on these frameworks, we select the following five LLMs to use in our evaluation:

- Google Gemini-2.0 Flash [2] because it provides a strong balance between speed and accuracy.
- OpenAI GPT4o-mini [3] because it offers a lightweight yet powerful alternative to larger models, maintaining high-quality code generation while being more cost-efficient.

[2] https://ai.google.dev/gemini-api/docs/models/gemini#gemini-2.0-flash

[3] https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence

Table 3: LLMs used in our experiments

Feature	Gemini 2.0 Flash	GPT-4o Mini	o3-mini	DeepSeek V3	Qwen2.5-32B-instruct
Model Provider	Google	OpenAI	OpenAI	DeepSeek	Alibaba
Inference Provider	Google	OpenAI	OpenAI	OpenRouter	OpenRouter
Model Open Source	No	No	No	Yes	Yes
Number of Parameters	Not Disclosed	Not Disclosed	Not Disclosed	671B	32.5B
Input Token Limit	1,048,576	128,000	200,000	131,000	131,000
Output Token Limit	8,192	16,000	100,000	131,000	131,000
Pricing (per 1M tokens)	In: $0.10 Out: $0.40	In: $0.15 Out: $0.60	In: $1.10 Out: $4.40	In: $0.90 Out: $0.90	In: $0.79 Out: $0.79
Knowledge Cutoff	August 2024	October 2023	October 2023	Not Disclosed	Not Disclosed

- OpenAI o3-mini [4] because it leads the results in RepairBench at the time of writing.
- Alibaba Qwen2.5-32b-instruct [5] because it is the leading open source model in the 32B parameter range at the time of writing in LiveCodeBench.
- DeepSeek V3 [6] because it has great cost-efficiency and robust performance.

Table 3 shows a comparison of the five models according to different aspects like their provider, number of parameters, and input and output token limits. Our study deliberately focuses on small and medium-sized, cost-effective models to enable large-scale, reproducible experimentation across many real breaking dependency updates and projects. To reduce sampling noise, we set the temperature to 0, which minimizes randomness; the remaining limitations of this choice are discussed in Subsection 7.1. This design choice aims to balance scalability, cost, and replicability in our evaluation.

5 Experimental Results

We now answer our three research questions introduced above.

[4] https://openai.com/index/openai-o3-mini

[5] https://huggingface.co/Qwen/Qwen2.5-32B-Instruct

[6] https://github.com/deepseek-ai/DeepSeek-V3

5.1 **RQ1** ***(Repair Success):*** *How effective is Byam in fixing compilation errors due to breaking dependency updates?*

To answer RQ1, we evaluate the build repair success of Byam using the different LLMs listed in Section 4.4 and the different prompt configurations defined in Section 3.3. We calculate the **Build Success Rate(BSR)** as defined in Equation 1. We summarize the results in Table 4.

Table 4: Build Success Rate (BSR) of Byam

Prompt ID	**Deepseek V3**	**Gemini 2.0-flash**	**Gpt 4o-mini**	**o3-mini**	**Qwen2.5-32b-instruct**
P_1	16/103 (16%)	15/103 (15%)	15/103 (15%)	24/103 (23%)	9/103 (9%)
P_2	15/103 (15%)	18/103 (17%)	18/103 (17%)	24/103 (23%)	11/103 (11%)
P_3	19/103 (18%)	21/103 (20%)	13/103 (13%)	26/103 (25%)	9/103 (9%)
P_4	14/103 (14%)	17/103 (17%)	16/103 (16%)	27/103 (26%)	11/103 (11%)
P_5	20/103 (19%)	16/103 (16%)	14/103 (14%)	19/103 (18%)	11/103 (11%)
P_6	22/103 (21%)	17/103 (17%)	16/103 (16%)	22/103 (21%)	12/103 (12%)
P_7	20/103 (19%)	7/103 (7%)	14/103 (14%)	25/103 (24%)	13/103 (13%)
P_8	19/103 (18%)	18/103 (17%)	15/103 (15%)	**28/103 (27%)**	11/103 (11%)

Table 4 presents the **BSR**, indicating the percentage of builds that are successfully repaired by each LLM under different prompt configurations. Each row of the table represents a different configuration (P_1 to P_8 shown in Table 2), while each column corresponds to one of the five evaluated LLMs presented in Table 3. Each cell indicates the corresponding build success rate by Byam after applying the generated code, we represent that value in the following format *total number of fixed builds / total number of broken builds (build success rate).*

The highest success rate in this evaluation is achieved by o3-mini using P_8 (CoT + Erroneous Line + API Diff), with 27%, meaning 28 out of the 103 original failing builds were completely repaired, with compilation and test execution succeeding. Overall, o3-mini had the highest build success rate across most prompts. Relatively high performance is also achieved by Deepseek V3 performing second-best in cases P_1, P_5, P_6 and Gemini-2.0-flash performing second-best in P_2, P_3, P_4, P_8

In contrast, Qwen2.5-32b-instruct has the lowest success rate across most prompts. This is because this is the smallest model considered, orders of magnitude smaller than the frontier models considered.

Meanwhile, Gemini-2.0-flash shows an increase when API Diff is introduced: for example, it improves from P_1 at 15% to P_3 at 20%. Interestingly, the performance of Gemini-2.0-flash improves when adding more context to the baseline prompt in P_2 and P_3.

The introduction of Chains-of-thought (CoT) in the prompt is beneficial for some models (o3-mini) and detrimental for others (Gemini-2.0-flash) In the latter case, the lowest performance for Gemini with P_7 at 7% contains CoT.

Looking at the open-source model, Qwen2.5-32b-instruct, it benefits a little from CoT and API Diff, improving from 9% in P_1 to 13% in P_7.

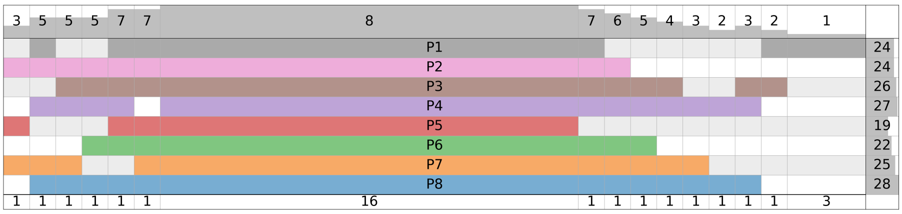


Fig. 11: Visualization of Build Success by o3-mini Across Prompts. Each row represents the set of commits repaired using the prompt labeled on that row. The numbers on the right indicate the total number of commits repaired by each prompt. The bottom number represents intersections ("chunks") of commits that were repaired by multiple prompts. The number at the top of each column indicates how many prompts are involved in that intersection. Overlapping areas correspond to the set intersections among the prompts.

Given that o3-mini has the highest build success rate, we further analyze its fixes in Figure 11. The figure illustrates the number of breaking commits successfully fixed by o3-mini across prompts, and the overlap between prompts. It shows each prompt (P_1 through P_8) as a horizontally colored row. The right-side number indicates how many commits each prompt successfully fixes in total. Along the upper part of the diagram, the numbers indicate how many prompts overlap in fixing a particular set of breaking updates, and the numbers at the bottom of the diagram are the actual count of these commits. The width of each 'chunk' represents how many breaking updates fall into those intersections between prompts. For example, the chunk labeled '8' at the top and '16' at the bottom indicates that there are 16 breaking updates successfully fixed by all eight prompts. Overall, P_8 fixed the most commits (28), while P_4 fixed the least breaking updates (20), confirming the importance of API Diff and Cot.

The diagram shows that out of the highest number of fixed breaking updates (28) by P_8, 16 of those were commonly fixed across every prompt (15% of total builds). Notably, some smaller chunks (e.g, labeled '3' at the top and '1' at the bottom) show that some breaking updates were only fixed by a couple of prompts. Oddly enough, three breaking updates were only fixed by the baseline simple prompt P_1.

Given that P_8 is the prompt that provides the highest number of build success, we now consider the output of P_8 across different LLMs in Figure 12. Similar to Figure 11, the figure illustrates the number of breaking updates successfully fixed by P_8 across LLMs. It shows the number of breaking updates the LLMs successfully fix with the prompt configuration P_8, as a horizontally colored rows. The right-side numbers indicate how many commits each LLM

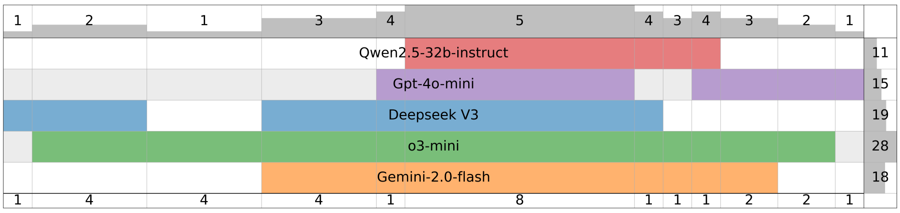


Fig. 12: Visualization of Build Success by P_8 Across LLMs. Each row represents the set of commits repaired by the LLM labeled on that row. The numbers on the right indicate the total number of commits repaired. The bottom number represents intersections ("chunks") of commits that were repaired by multiple LLMs. The number at the top of each column indicates how many LLMs are involved in that intersection. Overlapping areas correspond to the set intersections among the LLMs.

successfully fixes using P_8. The numbers in the upper part of the diagram indicate how many LLMs overlap in fixing a particular set of commits. The numbers at the bottom of the diagram are the actual count of these commits.

The chunk labeled '5' at the top and '8' at the bottom indicates that there are 8 commits successfully fixed by all five LLMs using the P_8 prompt. It also shows that there is one breaking update only fixed by Deepseek V3, and another one only fixed by Gpt4o-mini. There are four breaking updates that are only fixed by o3-mini which contribute to its top performance. Overall, Figure 12 confirms the superiority of o3-mini, which is capable of overlapping with other good models (DeepSeek-v3, gpt4o-mini, fixing unique breaking updates.

Answer to RQ1: It is possible to fix real-world breaking updates with LLMs. o3-mini with prompt P_8 (CoT + Erroneous Line + APIDiff) achieves the highest build success rate, fixing 27 out of the 103 original failing builds (28%). The addition of API Diff and Chains-of-thought makes a difference, yet a relatively small one. Open-source model Qwen2.5-32b-instruct, has the lowest success rate, as expected. Overall, the inherent capability of the model is more important than the prompting itself for fixing breaking updates.

5.2 **RQ2** ***(Partial Repair):*** *To what extent does Byam partially fix compilation errors at the file and error level?*

To address this research question, we evaluate the extent to which Byam can partially fix compilation errors in builds that still fail after applying the generated fixes. We analyze Byam's performance at two levels of granularity: the file level, which considers the number of completely fixed files, and the error level, which considers the number of individual compilation errors fixed.

Table 5: Partial Repair (RQ2): File Fix Success Rate of Byam over the failed cases

Prompt	**Deepseek**	**Gemini**	**Gpt**	**o3**	**Qwen2.5**
ID	V3	2.0-flash	4o-mini	mini	32b-instruct
P_1	48/252(19%)	48/251(19%)	40/256(16%)	75/242(31%)	41/262(16%)
P_2	61/254(24%)	54/248(22%)	46/253(18%)	66/241(27%)	44/263(17%)
P_3	62/249(25%)	82/246(33%)	53/262(20%)	89/246(36%)	61/265(23%)
P_4	64/255(25%)	87/251(35%)	53/255(21%)	**97/239(41%)**	46/263(17%)
P_5	52/251(21%)	59/260(23%)	36/254(14%)	72/252(29%)	63/268(24%)
P_6	42/248(17%)	39/243(16%)	40/254(16%)	74/243(30%)	62/263(24%)
P_7	59/244(24%)	71/253(28%)	57/260(22%)	88/238(37%)	61/264(23%)
P_8	71/248(29%)	76/250(30%)	53/256(21%)	92/246(37%)	54/267(20%)

Table 5 presents the **File Fix Success Rate (FFSR)**, indicating the percentage of files that initially contained compilation errors and were successfully repaired by each LLM under different prompt configurations on builds that failed after applying the generated code. Each row in the table refers to a different prompt configuration shown in Table 2 and each column refers to one of the five LLMs presented in Table 3. Each cell contains the respective success rate of the files fixed by Byam after applying the generated fixes code.

It is clearly that Byam gets lots of partial fixes, in the double digits range. For example, when it fails to completely repair the build, o3-mini with P_4 (Erroneous Line + API Diff) is able to fix 97 of the 239 original files with compilation failures (41%). All models are able to fix dozens of files.

At the prompt level, adding erroneous lines and APIDiff information improves partial repair, confirming RQ1. For instance, Gemini-2.0-flash improves from 19% (48/251) with the baseline prompt (P_1) to 35% (87/251) with P_4, which shows that the addition of APIDiff and erroneous lines in the configurations improves the performance of the model.

This is more evidence that the inclusion of APIDiff in the prompts assists the model to understand the nature of the breakage more clearly and to generate a more appropriate fix. On the other hand, GPT-4o-mini shows low values with the CoT prompt, achieving only 14% for P_5. However, its performance improves notably when using APIDiff and erroneous line information. This highlights how reasoning plays a significant role in influencing model performance. However, the addition of CoT improves performance in Qwen2.5-32b-instruct, from 16% in P_1 to 24% in P_5 and P_6. This indicates that CoT positively impacts the performance for fixing error files in specific models.

At the compilation error level, Table 6 shows the **Compilation Error Fix Rate (CEFR)** over failed repairs. Each row represents prompts, columns represent LLM, and each cell represents the fixed error rate. Strikingly, up to 78% of errors are repaired in unsuccessful builds. In other words, for those cases, the models are almost able to fix the build, a promising result for practitioners. o3-mini is again the best with P_8 prompt design, with 78%, fixing 741/955 errors, demonstrating high efficiency in fixing individual compilation failures.

Table 6: Partial Repair (RQ2): Compilation Error Fix Rate of Byam over failed cases

Prompt ID	**Deepseek** V3	**Gemini** 2.0-flash	**Gpt** 4o-mini	**o3** mini	**Qwen2.5** 32b-instruct
P_1	548/938(58%)	680/959(71%)	490/965(51%)	705/941(75%)	529/979(54%)
P_2	645/941(69%)	661/955(69%)	548/966(57%)	683/916(75%)	491/983(50%)
P_3	553/935(59%)	679/931(73%)	696/979(71%)	736/955(77%)	710/987(72%)
P_4	614/942(65%)	687/936(73%)	670/964(70%)	726/938(77%)	622/983(63%)
P_5	555/937(59%)	684/978(70%)	531/962(55%)	731/964(76%)	720/994(72%)
P_6	534/933(57%)	654/943(69%)	536/973(55%)	712/944(75%)	712/988(72%)
P_7	664/921(72%)	711/962(74%)	696/976(71%)	723/937(77%)	669/986(68%)
P_8	679/934(73%)	714/959(74%)	680/973(70%)	**741/955(78%)**	602/997(60%)

For Qwen2.5-32b-instruct, the behavior is inconsistent. Despite achieving its best result with P_3, P_5 and P_6 (72%), it drops to 60% in P_8, suggesting that prompts with multiple elements may overfit the model, decreasing its effectiveness. It even shows marked drops in advanced configurations such as P_6 and P_7.

Consider the case of the `openfire-hazelcast-plugin` project. Updating the `hazelcast` dependency from version `7.4.0` to `8.0.0` introduced a total of 15 buggy files and 91 compile errors. After applying the code generated by Byam (using o3-mini with P_8), 2 of the 15 buggy files are completely repaired, and 86 of the 91 compilation errors are successfully fixed. Although not all affected files are restored, Byam manages to resolve 94% of the individual errors, demonstrating a great ability to perform fine repairs even when a complete compilation fix is not reached. For example, in the `ChangeSkingSpponge` file, Byam correctly adapts the class signature, changing the generic type from `CommandSource` to `Audience` as shown in Listing 1 In another case, Byam fails to resolve the error `can't find symbol symbol: class Plugin`. Listing 2 shows the change proposed by Byam, which fails to fix the original error.

```
−  public class ChangeSkinSponge implements PlatformPlugin<CommandSource> {
+  public class ChangeSkinSponge implements PlatformPlugin<Audience> {
```

Listing 1: Changes introduced by Byam correctly fixing the individual error

```
−  @Plugin(id = ARTIFACT_ID, name = PomData.NAME, version = PomData.VERSION,
+  @Plugin(id = PomData.ARTIFACT_ID, name = PomData.NAME, version = PomData.
     VERSION, {
```

Listing 2: Changes introduced by Byam failing to fix an error

Answer to RQ2: Our results show that, even when the build is not fully repaired, a large number of errors are actually fixed. o3-mini, combined with prompts that include APIDiff and CoT, fixes 41% of files and an impressive 78% of individual compiler errors. This suggests that the top-of-the-line models are close to perfectly repair many more builds, a trend we expect to happen with the next generation of models, especially the reasoning models.

5.3 **RQ3** ***(New Error Introduction):*** *To what extent do language models tend to introduce new errors during the breaking update repair process?*

To answer this question, we evaluate the relative error fixed by Byam, considering the number of fixed errors as well as any new errors introduced by the LLM generated fix. We use the different LLMs listed in Table 4.4 and the different prompt configurations defined in Section 3.3. We calculate the **Relative Error Fixed Ratio (REF)** as defined in Equation 4 per breaking build for each configuration (Prompt and LLM). For this metric, the higher the better, meaning more fixed errors than introduced ones.

We summarize the results in Table 7, where we show the median value of the **REF**. Across all breaking builds, o3-mini performs the best, delivering the highest median REF on every prompt, peaking at 93.33% with P_4 (Erroneous Line and APIDiff). It also shows promising stability, never dropping below 66.67 %, which shows that the model is robust enough that providing it with basic information (P_1-P_5) or more advanced reasoning and contextual information ($P6$-$P8$) will not result in new errors.

In contrast, Qwen-2.5-32B-instruct fails according to this metric (with a median of 0% for all prompts), indicating that the model either does not help in fixing errors or introduces more new errors than it fixes.

DeepSeek-v3 benefits the most from the full context prompt P_8, with an increased median REF of 80.00. These results are consistent with the results of RQ1 we present in Section 5.1.

When we consider the results from the prompts viewpoint, we see that the improvement of P_2 (Erroneous Line) is consistently better, boasting the results of four of the five models. On the contrary, P_5 (CoT) is the least consistently effective, helping only o3-mini. This indicates that adding the exact erroneous line is a valuable prompt addition to most LLMs. CoT has an added value only when the model supports reasoning, such as o3-mini.

Next, we investigate why new compilation errors appear. To do so, we manually analyze a random sample of new errors. Our first example is from the `WorldwideChat` project, where the dependency update of `XSerie`s from version `8.5.0.1` to `8.6.0` fails because the method `parseEnchantment()` was removed in the new version of the dependency. o3-mini with P_8 introduces a new error when generating the patch to fix the error. The model replaces the call to the `parseEnchantment()` method with `getEnchantment()`. The

Table 7: Median of Relative Error Fixed Ratio of Byam

Prompt	**Deepseek**	**Gemini**	**Gpt**	**o3**	**Qwen2.5**
ID	V3	2.0-flash	4o-mini	mini	32b-instruct
P_1	50.00%	50.00%	25.00%	80.00%	0.00%
P_2	66.67%	79.17%	50.00%	80.00%	0.00%
P_3	60.00%	50.00%	33.33%	66.67%	0.00%
P_4	45.00%	40.00%	33.33%	**93.33**%	0.00%
P_5	50.00%	20.00%	20.83%	79.17%	0.00%
P_6	38.00%	50.00%	40.00%	80.00%	0.00%
P_7	60.00%	40.00%	50.00%	85.71%	0.00%
P_8	80.00%	40.00%	42.86%	80.00%	0.00%

```diff
- currentLangMeta.addEnchant(XEnchantment.matchXEnchantment("power").get().
  parseEnchantment(), 1, false);
+ currentLangMeta.addEnchant(XEnchantment.matchXEnchantment("power").get().
  getEnchantment(), 1, false);
```

Listing 3: Changes introduced By Byam to fix a real breaking dependency updates

method proposed by LLM does not exist in the new version of the dependency. Listing 3 shows the change applied by LLM that triggers the new compilation error.

Answer to RQ3: Our results show that o3-mini is the model that introduces the least new errors. Prompt-wise, providing the erroneous line (P_2) yields the most gains across all LLMs to avoid new error introduction. Models with reasoning abilities are the ones which introduce the least new errors, benefiting more from the added contextual information.

6 Discussion

In this section, we reflect on our findings and their implications for both future research and practical use. Subsection 6.1 compares Byam with prior work, situating our results in relation to existing approaches. Subsection 6.2 analyzes the main causes of unsuccessful repairs , highlighting the limits of fixes. Finally, Subsection 6.5 discusses how Byam can be integrated into developer workflows and outlines promising directions for future wrk.

6.1 Comparison with Prior Work

In Frunkte and Krinke (2025), the authors introduce two new approaches to fix breaking dependency updates using LLMs: a zero-shot prompting method and an agent system. Their agent iteratively attempts up to 30 fixes per case. The authors process the Bump dataset to produce two distinct evaluation data sets, the first set is considered as "light slice" with 65 projects with breaking dependency updates caused by only one file, and a "full slice" of 140 projects that has multiple files causing breaking in build. The authors test the zero-shot approach on the light-slice dataset while they test the agent system on both datasets. The zero-shot baseline uses no prompt engineering or design and is evaluated only on the light-slice dataset. The agent system, by contrast, is evaluated on both slices and operates iteratively, making up to 30 repair attempts per case. Importantly, the study does not report single-iteration success rates for the agent system, as its evaluation focuses on cumulative results across multiple trials.

Byam's key advantage over the work in Frunkte and Krinke (2025) is its context-rich prompting, evident in being able to achieve better performance using a less complicated approach . Byam fixed full builds that can consist of multiple file edits, achieving a success rate of 27% using the richest prompt (P8). Even when the full build does not succeed, Byam achieves a 78% fix rate of the individual compilation errors.

In contrast, the agent approach in Frunkte and Krinke (2025) fixes a maximum of 23% of builds in the full slice and a maximum of 19% in the light-slice, with the zero-shot approach achieving a maximum success rate of 19%. We note that Frunkte and Krinke (2025) does not report the average trials for fixes but allow iteration for up to 30 trials.

To provide a direct comparison of identical breaking dependency updates, we analyze the intersection of breaking dependency updates evaluated by both studies. The intersection between the 140 ("full slice") breaking updates presented by Frunkte and Krinke (2025) and the 103 breaking updates analyzed here yields 97 common breaking dependency updates. When evaluating Byam with o3-mini using the P_8 configuration on these 97 shared cases, Byam achieves a 27% (26/97) success rate, compared to the agent approach's 20% (19/97) performance on the same breaking dependency updates. This controlled comparison on the same failing builds eliminates dataset variability and demonstrates that our contextualized prompting strategy outperforms the agent-based approach.

To sum up, the key strength and novelty of Byam is contextualize the breaking update problem in the prompt with essential information: by considering the line that is causing the errors, structuring the API diffs to the parts relevant to the project under investigation, and utilizing chain-of-thought prompting.

6.2 Unsuccessful Repairs and Test Failures

Our work aims at repairing compilation failures. Even when full builds are not repaired, Byam often generates partial repairs that provide practical assistance. For example, with the o3-mini model, Byam fixes 78% of individual errors and 41% of erroneous files in otherwise failing builds. This suggests that many builds fail only due to a small number of remaining issues, and Byam can significantly reduce developer effort by handling the bulk of the errors automatically.

Yet, once compilation failures are resolved, we have sometimes test failures, indicating that semantic changes have occurred in the library. For example, in project `pay-adminusers`, the dependency update of `logback-classic` from version `1.2.11` to `1.4.5` fails due to compilation error `cannotaccess org.slf4j.spi.LoggingEventAwareclassfilefororg.slf4j.spi.LoggingEventAwarenotfound`. The compilation error is fixed by replacing the class `ch.qos.logback.classic.Logger` with the generic slf4j interface `org.slf4j.Logger`. However, during testing, the method `doAppend()` was never invoked, resulting in the error `Wantedbutnotinvoked:mockLogAppender.doAppend()`. This is an example where the updates by the LLM fix the compilation failure, but result in a test failure.

There are two ways in which we can interpret a test failure outcome after automated fixing by LLMs: (1) the LLM-generated code has introduced logical errors that pass compilation but fail functional tests, or (2) some unmodified parts of the code interact with the updated dependency, which has changed semantics. Distinguishing between these scenarios is a challenging and important direction for future work.

6.3 Recent Related Industry Tools

Over the past year, in parallel with this research, several industrial tools have emerged that use large language models and agents to support software development tasks, including dependency updates and related software maintenance tasks. Some of these tools are “general-purpose development agents” while some specifically focus on dependency updates.

General-purpose coding agents that can be assigned various tasks across the software development process have become increasingly available. Such agents can be standalone CLI tools (e.g., Co-pilot CLI [7] or Gemini CLI [8]) or can be directly integrated into development platforms such as GitHub (e.g. GitHub Copilot [9]) or Integrated Development Environments (IDEs) such as VSCode (e.g., VSCode Copilot [10] or Cursor [11]). Once prompted with a task, the agents

[7] https://github.com/features/copilot/cli

[8] https://geminicli.com/

[9] https://github.com/features/copilot

[10] https://github.com/features/copilot/ai-code-editor

[11] https://cursor.com/agents

can run in the background to complete the assigned task. These agents can be assigned issues to solve, whether the issue is about fixing a bug or implementing a new feature. For GitHub-integrated tools, the agent may create a PR with the required changes based on the prompt or assigned issue. If it is integrated into an IDE, it can start generating or editing files directly in the editor. While these agents do not specifically focus on fixing breaking dependency updates, the task can be viewed as a specific instance of program repair that these agents can handle. Generally speaking, these industrial agents are designed as end-to-end automation pipelines that integrate multiple stages of repair, such as fault diagnosis, patch generation, testing, and deployment—within real development workflows. Accordingly, while developers can use them to fix breaking dependency updates, we do not have empirical evidence about their effectiveness for this specific task.

On the other hand, there are some recent industrial tools that are more closely related to breaking dependency updates. For example, Dynatrace [12] and Develocity [13] provide in-depth monitoring of build status. These tools analyze build and test data, with the aim of providing insights into aspects like performance, reliability, and efficiency. While these tools do not attempt to fix the breaking update themselves, they can potentially provide richer context that can help the LLM provide a correct fix. Experimenting with the effect of this additional context is an interesting avenue for future work.

The closest to our setting are tools that specialize in fixing breaking dependency updates. For example, FOSSA [14] automatically reviews updates for breaking changes and analyzes the code impact of updates, working alongside tools like Dependabot and Renovate to help developers manage dependencies. Patchwork [15] is another example. These tools support automated repair by performing steps such as comparing dependency versions, generating patches, validating fixes, and potentially opening pull requests with proposed changes. However, similar to general-purpose agents, they are designed as end-to-end automation pipelines that integrate multiple stages of repair, rather than enabling in-depth experimentation and analysis of individual steps. In contrast, our study evaluates LLM-based repair under scientifically controlled conditions. We focus on empirical evidence for specific repair principles, including (i) the role of targeted contextual information about the breaking change, (ii) fault localization that highlights failing lines and relevant API changes, and (iii) prompt design choices and how they affect repair outcomes. Our goal is to isolate these factors and analyze how they influence repair effectiveness for breaking changes. Thus, rather than providing a full end-to-end automation solution, our work studies the repair step in isolation to understand which design choices contribute to successful outcomes. Our results can inform improvements to relevant academic and industrial tools.

[12] https://www.dynatrace.com/

[13] https://docs.gradle.com/develocity/current/

[14] https://fossa.com/products/fossabot/

[15] https://docs.patched.codes/patchwork/overview

6.4 Industry versus Academia in Software Research

The previous section demonstrates that since we began this research, the broader AI-assisted software development ecosystem has advanced quickly, especially with advances in the capabilities of large language models and agent-based systems. This forces us to reflect on these pace differences and their impact.

Generally speaking, industrial progress is fast paced with rapid updates in response to user feedback and needs. On the other hand, the requirements of academic research are different. It requires careful experimental design, dataset construction, controlled evaluation, peer review, and reproducibility. In other words, industrial tools drive adoption and deliver end-to-end automation, but they are typically evaluated through operational metrics and real-world usage rather than controlled experimental measurements of individual design choices. As a result, it remains difficult to assess how well their behavior generalizes or to reason about the causes of success and failure. Research, on the other hand, systematically investigates techniques and provides empirical evidence about what works, under which conditions, and why. These differing objectives naturally lead to different evaluation practices and timelines between industrial development and academic research.

6.5 Workflow Integration and Future Work

Applications in the field of tools like Byam require integration. We note that there exist Pull Request (PR) bots, which create pull requests to update dependencies (Dependabot, Renovate). Byam could be integrated into such systems to further automate the process of dependency update. For example, if a PR for a dependency update causes a break in the Continuous Integration (CI) environment, an integrated tool could attempt to fix the breaking dependency update. Such an end-to-end solution to dependency updates would further reduce developers' effort in keeping their dependencies up-to-date, thus making their projects more secure and reliable.

Future work should explore several directions. First, addressing semantic failures may require incorporating additional context from test suites or integrating program analysis to reason beyond syntax. Second, as more powerful reasoning-oriented LLMs emerge, they may further improve repair rates, particularly for complex or multi-file repairs. Finally, while our evaluation focused on Java/Maven projects, the core idea of contextual prompting is generalizable and could be applied to other ecosystems.

7 Threats to Validity

In the following, we discuss internal, construct, and external threats to the validity of our study.

7.1 Internal Validity

A threat identified in the study is the variability in LLM results, which can produce nondeterministic results due to inherent randomness in their operation. As an effort to limit this threat, we set a temperature of zero during inference, which reduces randomness and ensures that the model always chooses the most likely options.

7.2 Construct Validity

Memorization and data leakage are typically a concern in LLM-based techniques. We reviewed the release dates of the dependencies in the dataset, comparing them to the training cutoff dates of each LLM, and confirmed that all release dates precede the training cut-off. As such, there is a potential risk of data leakage, which is similar to that of research based on HumanEval, Defects4j, or SWE-bench. Future work in required to collect breaking updates with new dependency versions released after the models' training cutoff dates.

7.3 External Validity

We used the BUMP dataset, which focuses exclusively on breaking dependency updates in Java projects, which may limit the generalizability of our findings to other programming languages. In terms of generalization across application domains, the dataset was collected from 153 Java projects, and each project was filtered to ensure that it is not a toy project.

8 Related Work

Breaking dependency updates is a common problem in software development. (Brito et al. 2020, 2018b; Xavier et al. 2017) investigate the reasons behind the developer's decision to introduce updates that lead to breaking dependency updates. These reasons include introducing new features, refactoring the code, fixing bugs, and addressing security vulnerabilities. Hejderup et al. analyze how changes in method behavior trigger incompatibilities despite preserving the API contract. Such changes are difficult to detect by static analysis and often only become apparent at runtime (Hejderup and Gousios 2021). (Venturini et al. 2023) emphasized the challenges developers face in updating client code after dependency updates. Researchers investigated various approaches to address this issue, where some investigated detecting breaking dependency updates (Brito et al. 2018a; Mujahid et al. 2020). (Reyes et al. 2024a) recently introduced an automated approach to explain the breaking in the client code after the version update. Our approach differs from the previous studies as it uniquely addresses compilation failures caused by breaking updates through fixes generated using LLM. Byam originally exploits contextual information such as APIDiff and erroneous lines to augment prompts, a dimension unexplored in previous studies.

Different approaches to mitigate breaking dependency updates are built on rule-based program transformation techniques. (Dagenais and Robillard 2009) introduces a novel approach to API evolution by recommending adaptive replacements for deleted methods through analysis of repository history. (Xing and Stroulia 2007) presents Diff-CatchUp, a tool that assists client application migration by automatically identifying broken APIs, suggesting plausible replacements, and providing usage examples based on model differencing and the framework's actual working code. More recent research focuses on learning migration patterns from existing code samples (Lamothe et al. 2022), analyzing changes in customer projects that have already migrated (Xu et al. 2019), and leveraging documentation (Ni et al. 2021). These approaches are intended to reduce the manual effort required to update dependencies and increase the likelihood that developers maintain updated dependencies. Contrary to rule-based systems, our approach dynamically adapts to different types of breaking changes by integrating APIDiff and CoT reasoning. This eliminates the need for manual engineering of rules or migration examples by leveraging the generalization capabilities of LLMs.

Large Language Models (LLMs) have emerged as a solution to automate API migration and dependency updates. Almeida et al. explored the use of ChatGPT for these tasks (Almeida et al. 2024). Compared to rule-based approaches, LLMs can handle contextual variations in the code, which makes them more effective in certain transformations (Nikolov et al. 2025). Tools based on LLMs have been developed to address specific migration problems. RELANCER employs machine learning to predict repairs on deprecated APIs in Jupyter notebooks (Zhu et al. 2021). PCART automates compatibility assessment and repair of API parameter changes in Python (Zhang et al. 2024). In addition, Liu et al. note that LLMs can repair tests affected by code changes (Liu et al. 2024).

Our approach differs from previous research as it focuses on fixing Java compilation errors across build, file, and error levels.

The most closely related work to our study is presented in parallel research by (Frunkte and Krinke 2025). The authors explore automated build repair using either zero-shot prompting or agent-based iterative repair loops without explicit prompt engineering, achieving at most 23% repair on the full-slice dataset. Byam improves on this by incorporating contextual information, reaching 27% success on the same dataset of builds in a single iteration and fixing 78% of individual compilation errors.

9 Conclusion

In this paper, we have presented Byam, a novel approach that uses large language models to fix compilation failures caused by breaking dependency updates. We have provided an empirical analysis of Byam using the BUMP dataset, using five different notable models and experimenting with eight different prompt configurations. Our prompts incorporate contextual information such as the differences between the dependency versions, the lines causing the compilation failure, and providing a set-by-step reasoning in the prompt.

We have analyzed the results at three granularity levels: at the build level, the file level, and the individual compilation error level. Our results demonstrate the promising use of LLMs for fixing breaking dependency updates. In particular, the o3-mini model is able to successfully repair 27% of failed builds, fixing 41% of errors at the file level, and completely removing 78% of individual errors. The inclusion of APIDiff and CoT prompts improves performance, showing the importance of contextualized prompt designs to maximize the efficiency of LLMs to fix breaking dependency updates.

Overall, our results demonstrate the potential for LLMs to fix breaking dependency updates. This is an essential automation step to help developers with keeping their projects' dependencies up-to-date.

Declarations

Funding: This work was supported by the CHAINS project funded by Swedish Foundation for Strategic Research (SSF), the WebInspector project funded by Swedish Research Council (VR), as well as by the Wallenberg Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Some computation was enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS).

Conflict of Interest: None

Ethical approval The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper. All data and code related to this paper is available at https://github.com/chains-project /bacardi

Author Contributions **Frank Reyes** contributed to the experimental design, data collection, data analysis, and writing. **May Mahmoud** contributed to the experimental design, data collection, data analysis, and writing. **Federico Bono** contributed to the experimental design and data collection. **Sarah Nadi** contributed to data analysis and writing. **Benoit Baudry** contributed to data analysis and writing. **Martin Monperrus** contributed to data analysis and writing.

Generative AI: Generative AI was not used for the generation of any part of the content in this paper or fordata analysis. Grammarly, a tool that uses AI, was used for spell checking, grammar correction, and improving writing clarity.

Clinical Trial Number Not applicable

References

Almeida, A., Xavier, L., and Valente, M. T. (2024). Automatic Library Migration Using Large Language Models: First Results. In *International Symposium on Empirical Software Engineering and Measurement.*

Bono, F., Reyes, F., Sharma, A., Baudry, B., and Monperrus, M. (2024). Java-Class-Hijack: Software Supply Chain Attack for Java based on Maven Dependency Resolution and Java Classloading.

Brito, A., Valente, M. T., Xavier, L., and Hora, A. (2020). You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25:1458–1492.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018a). APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 507–511. IEEE.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018b). Why and how Java developers break APIs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265. IEEE.

Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., et al. (2024). A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45.

Dagenais, B. and Robillard, M. P. (2009). SemDiff: Analysis and recommendation support for API evolution. In *2009 IEEE 31st International Conference on Software Engineering*, pages 599–602. IEEE.

Dietrich, J., Pearce, D., Stringer, J., Tahir, A., and Blincoe, K. (2019). Dependency Versioning in the Wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 349–359.

Frunkte, L. and Krinke, J. (2025). Automatically Fixing Dependency Breaking Changes. In *Proc. ACM Software Engineer. 2.*

Hejderup, J. and Gousios, G. (2021). Can We Trust Tests To Automate Dependency Updates? A Case Study of Java Projects. *J. Syst. Softw.*, 183:111097.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. (2024). LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code.

Lamothe, M., Guéhéneuc, Y.-G., and Shang, W. (2021). A systematic review of API evolution literature. *ACM Computing Surveys (CSUR)*, 54(8):1–36.

Lamothe, M., Shang, W., and Chen, T.-H. P. (2022). A3: Assisting Android API Migrations Using Code Examples. *IEEE Transactions on Software Engineering*, 48:417–431.

Larios Vargas, E., Aniche, M., Treude, C., Bruntink, M., and Gousios, G. (2020). Selecting third-party libraries: The practitioners' perspective. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 245–256.

Le Goues, C., Pradel, M., and Roychoudhury, A. (2019). Automated program repair. *Communications of the ACM*, 62(12):56–65.

Liu, J., Yan, J., Xie, Y., Yan, J., and Zhang, J. (2024). Fix the Tests: Augmenting LLMs to Repair Test Cases with Static Collector and Neural Reranker. *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 367–378.

Maven (2022). japicmp-base.

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

Mujahid, S., Abdalkareem, R., Shihab, E., and McIntosh, S. (2020). Using others' tests to identify breaking updates. In *Proceedings of the 17th international conference on mining software repositories*, pages 466–476.

Ni, A., Ramos, D., Yang, A. Z. H., Lynce, I., Manquinho, V. M., Martins, R., and Goues, C. L. (2021). SOAR: A Synthesis Approach for Data Science API Refactoring. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 112–124.

Nikolov, S., Codecasa, D., Sjovall, A., Tabachnyk, M., Chandra, S., Taneja, S., and Ziftci, C. (2025). How is Google using AI for internal code migrations? *ArXiv*, abs/2501.06972.

Ochoa, L., Degueule, T., Falleri, J.-R., and Vinju, J. (2022). Breaking bad? semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study. *Empirical Software Engineering*, 27(3):61.

Reyes, F., Baudry, B., and Monperrus, M. (2024a). Breaking-Good: Explaining Breaking Dependency Updates with Build Analysis. In *Proceedings of IEEE International Conference on Source Code Analysis and Manipulation.*

Reyes, F., Gamage, Y., Skoglund, G., Baudry, B., and Monperrus, M. (2024b). BUMP: A Benchmark of Reproducible Breaking Dependency Updates. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 159–170.

Salza, P., Palomba, F., Di Nucci, D., D'Uva, C., De Lucia, A., and Ferrucci, F. (2018). Do developers update third-party libraries in mobile apps? In *Proceedings of the 26th conference on program comprehension*, pages 255–265.

Silva, A. and Monperrus, M. (2024). RepairBench: Leaderboard of Frontier Models for Program Repair. Technical Report 2409.18952, arXiv.

Vaswani, A. (2017). Attention is all you need. *Advances in Neural Information Processing Systems.*

Venturini, D., Cogo, F. R., Polato, I., Gerosa, M. A., and Wiese, I. S. (2023). I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–26.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. (2023). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.

Xavier, L., Hora, A., and Valente, M. T. (2017). Why do we break APIs? first answers from developers. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 392–396. IEEE.

Xing, Z. and Stroulia, E. (2007). API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836.

Xu, S., Dong, Z., and Meng, N. (2019). Meditor: Inference and Application of API Migration Edits. *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 335–346.

Zhang, L., Liu, C., Xu, Z., Chen, S., Fan, L., Chen, B., and Liu, Y. (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*, pages 1–12.

Zhang, S., Xiao, G., Wang, J., Lei, H., Liu, Y., and Zheng, Z. (2024). PCART: Automated Repair of Python API Parameter Compatibility Issues. *ArXiv*, abs/2406.03839.

Zhu, C., Saha, R. K., Prasad, M. R., and Khurshid, S. (2021). Restoring the Executability of Jupyter Notebooks by Automatic Upgrade of Deprecated APIs. *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 240–252.