Mining Reproducible Dependency Updates Across Ecosystems

What changes are made to dependency update pull requests before they are accepted?

Paras Khan

Supervisor(s): Sebastian Proksch, Cathrine Paulsen

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: <Paras Khan>
Final project course: CSE3000 Research Project
Thesis committee: <Sebastian Proksch>, <Cathrine Paulsen>, <Johan Pouwelse>

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Dependency management is a critical, difficult-to-automate task in software engineering. Researching automated dependency management requires reliable, reproducible datasets of dependency updates across ecosystems, but existing datasets fall short: they cover only specific update types (e.g. breaking updates) and log outcomes rather than the causal factors behind pull request (PR) acceptance and build outcomes. Closing this gap would let researchers determine not just whether a dependency update PR succeeded, but why — information needed to build dependency management tools that developers can trust. As a first step toward this, we construct a categorisation model that describes the code changes within accepted dependency update PRs on a commit level, developed using an established taxonomy-development methodology, and build a regex-based tool that automates this categorisation with 86% accuracy on hand-labelled data. Our results show that simple, deterministic techniques can reliably support transparent, automated change categorisation — a building block for future causal datasets of dependency updates.

1 Introduction

Dependency management is a critical task in software development, however modern attempts at automating it have proven to be flawed [1, 2]. The two main solutions so far have been automated dependency bots and semantic versioning. Automated dependency bots suffer from lack of trust by developers, and are also generally tedious to configure. Semantic versioning suffers due to non-compliance or protocol infractions by a significant portion of maintainers for dependency packages. To study and develop an effective automated dependency update tool that can fix the flaws of these two solutions, researchers need a dataset of transparent and reproducible dependency updates, especially one that covers a wide range of ecosystems, so that they can test and experiment with potential solutions using accurate real-world data.

The two current datasets in this field [3, 4] offer useful features such as reproducible build artifacts, or they cover multiple ecosystems e.g Maven for Java, pip for Python, npm for Javascript etc. The problem however, is that rather than focusing on code changes, they only log **outcomes** or high-level metadata e.g number of commits, comments, name of author etc. Reyes et al. [3] document the causes of build outcomes, e.g API changes, test failures/exceptions and so on, but this is only for **breaking builds**. This dataset does not describe the changes that happen in a project's code **generally**. This problem is the focus of this research paper.

The key gap therefore is the lack of general **causal** data: we can determine whether a dependency update broke a build and, to some extent, why, but not what was done to remedy a breaking PR so it could be accepted, nor the general changes made in non-breaking PRs, since causal data is currently extracted almost exclusively from breaking-build logs. The long-term vision is a dataset extension that documents, for every dependency update PR, both the causal **and** outcome data — describing not only whether and why a PR caused a breaking build, but also the general code changes made across all PR types, and what developers did to rectify them.

Closing this gap is valuable because researchers can not only determine whether a dependency update PR was accepted, but *why* it was accepted, which is vital information

needed to build trustworthy dependency management tools. According to Reyes et al. [3], most breaking dependency updates are caused by API changes that can be rectified in the project's own code; an automated tool could therefore still recommend an update provided the developer complies with the new API, which is a more accurate strategy than simply mapping build outcomes to PR acceptance decisions.

The focus of this paper is making the first step toward mining **causal** data for dependency update PR acceptances by examining the **changes** within PRs that get **accepted**. We break PRs down commit by commit and categorise each commit by the type of code change it contains (e.g. dependency update, source code change, or a mix of both). This categorisation narrows down the candidate causes behind a commit's contribution to a build outcome and also the PR acceptance. If a commit contains only a dependency update, that update is the sole candidate factor for that commit. If other change types are also present, our categorisation narrows down which changes are plausible causes. Actual causation can then be confirmed by pairing this categorisation with other features such as reproducing build outcome data as done by Reyes et al. [3], mapping call dependency graphs for deeper source code analysis as done in projects such as FASTEN-project [5], and mining metadata such as number of commits, comments, integrator information etc. This therefore gives deeper insight into how dependency updates are incorporated into projects and how project maintainers accommodate these changes.

The aim is to then create a tool that can be merged into a larger data mining tool for reproducible dependency updates which can be used across major programming ecosystems such as Java, Python, Javascript, Rust etc.

Our two main research questions are as follows:

1. RQ1: **How can we create a categorisation model that describes dependency update PRs that get accepted?**
2. RQ2: **To what extent can we automatically categorise dependency update PRs that get accepted?**

We ask RQ1 because differentiating code changes in a way that both explains their contribution to a PR and can be automated requires identifying recognisable patterns in those changes, which is precisely what categorisation provides. Categorisation is also a quantitative, deterministic procedure suited to a data-mining tool, unlike a qualitative description of each commit's exact changes and impact. RQ2 naturally follows, asking how to automate this categorisation in practice, and whether such a tool is feasible and reliable.

At the end of this paper, we find that we can create a model of categorisation for PRs that can be used to isolate build outcome factors and describe dependency update PR changes using code file analysis. We then find that we are able to create a classifier tool that can correctly and automatically categorise dependency update PRs according to this model. Our contributions are:

- Categorisation model that splits PRs into ordered lists of sets of labels, such that each set represents the type of code changes that happen in a commit e.g source code changes, dependency updates, a mix of both, etc.

- A categoriser tool that yields an accuracy of 86% on a random sample of non-bot dependency update PRs that were accepted

2 Related Works

Dependabot is a popular dependency management bot, but according to He et al. [2], it is riddled with "significance noise and workloads", and the bot's compatibility score or *confidence* score system is too scarce to effectively reduce update suspicion for most developers. For reasons such as this, automated dependency management is not always adopted.

Semantic versioning (semver) is a method to automatically track and understand dependency changes according to a major, minor and patch change format, but it also has weaknesses according to Raemaekers et al. [1]. For example, up to one-third of all library releases over seven years in Maven Central adhering to semver introduce one breaking change. For reasons such as this, the need for better methods for automated dependency management has been in demand.

Göçmen et al. [6] study the use of pull request-based change impact analysis to improve code reviews, although the focus of this study is creating an evaluative tool for PRs where risk scores are calculated to determine whether a PR should be merged, rather than a descriptive tool that tells the user what kind of changes happened in the PR on a higher level. The paper further highlights decades of research into CIA (change impact analysis), but the focus on PRs of dependency updates has not been particularly explored.

Rebatchi et al. [4] study the mining of PR-related issues on a large scale over a diverse ecosystem, however the study focuses on security rather than general changes, and only logs high-level metadata of dependency update PRs rather than analysing the deeper code/commit changes in the PRs themselves.

Reyes et al. [3] create a benchmark for reproducible breaking dependency updates for the Java ecosystem, but as is stated in the title, its focus is limited to both breaking changes and the Java ecosystem.

Zhang et al. [7] study the factors that influence PR decisions in general, and how they change with context, but this paper does not focus on dependency updates specifically, and does not dive deeper into code changes. Rather, it uses higher level PR data such as number of comments or whether the integrator is the same as the PR submitter and discovers their impact on PR decisions such as acceptances and rejections.

Overall, our approach differs to previous works by focusing on dependency update PRs that get accepted rather than general PRs, and describing the general changes in said PRs rather than evaluating confidence scores or logging build outcomes. Previous works have not touched on describing changes in dependency update PRs in the context of a mining tool for automated dependency management research. We want to create a tool that can analyse a dependency update PR, categorise the PR on a commit level, and isolate factors of build outcomes by analysing the history of the commits in the PR to reach a conclusion about how a dependency update was performed and what the developer did to ensure the dependency update PR could be merged.

3 RQ1: How can we create a categorisation model that describes dependency update PRs that get accepted?

In this section, we describe the methodology, experiment and results for RQ1.

3.1 Methodology

To create a suitable categorisation model for dependency update PRs, we must first decide on a system/framework which helps to not only construct the model but also validate it. Usman et al. [8] describes a development method for taxonomies in software engineering, which is rigorous, systematic and prescriptive in its clearly defined phases and steps. Nickerson et al. [9] provides a more iterative process of developing taxonomies that relies on defining meta-characteristics before conducting the taxonomy development process, which helps to guide the construction of the model. Neither taxonomy construction framework is necessarily better than the other, however for our experiment we focus on the method proposed by Nickerson et al. [9], coupled with a quantitative validation step that is fulfilled in the experiment in RQ2. The reason we use this framework is because the iterative nature better suits our problem as we explore the current datasets of dependency update PRs and use our meta-characteristic to guide the construction of our model, as opposed to creating a more rigid, systematic plan of our model that conforms to certain structures such as hierarchies or trees. Using a peer-reviewed taxonomy development framework enables transparency, reproducibility and validity of our categorisation model.

3.2 Experiment and Results

The dataset that we will be performing our analysis on is the dataset taken from Rebatchi et al. [4]. Namely, we are focusing on Partition (2) of Dataset (1), which constitutes all the dependency update pull requests of open-source projects mined on Github from 01/01/2023 to 30/09/2023. The reason we choose this partition is because it is data of the newest PRs in the overall dataset, which may be more relevant than if we had picked data from previous years. Older data has a higher chance of being more stale/irrelevant, or being affected by miscellaneous issues such as privated Github repositories. The total number of PRs in this partition is 3,342,829.

Step 1: First, we define our meta-characteristic. This characteristic is based on the purpose of the taxonomy, and how the taxonomy will be used. The user group of our taxonomy is researchers who are studying dependency update PRs for the purpose of developing an automated dependency management tool. Researchers want to be able to characterise and describe the contents of dependency update PRs to better reveal the context of a dependency update. Researchers already have access to reproducing build outcomes of dependency update PRs, however they also want to identify causal data, and differentiate between the different types of code changes that can determine a dependency update's PR eligibility to be merged and accepted into a project. Researchers want to use this taxonomy to automatically classify dependency update PRs and do data analysis on these PRs, which coupled with determining build outcomes, can reveal information about a dependency update in its entirety, especially in regards to why a dependency update was accepted in the first place. Thus, we define our meta-characteristic as 'Causes of dependency update PR acceptances'.

Step 2: Our ending conditions include all the objective and subjective conditions taken from Nickerson et al. [9], except for the objective condition of "All objects or a representative sample of objects have been examined", and the 'Comprehensive' subjective condition, because the dataset is quite large. This means that our end taxonomy will not be fully complete, however we aim to cover the vast majority of dependency update PRs to minimise this drawback.

3.2.1 Iteration 1

Step 3: We use an Empirical-to-Conceptual approach in our first step, because we have identified a major pattern after analysing the dataset and taking a convenience sample.

Step 4e: We collect two objects, which are 'Bot PRs' and 'Manual PRs'. Bot PRs are automatically created by common dependency update bots such as Dependabot, Renovate, Greenkeeper etc. They typically consist of single commit, single file changes where semver numbers are modified in the according dependency manifest files for that project e.g package.json for Javascript. Manual PRs are basically the PRs that are not Bot PRs, as they are made by human developers.

Step 5e: We identify two characteristics from this sample: first-party code changes and third-party code changes. These characteristics distinguish whether the introduced changes were made by a developer of the project or by external contributors. This distinction is relevant to our meta-characteristic because the source of changes may influence how a PR is reviewed and accepted. Zhang et al. [7] identifies that contributor-integrator relationship is one of the strongest factors influencing PR decisions, suggesting that source and ownership of PR changes is important contextual information for the acceptance of PRs. That being said, this applies for PRs in general, and there is a systematic difference between external *human* contributors and *bot* contributors.

Step 6e: We can group the two characteristics we found under one dimension: 'Source of Changes', to therefore construct our first taxonomy, which can be seen in Table 1.

Step 7: One dimension was added, so we need to iterate again.

For the sake of interest, we examine the balance of bot and manual PRs in the dataset by means of a filter. This rough filter tests a number of features of the PR, for example whether the author is of type 'Bot', whether their name ends with the word 'bot', and also use a reference list of common dependency bot names such as 'dependabot[bot]'. After running the filter, we collect statistics on how many of the PRs belong to each of the two object types, and the results are seen in Table 2. As we can see, a vast majority of dependency update PRs are actually automatic bot PRs. We will further expand on and discuss this fact later in Section 5.

3.2.2 Iteration 2

Step 3: We use a Conceptual-to-Empirical approach, because we determined a number of independent categories of code changes that we think may be common in the dataset.

Step 4c: We identify two characteristics, 'change in project code/functionality' and 'change

PRs	Source of Changes	
	FP	TP
Bot PRs		X
Manual PRs	X	

Table 1: First taxonomy.

Metric	Value
Total PRs	3,342,829
Filtered PRs	3,084,624
Filtering Rate	92.28%

Table 2: Bot Filter Statistics. The result shows that approximately 92.28% of the dataset consists of bot PRs.

in project configuration'. These characteristics are relevant to our meta-characteristic because they can change the outcome of a dependency update PR's build outcomes in independent ways, which may be relevant when trying to identify causal factors of a PRs acceptance. For example, if code functionality was modified, that affects the output of the project code, compared to if a dependency update occurred, which is more of a change in how the project is built and assembled rather than how the project code actually functions e.g its business logic. We group these characteristics together under the dimension 'Modification Type'.

Step 5c: We identify four types of objects as a result of these new characteristics: source code changes, dependency updates, changes in Continuous Integration (CI), and test code changes.

Step 6c: In attempting to revise the taxonomy model, we now encounter a problem. The new characteristics we added are not unique within their dimensions for all objects, namely Manual PRs and Bot PRs. So far, we have been treating objects on the PR level, however PRs typically consist of a mix of changes in practice that make them difficult to differentiate into a taxonomy. For example, a Manual PR can include both a change in configuration (dependency update) and a change in functionality (source code change). It is not necessary that these changes are mutually exclusive. To solve this problem, we revise and redefine how we view objects in our taxonomy model.

We can view PRs as an ordered list of sets of labels. A set of labels represents a commit with certain code changes within it, and an ordered list represents the commit history for that PR. We use a set because a commit's code changes cannot be ordered, as the commit is the smallest unit of ordered code change in a Git project. We also do not care about duplicates, since we want to identify which types of changes *at least* happened, as that gives us information about the possible factors that influenced a PR's acceptance and its build outcomes. Rather than creating a taxonomy of PRs, we now create a taxonomy of types of code changes (labels) within a commit, as a PR can be defined by its commit changes. This

Labels	Modification Type		Source of Changes	
	CNFG	FNC	FP	TP
Source Code		X	X	
Dependency Update	X			X
CI	X		X	
Test	X		X	

Table 3: Second taxonomy.

means that we remove the two objects 'Manual PR' and 'Bot PR' from our list, but this is valid, because with our new structure we can still define both types of PRs, however they may not necessarily be differentiated. If we still want to differentiate Bot PRs and Manual PRs, we make this distinction outside of the taxonomy, because both PRs can make the same types of code changes, for example a dependency update can occur both in a Manual PR and a Bot PR, or a source code change can happen in a Bot PR that was committed to by one of the developers after the initial dependency update commit initiated by the bot itself. In addition, we also reinterpret (or redesign) the 'Source of Changes' dimension to mean whether a code change modified project code (first-party) or third-party software e.g a dependency. We no longer differentiate *who* the contributor is. After making these changes, we come up with a new taxonomy as can be seen in Table 3.

Step 7: Since we have added new dimensions, and we have cell duplication, we must iterate again.

3.2.3 Iteration 3

Step 3: We again use Conceptual-to-Empirical approach as our current taxonomy does not meet our objective ending conditions.

Step 4c: We conceptualise 4 new characteristics grouped under two dimensions. The first dimension is 'Affects Code Results', where the characteristics are either 'Affects' or 'Does Not Affect'. These characteristics tell us whether this type of commit change actually affects the output/functionality of the project code itself. The second dimension is 'Affects Build Outcomes' which tells us whether this type of change potentially affects the **build outcomes** of the commit. Just like the former, the two characteristics are 'Affects' and 'Does Not Affect'. This is relevant to our meta-characteristic because the code and build output of a commit is naturally a significant factor in determining whether the PR will be accepted or not. A commit that fails the project's build, for example, makes the containing PR much less likely to be accepted compared to a commit that either doesn't fail the build or doesn't affect the build at all, provided that there is no commit afterwards that rectifies such an issue. This applies similarly for project code output.

Step 5c: We determine a new object after examining the dataset, which we refer to as 'Documentation'. These are the changes that solely have to do with files such as README or CHANGELOG files. These changes have no impact on the actual functionality of the

Labels	Modification Type		Source of Changes		Affects Code Results		Affects Build Outcomes	
	CNFG	FNC	FP	TP	A	DNA	A	DNA
Source Code		X	X		X		X	
Dependency Update	X			X	X		X	
CI	X		X		X		X	
Test	X		X			X	X	
Documentation	X		X			X		X

Table 4: Third taxonomy.

project code, nor the build outcomes. It is important to differentiate this object as our data mining tool should not misattribute the causal factors of a build outcome to a documentation change.

Step 6c: The new taxonomy can be seen in Table 4.

Step 7: Since we added two new dimensions, we iterate once more.

3.2.4 Iteration 4

Step 3, 4e, 5e, 6e, 7: We attempt to use further an Empirical-to-Conceptual based approach using convenience sampling, however we do not find any new objects from our samples. This does not guarantee our taxonomy is complete, however we can identify that we've met all of the ending conditions we set out to complete.

For the objective conditions, we see that our dimensions are unique, and so are the characteristics within the dimensions. This means our cells are all unique. We also have not added, merged, split or removed any objects, dimensions or characteristics, and at least one object is classified under every characteristic.

For the subjective conditions, we find that our dimensions are 'Concise' with there being just four dimensions. The dimensions and characteristics provide sufficient differentiation between the object types, as they explain how each type potentially contributes to the acceptance of a dependency update and also its build outcomes, thus making the model 'Robust'. For example, if we label a commit as having a 'documentation' change, then we know that that change has a much different contribution to the acceptance of a dependency update PR as compared to a source code change or a dependency update (and zero contribution to the code results or build outcomes). Although we cannot guarantee the 'Comprehensive' condition since we did not analyse all objects in the dataset, we can remedy this by adding a new label called 'Other' which automatically classifies any object that is not within the current taxonomy, however such a label does not technically fit into our taxonomy as it can create cell/characteristic duplication. Our model is 'Extendible', for example you can expand further the 'Source Code' object into different types of source code changes e.g 'method signature change' or 'class constructor modified', and then pair these with added dimensions

to keep the taxonomy valid under the objective conditions. You could also extend the model by analysing more objects, since the taxonomy is still technically incomplete. Finally, the model is 'Explanatory', because the characteristics explain how the specific object affects the outcomes of the dependency update and also where the object comes from.

To validate this taxonomy quantitatively, we apply the categorisation model in RQ2 and examine the results of our findings.

4 RQ2: To what extent can we automatically categorise dependency update PRs that get accepted?

In this section, we describe the methodology, experiment and results for RQ2.

4.1 Methodology

To answer this question, we aim to do three main things:

1. Develop a tool that can automatically label dependency update PRs with the categories found in RQ1.
2. Explain and justify each decision taken and technique used in the tool.
3. Test the tool using a hand-labelled ground truth and explore the limitations of our method based on the result.

We choose a regex-based, deterministic classifier as our primary approach over a machine-learning-based one for two reasons. First, it preserves full transparency in why a commit receives a given label, which matters for a tool meant to support causal analysis of PR acceptance rather than obscure it behind a learned model. Second, it requires no labelled training data, which is significant given the absence of existing labelled datasets for this categorisation task. Analysing code changes in commits can still be endlessly complex, since edge cases and unexpected scenarios may undermine a deterministic approach in practice; if our regex-based method proves insufficiently accurate, our fallback is to explore ML-based alternatives, tested against an independent hand-labelled ground truth to avoid data leakage from our initial results.

We apply our categorisation model from RQ1 by developing this automated algorithm in two stages: first developing it, hand-labelling a ground truth and testing the tool against it without further changes, to evaluate whether our approach works and identify its limitations. In the second stage, we repeat this process using a modified or different approach and test on a new hand-labelled ground truth. Transparency remains essential throughout, since a dataset with faulty labels is difficult to validate and would undermine further research; it should always be possible to identify why a given PR was categorised the way it was.

4.2 Experiment and Results

We create a categorisation tool that expands a PR's commits and their respective diff files, inspects them, then classifies all the commits in order. The code is available on a public GitHub which is published with this paper. We now describe the logic of our algorithm.

Category	Correct Label Ratio	Percentage
Dependency Update Only	10/10	100%
Source Code Only	7/10	70%
Test Code Only	5/5	100%
CI Only	4/5	80%
Other Only	1/10	10%
Mixed	9/10	90%
Overall	36/50	72%

Table 5: First experiment run on 50 quota-based samples based on hand-labelled ground truth.

First, we match file types with regular expressions for each type of category. For example, we check if a file matches package(-lock)?\.json and if it does then we know it is probably a file with a dependency update occurring inside it. We then check for white-spaces/comment lines to ensure that miscellaneous line changes do not affect the categorisation. If the file does not match any of our categories, we categorise it as 'Other'. Sometimes files which we usually consider as source code files can include dependency updates e.g setup.py. We make an extra check for source code files to check if there were only semantic version number changes inside the file. If there were, then we can assume that this file change was a dependency update and not a typical source code file change.

When we check for dependency updates, we strictly check if there was a line where a semantic versioning number was involved, and there was exactly one addition and one removal at that line, with the only difference being the version number. This should be the closest definition to a dependency update. We do not want to include dependency additions or removals for example, only dependency updates.

To test this tool, we randomly sample PRs from the chosen dataset using a quota sampling technique. Note that we filter out the Bot PRs using the rough filter described in RQ1, as otherwise the vast majority of PRs we test on would just be singular dependency update commits, and the results would therefore not be very interesting. We first take a pull request, classify it using our tool, and then add it to the chosen quota. We then go through the samples and manually test if the labels are correct. Afterwards, we also randomly sample PRs without the quota sampling technique, label them *first*, then test our tool on the samples to see if they are correct. The reason why we do it this way is to first take samples such that we get a relatively equal spread of each type of category of PR, since some categories are much rarer than others e.g 'CI' and 'Test'. We can then see how our tool then handles each category type. Afterwards, we randomly take samples to get a closer view of how our tool fares generally, since random sampling is closer to how the tool would actually be used. After this test, we examine the accuracy ratio, and also the causes for why our tool made incorrect labels. We then attempt to improve the tool to fix those causes.

For the quota sampling stage, we define 6 quotas. First, we take 5 dependency update

Category	Correct Label Ratio	Percentage
Dependency Update Only	17/20	85%
Source Code Only	3/3	100%
Test Code Only	0/0	N/A
CI Only	5/5	100%
Other Only	2/2	100%
Mixed	16/20	80%
Overall	43/50	86%

Table 6: Second experiment run on 50 random samples based on hand-labelled ground truth.

PRs with **only** 'Test' changes and 5 PRs with **only** 'CI' changes. Then we take 10 each for 'Dependency Update', 'Source Code', and 'Other' changes, such that each set of 10 PRs include **only** the according labels. Finally, we take 10 PRs that can have an assorted mix of different commit labels.
Pure 'Test' and 'CI' PRs are generally much rarer, which is why we sample 5 of each rather than 10. We also exclude the PRs with more than 6 commits for the sake of making it easier to manually label. This is safe to do because the labelling of each commit is independent of the commits before or after it, so the tool should work as accurately for a PR of 5 commits as for a PR of 100. The results are in Table 5. It is important to note that we classify correctness based on whether every file in every commit was labelled correctly. If a PR had one file misclassified in one commit, we label the whole PR as incorrectly classified.

For the second run of the experiment, after making improvements on the code logic, we get another set of results, which can be seen in Table 6.

The purpose is not to compare the results of the first and second run of this experiment, since they were sampled differently. The purpose is to first find faults in the classifier in the first run using quota-based samples, then test the classifier *generally* to see how well it fares. If we want to check whether our results in Table 6 are statistically significant, we use a hypothesis test, which we can do because we used random i.i.d samples. We check if our tool has a statistically significant correct classification rate of over 70%. We choose 70% because it approximates the accuracy observed in the quota-sampled experiment. This is an informal reference point and not a directly-comparable baseline, since the two experiments used a different sampling procedure. To our knowledge, a principled threshold accuracy for this domain of classification does not exist.

$$H_0 : p = 0.70, H_1 : p > 0.70$$

$$X \sim \text{Bin}(50, 0.7), P(X \geq 43) = 0.00726$$

If we choose a significance level of 5%, then $0.00726 \leq 0.05$, meaning our result is statistically significant. In addition, we can also compute the Wilson confidence interval for another view of our classifier score. For a sample size of 50, a confidence level of 95% and a classifier score of 86%, the confidence interval for our score ranges from 73% to 93%. The lower bound for

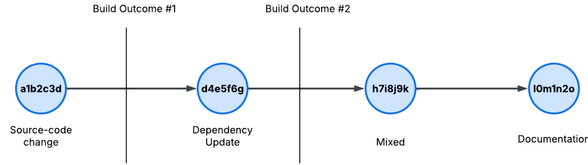

Figure 1: Example of a commit history with build factors isolated.

this interval is still greater than 70%, which is consistent with the results of our hypothesis test.

The biggest factor affecting the score in the first run is the mislabelling of the 'Other' category, which was caused by the fact that our logic for classifying dependency updates was too narrow, causing these dependency update files to be incorrectly classified as 'Other'. Rather than checking strictly if there is a line with a dependency update containing a semver number, and no other change, we correct this by just checking if a semver change happened, and allow other miscellaneous changes e.g commit SHA changes or timestamp changes. For files with a semver change and some other miscellaneous changes, we classify them as both 'Dependency Update' and whatever label is associated with the file type. For example, if a dependency update happens in a source code file but there's also some other miscellaneous changes, we will classify that commit as a 'Dependency Update' and 'Source Code' change file.

Aside from that, we find that the common pattern for false classifications throughout the experiment arises due to edge cases, or bugs such as Git file diffs/patches not rendering during the classification process. For example, we identify whether a dependency update happened by searching for a semver number change in a file, however not all dependencies follow the proper semver protocol e.g instead of writing '3.5.2', we instead see '3.4.LATEST_RELEASE'.

5 Discussion and Limitations

In RQ1, we developed a categorisation model for the commits of dependency update PRs using a peer-reviewed framework for developing taxonomies. Evidence for our model's validity was backed by a mix of objective and subjective validation conditions, which was further tested quantitatively in RQ2 by testing a categorisation tool using this model, which achieved a final accuracy score of 86%.

The first topic to discuss is the validity of our taxonomy: is the model valid, is it the only valid model for this domain, and is it actually useful? We do not have evidence that our model is the only valid one — another researcher applying the same framework may well arrive at different categories. We do, however, have evidence that our model itself is valid and useful, per the objective and subjective conditions covered in RQ1. We note again that we omitted the "comprehensive" condition, and we did not exhaustively sample every object type in the dataset, leaving the taxonomy partially incomplete.

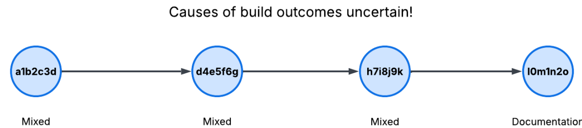

Figure 2: Example of a commit history with build factors that cannot be isolated.

That being said, the model works well in practice. In RQ2, a deliberately simple regex-based classifier reached 86% accuracy on a partition of the dataset from Rebatchi et al. [4]. This suggests either that the algorithm was well designed, or — more likely — that the model itself draws categories that are inherently easy to differentiate, since orthogonal, non-overlapping characteristics make classification logic simpler to implement once the characteristics are known in advance. This contrasts with typical machine-learning settings, where difficulty usually stems from overlapping categories rather than the classification logic itself.

We can illustrate the model's utility with an example. Figures 1 and 2 show two kinds of dependency update PRs. In Figure 1, each commit's code-change type is isolated — one commit is purely a source code change, the next purely a dependency update — so the contribution of each category to the build outcome can be identified separately. We call such PRs reproducible. In Figure 2, by contrast, commits mix multiple change types, so it is not clear (without manually reading the code) which change drove the build outcome — a non-reproducible PR. This distinction matters in practice: in a non-reproducible PR, a dependency update might introduce a breaking change that a subsequent source-code or CI change then repairs, letting the build pass and the PR merge — but our categorisation alone can't tell us this happened.

This is useful for researchers building a data-mining tool for reproducible dependency updates: by reading the ordered label sets for a PR, they can check reproducibility (do commits contain a single label type, excluding documentation?) or look for cases where a dependency update's breaking change was remedied by a source code fix — a signal that the update is worth investigating further.

The limitation is that not all PRs are reproducible under this definition, and those that aren't require other means of investigation. However, Table 2 in RQ1 shows that around 92.28% of dependency update PRs are simple Bot PRs — typically single commits containing only a dependency update — meaning most dependency update PRs are reproducible by default. This category is trivial to categorise, but still useful for examining causal factors across the majority of PRs.

The final point of discussion is in regards to the classifier we developed and its score of 86%. As mentioned before, part of the reason the score is so high is potentially due to the model. Upon further examination, we find that a lot of the PRs in the dataset still tend to be Bot PRs, as these are the PRs that made it past our rough filter. This is a limitation that can potentially skew our results since these Bot PRs are trivially categorisable, thus increasing the classification score. That being said, we find just as many Mixed PRs, and the

classification scores for the Mixed PRs and the Dependency Update Only PRs are similar within 5%, so perhaps the score is not skewed after all.

6 Conclusion and Future Work

In this paper, we identify the gap in current dependency update datasets: existing work captures build outcomes or high-level PR metadata, but not the general, causal code changes within accepted PRs. To address this issue, we developed a model and tool for the automatic categorisation of PRs on a commit level, so that the causal factors of PRs could be mined and differentiated. In conducting this research, we asked two research questions. The first question studied the method of constructing a valid and useful categorisation model of PRs, developed in the context of mining causal factors behind dependency update PR acceptances and build outcomes. This was based on the work of Nickerson et al. [9]. The second question studied the feasibility and accuracy of an automatic categorisation tool based on the model constructed in RQ1.

We found that we could construct a valid and useful taxonomy of dependency update PRs by organising them as ordered lists of sets of causal factors. We then used this model alongside a regex-based classifier on the code files of dependency update PR commits to achieve a classifier accuracy of 86% using hand-labelled test sets.

The results show that determining causal factors behind PR acceptances is feasible and automatable in practice. Although the majority of them are Bot PRs which are trivial to analyse, we can still meaningfully differentiate the characteristics within the remaining PRs using our constructed taxonomy. By viewing and reasoning about the contents of the ordered label sets, researchers can examine and extract the motivations behind PR acceptances. In the future, researchers can expand on this tool by creating a more complete and fine-grained taxonomy that splits commit categories into more specific changes, and pair the categorisation with other features such as reproducing build outcomes, mining PR meta-data, call dependency graph mapping and so on to create a holistic view of dependency update PRs that discern both causal factors and outcome data, thus creating full transparency behind dependency updates. This resulting data-mining technique can then produce datasets of reproducible dependency updates, mined across different ecosystems, which will fuel further research into automated dependency management tools.

7 Responsible Research

There are two main threats to reproducibility to our research:

1. Reproducing the categorisation model that was developed in answering RQ1
2. Reproducing the random samples and associated classification scores in the experiments from RQ2

For RQ1, we maintain that the focus of this section of the research is in **how** we create our categorisation model rather than the model itself. An independent researcher may arrive at different labels were they to conduct the same qualitative analysis done in RQ1, however the way they would do it would be the same as we did in this paper. This is ensured because not only do we outline the process in how we do our qualitative analysis, but also reference the framework used to motivate our choice of categorisation model at every step of the analysis.

For RQ2, we maintain both random seeds and the different versions of the code used for each experiment, which is numbered accordingly in the repository. This ensures the same code can be reran in a different environment and the results should stay the same.

Apart from reproducibility, we reflect on the ethical considerations of this research. As stated before, dependency updates are a critical component of software engineering, and therefore all types of software projects rely on good dependency management to prevent bugs and vulnerabilities. Large scale software services such as social media, government institutional services, data-sensitive services, operating systems and other important software projects all use dependencies, whether internal or external, to develop and grow. It is therefore not uncommon for bad dependency management to introduce issues that range anywhere from small bugs to severe security vulnerabilities that could potentially lead to thousands of users affected. A famous example is the xz Utils backdoor that happened in 2024, when a malicious attacker gained maintainer privileges over the xz compression library and introduced an OpenSSH backdoor in a dependency update, affecting numerous Linux distributions in development versions. Although it was caught early to prevent deployment to production, such a dependency update could have exposed hundreds of millions of devices to malicious attackers.

Therefore, it is crucial that when we develop automated dependency management techniques, we are careful about our assumptions and the impact of our research. If researchers were to haphazardly use a flawed data-mining tool that does not examine all facets of a dependency update and only focuses on features such as build outcomes instead of other features such as security, then their research could lead to flawed automated dependency management tools which could potentially affect millions who trust and use them. Therefore it is important to clarify that the data-mining feature that we are trying to develop simply categorises commits for the sake of determining the factors behind dependency update acceptances and build outcomes. It does not categorise the risk or validity of a dependency update itself. If this data mining tool were to be used in further research, special consideration should be taken in regards to the 86% classifier score, as this number does not tell the whole story. Instead, focus on the *method* behind the classification itself, as this method was specifically designed to be transparent. Future research should take care that the remaining 14% inaccuracy does not invalidate downstream findings.

A LLM Statement

I hereby state that I have not used LLMs whatsoever during the course of writing this paper, except for the following:

1. Searching for relevant literature to support my research.
2. Querying the LLM for LaTeX syntax to format my research paper.
3. Rephrasing paragraphs to shorten/make more concise. I started with writing rough drafts of all my paragraphs in this paper, then queried the LLM to help cut the word count.

B GitHub Repository

https://github.com/OmegaTurnip/RP

References

[1] Steven Raemaekers, Arie Van Deursen, and Joost Visser. “Semantic versioning versus breaking changes: A study of the maven repository”. English. In: *Proceedings - 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014*. 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014 ; Conference date: 28-09-2014 Through 29-09-2014. United States: IEEE, Dec. 2014, pp. 215–224. DOI: 10.1109/SCAM.2014.30.

[2] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. “Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot”. In: *IEEE Transactions on Software Engineering* 49.8 (2023), pp. 4004–4022. DOI: 10.1109/TSE.2023.3278129.

[3] Frank Reyes, Yogya Gamage, Gabriel Skoglund, Benoit Baudry, and Martin Monperrus. “BUMP: A Benchmark of Reproducible Breaking Dependency Updates”. In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Mar. 2024, pp. 159–170. DOI: 10.1109/saner60148.2024.00024. URL: http://dx.doi.org/10.1109/SANER60148.2024.00024.

[4] Hocine Rebatchi, Tégawendé F. Bissyandé, and Naouel Moha. “Dependabot and security pull requests: large empirical study”. In: *Empirical Software Engineering* 29.5 (2024). ISSN: 1573-7616. DOI: 10.1007/s10664-024-10523-y. URL: http://dx.doi.org/10.1007/s10664-024-10523-y.

[5] FASTEN-project. *Fasten*. Source code. n.d. URL: https://github.com/fasten-project/fasten.

[6] Ismail Sergen Göçmen, Ahmed Salih Cezayir, and Eray Tüzün. “Enhanced code reviews using pull request based change impact analysis”. In: *Empirical Software Engineering* 30.3 (Feb. 2025). ISSN: 1573-7616. DOI: 10.1007/s10664-024-10600-2. URL: http://dx.doi.org/10.1007/s10664-024-10600-2.

[7] Xunhui Zhang, Yue Yu, Georgios Gousios, and Ayushi Rastogi. “Pull Request Decision Explained: An Empirical Overview”. In: *CoRR* abs/2105.13970 (2021). arXiv: 2105.13970. URL: https://arxiv.org/abs/2105.13970.

[8] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. “Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method”. In: *Information and Software Technology* 85 (May 2017), pp. 43–59. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2017.01.006. URL: http://dx.doi.org/10.1016/j.infsof.2017.01.006.

[9] Robert C Nickerson, Upkar Varshney, and Jan Muntermann. “A method for taxonomy development and its application in information systems”. In: *European Journal of Information Systems* 22.3 (May 2013), pp. 336–359. ISSN: 1476-9344. DOI: 10.1057/ejis.2012.26. URL: http://dx.doi.org/10.1057/ejis.2012.26.