Automatically Fixing Dependency Breaking Changes

LUKAS FRUNTKE, University College London, United Kingdom
JENS KRINKE, University College London, United Kingdom

Breaking changes in dependencies are a common challenge in software development, requiring manual intervention to resolve. This study examines how well Large Language Models (LLMs) automate the repair of breaking changes caused by dependency updates in Java projects. Although earlier methods have mostly concentrated on detecting breaking changes or investigating their impact, they have not been able to completely automate the repair process. We introduce and compare two new approaches: an agentic system that combines automated tool usage with LLMs, and a recursive zero-shot approach, employing iterative prompt refinement. Our experimental framework assesses the repair success of both approaches, using the BUMP dataset of curated breaking changes. We also investigate the impact of variables such as dependency popularity and prompt configuration on repair outcomes. Our results demonstrate a substantial difference in test suite success rates, with the agentic approach achieving a repair success rate of up to 23%, while the zero-shot prompting approach achieved a repair success rate of up to 19%. We show that automated program repair of breaking dependencies with LLMs is feasible and can be optimised to achieve better repair outcomes.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Automated program repair, dependency management

ACM Reference Format:
Lukas Fruntke and Jens Krinke. 2025. Automatically Fixing Dependency Breaking Changes. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE096 (July 2025), 23 pages. https://doi.org/10.1145/3729366

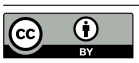
1 Introduction

Modern software development relies heavily on libraries and frameworks, packaging common functionality. Given their nature, these libraries and frameworks are called dependencies. Software in general, and dependencies specifically, evolve regularly. Therefore, there is a need to update these dependencies to ensure that the software remains secure, performant, and compatible with other components. However, updating dependencies is a challenging task, as it can introduce breaking changes that require manual intervention to resolve. This manual intervention can be time-consuming and error-prone, resulting in developers neglecting the update of their dependencies, with research demonstrating this for up to 82% of investigated cases [40]. This neglect could induce security vulnerabilities and compatibility issues, compromising integrity and functionality of software projects, potentially 'poisoning' the software supply chain.

The automation of such dependency updates has become a topic of ongoing research in both academia [12, 30] and industry, with common adoption of industrial tools like Dependabot[1] and Renovate[2]. A common limitation of these tools and this research is their inability to mitigate

[1] https://docs.github.com/en/code-security/dependabot
[2] https://www.mend.io/renovate

Authors' Contact Information: Lukas Fruntke, University College London, Department of Computer Science, London, United Kingdom, lukas.fruntke.23@ucl.ac.uk; Jens Krinke, University College London, Department of Computer Science, London, United Kingdom, krinke@acm.org.



ACM 2994-970X/2025/7-ARTFSE096
https://doi.org/10.1145/3729366

breaking changes in the Application Programming Interface (API) of dependencies, necessitating human intervention for such upgrades. The complexity of those breaking changes, coupled with their pervasiveness throughout the codebase, demands significant effort from developers [40].

The emergence of Large Language Models (LLMs) has opened new avenues for addressing complex software engineering challenges. These models have shown promise in a variety of software engineering tasks, with the application to the problem of the remediation of breaking changes being largely unexplored [42, 52]. This research aims to bridge this gap by leveraging LLMs to automate the process of updating dependencies, with a specific focus on resolving breaking API changes, as a sub-discipline of Automated Program Repair (APR). We present a novel system designed to facilitate the automated upgrade of software codebases in response to breaking dependency updates in Java projects. To evaluate the efficacy of LLMs in this domain, we investigate two distinct techniques: zero-shot prompting and an agentic approach. Our methodology includes establishing a controlled experimental setup to examine these approaches, concentrating on their capacity to repair breaking API updates. This comparison attempts to determine whether the simplicity of zero-shot prompting can match the performance of the more context-rich agentic method, and to analyse the efficiency of both techniques. By utilising the pre-existing knowledge of the LLM in zero-shot prompting, we assess the model's inherent ability without previous examples or instructions. Conversely, the agentic approach allows the LLM to independently refine its solutions based on environmental feedback, potentially enabling more sophisticated repairs. We evaluate repair success based on test suite passage. We also track test suite errors, indicating a repair that compiled, but did not pass the test suite for various reasons, potentially indicating near-misses. We also analyse the impact of factors such as dependency popularity and the complexity of breaking changes on the performance of our system.

Our findings demonstrate the potential of LLMs in addressing breaking API changes. The zero-shot prompting approach repairs up to 19% of the cases towards full test suite success, and additionally another 43% towards compilation success. While the agentic approach exhibits superior repair performance at up to 23% test suite success, it shows a lower compilation success rate of 15.4% in our evaluation. We demonstrate that the performance of both approaches is heavily model-dependent and influenced by the complexity of the breaking changes.

By automating the handling of breaking API changes, our work aims to significantly reduce the manual effort required in dependency updates. With the uptake of the approaches presented in this paper, updates could become more frequent, improving software security in the industry.

This paper contributes to the field of automated program repair in three key ways:

- We present two novel approaches for automated repair of breaking dependency updates in Java projects using LLMs.
- We provide a comprehensive comparison of zero-shot prompting and agentic approaches, offering insights into their relative strengths and limitations.
- We publish an improved version of the BUMP dataset, which we use for evaluation, and all artefacts from our research.

With this research, we contribute to the growing field of LLM-assisted APR, highlighting the feasibility and investigating optimisations of using LLMs to repair breaking changes, advancing automated software maintenance towards fully automated dependency management.

2 Related Work

Since this paper sits at the intersection of research areas including API evolution, dependency management, APR and LLMs, we provide a comprehensive background to contextualise the research.

2.1 Dependencies and Breaking Changes

According to Belguidoum and Dagnat [4] and Jezek et al. [32], vertical compatibility refers to the ability to replace one component with another without breaking existing contracts, encompassing the notion of backward compatibility. Horizontal compatibility includes behavioural constraints and even licence compatibility (for example, the exclusion of copy-left licences), ensuring that a component maintains proper interactions with other parts of the system [32]. Within this paper, we focus on vertical compatibility, as it is the most common form of compatibility in the context of dependency updates. We will refer to changes to dependencies/libraries/packages that violate vertical compatibility as breaking changes in this paper. The issue of breaking changes has been studied extensively within multiple studies, such as the study conducted by Xavier et al. [64], which investigates 317 real-world Java libraries, 9,000 releases, and 260,000 client applications for breaking changes. They show that 14.78% of API changes investigated break compatibility with previous versions and 2.54% of client applications are impacted. Meanwhile, Keshani et al. [38] found in their investigation that 67% of studied Maven packages violated semantic versioning at least once, thereby possibly introducing breaking changes in a release without prior warning. In a different study, Jayasuriya et al. [31] attempted to automatically update library-client pairs and found that 11.58% of their dataset were breaking changes, of which 41.58% were induced during a non-major version upgrade. They also show that transitive dependency changes are a leading cause for breaking changes. Meanwhile, Brito et al. [7] contributed a tool called APIDiff that detects breaking changes in Java APIs.

In their study, Rahkema and Pfahl [54] analysed the lag time (difference between release of a new dependency version and uptake in a client using said dependency) for projects leveraging three distinct package managers and found that on average it varied from 44 to 96 days. Decan et al. [13] show that for npm since 2015, only about 25% of the releases had a lag lower than 52 days.

He et al. [24] conducted a comprehensive study on Dependabot, a popular dependency management bot, using repository mining and a developer survey. The study found that Dependabot helps reduce lag, with projects decreasing their lag from 48.99 to 25.38 days within 90 days of adoption. While over 70% of Dependabot pull requests are merged, many developers prefer to perform manual updates after receiving notifications. In addition of course, Dependabot does not address breaking changes and at most notifies when the test suite breaks.

In the study by Dietrich et al. [14], an analysis was conducted on 109 programs with 212 program dependencies. They found as much as 75% of all version upgrades could become incompatible. In a separate study, Jezek et al. [32] examined 455 pairs of adjacent versions representing 'atomic' upgrades. Their analysis revealed that 375 version pairs exhibited vertical incompatibilities, while only 74 pairs were compatible, representing an incompatibility quota of 80% and a ratio of 5:1 incompatible to compatible upgrades.

APIFix [22] and MELT [55] use examples to learn relationships between adaptations of different library versions and how to generate transformation rules to automatically adapt other clients to a new version of a library.

Two different benchmarks/datasets are available for breaking changes and API compatibility. Durieux et al. [16] present a benchmark of pairs of 395 libraries and their 2874 respective clients, where each pair compiles and passes the respective test suite fully. Reyes et al. [57], however, present a reproducible benchmark of breaking changes mined from the broader Java Ecosystem, containing 571 breaking dependency updates from 153 projects. This benchmark has mostly been curated by mining Dependabot Pull Requests from GitHub. With their dataset, they not only present the causes of breakages and possible originations on the API-level, but they also make docker images accessible that allow the indicated failures to be replicated as checkpoints. A similar dataset

named Compsuite was presented by Xu et al. [66]. It contains 123 Java client-library pairs, where a library upgrade induces a breakage in the client and are surfaced by hand-constructed tests.

2.2 Automated Program Repair (APR)

Nielebock [50] introduced the concept of API-specifc APR, which includes the APIs that software dependencies expose. The paper goes on to show that API usage patterns are needed for APR to succeed. These patterns are learned through the corpora of LLM training data, with varying success, as a study by Zhong and Wang [69] found in a zero shot scenario that up to 62% of the code generated by LLMs was not correctly using APIs.

The type of Automated Program Repair we apply in this work falls under the behavioural repair category, as described by Monperrus [49]. It is particularly applicable to statically typed languages, where the compiler and/or test suite serve as the oracle to verify API compatibility with dependencies. Hejderup and Gousios [25] assess the effectiveness of test suites for identifying breaking changes arising from dependency updates of Java projects, contributing the success criteria of this paper. By examining 521 projects, they evaluate the test suite coverage and effectiveness. They show in their study that the median coverage of dependencies in their dataset by a client's test suite is at 58%. This was achieved, however, by artificially seeding faults, possibly not mirroring real-world conditions. It still raises an adequacy question of test suites as suitable oracles for APR for dependencies.

2.3 Large Language Models for APR

The application of LLMs within the field of APR is a field of current research, with a recent literature review finding 127 publications related to the field, where 37% of the publications focused on Java as a programming language specifically [68], with some notable publications including [15, 34, 36, 44, 48, 63]. In said literature review, 48% of the publications focused on the zero-shot methodology. Especially noteworthy is Joshi et al. [37], where the authors introduce a tool for APR called Ring, which they also prompt with the initial compilation error, inducing a repair.

Allowing the LLM to refine its solutions using the feedback from test suites or compilers, termed 'self-debugging', has been introduced by Chen et al. [11]. Leveraging agentic tools for the same purpose was investigated by Huang et al. [28]

In Xia and Zhang [65], the authors introduce an APR tool that incorporates back-tracking and evaluation feedback from the test suite to improve the quality of the generated patches. Hidvégi et al. [26] introduced CIGAR, a LLM based APR tool, which focuses on minimising token cost while maintaining effectiveness. The authors show in their study that CIGAR outperforms state-of-the-art APR tools by fixing more bugs while reducing token cost by 73%.

Agrawal et al. [2] investigate Language Servers and the Language Server Protocol to guide the decoding of LLMs, discarding infeasible options without compilation. In addition, they open-source their framework for interfacing with Language Servers like Eclipse's JDT for Java.

Chen et al. [11] show that a feedback loop utilising compilation can improve LLM performance in multi-shot code generation scenarios by up to 12%.

In the study by Horváth et al. [27], the authors show how LLM performance on APR tasks is directly dependent on the structure of the input (i.e., structuring code input as command sequences, Abstract Syntax Tree (AST) or text), which was however an optimisation task highly dependent on the LLM and dataset used. This study directly inspired the RQ3 in this paper, which also aims to find optimal representations for the task and dataset at hand.

In recent literature there are ample examples of agentic approaches to APR that are used to improve the performance of LLMs in the field of APR [10, 47, 61]. A survey into agentic approaches

conducted by Jin et al. [35] investigates applicability to Software Engineering problems, including APR. Meanwhile, Bouzenia et al. [6] demonstrate an LLM-based agent that performs APR autonomously, based on a custom JSON patch format, but with more tooling available to the agent than in our study and an iteration budget of 35 retries. Their tool, RepairAgent, outperforms the state of the art on the Defects4Jv2 benchmark.

2.4 Fixing Dependency Breaking Changes

Dann et al. [12] introduced UPCY, a graph-based approach to provide safer dependency update suggestions, minimising incompatibilities when updating libraries. Upon evaluation, the tool could find update paths with 70.1% of these updates having zero incompatibilities. Reyes et al. [56] introduce Breaking-Good, a tool that generates explanations for breaking dependency updates by analysing build logs and dependency trees. Similar to this paper, it relies on the BUMP dataset and was evaluated on 243 breaking updates, whereby it accurately explains and categories 70% of cases.

Jhamat et al. [33] introduce a tool called DepRefactor, which they claim can automatically refactor codebases to adapt to breaking changes in dependencies in C# projects. They do not conduct an evaluation of the tool's performance and do not substantiate their claim that the tool contains 'AI'.

The master's thesis by Bono [5] explored breaking change repairs via LLMs using the BUMP dataset. Their work concentrated on build failures, examining a filtered subset of 35 entries with four LLMs (Gemini, GPT-4, Llama, Mixtral). While both studies investigate few-shot prompting, Bono's approach targets specific failure types with tailored prompts, achieving repairs in 10 out of 35 projects. In contrast, our work examines a larger dataset slice, a broader spectrum of LLMs including several Claude variants, and introduces an agentic approach. We additionally analyse dependency popularity and function calling proficiency as repair performance factors, providing comprehensive insights into model capabilities and their contributing elements.

3 Approach

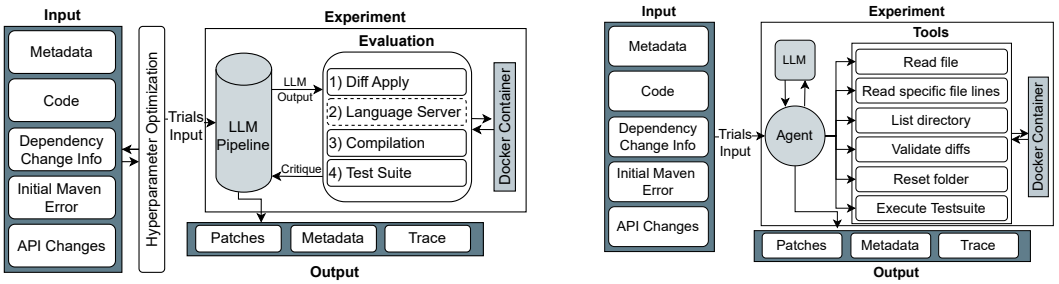
Within this section, we propose two approaches for automatically fixing breaking changes in dependency updates by leveraging LLMs. This section discusses the key components of our approach, focusing on the challenges of LLM interaction, the rationale behind our chosen methods, and the specific implementations for both zero-shot prompting and the agentic approach.

3.1 Architecture Overview

For later evaluation, we implement two different approaches to LLM interaction: zero-shot prompting and an agentic approach. These will be further discussed in subsection 3.2 and subsection 3.3. Within this section, we aim to outline the general architecture of both approaches, which is subtly varied in both implementations.

As we primarily designed the approaches to address breaking changes in Maven Java projects, we utilise diffs as an intermediary language for code modifications. The core workflow involves running these diffs against a containerised environment with an associated repository at a specific commit in it, against which we execute `mvn clean test`, triggering both compilation and test suite execution. The commit our approach is running against, and associated meta- and input data, can either be provided ad-hoc or in a batch fashion, for example in the form of a pre-compiled dataset. Both approaches share common phases: Input Processing and LLM Interaction, Validation & Feedback, and Output Generation.

Input Processing: The input for the LLM, which had to be pre-processed, is passed to the LLM in a structured format. This contains suspicious (suspiciousness here being included in Maven Errors and/or API Change metadata) code files, dependency change information for the commit



(a) Zero-Shot Approach (b) Agentic Approach

Fig. 1. Architectural comparison of the Zero-Shot and the Agent-based approach.

(i.e., which dependency was upgraded), the initial Maven errors (test suite error and/or compilation error), and structured API changes of the dependency that was updated.

LLM Interaction: Finding an appropriate editing format for LLMs to communicate code changes is challenging due to their tendency to produce partial or altered code fragments [62]. To mitigate risks of hallucinations [1, 29] or irrelevant edits, we use unified diffs as an intermediate language, following observations by Gauthier [23]. This approach is supported by ongoing research on using diffs for code review tasks [18]. Based on work by Gauthier [23], we developed an improved diff parser with optimistic error correction.

Validation & Feedback: A multi-stage process that includes diff parsing, compilation, and test suite execution. This can be either in the form of a static pipeline or as a subagent, depending on the approach. Feedback is given to the LLM based on the validation results for refinement.

Output Generation: Produces the final patches and metadata.

3.2 Zero-Shot Prompting

Following the definition of Brown et al. [8], we define zero-shot prompting as a natural language description of a task specifically without any examples, which is the key-differentiator to few-shot prompting. In our definition, while back-tracking with a subsequent refinement of the prompt induces a sense of conversationalness, we still count it as zero-shot prompting, as only refining guidance is given, not examples.

Our experimental architecture for the zero-shot approach relies heavily on backtracking, implemented using DSPy [39] for inference. We enhanced this with the Assertions functionality for backtracking in DSPy [60], allowing us to build a flexible inference pipeline. In this context, backtracking refers to the process of prompting the LLM with its last solution and the reason it failed, eliciting a refinement, as the pipeline execution is backtracked. Our implementation generates an initial solution attempt using the LLM, which is then evaluated against a set of assertions – predefined conditions that must be met for the solution to be considered valid. Upon assertion failure, the system backtracks by regenerating a new solution, taking into account the failed assertion. To avoid infinite back-tracking there is a maximum limit to attempts.

The baseline configuration for the zero-shot prompting approach, as shown in Figure 1a, includes a disabled Language Server integration, 30 backtracking iterations, the full code file, the custom dependency update representation, the minified error and the API Changes as reported by Japicmp and RevAPI. Both tools are widely used open-source tools for Java API compatibility analysis [51, 56, 57] – detecting breaking changes across library versions by analysing bytecode modifications. These include critical API alterations like method/field additions/removals, signature updates, and

constructor changes. In the dataset, Reyes et al. [57] leverage both tools to systematically map API changes to downstream compilation errors, isolating the root causes (suspicious files) that trigger failures.

3.3 Agentic Approach

An agent, as defined by Franklin and Graesser [19], differs from a typical program in its ability to interact with its environment and other agents over time, maintaining a sense of statefulness. It autonomously selects and executes actions, responding to subsequent changes. Applying this concept to LLMs requires the ability to interact with the environment. As shown by Schick et al. [58], this capability is learnable and was integrated in many LLMs. We implement a tool-using agent via LangGraph [41]. Our agent integrates a Large Language Model (LLM) as its core decision-making component. The agent operates through LangGraph's event loop as shown in Figure 2. From the initial start state, the LLM is invoked with an initial prompt instructing it to create a patch. It then can choose to invoke maven as our test oracle to validate the solution (via the *test* node), or it can invoke a specific tool (via the *tools* node). The feedback for the (failed) test or the result from the invoked tool is then used by the LLM to decide on the next step. If the validation step is successful (the test is passed), the agent completes with success and the agent is ended. If the agent cannot find a successful solution within 30 executions of a node, the agent is aborted (by LangGraph).

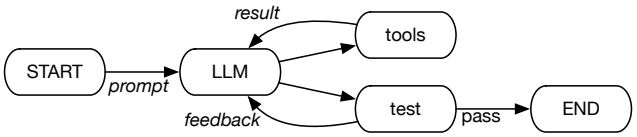


Fig. 2. LangGraph Execution Graph of the Agent System

The agent has access to several tools, as shown in Figure 1b. These include *read_file* (reads a file from a given path), *read_file_lines* (reads specific lines from a file), *get_directory_tree_for_path* (returns the directory structure), *validate_diffs* (checks if a diff can be applied), *reset_repo* (resets the project repository), and *compile_maven_stateful* (compiles the project with applied diffs, persisting changes unless reset, and executes the test suite). The *compile_maven_stateful* tool is separated in its own *test* node as it decides on ending the iteration (on pass) or continuing (on fail). The tools return the results or descriptive error messages to aid the LLM in its decision-making process.

To explain how the agent system works, we provide an example run for a successful repair of a failed dependency update for `net.sf.jasperreports:jasperreports` from version 6.18.1 to 6.19.1 involved 6 iterations: The LLM is initially given a prompt consisting of the fixed system prompt in Figure 3 and the additional information about the dependency breakage shown in Figure 4, consisting of the maven error messages and the Japicmp/RevAPI results. The LLM first invokes the *read_file* tool to retrieve the content of `ReportBuilder.java`, the file causing a compilation error. It then generates a patch and invokes the *validate_diffs* tool to check the generated patch, which is successful. The LLM then invokes the *compile_maven_stateful* tool to compile the project with the applied patch and execute the test suite. However, the compilation still fails. With the feedback from the failed build, the LLM creates a new patch and tests it again. This time the test fails because the patch cannot be applied. The LLM then invokes *read_file* again and generates a new patch. The next invocation of *compile_maven_stateful* is successful (all tests are passed), and the agent ends successfully.

```
Act as an expert Java software developer.
The program has issues after a version upgrade of a dependency.
Try using minimal changes to the code to fix the issues. Do not explain your actions or ask
questions, just provide diffs that always adhere to the rules. When you think you are done, reply
with the diff that fixes the issues, after that a final verification step will happen and the
conversation will be ended if it was successful. If not you get the error back.

# File editing rules:
Return edits similar to unified diffs that `diff -U0` would produce.
The diff has to be in a markdown code block, like this: ```diff ```.

Make sure you include the first 2 lines with the file paths.
Don't include timestamps with the file paths.

Start each hunk of changes with a `@@ ... @@` line. Don't include line numbers like `diff -U0` does.
The user's patch tool doesn't need them.

The user's patch tool needs CORRECT patches that apply cleanly against the current contents of the
file! Think carefully and make sure you include and mark all lines that need to be removed or
changed as `-` lines. Make sure you mark all new or modified lines with `+`. Don't leave out any
lines or the diff patch won't apply correctly. Don't add in new comments or change existing
comments. Make sure the diff is minimal and only includes the changes needed to fix the issue plus
at least one context line so the tool can apply the diff correctly. Indentation matters in the
diffs!

Start a new hunk for each section of the file that needs changes. Don't include unnecessary context,
but include at least one line of it. If no context is included, the tool will try to apply the
changes at the end of the line.

Only output hunks that specify changes with `+` or `-` lines. Skip any hunks that are entirely
unchanging ` ` lines. Output hunks in whatever order makes the most sense. Hunks don't need to be
in any particular order.

When editing a function, method, loop, etc use a hunk to replace the *entire* code block. Delete
the entire existing version with `-` lines and then add a new, updated version with `+` lines.
This will help you generate correct code and correct diffs.

To make a new file, show a diff from `--- /dev/null` to `+++ path/to/new/file.ext`.
```

Fig. 3. The prompt template for the agent system.

4 Evaluation

We designed a controlled experimental setup that allows us to evaluate the effectiveness of both zero-shot prompting and the agentic approach. We aim to address the following research questions, comparing both approaches and LLMs across multiple dimensions:

RQ1: Is an agentic approach more effective than zero-shot prompting? This question investigates and compares the efficacy of the two approaches, zero-shot and agentic, in addressing breaking API changes, discovering insights into their relative strengths and limitations.

We expect the agentic approach to outperform the zero-shot approach, as it can refine its solutions based on environmental feedback. Therefore we aim to analyse factors that may influence the performance of the agentic system in RQ2. Given the nature of the zero-shot approach, we suspect a sensitivity to the input structure and format, which we aim to explore in RQ3.

```
Updated Dependency Details: net.sf.jasperreports:jasperreports 6.18.1 -> 6.19.1

Initial Error: [ERROR] Failed to execute goal
↪  org.apache.maven.plugins:maven-compiler-plugin:3.10.0:compile (default-compile) on project
↪  biapi: Compilation failure
[ERROR] src/main/java/xdev/tableexport/export/ReportBuilder.java:[369,81] incompatible types: int
↪  cannot be converted to java.lang.Float
[ERROR] -> [Help 1]

Japicmp/RevAPI API Changes, which describe changes in the APIs used by this project:
{"japicmpResult":
  {"getLineBox": ["METHOD_REMOVED"], "getPen": ["METHOD_REMOVED"],
   "getLineWidth": ["METHOD_REMOVED"], "setLineWidth": ["METHOD_REMOVED"]},
 "elementLines":
  {"getLineBox": "[ERROR] /biapi/src/main/java/xdev/tableexport/export/ReportBuilder.java:[369,81]
  ↪  incompatible types: int cannot be converted to java.lang.Float",
   "getPen": "[ERROR] /biapi/src/main/java/xdev/tableexport/export/ReportBuilder.java:[369,81]
   ↪  incompatible types: int cannot be converted to java.lang.Float",
   "getLineWidth": "[ERROR]
   ↪  /biapi/src/main/java/xdev/tableexport/export/ReportBuilder.java:[369,81] incompatible types:
   ↪  int cannot be converted to java.lang.Float",
   "setLineWidth": "[ERROR]
   ↪  /biapi/src/main/java/xdev/tableexport/export/ReportBuilder.java:[369,81] incompatible types:
   ↪  int cannot be converted to java.lang.Float"},
 "revapiResult":
  {"getLineBox": ["java.method.removed"], "getPen": ["java.method.removed"],
   "getLineWidth": ["java.method.removed"], "setLineWidth": ["java.method.removed"]},
   "allPotentialBreakingElements":
    ["getPen()", "setLineWidth()", "getLineBox()", "getLineWidth()"],
   "elementPatterns":
    {"getLineBox": "incompatible types: cannot be converted to",
     "getPen": "incompatible types: cannot be converted to",
     "getLineWidth": "incompatible types: cannot be converted to",
     "setLineWidth": "incompatible types: cannot be converted to"}, "project": "biapi"}
```

Fig. 4. Information attached to the prompt template.

RQ2: How does the performance of the agentic system vary across different language models, and which external factors influence it? This question examines the performance variations among different language models within the agentic system and explores external factors, such as dependency popularity, that may impact the system's effectiveness.
RQ3: Which input variation yields the best results for the zero-shot system? This question aims to identify optimal input structures that maximise the zero-shot approach's performance in resolving breaking API changes.

In the following subsections we discuss the design and setup of the evaluation framework based on a structured dataset.

4.1 Dataset

This paper mainly explores Java and necessitates a benchmark specifically designed for Java. We use the BUMP benchmark specified by Reyes et al. [57], which offers reproducible Maven failures resulting from dependency updates within Dependabot pull requests. Each entry in the dataset represents a single commit per repository.

However, the BUMP dataset had limitations for our research. It lacked a proper categorisation of the entries per failure type (compilation, JDK issues, maven plugin failures, ...). Additionally, the absence of license information for mined repositories raised ethical concerns, and some entries contained flaky test suites, potentially leading to false negatives. Flaky tests can yield inconsistent results for the same code under identical conditions, complicating fix evaluations.

To address these issues, we created a forked version of the dataset with added categorisation and license information, providing a more robust foundation for our research.

We began with 571 successfully replicated entries from the original BUMP dataset. Of these, 243 entries were compilation failures and 328 were other error types (e.g., Maven Enforcer issues requiring `pom.xml` modifications rather than code repairs). To refine our dataset, we filtered the 243 compilation failures by (i) removing 78 entries caused by JDK version mismatches (specifically, `class file has wrong version` errors) and (ii) discarding 25 entries whose associated GitHub commits were no longer accessible. This process yielded a *full slice* of 140 entries.

From this full slice, we filtered for projects that had an associated Japicmp/RevAPI log detailing bytecode-level breaking changes—information essential for our zero-shot approach (yielding 85 results) and discard 20 entries which have multiple suspicious files. This yields a *light slice* of 65 entries. We focus in the light slice on files with single suspicious files to avoid the risk of overrunning the context window limitations of the LLM used in the zero-shot approach. Notably, in the full slice, 68% of compilation errors affected a single file (with 32% spanning multiple files), whereas in the light slice, 83% were single-file errors (only 17% involved multiple files). This creates an apparent discrepancy: The 17% involving multiple files in the naïve maven compile error statistics trace back to a single suspicious file containing the root API consumer. Compiler errors elsewhere occur because of inheritance – a nuance that is captured by using the Japicmp/RevAPI-based approach from Reyes et al. [57]. On average, entries in the full slice had 2.56 Maven errors (SD = 5.54), compared to 1.22 errors (SD = 0.52) in the light slice.

Given that the majority of errors in both sets were in a single file, we believe the limitation of the zero-shot approach to be justified. To compare the performance of the approaches, both the zero-shot and agentic approaches are evaluated on the light slice, with the agentic method additionally assessed on the full slice.

To support our experiments, we generated multiple input variations for both slices. Code was provided either in original form or processed to retain only the relevant AST nodes surrounding the lines with Maven errors. This processing, performed using Spoon [53], trims the code to the method where the initial compilation error occurred (preserving necessary overloads and imports only as needed). We also created versions with and without comments to assess their impact on the LLM's repair performance.

Since every BUMP commit involves a version change, we produced two representations of the version change: a diff of the `pom.xml` file triggering the change, and a custom-parsed syntax (e.g., `org.yaml:snakeyaml 2.0 -> 2.1`). Additionally, the Maven error logs were processed in various ways (with and without line/column numbers) to preserve context.

4.2 Experimental Setup

To judge the effectiveness of any such approach with a Language Model, we selected **test suite success** (i.e., running *mvn clean test*) as success metric. As shown by Hejderup and Gousios [25], test suite success can be a somewhat reliable indicator of the success of fixing the issue. It is not a perfect indicator given the fact that some elements in the dataset lack tests altogether. As flaky and missing test suites have been observed in parts of the benchmark, the test error metric has been introduced as well, measuring compilation success of entries failing the test suite after.

Table 1. Manual Evaluation Results of Test Suite-Passing Patches

	Zero-Shot ($n = 10$, $N = 48$)		Agentic ($n = 16$, $N = 78$)	
Dimensions	R #1	R #2	R #1	R #2
Correct	9/10	8/10	13/16	11/16
Minimal	8/10	4/10	10/16	9/16
Maintainable	8/10	8/10	10/16	14/16
Agreement	$\kappa_{\text{correct}} = 0.435$ (81%) – Moderate $\kappa_{\text{minimal}} = 0.308$ (65%) – Fair $\kappa_{\text{maintainable}} = 0.161$ (69%) – Slight			

To achieve replicability, the experimental system is designed to log all inputs and outputs of the LLM and the experiments. The system is also designed to log the full OpenInference traces, as well as a custom output format showing all inputs and outputs per experimental trial. In addition, the solutions for both approaches that passed either compilation or the test suite are published in a Docker image, which is based on the original dataset, but adds the modified files in the container, aiding verification of the reported results. All quantitative analysis is performed on git diffs of the workspace after application of the patches, excluding any whitespace changes introduced by the more lenient diff parser.

We experimented with a mix of open- and proprietary models. We mostly employed recent releases of proprietary models of varying sizes, as well as Llama 3.1 70B and Mistral NeMo 12B as open models. To aid comparability, we employed a temperature setting of 0 for all providers. While setting temperature to zero minimises output variability, inherent model randomness remains. More extensive randomness analysis would require substantial computational resources beyond our current scope, but our setup provides a reliable and reproducible baseline. We exclusively leverage hosted model versions at different providers. The OpenAI family models were used via Azure and the OpenAI API, the Anthropic and Google Models were used via Google Cloud, Mistral NeMo was used via the Mistral API and Llama was used via the together.ai API.

4.3 Evaluation of Patches

To validate the effectiveness of both approaches beyond test suite metrics, we conducted a manual evaluation of generated patches. A stratified random sample of 20% from the test suite-passing patches was selected, with 10 patches from the zero-shot approach and 16 from the agentic approach. Two experienced software engineers independently and blindly (no association to model name or approach type was possible) evaluated each patch across three key dimensions: correctness (proper resolution of breaking changes without introducing new issues), minimality (containing only necessary modifications), and maintainability (adherence to coding practices and review acceptability). The evaluation results, presented in Table 1, show different agreement between reviewers (Cohen's $\kappa = 0.435$ for correctness, $\kappa = 0.308$ for minimality, and $\kappa = 0.161$ for maintainability). For the zero-shot approach, reviewers found 85–90% of patches to be correct, 40-80% to be minimal, and 80% to be maintainable. The agentic approach achieved 69–81% correctness, 56–63% minimality, and 63–88% maintainability ratings. While both approaches produced largely correct patches, they differed notably in minimality and maintainability assessments.

Table 2. Comparison of Language Models using the Zero-Shot and Agentic Approach

	Zero Shot (`light`, $n = 65$)						Agent (`light`, $n = 65$)						Agent (`full`, $n = 140$)					
LM	**Fixed**	Test Errors	Other Errors	Time (h)	Token Sum	Avg. Cost/Try	**Fixed**	Test Errors	Other Errors	Time (h)	Token Sum	Avg. Cost/Try	**Fixed**	Test Errors	Other Errors	Time (h)	Token Sum	Avg. Cost/Try
Sonnet 3.5[a]	12	28	25	12.6	39M	$1.98	12	10	43	4.2	17M	$0.85	32	12	96	22.5	42M	$0.39
GPT4o[a]	8	26	30	17.4	32M	$2.55	9	9	47	4.8	13M	$1.00	17	11	112	23.8	29M	$0.47
Gemini Pro[a]	10	24	31	16.8	121M	$9.45	4	0	61	3.7	5M	$0.30	7	2	131	15.2	14M	$0.14
GPT4o mini	4	11	50	12.7	22M	$0.06	6	7	52	3.6	11M	$0.03	9	8	123	21.1	31M	$0.01
Haiku 3[a]	7	22	36	13.1	33M	$0.14	2	0	63	2.1	12M	$0.06	6	1	133	7.8	26M	$0.03
Llama 3.1[a]	4	7	54	2.1	10M	$0.14	2	1	62	2.6	11M	$0.14	6	2	132	7.9	22M	$0.07
NeMo[a]	3	14	48	5.0	27M	$0.13	0[b]	1	63	1.6	7M	$0.03	1[b]	1	130	9.1	14M	$0.02

[a]The models used are claude-3.5-sonnet@20240620, gpt-4o-2024-05-13, gemini-1.5-pro-001, gpt-4o-mini-2024-07-18, claude-3-haiku@20240307, llama-3.1-70B and mistral-nemo respectively.

[b]NeMo reproducibly failed to execute on 1 out of the 65, 8 out of the 140 projects with an http timeout error.

5 Results and Discussion

To answer the three Research Questions, we analyse the evaluation data from the zero-shot and agentic approaches.

5.1 Is an Agentic Approach More Effective than Zero-Shot Prompting? (RQ1)

Our evaluation of zero-shot prompting and agentic approaches for automated dependency updates in Java projects revealed significant differences in their effectiveness. We compare the performance of seven language models in Table 2 across different dataset slices. We show how many repair attempts fixed/repaired the breaking change, how many attempts failed either with a test suite error (a failing test suite, but a successful compilation) or any error, we show the total runtime, the token sum and the cost per repair attempt in US Dollars. We first contrast the performance of the zero shot approach and the agentic approach within the `light` slice, extending the analysis to the `full` dataset slice for the agentic approach.

5.1.1 Analysis of the Light Slice. For the `light` slice, we can directly compare the zero-shot and agentic approaches:

Success Rates: Sonnet 3.5 achieved 12 successful repairs (18.5%) with 28 test errors in zero-shot versus 12 fixes with 10 test errors in agentic approach (Table 2). GPT-4o showed modest improvement (+1 fix) while GPT-4o-mini improved from 4 to 6 fixes in agentic mode. Gemini Pro declined sharply from 10 to 4 fixes, and NeMo failed completely in agentic mode.

Efficiency: Agentic Sonnet 3.5 reduced average cost from $1.98 to $0.85 (-57%) and time from 12.6h to 4.2h (-67%), though Llama 3.1-70B required 1M more tokens (10M→11M) and +0.5h (Table 2). Positive correlations for both approaches were found between success rates, cost, and time in both approaches. While the zero-shot approach exhibited a moderate positive correlation between fixed cases and average cost per attempt ($r = 0.602$, $p = 0.1529$), this was not statistically significant. Conversely, the agentic approach demonstrated strong and statistically significant correlations: fixed cases correlated positively with both average cost per attempt ($r = 0.845$, $p = 0.01666$) and time ($r = 0.885$, $p = 0.008103$). The findings suggest a strong correlation between resource investment and improved performance in the agentic approach.

Error Distribution: Zero-shot approaches exhibited higher test errors where compilation succeeded but tests failed (Sonnet: 28 vs 10 test errors in agentic), despite matching success counts

Table 3. Diff Metrics across all slices and approaches

LM	Zero Shot (`light`, $n = 65$)			Agent (`light`, $n = 65$)			Agent (`full`, $n = 140$)		
	Format: Mean ± Standard Deviation								
	Files Mod.	Operation Count	Hunk Count	Files Modified	Operation Count	Hunk Count	Files Modified	Operation Count	Hunk Count
Sonnet 3.5[a]	1	3.83 ± 1.59	1.75 ± 0.62	**1.27 ± 0.63**	**10.73 ± 7.88**	**1.82 ± 1.53**	**1.75 ± 2.14**	13.30 ± 17.19	**2.55 ± 2.35**
GPT4o[a]	1	**4.12 ± 2.70**	1.62 ± 0.52	1.06 ± 0.24	9.28 ± 7.31	1.17 ± 0.51	1.04 ± 0.19	7.43 ± 6.67	1.32 ± 0.72
Gemini Pro[a]	1	2.80 ± 1.48	1.10 ± 0.32	1	2	1	1	2.78 ± 1.30	1
GPT4o mini[a]	1	3.75 ± 1.26	**2 ± 0.82**	1.10 ± 0.32	9.30 ± 5.01	1.10 ± 0.32	1.07 ± 0.27	7.79 ± 5.22	1.21 ± 0.58
Haiku 3[a]	1	3.14 ± 0.90	1.86 ± 0.69	1	2	1	1	**13 ± 29.10**	1
Llama 3.1[a]	1	2.25 ± 0.5	1	1	4.33 ± 2.52	1.33 ± 0.58	1	2.88 ± 1.81	1.12 ± 0.35
NeMo[a]	1	3.33 ± 0.58	1.33 ± 0.58	N/A[b]	N/A[b]	N/A[b]	1	2	1

[a]The models used are claude-3.5-sonnet@20240620, gpt-4o-2024-05-13, gemini-1.5-pro-001, gpt-4o-mini-2024-07-18, claude-3-haiku@20240307, llama-3.1-70B and mistral-nemo respectively.

[a]NeMo did not produce any successful patches to analyse.

(12 repairs) (Table 2). Therefore, the zero-shot approach managed to clear the compilation hurdle more often, while still failing the test suite.

Patch Complexity: The agentic approach produced patches with higher operation counts (where each operation represents an individual line addition, deletion, or modification in the diff) compared to zero-shot, as evidenced by Sonnet 3.5's average of 10.73 ± 7.88 operations versus 3.83 ± 1.59 in zero-shot (Table 3). While technically capable of multi-file modifications, 75% of agentic repairs in the `light` slice modified ≤ 2 files (average 1.27 ± 0.63). Manual evaluation (Table 1) reveals a complexity-minimality tradeoff: agentic patches were rated minimal in 56–63% of cases versus zero-shot's 40–80%, though maintainability remained comparable (agentic: 63–88%, zero-shot: 80%). The data suggests agentic approaches may handle complex edits better when required, but this doesn't universally correlate with success rates – Sonnet's higher complexity accompanied improved results, while Gemini Pro's increased operations coincided with failure rate increases.

Model-Specific Performance Shifts: Some models showed pronounced differences in performance between approaches. As evident from Table 2, GPT4o mini improved from 4 to 6 successful repairs in the agentic approach. Conversely, Gemini Pro's performance declined sharply (10 to 4 fixes). For the agentic approach, GPT4o mini performed better than Gemini, with Gemini performing better in the Zero-Shot approach.

5.1.2 Analysis of the `Full` *Slice.* Expanding the evaluation of the agentic approach to the `full` slice yielded additional insights.

Scalability and Efficiency: As shown in Table 2, Sonnet 3.5's performance improved significantly, repairing 32 projects (22.9%) compared to 12 (18.5%) in the `light` slice, while further reducing the average cost per attempt to $0.39. This suggests potential for scaling, though model variance remains.

Scaling Patterns: While some models, such as Sonnet 3.5 (+4.4%), Haiku (+1.2%), Llama 3.1 (+1.2%), NeMo (+0.7%), showed improved performance on the `full` slice, compared to the `light` slice, the success rate of the GPT-4o mini (-2.8%), GPT-4o (-1.7%) and Gemini Pro (-1.2%) declined relative to the dataset. This suggests that the benefits of the agentic approach may be model-dependent.

Complexity-Performance Trade-off: The increase in patch complexity from the `light` to `full` slice (Table 3) occurred with improved success rates for Sonnet 3.5. However, other models like GPT-4o showed reduced complexity without success rate improvements. For instance, Sonnet 3.5 shows an increase in average files modified (1.75 ± 2.14 vs 1.27 ± 0.63) and operation count (13.30 ± 17.19 vs 10.73 ± 7.88), corresponding to its improved success rate.

Table 4. Frequency Distribution of Intersections between Language Models for Test Success and Test Error

LLM-Intersections	0	1	2	3	4
Test Error (%)	8.3	16.7	**41.7**	25.0	8.3
Test Success (%)	**52.5**	17.5	17.5	7.5	5.0

Token Efficiency at Scale: Despite the increased complexity of repairs, some models showed improved token efficiency compared to the `light` slice. This is especially apparent for GPT-4o-mini (-23.6% tokens), Gemini Pro (-23.1%) and Sonnet 3.5 (-12.8%). Meanwhile, the trend was reversed with Llama 3.1 (+7.7%) and NeMo (+7.7%). This might indicate the agentic approach becoming more efficient in its use of context and generation within the `full` slice. There is, however, no correlation to the success rate.

Real-World Applicability: Agentic repairs modified an average of 1.75 ± 2.14 files in the `full` slice (Table 3), though 68% still involved ≤ 2 files. When tests passed, both approaches showed acceptable correctness (zero-shot: 85–90%, agentic: 69–81% in Table 1), suggesting reliable functionality for successful patches. Notably, 43.1% of zero-shot attempts passed compilation but failed tests (vs 15.4% for agentic), indicating agentic methods more effectively align fixes with functional correctness. Observed efficiency gains (agentic cost reduction: 57–80% across models) combined with maintainability ratings (63–88%) suggest practical utility for contained dependency updates, though the approach's 1.75 average file modifications and model-dependent performance (Sonnet: 32 fixes vs NeMo: 1) indicate current limitations in handling complex scenarios.

Answer to RQ1: Agentic approaches show variable efficiency improvements, with Sonnet 3.5 reducing costs by 57% ($1.98→$0.85) and time by 67% (12.6h→4.2h) in the `light` slice (Table 2). Model dependence remains significant – GPT-4o-mini improved fixes (+50%) while Gemini Pro declined (−60%). The manual evaluation (Table 1) suggests comparable patch quality when tests pass (69–90% correctness), though minimality varies. The agentic approach becomes more cost-efficient when scaled to larger datasets (as shown by Sonnet 3.5's reduced cost of $0.39 per repair attempt in the full dataset), but effectiveness varies significantly between language models, requiring careful model selection for optimal results.

Our analysis suggests agentic approaches may offer efficiency advantages over zero-shot methods (57% cost reduction, 67% faster processing for Sonnet 3.5 in `light` slice), though performance varies substantially across models (+50% fixes for GPT-4o-mini vs -60% for Gemini Pro). Both represent early examples of LLM-based dependency repair, with agentic methods showing particular promise in balancing compilation success with functional correctness (28⟶10 test errors for Sonnet). While demonstrating capability for multi-file edits (1.27–1.75 files on average), the approaches currently show more consistent effectiveness for localized repairs. The 22.9% success rate on the `full` dataset (Sonnet 3.5) suggests potential for practical application, though model selection and error handling require careful consideration given the 43–131 other errors observed in failed attempts.

5.2 How Does the Performance of the Agentic System Vary Across Different Language Models, and Which External Factors Influence It? (RQ2)

To address this question, we conducted a comprehensive analysis of influence factors on the performance of the agentic approach on the `full` dataset slice.

Specifically, 52.5% of test successes and 8.3% of test errors were unique to a single model. This finding suggests that different language models exhibit distinct strengths in addressing various

types of breaking changes. Comparing these findings to Table 2, the difference of Sonnet 3.5 (32 repairs) and GPT-4o (17 repairs) is represented in the intersection data.

Intersection Analysis of Successful Repairs. We first examined the degree of overlap in successful repairs across different language models. As shown in Table 4, the majority of successful repairs were unique to individual models or shared between three models. Specifically, 52.5% of test successes and 8.3% of test errors were unique to a single model. This finding suggests that different language models exhibit distinct strengths in addressing various types of breaking changes. Comparing these findings to Table 2, the stark difference of Sonnet 3.5 (32 repairs) and GPT-4o (17 repairs) is represented in the intersection data.

Correlation with Berkeley Function Calling Benchmark. As our agentic approach uses the capability of tool calling for all interactions with the environment, its performance may be influenced by the proficiency of the language model in this area. Function calling by an LLM is modelled via the output language, thereby being susceptible to hallucinations similar to other tasks. Therefore the proficiency in function calling could be a factor in the success of the agentic approach. We investigated potential correlations between our success metrics and the Berkeley Function Calling Leaderboard v1 [67] in its version from 2024-08-11. The Function Calling Leaderboard measures the proficiency of language models in correctly formulating and executing function calls. The Java Simple Function AST task is a component of the Berkeley Function-Calling Leaderboard that evaluates models' ability to generate syntactically correct Java function calls when presented with function signatures. The task uses Abstract Syntax Tree parsing to verify correctness of 100 Java function call examples, with emphasis on Java-specific type handling (e.g., HashMap, primitives with type suffixes like long with "L").

Our analysis shows this Java-specific function calling capability strongly correlates with repair success (r = 0.702), exhibits negative correlation with repair errors (r = -0.653), and positively correlates with test errors (r = 0.677). In contrast, the Overall Accumulated metric from the leaderboard shows weaker correlations with repair performance (r = -0.423 for errors, r = 0.577 for test errors, r = 0.298 for success), suggesting that Java-specific function calling proficiency is more predictive of repair capabilities than general function calling ability across languages.

These findings highlight that function calling benchmarks serve as a useful predictor of a model's repair capabilities in our context. As new models are released, improvements in Java AST-related tasks on the Berkeley Function Calling Leaderboard may indicate better overall repair performance. This would allow industry practitioners to potentially predict the performance of new models on the task of automated dependency repair.

Correlation Analysis between GitHub Popularity and Repair Success. To examine whether dependency popularity influences repair success, we analysed GitHub stargazers as a proxy metric. Pearson correlation analysis showed no correlation (r = 0.0186, p = 0.5632) between stargazers and detailed success levels, and not even a correlation (r = 0.0728, p = 0.0232) when simplifying success into binary. These results indicate that dependency popularity does not predict repair success.

Answer to RQ2: The performance of the agentic system varies significantly across language models, with 52.5% of test successes and 8.3% of test errors being unique to one language model. Language-specific capabilities, particularly Java AST manipulation skills, show strong correlations with repair success (r = 0.702 for test success), while GitHub popularity of dependencies exhibits no correlation with overall repair success.

The performance of language models in the agentic approach is highly variable, albeit influenced to varying degrees by external factors like the LLM's proficiency in function calling. We establish that the popularity of a dependency does not influence the repair outcome.

Table 5. Parameter importances and Trial values

Parameter	Importance	Baseline	T 1	T 2	T 3	T 4	T 5	T 6	T 7	T 8
API Change	0.542380	REVAPI					**OMIT**			
Mvn Error	0.216602	MINIFIED		**SMIN**						**OMIT**
Code	0.082599	ALL			**MIN**				**MIN**	
Dep. Change	0.070491	MINIFIED_PARSED						**DIFF**	**OMIT**	**DIFF**
Max. Hops	0.059507	30				**40**				**10**
LSP Check	0.028420	False	**True**						**True**	
Test Success (n=65)		7	7	4	5	7	2	6	5	5
Test Error (n=65)		22	22	23	21	22	21	22	15	19
Duration (hours)		13.1	32.2	8.6	6.8	13.0	8.1	10.1	33.0	5.3
Input Tokens		32.4M	30.4M	32.2M	25.8M	43.1M	27.9M	41.0M	23.3M	10.2M
Output Tokens		977.9K	953.4K	1.2M	1.2M	1.3M	1.5M	1.4M	10.2M	434.1K

SMIN = Super Minified, **MIN = MINIFIED**

Analysing this data, it becomes clear that the variability from external factors such as the BFCL leaderboard and the patch metrics is not high enough to explain the exceptionally high performance of Sonnet 3.5. This suggests that the model selection is the most important factor in the performance of the agentic approach.

5.3 Which Input Variation Yields the Best Results for the Zero-Shot System? (RQ3)

We aim to explore whether we can systematically improve the performance of the zero-shot system, to ideally be competitive with the agentic approach. To address this question, we conducted a comprehensive hyperparameter analysis of the zero-shot system using different prompt variations. We employed claude-3-haiku for this analysis due to its optimal balance between success metrics and cost-effectiveness. While we acknowledge that our findings in RQ2 identified model selection as a critical success factor, we chose to conduct hyperparameter optimisation on a single model to manage computational costs and establish a baseline for optimisation strategies. We note that the hyperparameter importance findings may be model-specific, and the relative importance of different input representations could vary across models with different capabilities and architectures. The results of this analysis and the variations employed are presented in Table 5 the table shows both the parameter variations employed per trial, as well as the Baseline. For all trials it shows the success metrics for comparison purposes, as well as the duration and the input and output tokens. We evaluate the trials for repair effectiveness as well as efficiency in resource usage, across both duration and token usage. For the optimisation process, we leverage Optuna, a popular hyperparameter optimisation framework, which allows to efficiently explore the search space of the hyperparameter combinations [3]. While trials one to six in the Table were designed to test specific parameters, trials seven and eight leveraged Optuna's integrated Tree-Structured Parzen Estimator to explore promising parameter combinations.

Our investigation revealed variations in performance across different input configurations. Table 5 summarises the importance scores of various factors influencing the system's performance.

API Change Representation emerged as the most influential factor (importance: 0.54). The baseline configuration used the REVAPI representation, while Trial 5 omitted this information. This omission led to a notable drop in test success (2 vs. 7) and a slight reduction in test errors (21 vs. 22), indicating its key role. **Maven Error Representation** was the next most important (importance: 0.22). The baseline employed a MINIFIED error representation, and Trial 2 used a SUPER_MINIFIED version. This change resulted in lower test success (4 vs. 7) and a minor improvement in test errors (23 vs. 22), as well as a reduction in total duration by about 4.5 hours.

The **Code Representation** had an importance score of 0.083. The baseline used the complete code, whereas Trial 3 tested a MINIFIED version. This adjustment slightly reduced both test success (5 vs. 7) and test errors (21 vs. 22). Although this trial achieved the largest runtime reduction (6.31 hours), the decrease in success rates suggests that the backtracking limit is important for achieving success beyond 10 retries. **Dependency Change Representation** (importance: 0.07) and **Maximum Hops** (importance: 0.06) also affected performance. For instance, Trial 8, which combined adjustments to these parameters, saw declines in both test success (5 vs. 7) and test errors (19 vs. 22). Integration of **Language Server Protocol (LSP) Checks** showed the least influence (importance: 0.028). Activating this feature in Trial 1 maintained the baseline test error (22) and test success (7) rates, suggesting a stabilizing effect; however, it increased the duration significantly (32.24 hours vs. 13.11 hours). The integration of a language server-based linting and static analysis pipeline, inspired by the work of Agrawal et al. [2], represents a novel approach in our study. Unlike previous implementations that directly modified LLM output probabilities, our LLM-agnostic approach feeds the language server's diagnostics back to the LLM, allowing for iterative improvement of the generated patches.

Given its distinctive features, the outcomes of Trial 7 need particular consideration. Through the combination of several parameter modifications such as minified code representation, omission of dependency change information, and activation of LSP checks, this experiment yielded an unforeseen result in terms of output tokens. While most trials generated between 1–1.5M output tokens, Trial 7 produced a remarkable 10.2M output tokens, more than ten times the baseline.

In terms of overall efficiency, Trial 8 stands out as the most promising configuration. It achieved 5 test successes and 19 test errors, which is only slightly lower than the baseline (7 and 22, respectively). However, it accomplished this with significantly reduced resource usage: 5.34 hours of runtime (compared to 13.11 hours for the baseline), 10.18M input tokens (compared to the baseline of 32.36M), and 434.1K output tokens (compared to 977.9K).

We recognise that prompts themselves are brittle, and future work into optimising the prompt generation process could yield significant improvements. Since the prompt produces rather high input token usage, common LLM issues such as confusion due to long contexts [43, 59] could artificially deflate performance. Additionally, future studies could investigate the role of prompt parameter ordering, as the order of few-shot examples has been shown to influence performance [46], suggesting that zero-shot prompt order might also be an influential factor, as well as general prompt quality. Further experiments could also be tiered towards a systematic exploration of few-shot examples to improve the zero-shot approach's performance.

Answer to RQ3: API change representation and Maven error representation are the most crucial factors influencing the zero-shot approach's performance. The baseline was usual the best performing configuration, with the exception of Trial 2, which repaired one more candidate towards compilation success/test error. We identified a configuration (Trial 8) that significantly reduces resource usage (59% less runtime, 68% fewer input tokens) while maintaining reasonable performance (71% of baseline test success, 86% of baseline test errors/compilation success).

For claude-3-haiku, we discovered that hyperparameter optimisation is unable to improve the baseline's performance, thereby not improving the competitiveness to the agentic approach. Moreover, it indicates that the weak performance of the zero-shot approach is not caused by improper prompt inputs, but rather by the inherent limitations of the zero-shot approach.

5.4 Discussion

In our work, LLMs have shown the ability to effectively handle breaking changes in dependencies, which could result in more frequent and lower-risk updates in software projects. Concretely, in fixing abandoned Dependabot-initiated pull requests from the BUMP dataset, our solution helps in addressing potential security vulnerabilities from outdated dependencies. As a result, software security and stability could be greatly improved throughout the industry. Nevertheless, the shift from research to practical application poses numerous challenges that require careful consideration. When incorporating LLM-based repair tools into existing development workflows, it is important to carefully evaluate the strengths and limitations of these models. One potential avenue to explore is the creation of semi-automated systems that present LLM-generated patches to developers as suggestions, similar to existing code review tools. This is especially apparent when discussing the repairs that failed the test suite but passed the compilation. They could contain numerous semi-optimal solutions, that should be refined by human intervention.

This research has produced several tangible resources that contribute significantly to the research community:

- An enhanced version of the BUMP dataset, originally introduced by Reyes et al. [57], providing a more robust foundation for future studies in automated dependency updates.
- A substantially modified fork of the multilspy tool, initially developed by Agrawal et al. [2], optimised for Eclipse JDT-LS and packaged as a Docker image, facilitating easier integration of language server capabilities in future research.
- Docker images of the agentic upgrades, enabling straightforward replication and extension of our work by other researchers.
- Complete OpenInference traces, along with the experimental dataset.

These resources not only enhance the reproducibility of our findings but also provide a solid foundation for future research in this domain. Future studies could leverage these resources to explore questions related to the interpretability and reliability of LLM-generated code changes. The implications of this research extend beyond academia to industry practices. As LLM-based repair tools mature, they have the potential to significantly reduce the time and effort required for dependency management, allowing development teams to focus on more creative and high-value tasks. However, this transition will require careful consideration of ethical implications, such as the potential over-reliance on automated systems and the need for maintaining human oversight in critical code changes.

6 Threats to Validity

This study, like all empirical research, is subject to several threats to validity that must be acknowledged and addressed.

6.1 Internal Validity

Internal validity concerns the causal relationships inferred from our study. We identify the following threat to internal validity:

LLM Training Data: The knowledge cutoff dates of the LLMs used in this study encompass the majority of Java repositories on GitHub. This temporal overlap between the LLMs' training data and our test data could lead to inflated performance metrics if the models have prior exposure to similar codebases or update patterns and could threaten out-of-distribution applicability. We recognise however, that the majority of Java projects on GitHub are centered around a few popular libraries, making the case for very little need for generalisability for real-world applications.

Limitation of the Zero-Shot Approach: We limited the zero-shot approach to process one suspicious file at a time, since we lacked API Change information for the excluded projects of the `full` slice. This potentially could have led to an underestimation of the performance of the zero-shot approach, irrespective of poor outcomes with the omission of API Change information in the hyperparameter trials in RQ3.

6.2 External Validity

External validity relates to the generalisability of our findings beyond the specific context of our study:

Dataset The relatively small size of our dataset (65 projects for the `light` slice and 140 for the `full` slice) limits the generalisability of our findings. The performance characteristics observed may not hold for a broader, more diverse set of Java projects and dependency updates. In addition, the complexity of the projects in our dataset may not be representative of the ecosystem. We acknowledge that the underlying BUMP dataset could have had a selection bias towards certain projects and unknown limitations, which we could not control for.

Test Flakiness We cannot fully rule out that the test suite success rates are under-reported due to test suite flakiness. This could lead to an underestimation of the success rates.

LLM API Rate-Limits The use of LLMs in this study is subject to API rate limits, which could have affected the results. We mitigated this by using generous exponential retries and reporting of errors within our data collection pipeline. Yet, the rate limits could have affected the results in unforeseen ways.

LLM Selection The selection of LLMs used in this study, while diverse, does not encompass the entire landscape of available models. The performance characteristics we observed may not generalise to other versions or different LLMs. This paper does not use Chain-of-Thought prompting, or 'reasoning' models, which could potentially perform better (at higher cost).

6.3 Conclusion Validity

Conclusion validity relates to the reliability of our conclusions based on the observed data:

General Applicability There is a certain threat to the applicability of the LLM-based tooling presented in this paper, as the study by Lu et al. [45] shows that the emergent abilities of LLMs are mostly in-context learning from instruction tuning of the LLMs, which might not be applicable to the task of code repair, as the LLMs training dataset might not fully sample less common libraries, errors, etc. This is further supported by the study of Bubeck et al. [9], which shows that the planning capabilities of LLMs are not yet sufficient for complex tasks. This might be a reason for the low success rates of the LLMs in the agentic approach, as the planning capabilities are not sufficient to plan the repair of the code.

Model-Specific Findings in RQ3 The generalisability of our findings in Research Question 3, which examined input variations for the zero-shot system, may be limited due to the use of a single language model (claude-3-haiku) for all experiments. While this approach ensured comparability across trials, it potentially constrains the broader applicability of these results across different language models.

6.4 Mitigation Strategies

To overcome some of the identified limitations we included several mitigation strategies in our study: To address LLM selection bias, we experimented with a mix of open and proprietary models, including recent releases of varying sizes. To aid comparability between LLMs, we also used consistent temperature settings (T=0) across all providers to ensure comparability. We also measured test errors, i.e., compilation success as a secondary metric, recognising the limitations of test suites

as identified by Hejderup and Gousios [25]. By establishing that repair success is not correlated to dependency popularity, we reduced the risk of overfitting to popular dependencies.

7 Conclusion

Our examination of zero-shot prompting and agentic methods employing Large Language Models (LLMs) shows clear advantages: the agentic method shows remarkable efficacy in successfully passing test suites (up to 23%), while zero-shot prompting performs well in generating compilable repairs (up to 43%). The fix-rate of up to 23% is in line with the current state of the art [17]. We identified notable model-specific capabilities, namely in the manipulation of Java AST, which exhibit a robust connection (r = 0.702) with the success of repairs. The analysis of input variations for the zero-shot system highlights the significance of API modification details and Maven error representations on the system performance.

Overall, this study offers three important contributions to the field of automated program repair. Our study showcases the practicality of employing LLMs to automatically fix breaking changes in Java dependencies. The results highlight the potential of LLMs in effectively addressing practical software maintenance challenges. Additionally, we analyse the factors that impact repair performance. These factors include a specific agentic tool usage capability related to Java AST manipulation (which has a strong correlation of 0.702 with repair success) and the significance of representing API changes in zero-shot inputs. In conclusion, our study presents opportunities for further investigation, as well as industrial uptake.

Data Availability

The code is available on Zenodo under the DOI 10.5281/zenodo.14926755 [20]. The dataset with the full traces used in this study is available via Zenodo under the DOI 10.5281/zenodo.13686296 [21]. All Zenodo artifacts have README files with instructions on how to reproduce the results.

- The **enhanced version of the bump-dataset** is available as part of the code package, but the specific improvements via Pull Request are also available at GitHub:
  - https://github.com/chains-project/bump/pull/187
  - https://github.com/chains-project/bump/pull/186
- The LSP tool is accessible at https://github.com/LukvonStrom/multilspy-java.
- **Docker images** of the agentic upgrades are available via Zenodo (10.5281/zenodo.13686296).
- The **OpenInference traces** are available via Zenodo (10.5281/zenodo.13686296).
- The **experimental dataset**, including all inputs and final results, is available as part of the code package. (10.5281/zenodo.14926755)

References

[1] Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. 2024. CodeMirage: Hallucinations in Code Generated by Large Language Models. arXiv:2408.08333

[2] Lakshya Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Thirty-seventh Conference on Neural Information Processing Systems*.

[3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. doi:10.1145/3292500.3330701

[4] Meriem Belguidoum and Fabien Dagnat. 2007. Formalization of Component Substitutability. In *Proceedings of the 4th International Workshop on Formal Aspects of Component Software (Electronic Notes in Theoretical Computer Science, Vol. 215)*. doi:10.1016/J.ENTCS.2008.06.022

[5] Federico Bono. 2024. *Automatic Program Repair For Breaking Dependency Updates With Large Language Models*. Master's Thesis. KTH Royal Institute of Technology, Stockholm, Sweden. https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-354835

[6] Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. arXiv:2403.17134

[7] Aline Brito, Laerte Xavier, André C. Hora, and Marco Túlio Valente. 2018. APIDiff: Detecting API breaking changes. In *25th International Conference on Software Analysis, Evolution and Reengineering*. doi:10.1109/SANER.2018.8330249

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Annual Conference on Neural Information Processing Systems*.

[9] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712

[10] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. arXiv:2406.01304

[11] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. arXiv:2304.05128

[12] Andreas Dann, Ben Hermann, and Eric Bodden. 2023. UPCY: Safely Updating Outdated Dependencies. In *45th IEEE/ACM International Conference on Software Engineering*. doi:10.1109/ICSE48619.2023.00031

[13] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Evolution of Technical Lag in the npm Package Dependency Network. In *2018 IEEE International Conference on Software Maintenance and Evolution*. doi:10.1109/ICSME.2018.00050

[14] Jens Dietrich, Kamil Jezek, and Premek Brada. 2014. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. doi:10.1109/CSMR-WCRE.2014.6747226

[15] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*. doi:10.1145/3460945.3464951

[16] Thomas Durieux, César Soto-Valero, and Benoit Baudry. 2021. Duets: A Dataset of Reproducible Pairs of Java Library-Clients. In *18th IEEE/ACM International Conference on Mining Software Repositories*. doi:10.1109/MSR52588.2021.00071

[17] Hadeel Eladawy, Claire Le Goues, and Yuriy Brun. 2024. Automated Program Repair, What Is It Good For? Not Absolutely Nothing!. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. doi:10.1145/3597503.3639095

[18] Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. 2024. Exploring the Capabilities of LLMs for Code Change Related Tasks. arXiv:2407.02824

[19] Stan Franklin and Arthur C. Graesser. 1996. Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. In *Intelligent Agents III, Agent Theories, Architectures, and Languages (ATAL)*. doi:10.1007/BFB0013570

[20] Lukas Fruntke and Jens Krinke. 2024. Automatically fixing dependency breaking changes (Code). doi:10.5281/zenodo.14926755

[21] Lukas Fruntke and Jens Krinke. 2024. Automatically fixing dependency breaking changes [Data set]. doi:10.5281/zenodo.13686296

[22] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021. APIfix: output-oriented program synthesis for combating breaking changes in libraries. In *Proceedings of the ACM on Programming Languages*. doi:10.1145/3485538

[23] Paul Gauthier. 2023. Unified Diffs Make GPT-4 Turbo 3X Less Lazy. https://aider.chat/2023/12/21/unified-diffs.html

[24] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot. *IEEE Trans. Software Eng.* 49, 8 (2023). doi:10.1109/TSE.2023.3278129

[25] Joseph Hejderup and Georgios Gousios. 2022. Can we trust tests to automate dependency updates? A case study of Java Projects. *J. Syst. Softw.* 183 (2022). doi:10.1016/J.JSS.2021.111097

[26] Dávid Hidvégi, Khashayar Etemadi, Sofia Bobadilla, and Martin Monperrus. 2024. CigaR: Cost-efficient Program Repair with LLMs. arXiv:2402.06598

[27] Dániel Horváth, Viktor Csuvik, Tibor Gyimóthy, and László Vidács. 2023. An Extensive Study on Model Architecture and Program Representation in the Domain of Learning-based Automated Program Repair. In *IEEE/ACM International Workshop on Automated Program Repair*. doi:10.1109/APR59189.2023.00013

[28] Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. arXiv:2312.13010

[29] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2023. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. arXiv:2311.05232

[30] Dhanushka Jayasuriya. 2022. Towards Automated Updates of Software Dependencies. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. doi:10.1145/3563768.3565548

[31] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. Understanding Breaking Changes in the Wild. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. doi:10.1145/3597926.3598147

[32] Kamil Jezek, Jens Dietrich, and Premek Brada. 2015. How Java APIs break – An empirical study. *Inf. Softw. Technol.* 65 (2015). doi:10.1016/J.INFSOF.2015.02.014

[33] Naveed Jhamat, Zeeshan Arshad, and Kashif Riaz. 2020. Towards Automatic Updates of Software Dependencies Based on Artificial Intelligence. *Global Social Sciences Review* V, III (2020). doi:10.31703/gssr.2020(V-III).19

[34] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering*. doi:10.1109/ICSE48619.2023.00125

[35] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. arXiv:2408.02479

[36] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. doi:10.1145/3611643.3613892

[37] Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radicek. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence*. doi:10.1609/AAAI.V37I4.25642

[38] Mehdi Keshani, Simcha Vos, and Sebastian Proksch. 2023. On the relation of method popularity to breaking changes in the Maven ecosystem. *J. Syst. Softw.* 203 (2023). doi:10.1016/J.JSS.2023.111738

[39] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714

[40] Raula Gaikovina Kula, Daniel M. Germán, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empir. Softw. Eng.* 23, 1 (2018). doi:10.1007/S10664-017-9521-5

[41] LangChain Inc. 2024. LangGraph. https://langchain-ai.github.io/langgraph/

[42] Zhihao Lin, Wei Ma, Tao Lin, Yaowen Zheng, Jingquan Ge, Jun Wang, Jacques Klein, Tegawendé F. Bissyandé, Yang Liu, and Li Li. 2024. Open-Source AI-based SE Tools: Opportunities and Challenges of Collaborative Software Learning. arXiv:2404.06201

[43] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Trans. Assoc. Comput. Linguistics* 12 (2024). doi:10.1162/TACL_A_00638

[44] Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. 2024. MarsCode Agent: AI-native Automated Bug Fixing. arXiv:2409.00899

[45] Sheng Lu, Irina Bigoulaeva, Rachneet Sachdeva, Harish Tayyar Madabushi, and Iryna Gurevych. 2024. Are Emergent Abilities in Large Language Models just In-Context Learning?. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. doi:10.18653/v1/2024.acl-long.279

[46] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. doi:10.18653/V1/2022.ACL-LONG.556

[47] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? arXiv:2406.01422

[48] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories*. doi:10.1109/MSR52588.2021.00063

[49] Martin Monperrus. 2018. Automatic Software Repair. *ACM Computing Surveys (CSUR)* 51, 1 (2018). doi:10.1145/3105906

[50] Sebastian Nielebock. 2017. Towards API-specific automatic program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. doi:10.1109/ASE.2017.8115721

[51] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen J. Vinju. 2022. Breaking bad? Semantic versioning and impact of breaking changes in Maven Central. *Empir. Softw. Eng.* 27, 3 (2022), 61. doi:10.1007/S10664-021-10052-Y

[52] Behrooz Omidvar-Tehrani, Ishaani M, and Anmol Anubhai. 2024. Evaluating Human-AI Partnership for LLM-based Code Migration. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. doi:10.1145/

3613905.3650896

[53] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* (2015). doi:10.1002/spe.2346

[54] Kristiina Rahkema and Dietmar Pfahl. 2022. Analysing the Relationship Between Dependency Definition and Updating Practice When Using Third-Party Libraries. In *Product-Focused Software Process Improvement – 23rd International Conference*, Vol. 13709. doi:10.1007/978-3-031-21388-5_7

[55] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2023. MELT: Mining Effective Lightweight Transformations from Pull Requests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1516–1528. doi:10.1109/ASE56229.2023.00117

[56] Frank Reyes, Benoit Baudry, and Martin Monperrus. 2024. Breaking Good: Explaining Breaking Dependency Updates with Build Analysis. arXiv:2407.03880

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

[58] Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*.

[59] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. 2023. Large Language Models Can Be Easily Distracted by Irrelevant Context. In *International Conference on Machine Learning*. https://proceedings.mlr.press/v202/shi23a.html

[60] Arnav Singhvi, Manish Shetty, Shangyin Tan, Christopher Potts, Koushik Sen, Matei Zaharia, and Omar Khattab. 2023. DSPy Assertions: Computational Constraints for Self-Refining Language Model Pipelines. arXiv:2312.13382

[61] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741

[62] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2024. Where Do Large Language Models Fail When Generating Code? arXiv:2406.08731

[63] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. doi:10.1145/3597926.3598135

[64] Laerte Xavier, Aline Brito, André C. Hora, and Marco Túlio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*. doi:10.1109/SANER.2017.7884616

[65] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 Bugs for $0.42 Each Using ChatGPT. arXiv:2304.00385

[66] Xiufeng Xu, Chenguang Zhu, and Yi Li. 2023. Compsuite: A Dataset of Java Library Upgrade Incompatibility Issues. In *38th IEEE/ACM International Conference on Automated Software Engineering*. doi:10.1109/ASE56229.2023.00127

[67] Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Berkeley Function Calling Leaderboard. https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html

[68] Quanjun Zhang, Chunrong Fang, Yang Xie, Yuxiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A Systematic Literature Review on Large Language Models for Automated Program Repair. arXiv:2405.01466

[69] Li Zhong and Zilong Wang. 2024. Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*. doi:10.1609/AAAI.V38I19.30185

Received 2024-09-12; accepted 2025-04-01