Exploring Architectural Design Decisions in Issue Tracking Systems

Sarah Druyts (S4489691)
Supervisor: Dr. M.A.M. Soliman

September 26, 2022

Contents

1 Background

1.1 Knowledge Types

Software is created through making a series of decisions, which are then implemented in the source code of a system, or a component of the system. According to Kruchten (2006), there are four types of decisions that can be made, with a difference in how the decision is made and documented.

The first distinction is how the decision is made. An architect can make **implicit** decisions: if the decision was so 'obvious' that the architect did not consider any alternatives, or if they did not realize they were making a decision at all. These decisions never become documented.

All other decisions are made **explicitly**. Here, we make distinctions in how the decision is documented. Firstly, an architect can simply forget or forego documentation, due to time constraints or other reasons, in which case the decision remains accidentally undocumented and the reasoning behind it will likely become lost in time. Next, an architect can also choose explicitly not to document a decision: the purpose behind such a conscious decision is often obfuscation, either for the benefit of the company (in order to give the company a tactical advantage over competitors) or of the architect (to protect their position within the company). Lastly, the architect can also choose to document their explicitly made decision, which is the preferred outcome.

The first and third types of architectural knowledge are, by definition, impossible to find. The last type should be easy to discover. The process of finding knowledge of the second type, that is, the explicitly made decisions that remained undocumented but not on purpose, is the focus of this report.

1.2 Architectural Decision Types

Kruchten (2006) also defines four types of architectural decisions that can be applied to software. For this project, two have been grouped together, as they are very similar. How to discover if a decision is one type or another will be described in the Coding Book section of this report.

1. **Ontocrises and Anticrises, or Existence and Ban Decisions.**

   These decisions talk about the presence (or non-presence) of elements in (a component of) the system. These can further be split up into Structural, which relate to the structure of the software system and can lead to the creation of layers, subsystems and components, and Behavioural, which relate to how the system or its relevant component behaves.

2. **Diacrises, or Property Decisions.**

   Property decisions are characterized by their obvious motivation to change something about an overarching quality of the system. Examples of such qualities are performance, scalability, availability, and security.

3. **Pericrises, or Executive Decisions.**

   This type of decision does not directly impact the outward functionality of the software, but is characterized by having to bend to external pressures, both from within the business environment and the wider coding community. Executive decisions can be of various subtypes, and two common ones are Technology, where the architect decides to use an external technology in the system to fulfill a requirement, and Tools, where the architect uses an external technology to develop the system.

1.3 Jira Issue Characteristics

This paper specifically focuses on knowledge as found within issues on the issue tracking system Jira. Primarily, they are useful for the developers working on the system to quickly be able to tell what the issue is about, and they might give us a hint as to if there is architectural knowledge in the issue, and if so, which type. The characteristics looked at in this report are as follows:

1. Issue Type: E.g. “Improvement”, “Bug”, “New Feature”, ...
2. Status: Where this issue currently is in the pipeline of “newly reported” to “finished with”, which can range from “won’t fix” to “resolved”.
3. Resolution: How this issue was resolved, e.g. “fixed”.
4. Description: Reporter’s description of the issue. Can range from just describing a functional requirement to giving stack traces or proposing a fix.
5. Attachments: Files that users have attached to this issue, e.g. error dumps, comparative performance graphs.
6. Comments: Other users’ input on this issue.
7. Parent Issue: Some issues exist under the umbrella of an overarching epic, e.g. a parent issue called “Improve Performance of System X” with sub-issues of “Change y to z to improve performance” or “Improve performance in Subsystem W”.

1.4 The Maven Dependency Method

The method under investigation in this report is called the Maven Dependency Analyzer method. Through Maven, this method analyzes how many changes were made to pom files in one commit and, if more than one change is present, tries to find the relevant Jira issue for the commit. Then, the issues are manually analyzed.

For each project, the Maven method has discovered the following amounts of issues:

	Cassandra	Hadoop	HDFS	Yarn	Map-Reduce	Tajo
Architectural	127	22	10	6	16	67
Non-Architectural	57	28	7	3	19	41
Total	184	50	17	9	35	108

2 Coding Book

This section will provide a guide to differentiating between the different architectural decision types, by going into more detail and giving examples from analyzed projects.

2.1 Existence Decisions

These decisions are characterized by a thought process of “this (component of the) system exists”, or in the case of a ban, “does not exist”. It is important that the issue in which this decision is codified describes the new (component of the) system in some amount of detail: an issue that is only the functional requirement is not enough documentation to gain useful knowledge from.

An example of such an under-detailed issue is CASSANDRA-7562: it only mentions the functional requirement, and not any implementation details, such as which component of the system will be affected. The implementation details can however be found in the comments of this issue, which does turn this issue into an existence decision.

A different example of an existence decision is CASSANDRA-1075. This issue declares the existence of (certain functionalities of) an API in order to solve a problem.

When found in an issue description or summary, the word ‘encapsulate’ may signal the presence of existence decisions: encapsulation is a structural decision being taken, in order to increase maintainability of the system, so if an issue is describing to encapsulate something, there is a high chance it is an existence decision. An example of this is issue CASSANDRA-7443.

2.2 Property Decisions

Property decisions are always motivated by a desire to change something about a quality of the system. Qualities can be performance, availability, scalability, security, etc. They are not necessarily about ‘properties’ of the system, such as the ability to support a certain Java version.

For example, the quality ‘performance’ is a common one to make property decisions about. A ticket that changes a compression algorithm out for a faster one, while specifically stating that the intent behind this change is to improve performance, is a property decision. An additional rule for performance decisions is that the change in performance must be noticable by the end user of the software. An additional property decision example is CASSANDRA-11040, where the developers clearly state the desire to improve the security of the system.

It is not true that any decision that changes anything about performance, or any other quality of the system, is automatically a property decision. As stated above, the intent is key, and must be stated explicitly. For example, in CASSANDRA-14427, the developers update a tool, and state that as a consequence, security should also be improved. This is not a property decision: security was not the motivation behind it, the improvement to it was a coincidence.

The technology Netty is popular and almost always used to increase the performance of a system. Hence, if an issue mentions this technology, it is likely that this ticket is trying to do something with the performance of the system.

2.3 Executive Decisions

Executive decisions are motivated from the external development environment. Financial, methodological, educational or training-related factors can play a role in executive decisions. As such, from the viewpoint of the software system, these decisions always have an element of “out of our control”.

The most common and easily diagnosable executive decision type is technology. For example, because a certain library already exists with active support, and we do not want to reinvent the wheel, we will use this external technology in our project. This is one executive decision that can set us up to create many more in the future: if this technology receives an update, for example to fix some security problems, we need to update it in our project. An example of an executive decision updating a technology is CASSANDRA-12032. If the technology becomes deprecated, we need to find or create a replacement.

Another frequently seen executive decision type is tools. Let us take as example CASSANDRA-12197: in this issue, the proposal is to change something about the development environment of the software system, to make the job of the developers easier.

2.4 Multiple Types At Once

It is possible for one issue to contain multiple types of decisions at once. For example, there is CASSANDRA-10528, which describes both an executive and an existence decision at once: it proposes to integrate an external technology, which is the executive element, in order to be able to implement certain feature, which is the existence decision. On the executive front, the author even explicitly states that they do not want to rewrite such a framework, necessitating the new technology, and they also discuss the technology itself in terms of benefits, drawbacks and features. Existence-wise, the systems that are to be modified are mentioned, as well as define code conventions, such as "avoid code complexity/readability issues".

Very rarely, an issue can be all three types at once. One such issue is CASSANDRA-7039: the initial motivation for this issue is to increase performance, which makes it a property decision. In order to achieve this performance increase, the developers are looking at integrating an external technology and discussing it, which makes it an executive decision. They are even forking the technology, participating in its open source development, in order to speed up the development of the features they need in it. Lastly, the issue implies that existing interfaces need to be edited, which also makes it an existence decision.

As discussed in the executive decision section, some technologies may need to be updated because of security concerns. This type of issue is by definition both executive and property, because it is improving something about the security quality of the system. Examples of this dual-type issue are CASSANDRA-16462, CASSANDRA-14183 and CASSANDRA-16150.

3 Precision Analysis

In this section, we take a closer look at the efficiency of the Maven method for discovering architectural knowledge, as compared to other methods. There are two main other methods for this: static source code analysis, which may also be referred to as 'Static SC Analysis', 'SSC' or 'SC Analysis', which analyzes the source code as its basis for finding potential architectural knowledge, and Keywords Searches, which may also be referred to as 'KS' or 'Keywords', which comprises four carefully created queries that contain different categories of keywords. These different queries serve to analyze the connection between the presence of different types of keywords and architectural knowledge: Components and Connectors (CAC), Decision Factors (DF), Rationale (R) and Reusable Solutions (RS).

Additionally, as a baseline, the Bhat dataset of randomly selected issues, which may also be referred to as 'Random', has been added. It is worth noting that, while Keywords Searches, Static SC Analysis and Maven Dependencies were all used on Cassandra, Hadoop and Tajo, Bhat's issues came from Hadoop and Spark, and seem to only have been classified as either Existence or Non-Architectural.

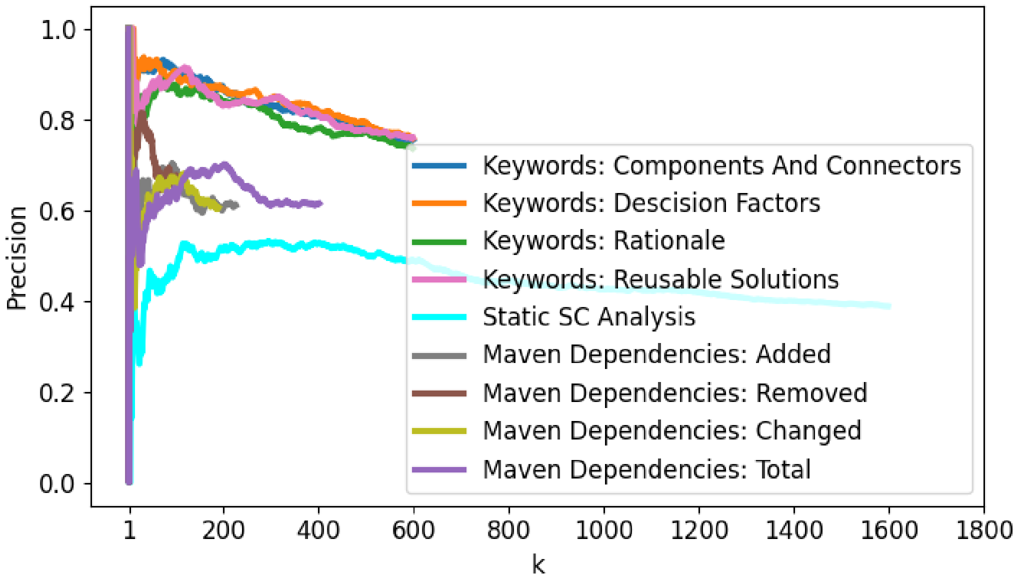
With the Maven method, we have four rankings, based on how a pom file was changed: a dependency could have been added, removed or changed, and the final ranking is the total amount of changes, so all previous three rankings added up.

3.1 Overall Precision

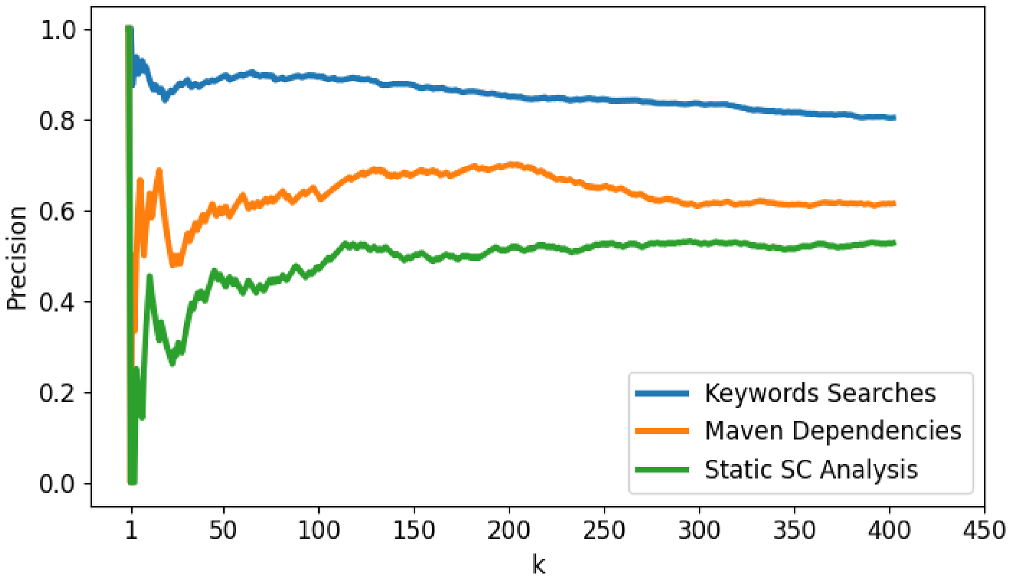
In Figure 1, the total precision for all methods is depicted: the fraction of issues found through the method that did contain architectural knowledge. Maven looks to be on par with Keywords, however does not seem to lose accuracy as k grows.

Figure 2 shows the same graph but with exclusively the Maven methods. There does not seem to be a significant difference between the individual rankings, in figure 2b, but in 2a, the Total ranking seems to do better than the others. If we take into account the fact that Added and Changed had more hits than Removed, we see that where Removed ends at $k = 100$, Added and Created drop a little in precision after that point.

Figure 3 shows a more detailed view of the Keywords Searches queries. We see that whereas they have high precision before $k = 100$, they then start to decline steadily. This may be attributed to

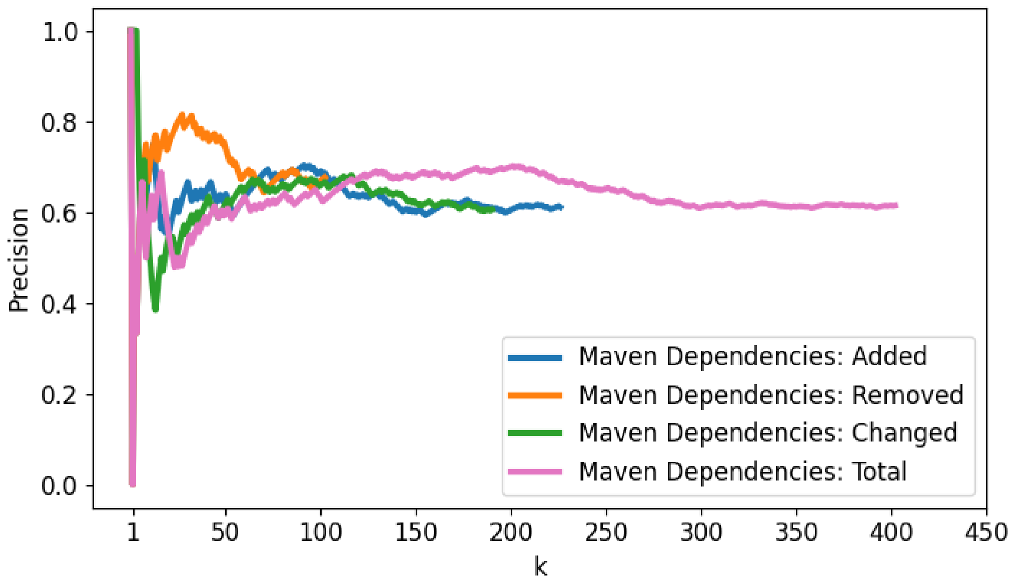


(a) Total precision of all methods and their comprising elements.

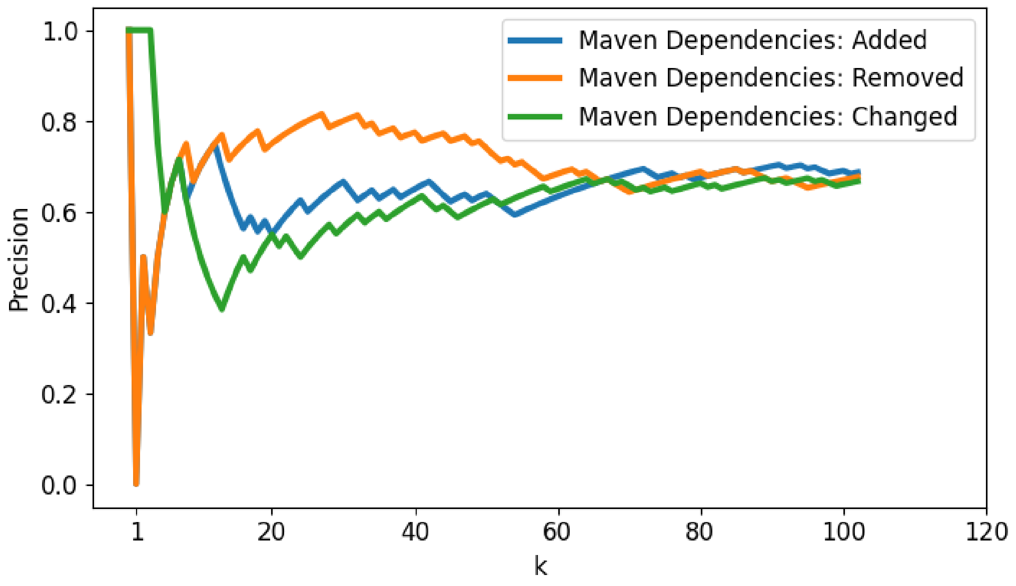


(b) The same information as on Figure 1a, with the Keywords queries averaged, the Maven dependencies represented by the Total ranking, and the graph cut off at 400 to allow us to see more details.

Figure 1: Total precision for all projects and methods.

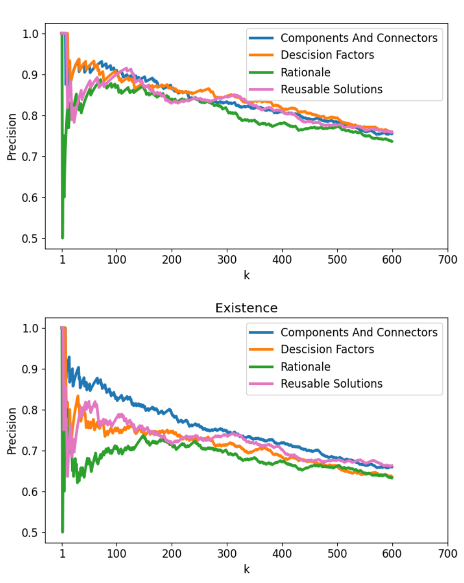


(a) All Maven rankings.



(b) The three Maven rankings Added, Removed and Changed, without Total, allowing the graph to show more detail.

Figure 2: Total precision for all projects, for only the Maven methods. The second graph omits the total, in order to get a more detailed view.



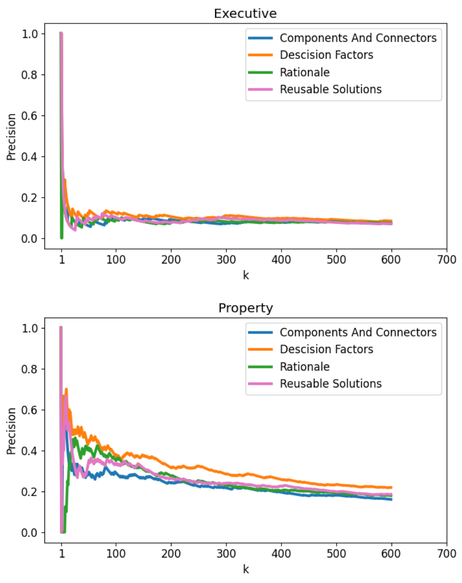


Figure 3: A more detailed precision overview of only the Keywords Searches query results, in general and per decision type.

the algorithm that produced the search results: it presumably shows the most relevant issues first, which appear to be more likely to contain architectural knowledge.

3.2 Type-Specific Precision

In figures 4 and 5, we see the precision for each studied decision type individually. In the first graph, we see that Maven Dependencies is much more efficient at finding executive decisions than the other two methods. It is worse at finding existence decisions than Keywords Searches, but seem mostly on par with Static SC Analysis. The property decision graph looks similar to the existence graph: Maven sits below Keywords, but at the same level as SC Analysis.

Figure 6 shows us the Maven rankings separately again. Interestingly, for executive decisions, the ‘changed’ ranking seems much more efficient than the ‘added’ and ‘removed’ rankings. This situation is reversed for existence decisions: ‘added’ and ‘removed’ stick out above ‘changed’. For property decisions, the Maven rankings do not have a significant difference.

3.3 Decision Type Quantities

The graphs in figures 7-12 represent the amount of decisions found in a project, by each method separately, colored according to decision type. We can immediately see across all projects that the Maven method did indeed yield more executive decisions than the other methods, and comparatively less existence and property decisions.

From these graphs we can also conclude things about the projects and the method of documentation used for each project. The Hadoop projects (Hadoop, HDFS, Map Reduce and Yarn) seem to not have had as many issues picked up by the Maven method as the other methods, which indicates that the developers of Hadoop perhaps did not add issue IDs into their commit messages as frequently as others. They did however seem to use the correct keywords a lot. Tajo is the mirror image: not a lot of keywords search results, but a lot of issues found through Maven. Cassandra seems more balanced.

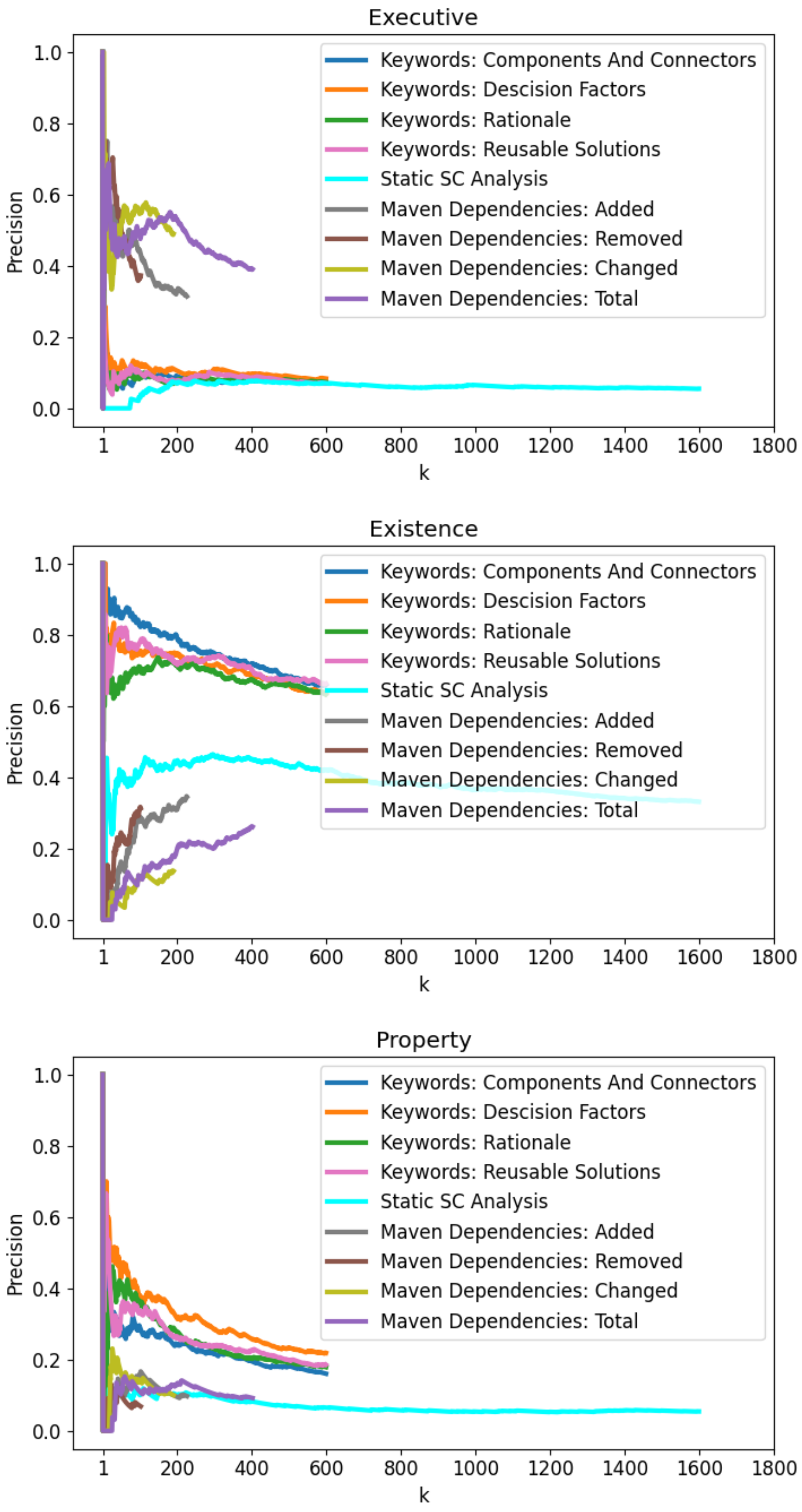


Figure 4: Type-specific precision graphs for executive, existence and property decisions, for all projects and methods, and all comprising elements of the methods.

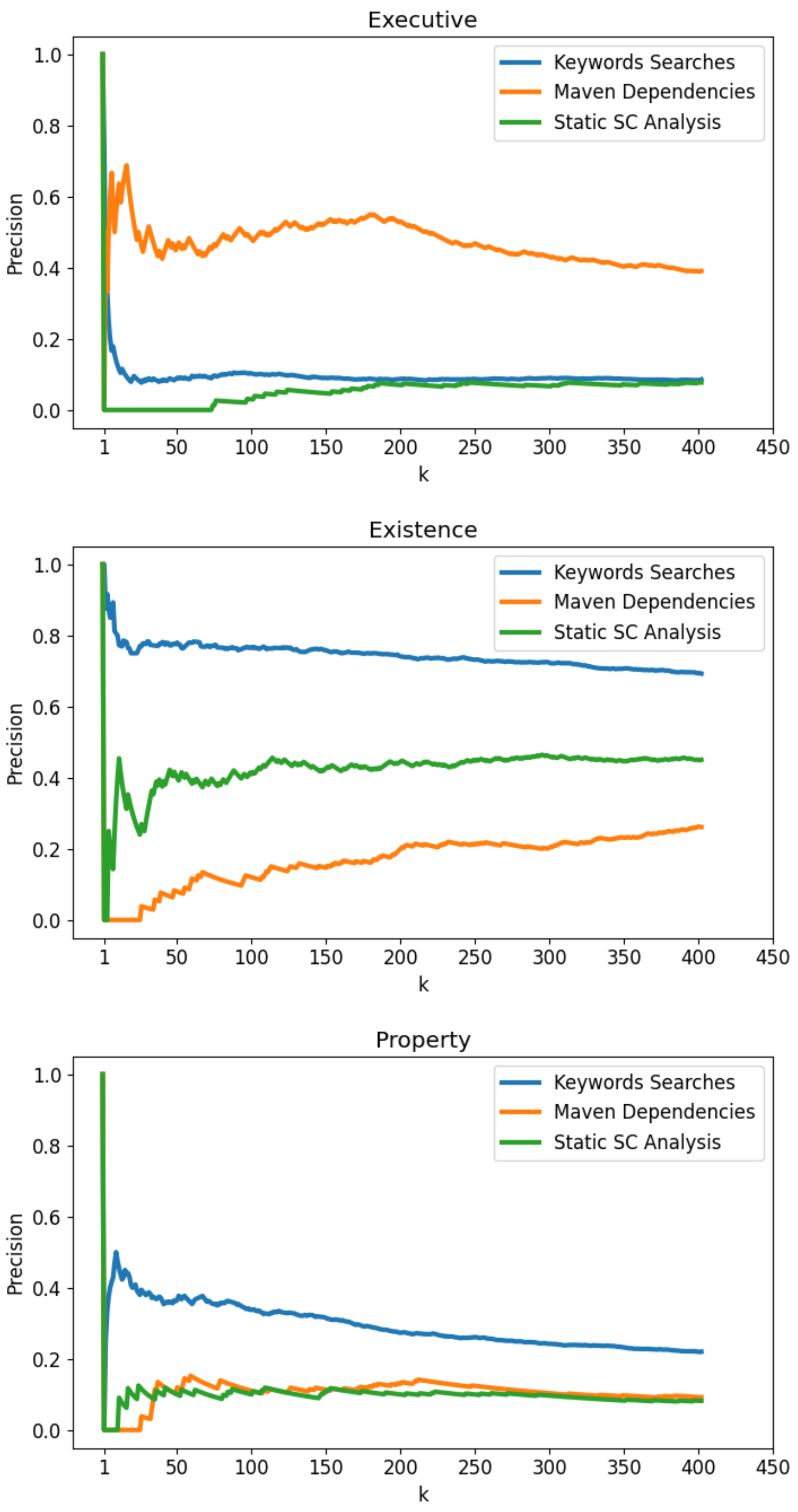


Figure 5: Type-specific precision graphs for executive, existence and property decisions, for all projects and methods, with Keywords Searches averaged and Maven Dependencies represented by the Total ranking, and k limited to 400.

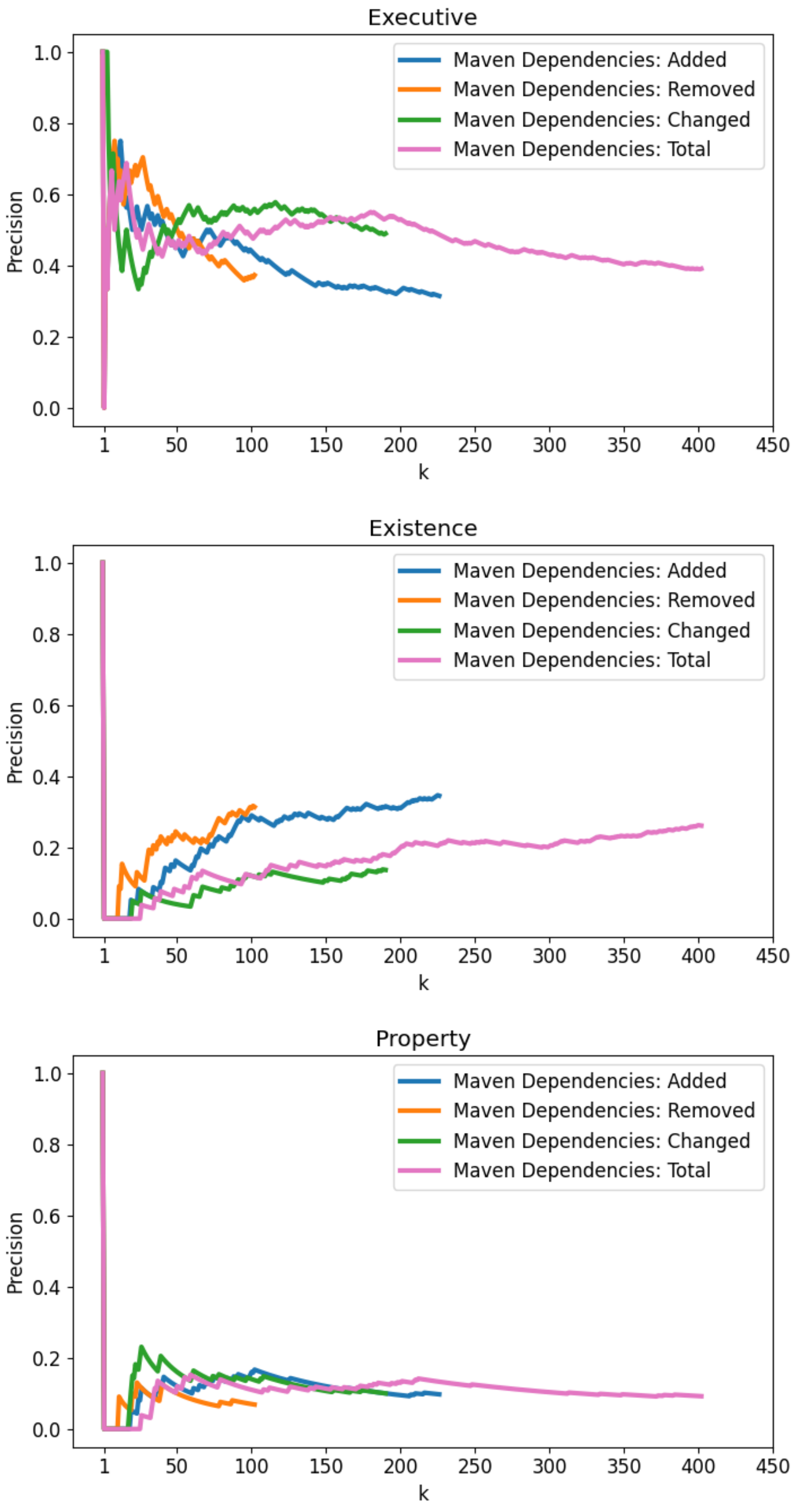


Figure 6: Type-specific precision graphs for executive, existence and property decisions, isolating the four Maven Dependency rankings.

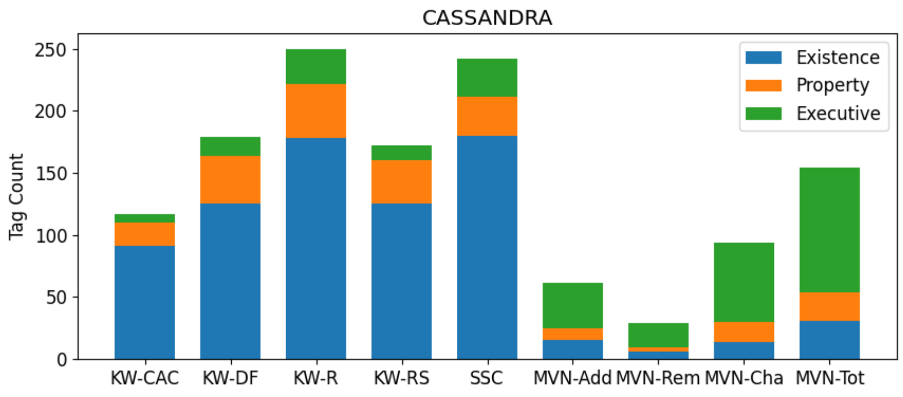


Figure 7: The quantity of decisions with a certain type found in Cassandra by all methods.

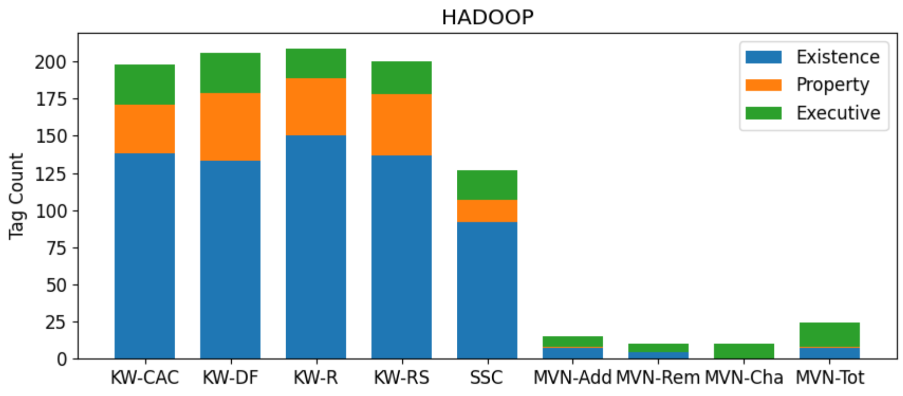


Figure 8: The quantity of decisions with a certain type found in Hadoop by all methods.

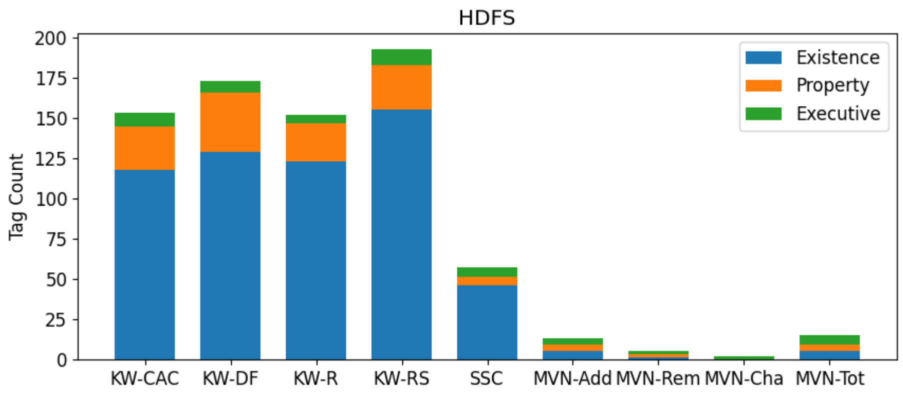


Figure 9: The quantity of decisions with a certain type found in HDFS by all methods.

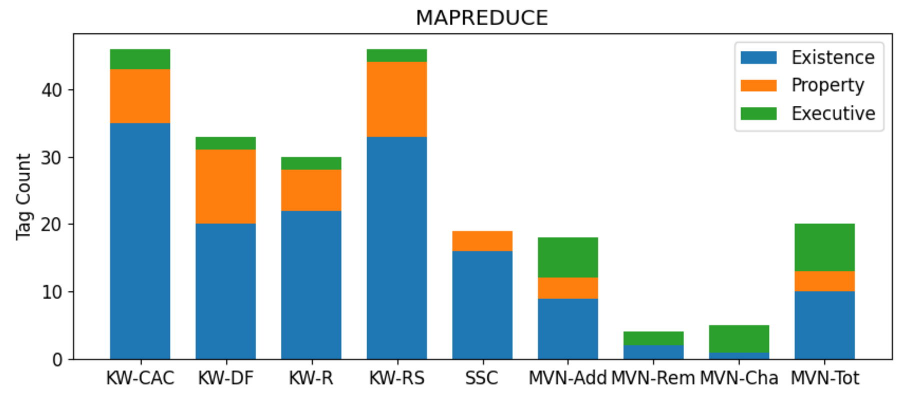


Figure 10: The quantity of decisions with a certain type found in Mapreduce by all methods.

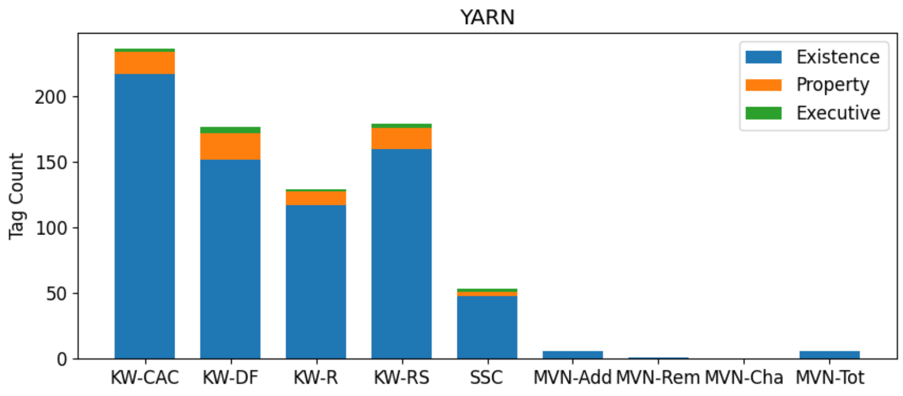


Figure 11: The quantity of decisions with a certain type found in Yarn by all methods.

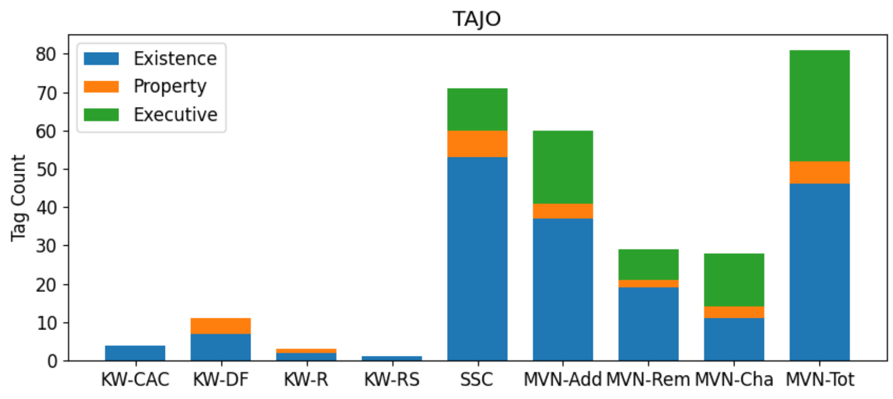


Figure 12: The quantity of decisions with a certain type found in Tajo by all methods.

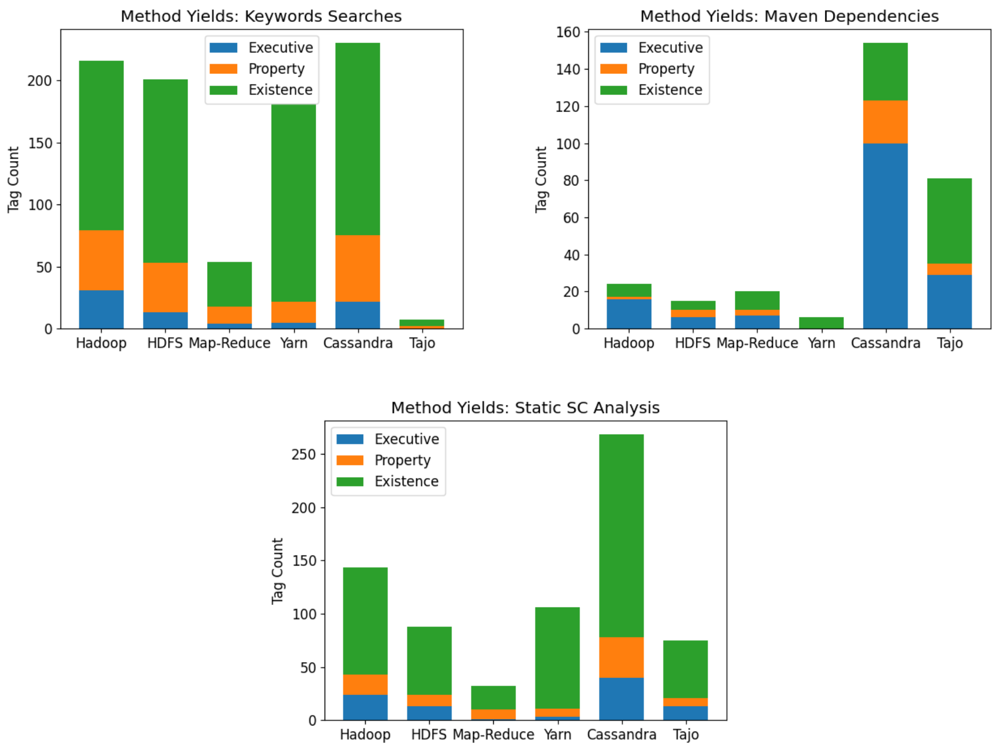


Figure 13: Results yielded by each method, displayed as amount of tags found per project.

We can conclude that the Maven method works best if the developers diligently fill in issue IDs in their commit messages.

3.4 Comparing Methods with Tag Counts

Figure 13 shows a comparison between the three discussed methods, focused on the differences per project. The differences in it, combined with the remainder of results in this report, suggest that different methods are more efficient for different types of decisions, but also for different projects with different documentation practices.

4 Architectural issue characteristics from different approaches

In this section, bar charts and box plots have been generated for eight issue characteristics, from the four approaches already described above: Maven Dependencies, Static SC Analysis, Keywords Searches and Random. This will allow us to understand differences between results yielded from these different approaches, with the random approach as baseline. Additionally, for these characteristics, there are graphs of both each approach separately and the overlaps in results between approaches.

Each method may find issues that do or do not contain architectural knowledge, and each issue found may or may not also be found by other methods. Additionally, keyword searches were limited to 600 issues each, and the SC analysis was limited to 1600 issues. Below is a table with the unique issue counts per category:

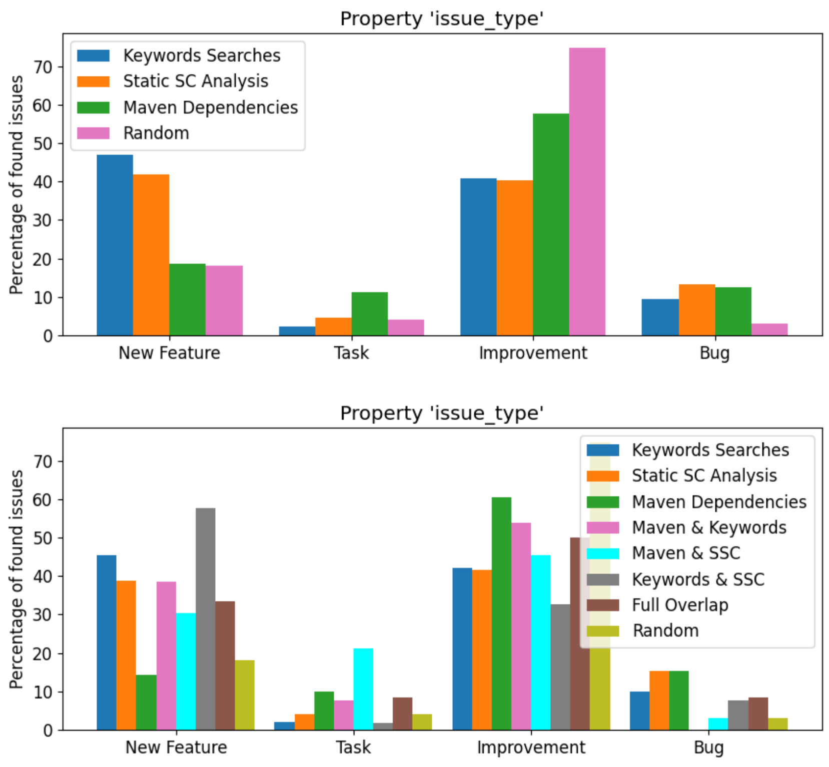


Figure 14: Issue Type per AK discovery method.

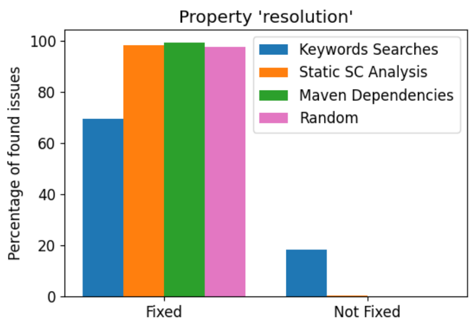
	KW	SSC	MVN	KW+SSC	KW+MVN	SSC+MVN	All
Architectural	604	460	190	116	13	33	12
Non-Architectural	279	927	137	36	2	16	0
Total	883	1387	327	152	15	49	12

4.1 Issue Type

The issue type of an issue in Jira seems to be mostly up to the users' interpretation. As such, it is difficult to draw a conclusion from the consistency of these results.

In Figure 14, we see that Keyword Searches generates the most New Feature-type results out of all four methods, with SC Analysis close behind. This makes sense, since issues labeled as New Features will often contain architectural language, which is something that the Keywords queries will pick up on, and the commits associated with the New Feature are likely to be picked up by SC Analysis. Maven does no better than the random method.

The Random method generates the most Improvement-type issues out of all methods, which can be due to either the fact that the projects examined by Random (Hadoop and Spark) had a larger percentage of Improvement-type issues than the others, or that the custom methods are worse at picking up Improvement-type issues than a random method. Maven is better than the other two custom methods at finding Improvements, however: this is likely due to the fact that changing a dependency, which is Maven's method of discovery, is something that often happens while making an improvement.



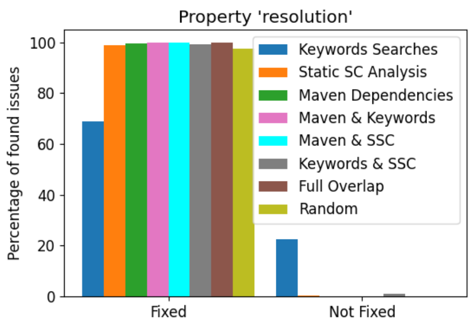


Figure 15: Issue Resolution per AK discovery method.

4.2 Resolution

The resolution of most issues in the chosen projects as depicted in Figure 15 seemed to have been Fixed. Only the Keywords method finds significantly different results: its discovery of issues that were marked as Won't Fix, None or Duplicate is likely due to the fact that SC Analysis and Maven rely on committed code changes, which do not usually exist for issues that have not yet been Fixed.

A note on the labels of the Resolution property: "Fixed" and "Won't Fix" have been merged into "Fixed", while "Duplicate" and "None" have been merged into "Not Fixed".

4.3 Status

For the status of an issue, in Figure 16, we see a similar result to the resolution. Keywords again has several still-Open issues, while Maven and SC Analysis remain focused on Resolved and Closed.

The difference between 'Closed' and 'Resolved', according to the Jira documentation, seems to be mostly up to the users themselves.

4.4 Description Size

Figure 17 shows that Keywords has, on average, the largest descriptions. This is likely because it searches for keywords, and a bigger description has a greater chance of containing relevant keywords. We can also see the inclusion of Keywords skewing up the averages of other issue sets in the chart that takes intersections into account.

As for Maven detecting issues with on average smaller descriptions than Keywords and SC Analysis, this is likely due to the type of issues that it detects. Previous graphs have established that Maven is more likely to detect improvement-type issues, rather than new features. One very prevalent type of improvement issue detected by Maven is a very short ticket, to update a technology to a new version. It is likely that the presence and amount of these types of issues skews the average description size downwards for Maven.

4.5 Average Comment Size

The average comment size, depicted in Figure 18, seems to follow the same logic as the description size.

4.6 Comments Count

The comment count of issues per discovery method, depicted in Figure 19, also follows the same logic as the description size and average comment size.

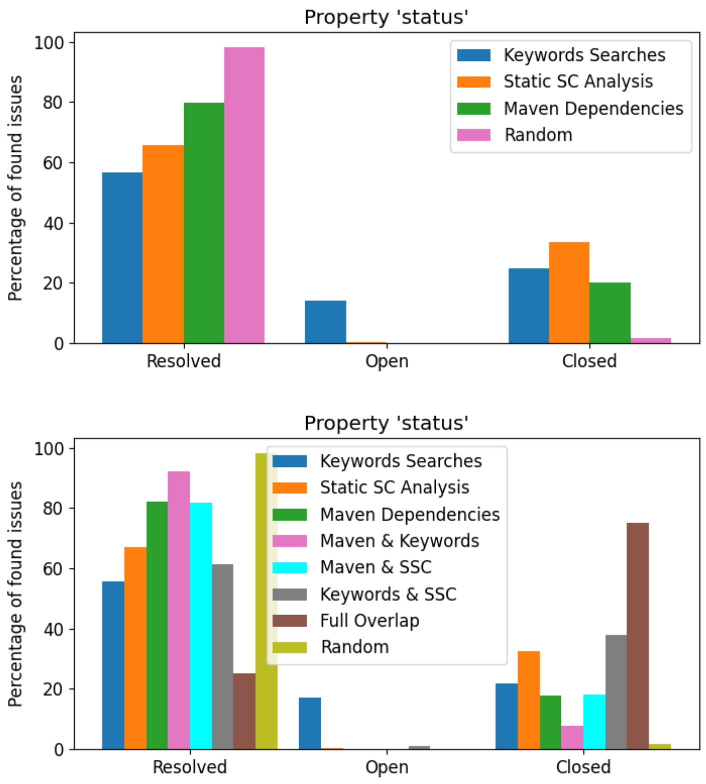


Figure 16: Issue Status per AK discovery method

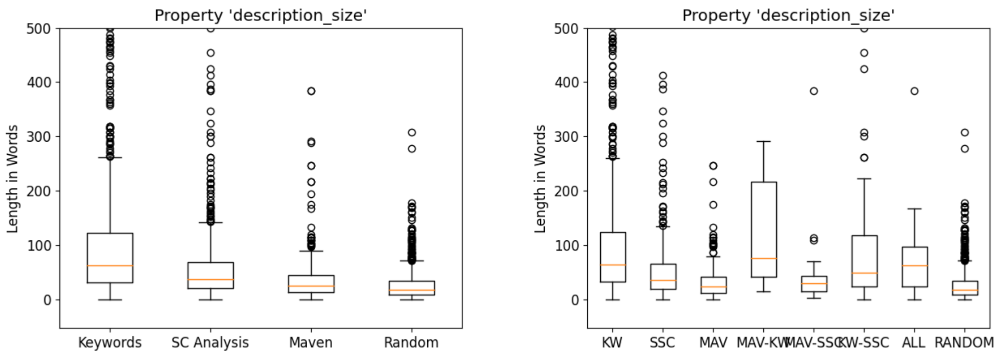


Figure 17: Issue Description Size per AK discovery method

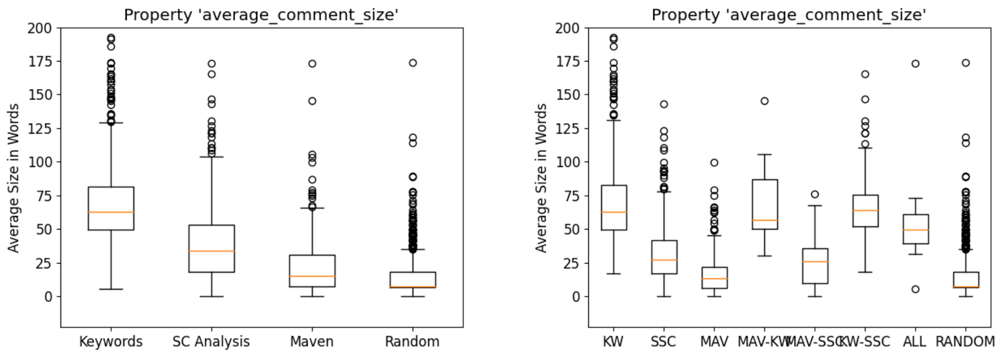


Figure 18: Issue Average Comment Size per AK discovery method

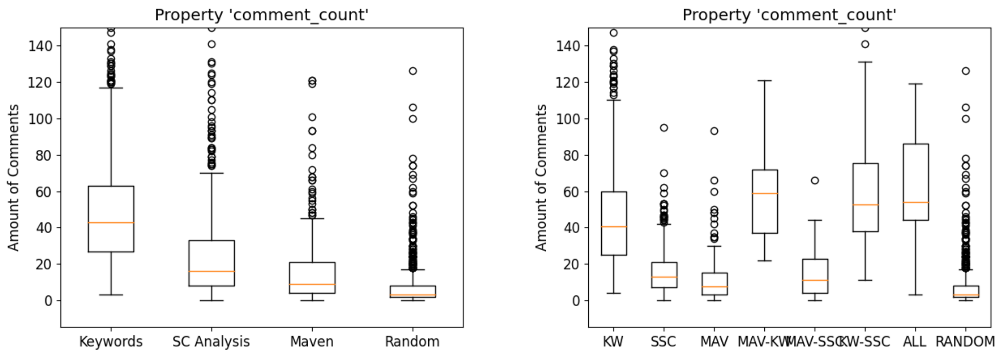


Figure 19: Issue Comment Count per AK discovery method

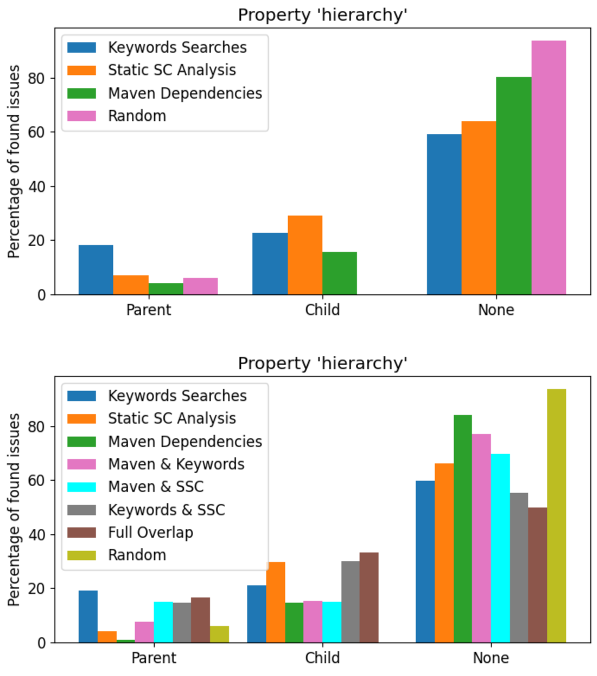


Figure 20: Issue Hierarchy per AK discovery method

4.7 Issue Hierarchy

This characteristic was determined by generating a list of parents by going through all issues and checking each issue for the presence of either sub-issues or a parent task ID. The result is Figure 20.

This bar chart illustrates Keywords having found the most Parent issues, which is likely because parent issues are more likely to contain architectural knowledge and language, rather than their children. On the other hand, SC Analysis found a lot of child issues, which is likely because it is in these child issues that the code gets implemented.

Contrasting Keywords and SC Analysis, Maven finds about as many Child issues as Keywords and barely any Parent issues. It seems to find more non-hierarchical issues than either Keywords or SC Analysis. As with Description Size, this is likely due to the fact that Maven detects dependency changes, which are mostly either child tasks, as with SC Analysis, or small improvement tasks that are not connected to anything.

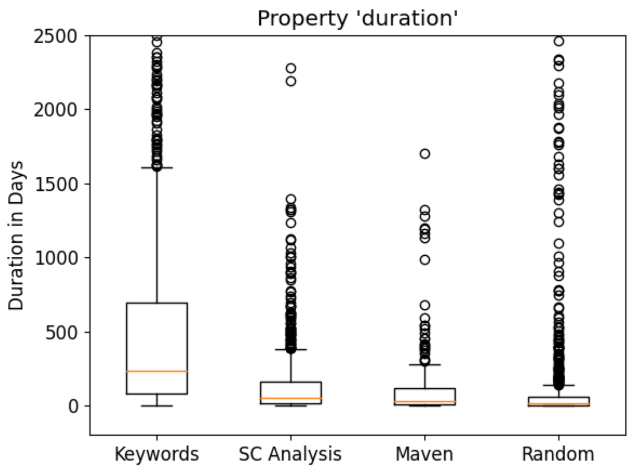
4.8 Issue Duration

Duration means the amount of time in days that passed between an issue's creation date and it's resolution date, or if it is not yet resolved, today's date. This statistic is described in Figure 21.

In this figure, we see that Keywords leads by a large margin. This may be due to the nature of this characteristic, and the fact that Keywords may also find non-finished issues, as opposed to the other methods.

5 Comparing architectural and non-architectural issues

In this section, I compare all issues, now sorted into containing architectural knowledge or not, on the same eight characteristics.



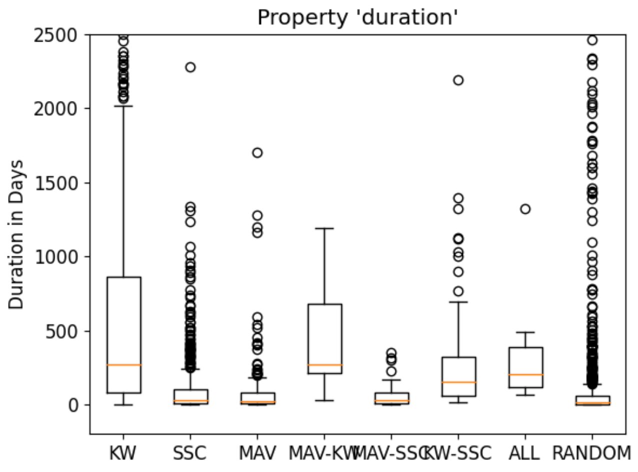


Figure 21: Issue Duration (from creation until resolution or now, in days) per AK discovery method

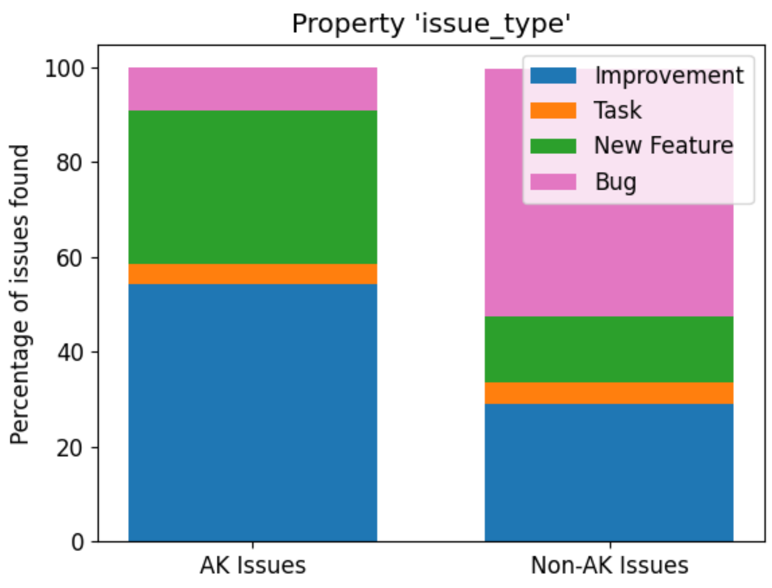


Figure 22: Comparison of Issue Type between AK and non-AK Issues

5.1 Issue Type

In Figure 22, issues with AK are more often of type Improvement than non-AK issues. Most non-AK issues seem to be bugs. AK issues also contain a higher portion of New Features than non-AK issues.

5.2 Resolution

Figure 23 shows this characteristic. Mostly, issues are Fixed. There is a small majority of None issues on the AK side, which might be the influx of issues from Keywords, which doesn't detect if a decision has been implemented yet or not.

5.3 Status

In Figure 24, which describes the Status characteristic, AK and non-AK issues are mostly the same, except as we saw in Resolution, a few more are still Open on the AK side.

5.4 Description Size

In Figure 25, at the scale of the graph, AK and non-AK issues seem to have similarly large description-sizes, though the non-AK issues seem to have more outliers.

5.5 Average Comment Size

Figure 26, this graph is similar to Description Size, though AK issues seem to have on average slightly larger comments than non-AK issues.

5.6 Comments Count

Figure 27 shows us that AK issues have on average a slightly higher comment count than non-AK issues.

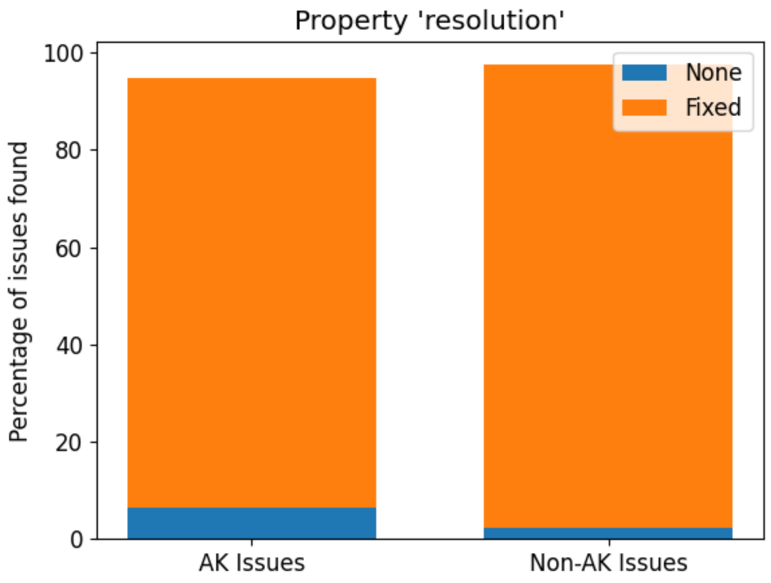


Figure 23: Comparison of Resolution between AK and non-AK Issues

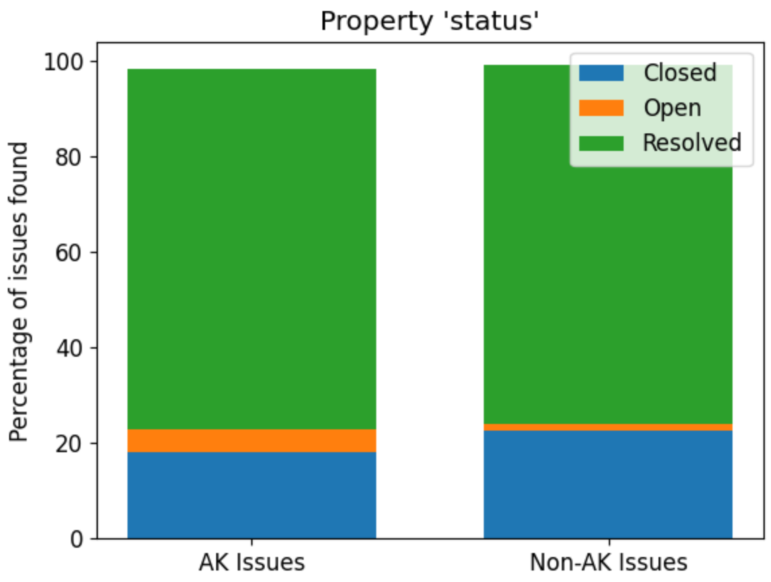


Figure 24: Comparison of Status between AK and non-AK Issues

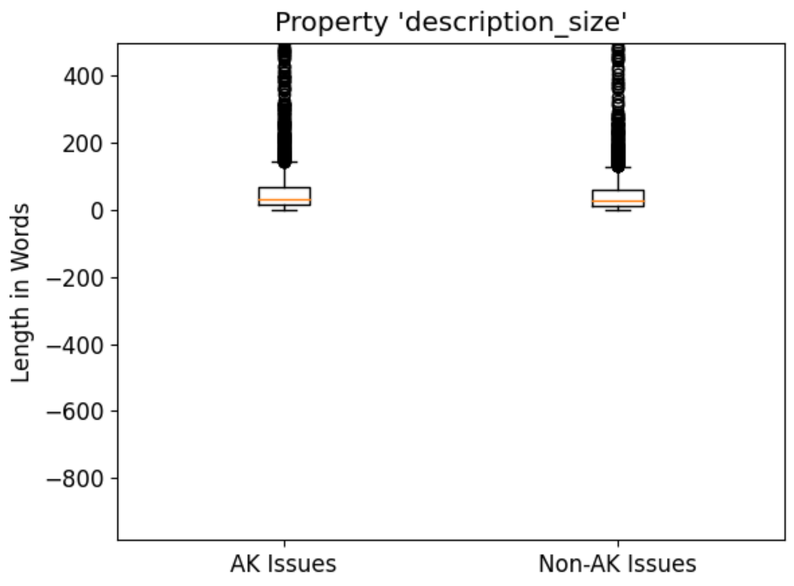


Figure 25: Comparison of Description Size between AK and non-AK Issues

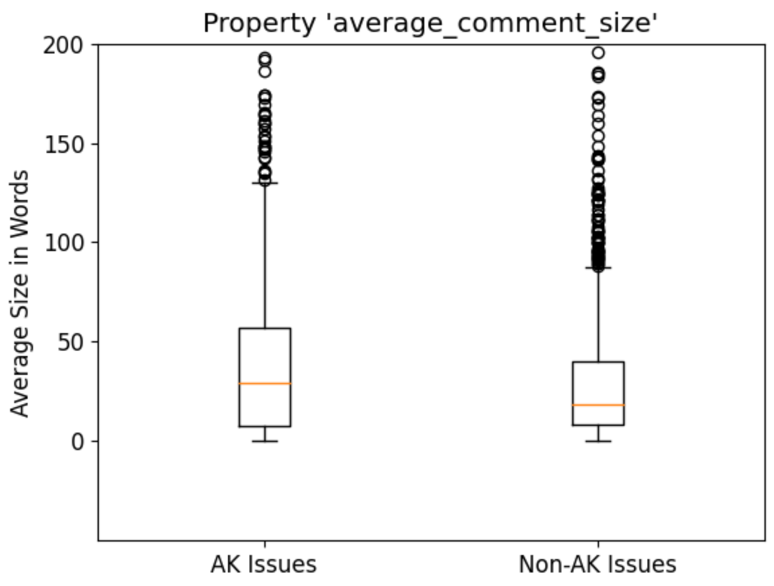


Figure 26: Comparison of Average Comment Size between AK and non-AK Issues

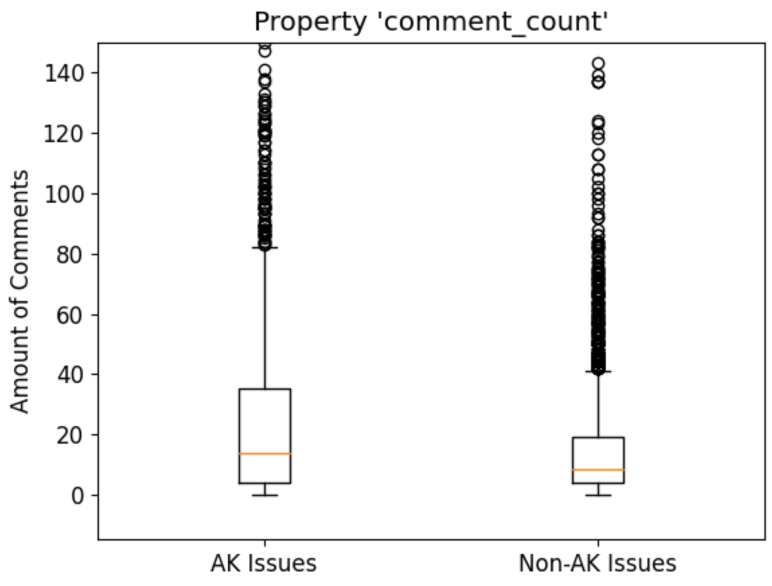


Figure 27: Comparison of Comments Count between AK and non-AK Issues

5.7 Issue Hierarchy

In Figure 28, AK issues are more often parent issues than non-AK issues.

5.8 Issue Duration

In Figure 29, AK issues seem to take just a little bit longer than non-AK issues.

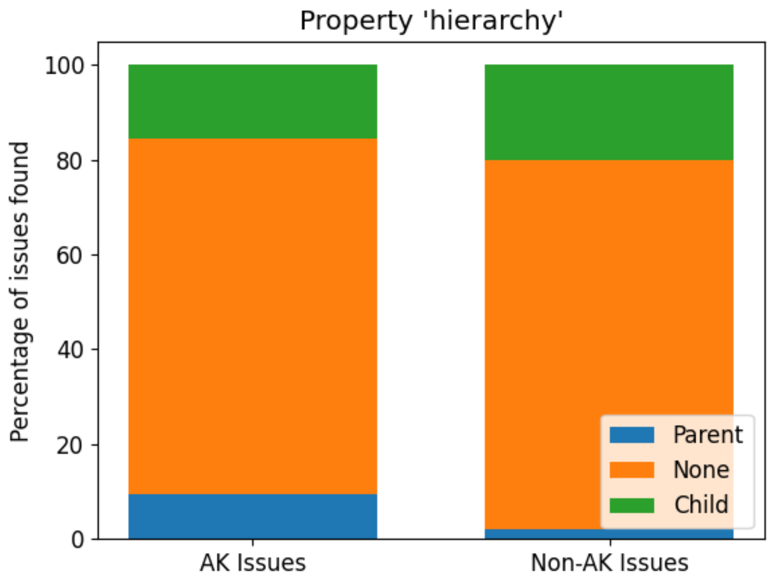


Figure 28: Comparison of Issue Hierarchy between AK and non-AK Issues

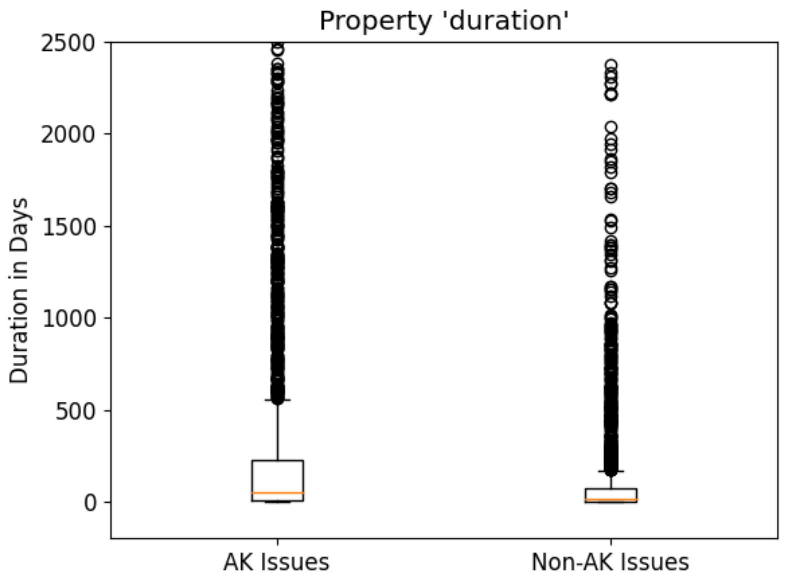


Figure 29: Comparison of Issue Duration between AK and non-AK Issues

References

[1] Kruchten, P., Lago, P., Vliet, H.: Building Up and Reasoning About Architectural Knowledge. In: Hofmeister, C., Crnkovic, I., Reussner, R. (eds.) Quality of Software Architectures, Lecture Notes in Computer Science, vol. 4214, pp. 43–58. Springer Berlin Heidelberg (2006)

[2] Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: WICSA. pp. 109–120 (2005)

[3] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 3rded. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2012.

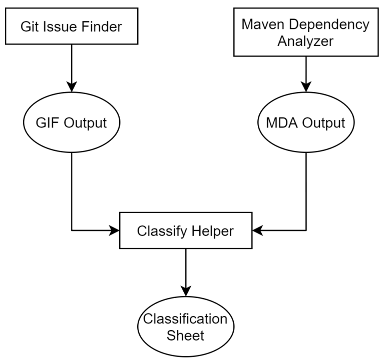


Figure 30: A graph of the structure of the detection method programs.

Appendices

A Programs Structure

In order to generate, classify, organize and develop this statistical data, multiple existing systems were used and in some cases modified or extended. This appendix serves as an explanation of these systems and how they work together.

The actual collection of programs as described here is in a GitHub repository with relevant readmes in most directories.

A.1 Detection Method: Maven

The Maven method works through finding edits to the pom file in a repository, and then connecting those commits to relevant Jira issues.

In order to achieve this goal, there are three programs. The first, titled Maven Dependency Analyzer, generates a list of all commits with their relevant pom file changes, counted as adding, removing, updating, and total changes. The second is the GitIssueFinder, which goes through all commits in a repository and looks through the commit message to find a Jira issue key. Lastly, there is a program called ClassifyHelper which is a commandline tool that, firstly, combines the outputs of the first two programs into one that can be used to classify issues manually, and next, contains functionality to help with the more tedious parts of classification.

The Git Issue Finder and the Maven Dependency Analyzer were created in Java. The ClassifyHelper was created in Python.

A.2 Search Engine

With the discovered issues classified, we can head to the search engine, upload them, and then search the results for similar issues so we can find a solution to a new problem, or do any other type of research.

The backend is a Java server, which accesses the Jira API and, through an Apache Lucene index, the issue database. The frontend, a Vue.js application, communicates with the backend through HTTP queries. The frontend can be used to search issues, upload classification sheets, and export search query results as JSON. The actual projects themselves contain usage documentation.

For now, this search engine is exclusively run locally.

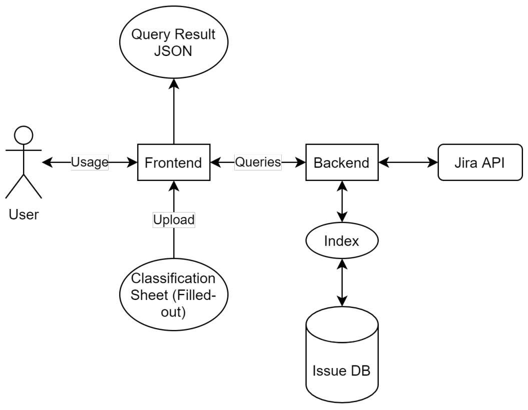


Figure 31: A graph of the structure of the search engine programs.

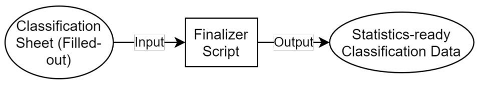

Classification Sheet (Filled-out)
Input
Finalizer Script
Output
Statistics-ready Classification Data

Figure 32: A graph of the structure of the helper script.

A.3 Helpers

The finalizer helper script is small: it takes the filled-out classification sheet produced by the ClassifyHelper and puts out the same data but in a format that is more straightforward to integrate into the statistics script.

A.4 Statistics

There are several statistics scripts, created in Python. The input of this collection of programs is all the classification data it is currently configured to accept, as can be seen in Figure 30: Top-Down, Bottom-Up, Maven and Bhat data. This data is fed into the first script, analysis, to generate data that can then be used by all the other scripts to generate various graphs. All of these graphs can be found back in this report, in their relevant sections.

There is another script, script 0, which ensures you have the required NLTK package installed. Additionally, there is a python library included with the scripts that contains common functionality.

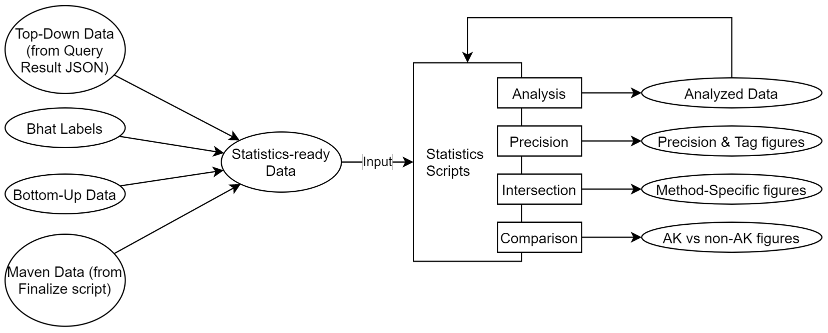


Figure 33: A graph of the structure of the statistics scripts.