Mining architectural knowledge in issue tracking systems

university of groningen / faculty of science and engineering

Said Faroghi

Supervisors: Dr. Mohamed Soliman, Prof. Paris Avgeriou

03 02 2022

Abstract

Recording and accessing architectural knowledge (AK) is not a trivial task. A promising source of AK could reside in issue tracking systems, which are platforms for developers to coordinate building software, and therefore it is a hotspot for software-related discussions. We have evaluated two tools that specialize in extracting AK issues, by annotating the generated issues with architectural design decisions (ADD). We developed a coding book to help the annotation process. Furthermore, we analyzed the issues to see how their AK characteristics and other properties differ based on the tools they came from.

Contents

1 Introduction **4**
- 1.0.1 Proposed Solution . . . 4
- 1.0.2 Thesis Structure . . . 5

2 Background **6**
- 2.1 Software Architecture and its Knowledge . . . 6
- 2.2 Issues and Issue Tracking Software . . . 6
- 2.3 Effectiveness Metrics . . . 7
  - 2.3.1 Precision and Recall . . . 7
  - 2.3.2 NDCG . . . 7
  - 2.3.3 Issue Properties . . . 8
  - 2.3.4 Architectural Design Decisions (ADD) . . . 9
- 2.4 AK-Mining tools . . . 9

3 Related Work **10**
- 3.1 Representation of Architectural Design Decisions . . . 10
- 3.2 Retrospective Recovery of Architectural Design Decisions . . . 11
  - 3.2.1 Using issue tracking systems . . . 11

4 Methodology **12**
- 4.1 Issue collection strategy . . . 12
  - 4.1.1 Top-Down Issue Collection . . . 12
  - 4.1.2 Bottom-Up Issue Collection . . . 13
- 4.2 Annotation . . . 13
  - 4.2.1 Iterative annotation process . . . 14
  - 4.2.2 Number of issues . . . 14
  - 4.2.3 Differences in the labeling process between the top-down and the bottom-up approaches . . . 14
- 4.3 Measuring the effectiveness . . . 15
  - 4.3.1 Precision . . . 15
  - 4.3.2 NDCG . . . 15
  - 4.3.3 Missed Parents . . . 15
- 4.4 ADD type distributions . . . 15
- 4.5 Issue Property Distributions . . . 16

5 Results **17**
- 5.1 RQ1: Bottom up And Top Down Effectiveness . . . 17
  - 5.1.1 Precision Results . . . 17
  - 5.1.2 NDCG Results . . . 19

5.1.3 Missed Parents Results . . . 21
5.2 RQ2: ADD Type distributions . . . 22
5.3 Issue Property Analysis Results . . . 25
5.3.1 RQ2: AK-Issues Only . . . 25
5.3.2 RQ3: AK-Issues vs. Non-AK-Issues Results . . . 28

6 Discussion **30**
6.1 How effective are top-down and bottom-up approaches in finding architectural issues in issue tracking systems? . . . 30
6.2 What are the differences in AK characteristics and issue properties between the top-down and bottom-up approaches? . . . 30
6.3 What are the differences in issue properties between AK issues and non-AK issues? . . . 31
6.4 Threats to Validity . . . 31

7 Conclusion **32**
7.0.1 Future Work . . . 32

A Code Book **36**
A.1 General . . . 36
A.1.1 Recurring architectural changes . . . 36
A.2 Existence Decisions (Ontocrisis) . . . 36
A.2.1 Design Specifications . . . 36
A.2.2 New Architectural Component(s) . . . 37
A.2.3 Configuration descriptions . . . 37
A.2.4 Bans or Non-existence . . . 37
A.3 Property Decisions (Diacrises) . . . 37
A.4 Executive Decisions (Pericrises) . . . 38

B Search Queries **39**
B.1 Decision Factors . . . 39
B.2 Reusable Solutions . . . 39
B.3 Components and Connectors . . . 40
B.4 Rationale . . . 40

Chapter 1

Introduction

A software's architecture concerns elements and behavior at the highest level, providing the foundations upon which the software is implemented. Architectural design decisions (ADD) usually happen early in a project's life cycle, so bad decisions here are costly. It takes a hefty amount of expertise to design and maintain a solution as a software architect, especially since different business goals suggest using alternative design decisions. It usually boils down to the developer's experience and all the tacit knowledge they have acquired during their career to make these important decisions. [1]

Unfortunately, pulling that knowledge out of the developer's heads and into some form of systematic documentation is rarely done. As a result, both the knowledge of successful design decisions and failed ones will eventually be lost to time, and other developers can repeat mistakes. There have been attempts to capture architectural knowledge (AK), usually in the form of design decisions, to facilitate the reuse of the knowledge in future solutions. AK management tools [2, 3, 4] can help re-usability by providing features to organize and share AK. Work has also been done in trying to recover AK from project artifacts like source code changes, issues, commit messages, mailing lists, as well as online software communities [5, 6, 7, 8].

The goal of this thesis is to provide valuable data regarding the collection of AK from existing projects, in the form of: architectural issue data-sets, information on the viability of new mining tools, and the documentation of how AK is represented in issues.

1.0.1 Proposed Solution

First, we will evaluate the effectiveness of two tools that specialize in collecting architectural issues from the Apache JIRA issue tracking system. The first tool uses the top-down approach, which means it uses a search engine to find issues related to the search terms. The second tool uses the bottom-up approach, which analyzes source code changes and eventually links them to the issues that the changes are related to. We annotate the resulting issues with the types of ADDs they contain.

Since we now have a dataset of annotated architectural issues, we check the distributions of ADD types among these issues and compare them across the different approaches. Additionally, various properties of the issues will be compared as well.

Furthermore, we can also compare issue properties among architectural issues and non-architectural issues.

This research will answer the following research questions:

1. How effective are top-down and bottom-up approaches in finding architectural issues in issue tracking systems?

2. What are the differences in architectural issue characteristics and issue properties between the top-down and bottom-up approaches?

3. What are the differences in issue properties between architectural issues and non-architectural issues?

1.0.2 Thesis Structure

In Chapter 2, the relevant background is defined: Software architecture, architectural knowledge, issue tracking software. In Chapter 3, relevant literature is explored concerning the topic at hand: Existing ways to document/recover architectural knowledge and complementary work to this paper. Chapter 4 reviews the research process followed in other to arrive at our results. Chapter 5 displays the results themselves with short explanations describing the data. Chapter 6 interprets these results and discusses threats to validity. Chapter 7 concludes the paper and explores future work.

Additionally, Appendix A provides the coding book developed and used to annotate issues, and Appendix B lists the queries used for the search engine in the top-down approach.

Chapter 2

Background

2.1 Software Architecture and its Knowledge

Software Architecture (SA) refers to the structure/organization of software components and their behavior, which together create a solution meeting a set of business requirements [9]. Architectural decisions are typically made early into a project's life cycle. They involve technology decisions (Should we use Java EE or ASP.NET?) and high-level structural decisions; however, they are challenging to change retrospectively. The architecture is closely tied to the requirements it is trying to fulfill, therefore making the correct choices at this level is necessary for a successful outcome.

Many definitions of **Architectural Knowledge** (AK) have been stipulated in literature [10]. The same is true for SA; however, the intuition behind it is helped by drawing comparisons to architecture in other domains such as network or building [11]. For this study, we can conform to the following definition: AK is the architecture as well as the design decisions that lead to it [3]. It is a focused definition, where its scope is related to how rigorously design decisions are captured and documented. Design decisions reveal the "why?" of various parts of the architecture, and documenting them can help build architectural knowledge management tools [12] to support the architectural design process.

2.2 Issues and Issue Tracking Software

An **issue** in the context of software development is a request for improvement to the current system, which could be adding a new feature, resolving a bug, among others. **Issue tracking systems** are software solutions aimed at creating and managing these issues in order to collaborate on projects efficiently and keep track of the history of changes to issues. Issues can be tagged with priority, completion status, development time estimations, and many other issue tracker specific information that help organize and describe issues.

An important feature that many issue tracking systems offer is commenting and starting threads on specific issues, which attracts discussions among developers and even stakeholders, therefore being a potentially valuable resource for information mining.

2.3 Effectiveness Metrics

In order to compare the performance of the top-down and bottom-up approaches, a couple of promising metrics have been selected. In this section, these metrics are explained in order to gain an understanding of how they will be used in this context, and to make sense of the values outputted.

2.3.1 Precision and Recall

Precision is a metric that calculates the percentage of relevant instances returned by some model. It is a useful metric to determine the quality of the results returned. Within the results, there exist *true positives* (TF), and *(*false positives) (FP). True positives are the relevant instances the model correctly assigns, and false positives are instances where the model assigns as relevant but is not.

$$Precision = \frac{TP}{TP + FP}$$

Recall is a metric that tells us what fraction of the total relevant instances are returned.

$$Recall = \frac{TP}{TP + FN}$$

where FN = *false negatives*, which are relevant values the model did not label as relevant. $TP + FN$ contain all the relevant values of the dataset.

2.3.2 NDCG

A metric designed to grade the quality of a search result is the **Normalized Discounted Cumulative Gain**. To understand what it does, we can analyze how the **Discounted Cumulative Gain** works and how it leads to the NDCG:

$$DCG(Q) = \sum_{i=1}^{p} \frac{rel(Q_i)}{\log_2(i+1)}$$

Where p is the rank position being evaluated, Q is the list of results, Q_i is the result at position i, and $rel(Q_i)$ is the result's relevance.
An alternative formulation for DCG (let us call it DCGa) exists that emphasizes results appearing higher up in the list:

$$DCGa(Q) = \sum_{i=1}^{p} \frac{2^{rel(Q_i)} - 1}{\log_2(i+1)}$$

Because the $\log_2(i+1)$ term smoothly gets more significant at each successive position, it penalizes results appearing lower in the result list. But how do we tell what a good DCG value is from a bad one?

One approach is to compare it to the results of the ideal query result. The ideal query result I is the result set ordered by relevance. If we run the DCG on this list, we get the Ideal DCG. Dividing the DCG by the ideal DCG will give us a value between 0 and 1, with 1 being the best result possible. Hence we get the NDCG:

$$NDCG = \frac{DCG(R)}{DCG(I)}$$

Where R is the original result list, and I is the ideal result list.

2.3.3 Issue Properties

Issues contain a description by the author and a comment thread associated with it. However, there are many more properties associated with issues. Properties may be fields themselves or a derived value from the fields. Here is the breakdown of how different pieces of information from the issues are summarized and compared across different sets of issues.

- **Status** - The status of an issue depicts the current state of the task, whether it is done or in progress, et cetera. The number of occurrences of each status type will be recorded.

- **Resolution** - This field depicts the specific outcome of an issue that has been closed. An example resolution is 'Fixed'. The number of occurrences of each resolution type will be recorded.

- **Type** - Examples include Feature or Bug. Sometimes the type is a sub-task. It does not tell us much about the actual type, but because a sub-task is a child of a parent issue, we instead grab the type from the parent issue. The number of occurrences of each issues type will be recorded. If the issue is a sub-task, we also record this fact to avoid losing information about which issues are sub-tasks.

- **Description Size** - One way to extract some meaning out of this field is to measure its size - the bigger it is, the more content it has. A good way to measure the 'size' is to involve some basic NLP - we count the number of words after filtering the text of stop words since stop words are meaningless when it comes to AK.

- **Comments** - Two meaningful pieces of information can be extracted from the comment section. First, we can add up the number of comments in total for each issue. Second, we can derive the average size of the comment section. The size of each comment is calculated similarly to the size of the description field and then averaged. Additionally, bots' comments are filtered out not to obfuscate the results.

- **Attachments** - Most issues come with attachments, which are just files. Any type of files can be attached, but the most common file types are .patch, .txt, .doc, .pdf. Patch files are just code changes, and some txt files are also code changes, so we do not look at these. However, doc and pdf files may indicate a larger, more involved issue. We do not analyze the contents of the files, but just the number of them.

2.3.4 Architectural Design Decisions (ADD)

Presented in the paper Krutchten et al. [3] is the concept of ADDs - a way to structure the rationale aspect of architectural knowledge. Important to this thesis are 3 types of decisions:

- **Existence** - An element/artifact will exist in the system's design or implementation. This entails the structure (modules, components, layers, et cetera) and behaviour (the interaction between different structural entities). Also included in this definition is the opposite: An element/artifact will *not* exist in the system's design or implementation.
- **Property** - A property decision states an enduring, overarching trait or quality of the system. Perhaps a suitable way of representing a property decision is when it is related to one or more quality attributes [9] like performance, scalability, et cetera.
- **Executive** - Decisions related to the environment the project exists in, which affects its development process - such as business requirements and the choice of tools and technologies to use.

2.4 AK-Mining tools

Top-down approach: The **Archedetector** tool [13] is an optimized search engine (powered by Lucene) over Jira issues and mailing lists, responsible for extracting architecturally relevant data employing specific queries. It also contains faculties that help navigate issues and annotate them.

Bottom-up approach: The source code analyzer tool [7] analyzes the commits made to specific Jira projects and then assigns them a so-called **A2A** metric - a value that is calculated by looking at the code changes in the commit and determining to what degree these changes may be architectural. The commits are then linked to the issue that the commit is a part of if the issue exists (some commits are not related to specific issues). As a result, it is hoped that the issue has a higher chance of containing AK. Positive A2A values indicate architectural relevance, and negative values do not.

Chapter 3

Related Work

3.1 Representation of Architectural Design Decisions

The problem that can arise when creating architectures is that the design decisions used to drive the implementation are typically lost inside the architecture or otherwise poorly documented. Jansen et al. [2] has tackled this problem by treating software architecture as a set of design decisions. Knowledge vaporization (the loss of AK over time) is reduced by representing design decisions as first-class entities (objects containing explicit data such as problem statements, motivation, et cetera, and a set of potential solutions), knowledge vaporization (the loss of AK over time) is therefore reduced. In Jansen et al., the use of such a design decision model is exemplified by integrating it into an architectural meta-model that unites the architecture model and the design decision model in order to express changes in functionality.

How may collecting a set of design decisions help create a so-called architectural knowledge management tool? Kruchten et al. [14] exemplifies such an approach by stating the attributes which are essential to a design decision (Epitome, Rationale, Scope, Author, Time-stamp, History, State, Category(s)) and type of relationships between design decisions. Furthermore, a list of actors (what kind of users will benefit from AK) and use cases (to what purposes the AK will be used) is specified. Finally, a visualization tool is presented, where the design decisions are topographically displayed in a cluster map based on the relationships between the decision types and their statuses.

Zimmerman et al. [4] concluded that the design decision models presented in Jansen et al. [2], Krutchten et al. [14], and other similar approaches had vagueness issues and were not formalized enough. They then formalized the architectural decision model in the mathematical paradigm - in terms of set theory, relations, integrity constraints, production rules, and graph theory.

3.2 Retrospective Recovery of Architectural Design Decisions

Applying the processes outlined in the section above during the project's life-cycle is an effective AK documentation tool that facilitates reuse. What about trying to mine architectural resources based on existing data about the project (version-control commits, issues, existing existing existing documentation, et cetera)? Outlined next are some attempts of this in the relevant literature.

A study by Shahbazian et al. [5] developed a tool to try and recover design decisions using a project's version control commits and mapping them to issues, alongside recovering entire architectures on different versions/releases and analyzing the changes between them. It is a lossy analysis (not all relevant decisions are captured), but the tool still averaged a recall of 75% and precision of 77%, which is (put colloquially) *not bad!* This tool could also be applied during the life-cycle of the project in iterative steps, providing a helpful source of auto-documentation.

3.2.1 Using issue tracking systems

Because issues may contain a relatively large amount of knowledge [15], the focus of many studies has been spent analyzing issue tracking systems. For example, Bhat et al. [6] has approached the situation via machine learning. A supervised classification model has been trained on 1500 correctly labeled issues and used to classify novel issues, with good results.

A collection of recent student papers have explored architectural knowledge in the Jira issue tracking system. In A. Fydorov [16], issues were linked together by the relevance of their discussions, and explored ways of pooling the knowledge from different sources into one place, providing a useful way to extract architectural knowledge from. Additionally, a software tool was developed to link the issues inside a project. In A.Dekker [17], another tool was developed which fetches commits that contain changes in a project's dependencies, and links them to issues, with the goal of documenting the architectural knowledge concepts present in the issues. In T. Boon [13], the different types of architectural design decisions was explored in both Jira and mailing lists, by developing a searching tool that helps find and label these resources with the appropriate decision type (further explained in the background).

An exploratory study by Soliman et al. [7] defines AK concepts and documents their appearance in Jira issues. A coding book was developed, which contains guidelines on identifying AK concepts in the text of the issues. A source code analyzer was made as an effort to identify relevant architectural changes and link them to issues (further explained in the background). The study focuses on examining the representation of AK concepts, the frequency of appearance of each AK concept, and which AK concepts tend to occur together.

Chapter 4

Methodology

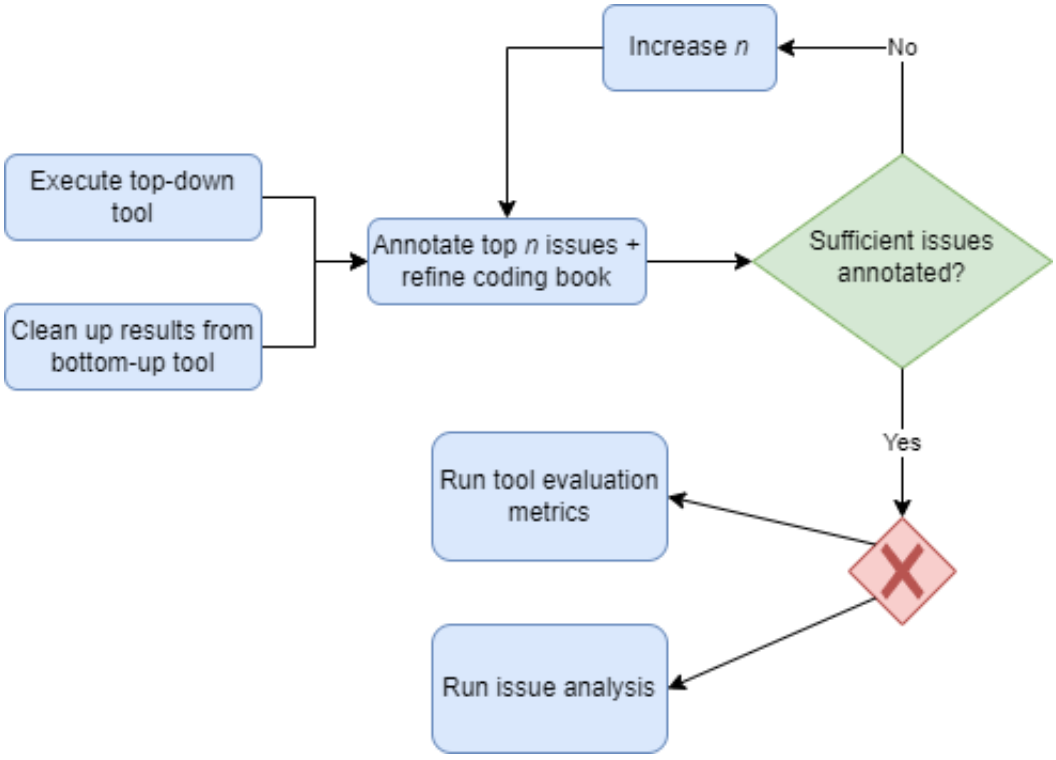


Figure 4.1: Research process diagram

4.1 Issue collection strategy

Jira was selected as the source of issues because previous research in AK-mining has used this issue tracking system with success [7], due to its repository of large projects producing many issues to analyze.

4.1.1 Top-Down Issue Collection

To collect top-down issues, a tool called the **Archedetector** [13] was used. The software collects issues from specified Jira projects, and then a Lucene-powered searcher in the Archedetector filters the results and orders them by the significance related

to the search query used.

The list of Jira projects that were selected to be analyzed are **HDFS, Mapreduce, Tajo, Cassandra, Yarn, Common**. All of these projects (except **Cassandra**) are sub-projects stemming from the larger project **Hadoop**. This collection of projects provided a sufficient issue repository for the searcher to index.

We ran the Lucene searcher with four different queries - **Decision Factors, Reusable Solutions, Components and Connectors, Rationale** (Appendix B). Each query provided a dataset of issues to annotate on. The queries contain vocabulary that have a high chance of appearing in text containing AK. The contents of an issue that the searcher applies the query to consist of the issue description and the comments attached to the issue. The number of terms from the query matched to the contents of the issue is positively correlated to its final ranking: more terms, higher ranking, and presumably the more AK-related the issue is.

4.1.2 Bottom-Up Issue Collection

The source code analyzer tool [7] was used to collect bottom-up issues. Running this tool is not part of the process - we started with a ranked list of commits containing the output of this tool along with various data on each commit. The essential data is the commit's issue and the commit's A2A metric. After filtering, we got a list of issues ordered by its commit with the highest A2A metric.

4.2 Annotation

Once we collected the set of issues as described above, we began the annotating process. The first decision taken is: what kind of annotation tags do we settle on? In the works of Kruchten et al [3], we have four main definitions: **Existence Decisions**, **Bans or Non-existence Decisions**, **Property Decisions**, **Executive Decisions**. We have decided to use three definitions as tags: Existence, Property, Executive, with the fourth definition (Bans) acting as a sub-definition of Existence.

A Coding Book (Appendix A) has been written and updated during the annotation process to record the labeling process systematically. It helps the annotating process improve over time. Future annotators can also use the coding book as a manual. It gives a short definition of each ADD type and how to identify them in an issue.

Two extra tags not related to the type of AK were also added to the set of possible tags: One tag if the issue was present in the top-down collection of issues, and another tag if the issue was present in the bottom-up collection of issues.

The text most focused on for the source of AK is the descriptions of the issues. It was done for two reasons: As a time saver (since some comment chains are long), and also because, in many cases, the range of topics discussed in the comments may be too broad to narrow down details about the potential AK discussed reliably. One exception to this rule: sometimes the description is part of the first comment,

authored by the publisher of the issue. In that case, we took that comment into account when annotating.

4.2.1 Iterative annotation process

The goal was to annotate a certain number of top-ranked issues in the bottom-up and top-down results. But we did not know how many issues to annotate beforehand, so it was done in an iterative fashion. It was decided that if further iterations will not yield significantly more architectural issues, then the annotation process should stop. What ended up happening is that the densities of architectural issues (especially for the top-down, as seen in the Results section) remained high, so we stopped annotating when we decided that enough architectural issues were collected.

In the case of the top-down annotations, the iteration process went like so: Top 50 for each query, top 150 for each query, top 250 for each query, top 400 for each query, top 600 for each query. Within some of these iterations, updates have been made to the ADDs of some issues too, because the knowledge of how to label the issues have increased over time, hence allowing previous annotations to be fixed in later iterations.

The bottom up was only iterated on twice - the first time, we annotated the same number of issues as the top-down, but was decided that this yielded too little architectural issues compared to the top-down so in the second iteration we annotated further.

4.2.2 Number of issues

During the annotation process of the top-down results, the top 600 results returned by the Archedetector searcher have been annotated successfully. However, this does not translate to 600*4 = 2400 unique issues because there is an overlap of issues returned by the queries. The total amount of unique issues annotated is 1062.

During the annotation process of the bottom-up results, the top 1600 results have been successfully annotated.

4.2.3 Differences in the labeling process between the top-down and the bottom-up approaches

The two batches of data come from different software solutions, and the output formats are different. They yield a slightly different labeling process for each:

- **Top-Down** - The Archedetector itself has an interface to browse issues, define tags and add them to issues, which then get stored into its PostgreSQL database.
- **Bottom-Up** - The issues were labeled inside the excel sheet itself. However, this was not the only operation done - we also wanted a way to save these values into the Archedetector Database. For this, the labeling was performed twice per issue - first on the excel sheet, then on the Archedetector.

4.3 Measuring the effectiveness

After collecting and annotating issues, we had two sets of data: A collection of labeled top-down issues and another collection of labeled bottom-up issues. In order to calculate the AK-retrieval effectiveness, two suitable metrics were chosen. Using these metrics, we can answer Research Question 1.

4.3.1 Precision

Issues that contain AK were counted as positive instances for any particular output generated by the search query for the top-down method and the A2A ordered output of the bottom-up method. Let AK = issues containing AK. Then:

$$Precision = \frac{AK}{AK + \neg AK}$$

We evaluated the precision of the returned results at each rank until a maximum rank. This is called *precision at n*. The calculation stays the same, except for each rank n, $AK + \neg AK$ contains all the issues until n.

Recall is a metric that could have been useful, but unfortunately, we have no way of knowing how many total AK issues there are - the dataset of issues is too big to go through all issues. One can potentially use this metric on a smaller subset of issues with a known amount of positive instances, but that is not within the scope of this research.

4.3.2 NDCG

The relevance was chosen to be 1 for an AK issue or 0, so it is a binary value. Because the $\log_2(i + 1)$ term smoothly gets larger at each successive position, the overall contribution of each relevant AK issue strictly gets smaller.

As with precision, the *NDCG at n* was evaluated. In order to determine what the IDCG is for each rank n, the results until n were sorted by relevance.

4.3.3 Missed Parents

While not an effectiveness metric, we still wanted to know how well the methods are at finding the parents of issues that are marked as sub-tasks.

There can be cases where we have an issue containing AK, but it is a sub-task issue, meaning that it has a parent issue. Furthermore, the analysis methods never caught the parent of the issue. We checked every labelled issue's parents and recorded whether they were already previously found or not

4.4 ADD type distributions

We have dealt with issues that either contain AK or do not contain AK. We also checked how the distribution of ADD in AK issues differs across different queries in

the top-down and bottom-up approaches. For this goal, the analysis was straightforward. We collected the total amount of Existence, Property, and Executive tags used for each list of results and then compared them.

There were many instances where issues are labeled with more than one tag. For this purpose, we checked distributions of all the tag combinations possible for each list of results and then compared them. Additionally, the same Precision and NDCG analysis performed on the whole data was performed on specific ADDs. We also analyzed the distribution of ADDs across specific projects.

Comparing the distribution of ADDs in the issues across the top-down and bottom-up methods will answer Research Question 2.

4.5 Issue Property Distributions

In order to get some more answers for Research Question 2, we made three sets of data - top-down-only, intersected, and bottom-up-only. The contents are self-explanatory: the first set deals with issues only present in the top-down analysis, the last set deals with all the issues only present in the bottom-up analysis, and the middle set are the issues present in both. The issue properties described in the background section were then compared across these different sets of issues.

Next, we would like to try and answer Research Question 3. For this purpose, we made two sets: AK issues and non-AK issues, from both the bottom-up and top-down approaches. The property analysis was ran and a comparison was made between these sets.

Chapter 5

Results

5.1 RQ1: Bottom up And Top Down Effectiveness

5.1.1 Precision Results

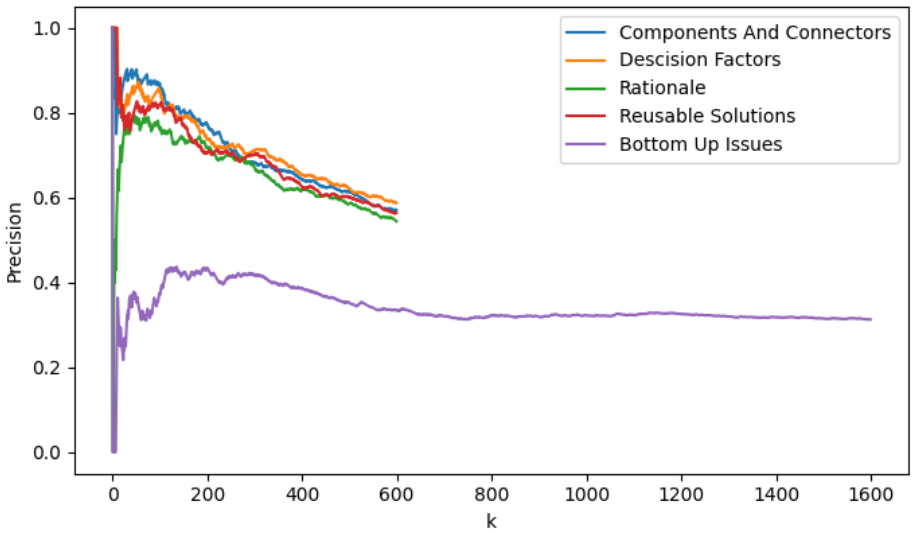


Figure 5.1: Precision at k for the top 600 results of each top-down query and the top 1600 results for the bottom up results

In Fig 5.1, a violent variation of precision occurs for all result sets, with a quick stabilization into a gradual decline as k increases. First, let us understand why the precision values evolve the way they do. When k is very low, the data is jagged and violently shifting directions. It is because new entries are affecting the precision by a large magnitude. At larger k values, every new contribution minimizes, smoothing out the line.

All top-down queries perform similarly well, but the precision of the bottom-up issues is much lower. This likely happens because a lot of the commits have a high

A2A metric despite lacking in architectural changes, and this occurs in cases where a simple operation like renaming a variable occurs over a large portion of the code.

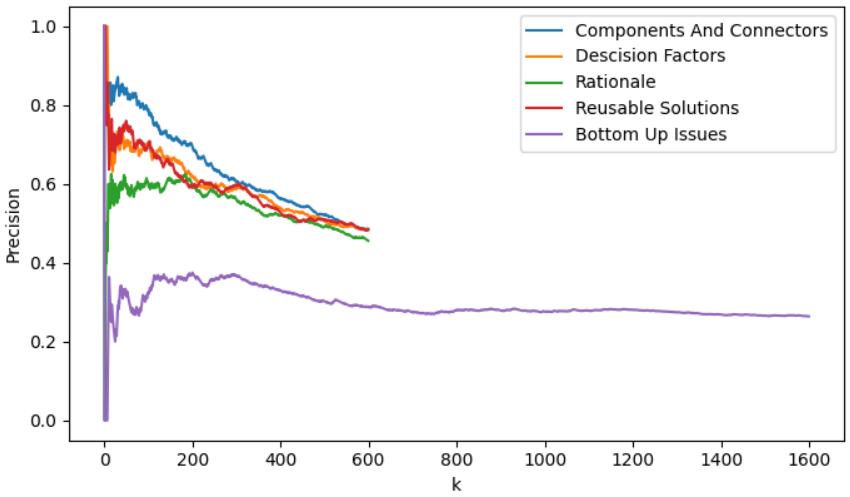


Figure 5.2: Same setup as Fig. 5.1, only counting the issues with an *Existence* tag

In Fig 5.2, we see the same patterns as 5.1, but with one difference: The queries of the top-down results start stabilizing with a significantly different value (with *Components and Connectors* starting with a higher value, and *Rationale* starting with a lower value) before eventually converging, where they remain at roughly the same (slight) difference of precision. Because of the definition of the *Existence* ADD, it makes sense why the *Components and Connectors* is high performing early on - this search query contains terms related to structural elements of the architecture, while *Rationale* does not.

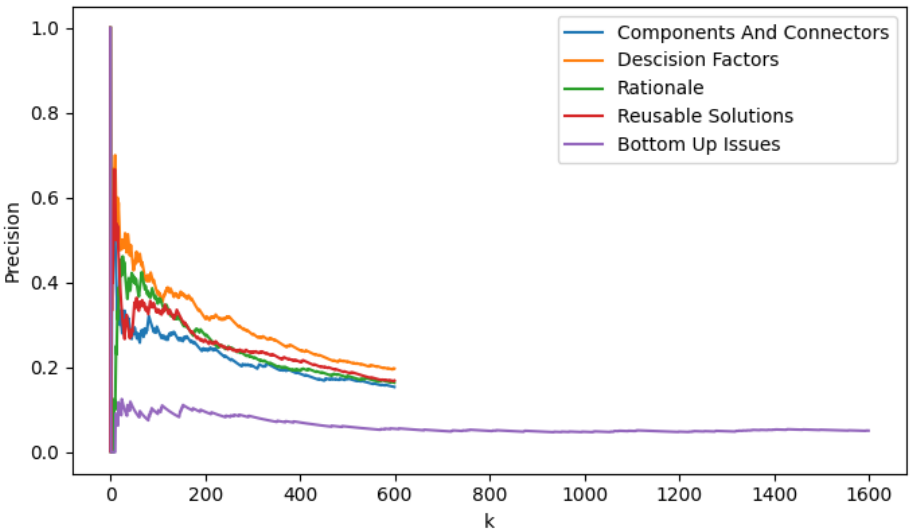


Figure 5.3: Same setup as Fig. 5.1, only counting the issues with a *Property* tag

In Fig 5.3, we see a similar progression of precision to the figures compared to Fig 5.1 and 5.2, but with a notably lower overall precision for all data.

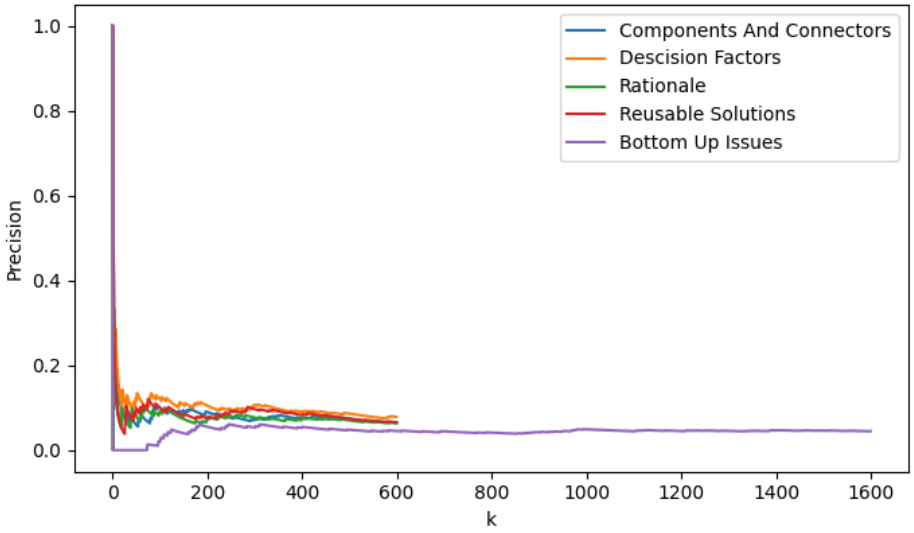


Figure 5.4: Same setup as Fig. 5.1, only counting the issues with an *executive* tag

Here we see record low precision values. A notable feature is that the bottom-up precision stabilizes much closer to the precision of the queries.

5.1.2 NDCG Results

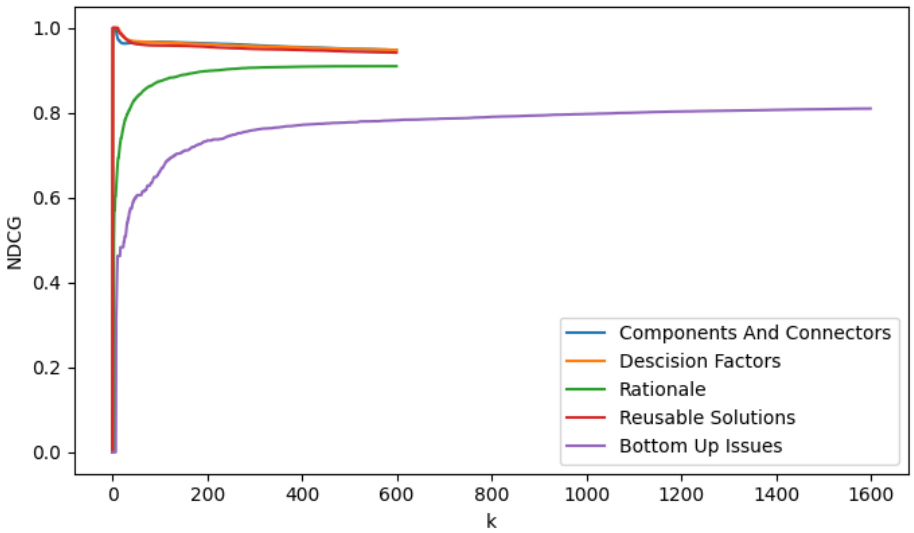


Figure 5.5: Precision at k for the top 600 results of each top-down query and the top 1600 results for the bottom up results

In Fig 5.5, the NDCG of the top-down stabilizes at quite a high value compared to the bottom up. Although the *Rationale* query seems to lag a little behind the other three queries. This is because the NDCG is very sensitive to the first few results at the top of the list, and the *Rationale* query happens to have fewer architectural issues at the very top compared to the other queries.

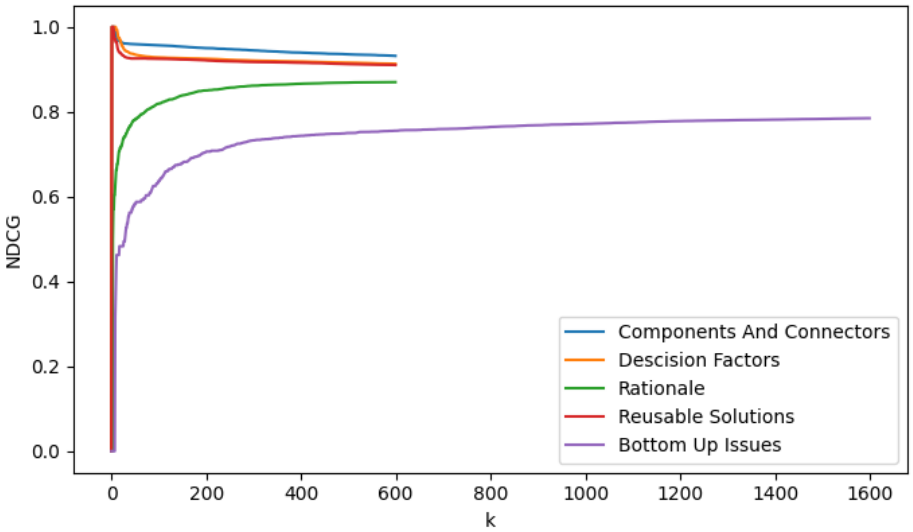


Figure 5.6: Same setup as Fig. 5.5, only counting the issues with an *Existence* tag

Fig 5.6 looks strikingly similar to figure 5.5, with maybe ever so slightly lower NDCG values across the board.

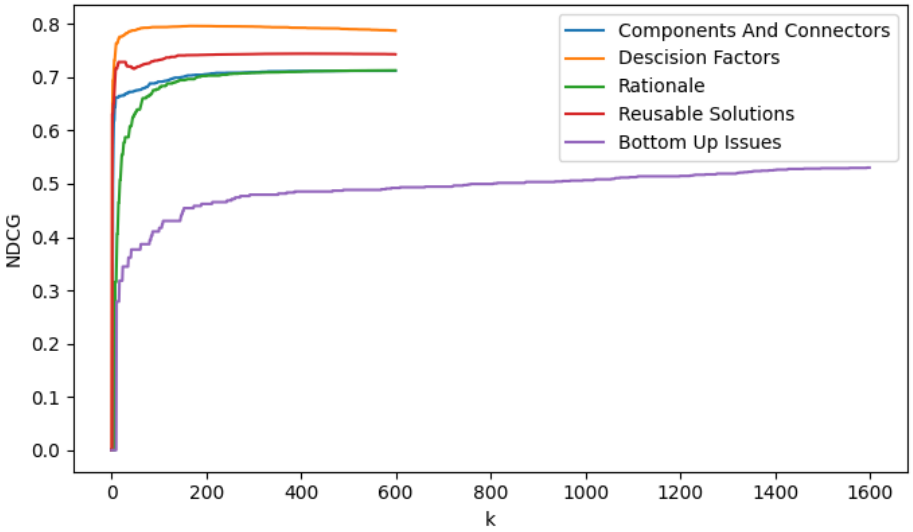


Figure 5.7: Same setup as Fig. 5.5, only counting the issues with a *Property* tag

In Fig 5.7, we see more lowering of the NDCG values and a small but not very

significant spread on the NDCG of the top-down queries.

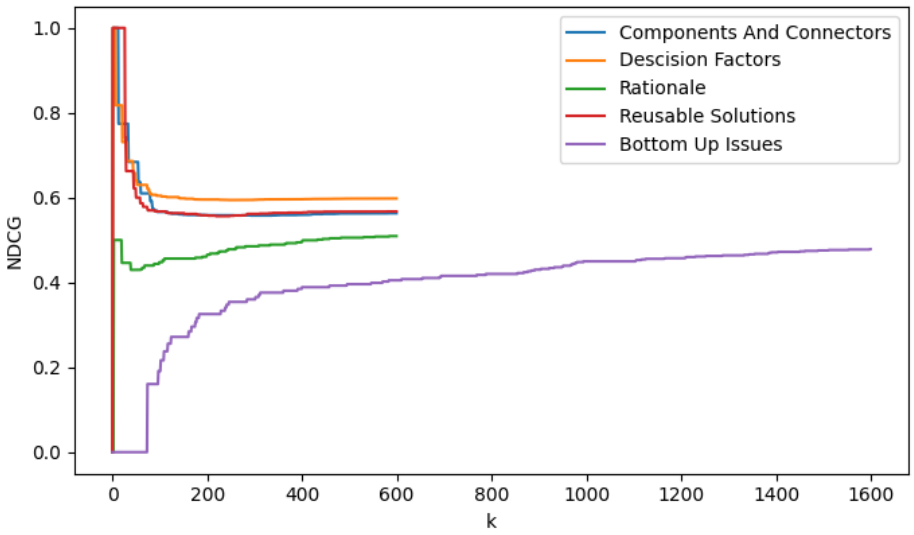
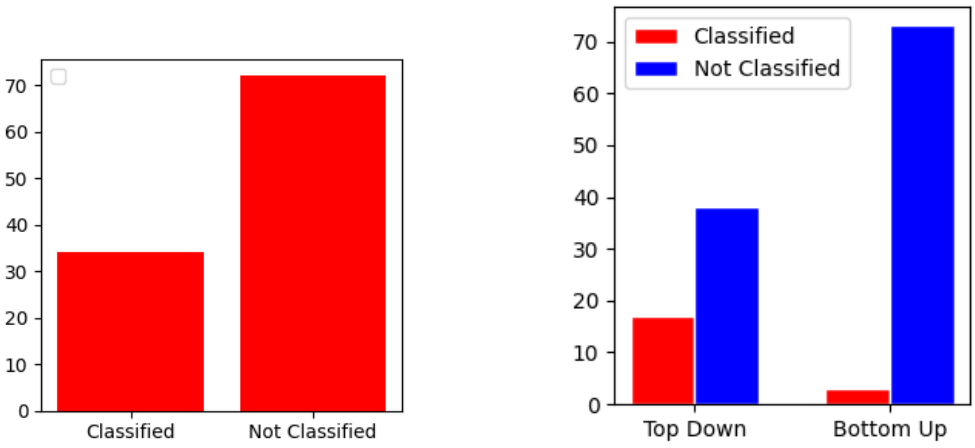


Figure 5.8: Same setup as Fig. 5.1, only counting the issues with an *Executive* tag

Somewhat unsurprisingly, Fig 5.8 displays the lowest NDCG scores. Here we also see the same pattern occur as with the precision graph of the same tag (Fig 5.4), where the bottom-up scores get relatively closer to the scores of the queries.

5.1.3 Missed Parents Results



(a) Top-down and bottom-up inclusive (b) Top-down and bottom-up separate

Figure 5.9

Fig5.9 (a) is pretty straightforward: for every AK-positive issue's parent analyzed, there is a higher chance that it has not been found by either method.

Fig 5.9 (b) shows that the top-down method classifies fewer new parents than the bottom-up.

The bottom-up seems to do much worse at finding parents. This is because code committed is not usually linked to parent issues, but instead to the parent's subtask issues. The parent issue is mostly responsible for managing the task at a higher level.

5.2 RQ2: ADD Type distributions

It is important to note that for this analysis, the amount of bottom-up issues that are checked has been limited to 1200 in order to roughly match the amount of AK issues found in the top 600 of each query.

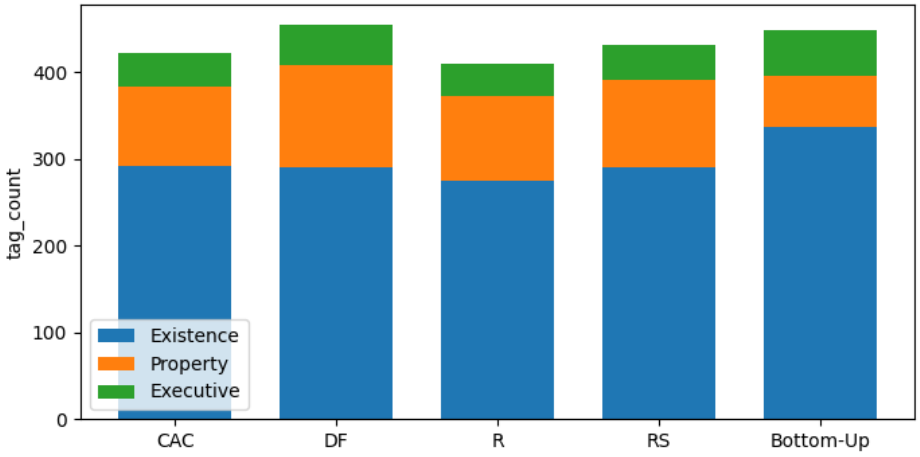


Figure 5.10: The distribution of ADDs found for each query for the top down issues with the bottom up issues alongside. CAC: Components and Connectors, DF: Decision Factors, R: Rationale, RS: Resusable Solutions

In Fig 5.10, there are similar parallels to the general number of each tag present when compared to the precision results in the previous section. There are relatively similar proportions of ADDs for each query in the top-down result set. *Existence* ADDs are predominant, followed by the less common *property* ADDs, which is then followed by the least common *executive* ADDs. These results are consistent with the fact that the issue tracking system is mostly used by developers creating project-building tasks, and that means an overall focus on the structural and behavioural elements added or removed from the project, while *executive* ADDs like business decisions are usually kept out of this system, with the exception of third party software discussions.

The bottom-up set is distributed similarly but not quite the same - the amount of *existence* ADDs are also large, but the *property* and *executive* ADDs are roughly

the same in number relative to each other. Relative to the top-down queries, there are less bottom-up *property* ADDs found and slightly more *executive* ADDs.

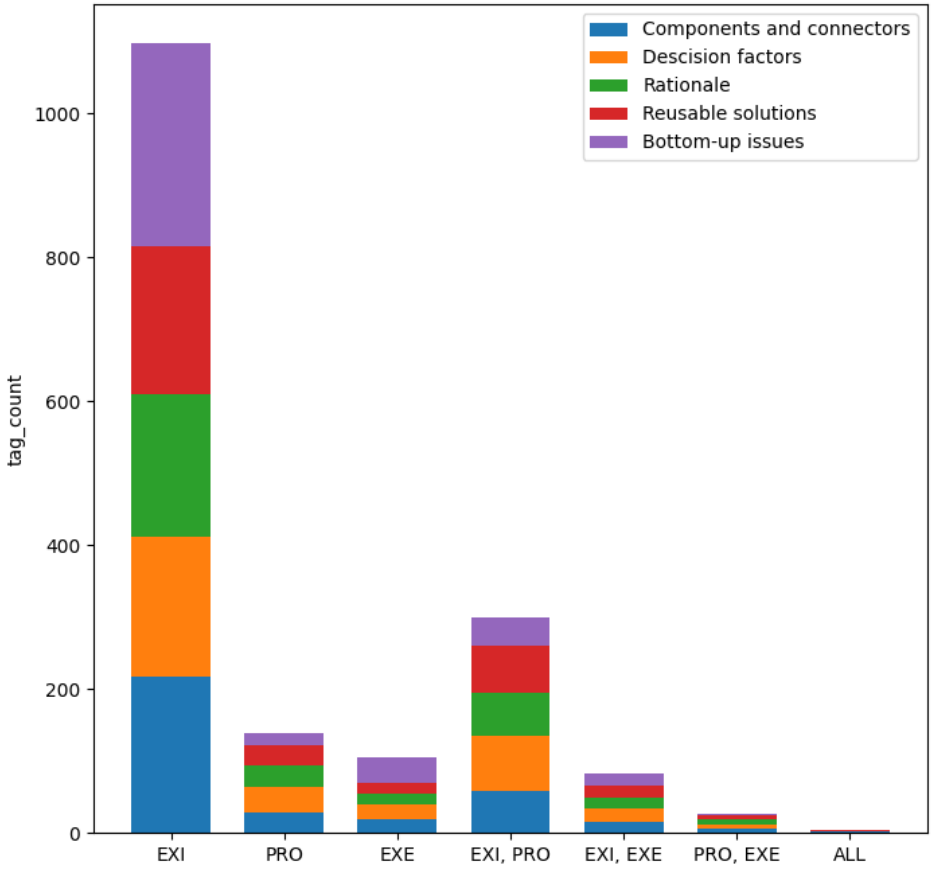


Figure 5.11: The distribution all the combinations of ADDs found for each query for the top down issues and bottom up issues

In Fig 5.11, we see that the *existence* ADD once again predominates, primarily by itself, but also co-occurring with *property* a significant amount of times compared to the other combinations. Issues containing all three ADDs are very scarce. The reasoning for these distributions mirrors the ones shown in the previous figure, but with a few differences: there is a decent proportions of results with the *existence, property* combination, and very few with the *property, executive* combination. This is likely because most issues suggest structural and behavioural changes as a solution, so property/executive-focused tasks are no exception to this.

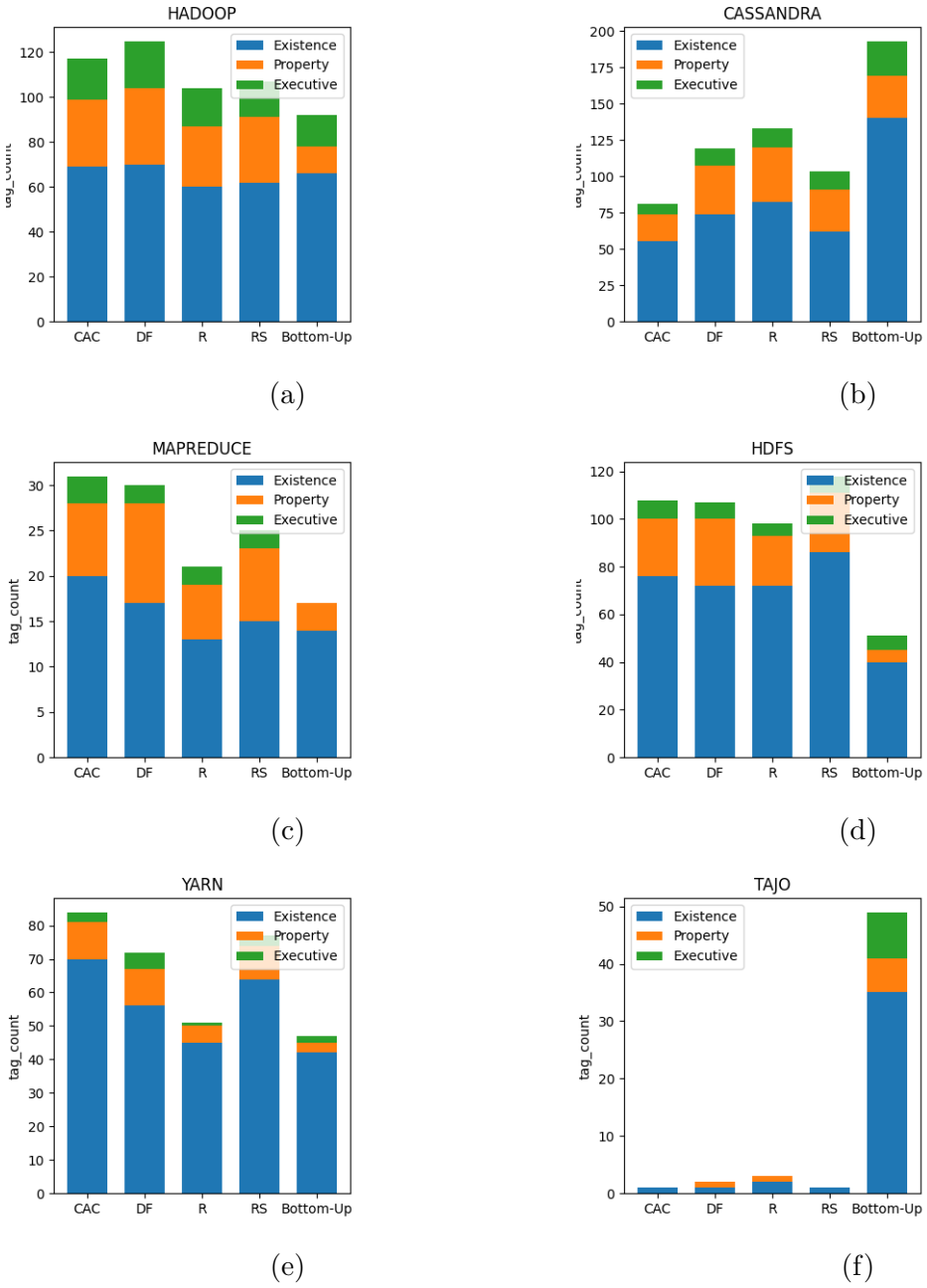


Figure 5.12: Project specific distributions of ADDs

For project-specifc ADDs in Fig 5.12, we summarize some notable details:

- the *rationale* query has the fewest ADDs compared to the other top-down queries, except for CASSANDRA, where it has the most.
- The YARN project has an even larger tendency for *existence* ADDs compared to the other ones.
- The bottom-up ADDs have significant variations of total ADDs compared to the top-down ADDs. The most extreme difference is present in TAJO, where the top-down found almost no issues but are much more common in bottom-up.

- The total number of ADDs across all top-down and bottom-up approaches for each project is shown to be varying a lot, with projects like TAJO and MAPREDUCE containing much fewer issues with ADDs compared to the other projects.

The reasons as to why these variations occur is uniquely project-specific.

5.3 Issue Property Analysis Results

5.3.1 RQ2: AK-Issues Only

First, we look at the results we obtained when we compare AK issues from the bottom up against the top down. The number of AK issues unique to top-down is 388, unique to bottom-up is 409, and present in both is 122.

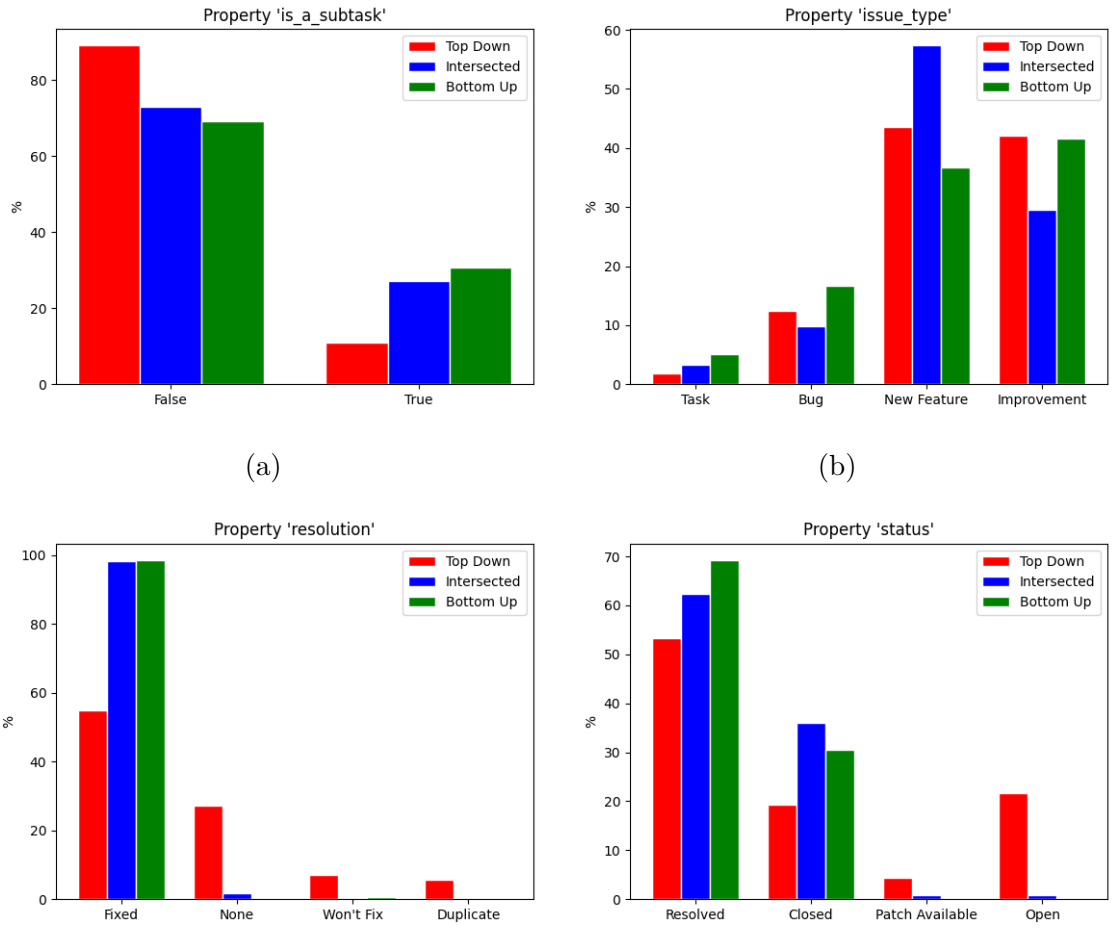


Figure 5.13

Fig 5.13 shows us the percentage of issues of a particular value for each subset of AK issues created. Percentages, rather than frequency, were chosen for this graphic because a fraction of the total issues better represents how dense the issues are with

a specific property when each set contains an unequal amount of issues.

Fig 5.13 (a) shows us that most of the issues from both sets are not a sub-task. This difference is emphasized a little stronger for top-down issues compared to bottom-up issues.

Fig 5.13 (b) has some peculiar details - while there are not any significant differences between the top-down and the bottom-up sets, the intersected sets behave less predictably. Apparently, if the issue appears in the intersected set, it has a higher chance of being a new feature than the top-down only or the bottom-up only. For improvements, the reverse is true. The reasoning for this is not obvious, but its an interesting result nonetheless.

Fig 5.13 (c) reveals a significant difference between the bottom-up and the top-down approaches. Almost all of the bottom-up issues are marked as 'Fixed', while only over half of the top down issues are fixed. All the other property values present also indicate that the relevant changes are not/will not get implemented for top-down issues. This is likely because top down issues does not care whether issues have actually been completed or not, while bottom-up issues are extracted from code changes, which means that the issue has already been worked on.

Fig 5.13 (d) follows a similar narrative to (c) - unfinished top-down issues. Almost only top-down issues exist as "patch available" and "open", in addition to slightly fewer issues being closed and resolved compared to bottom-up.

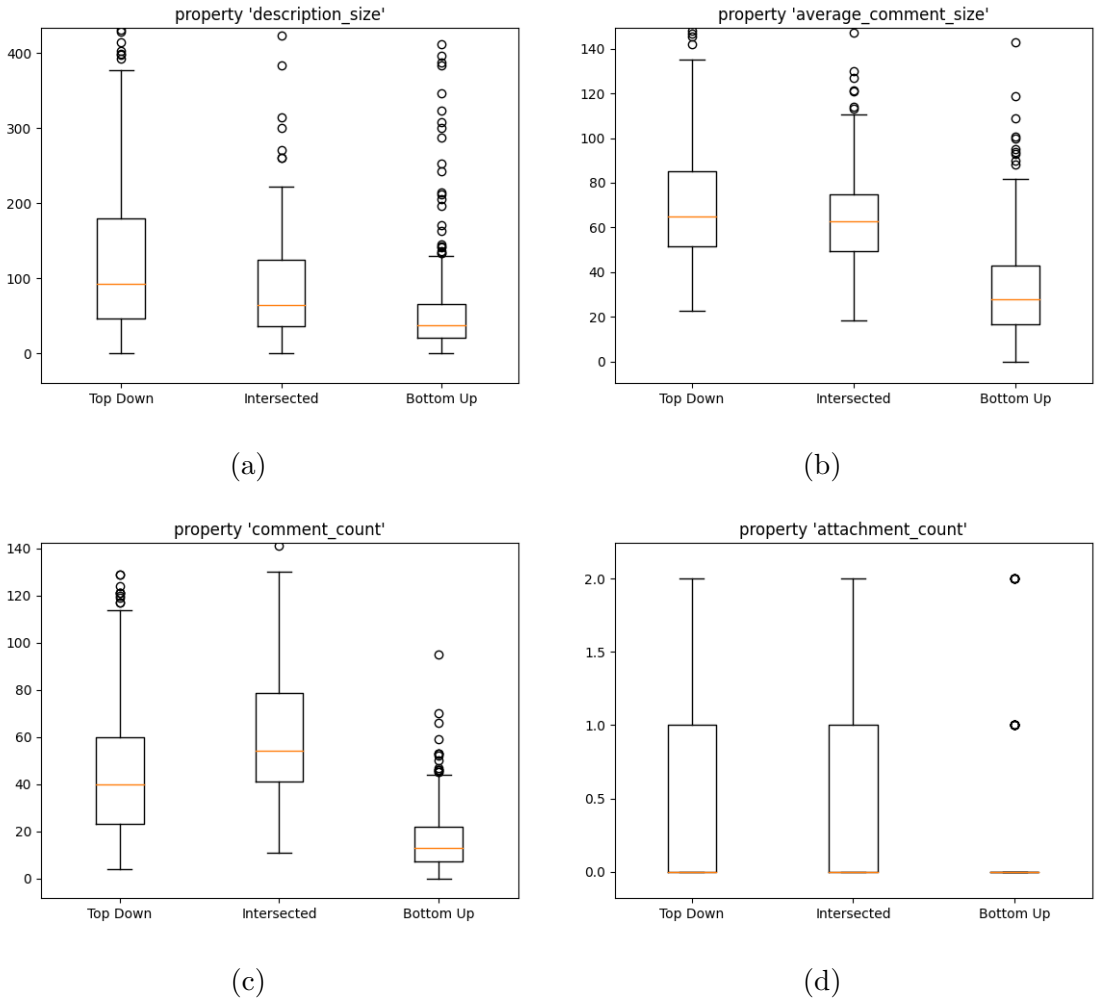


Figure 5.14

Fig 5.14 (a) shows a difference in the length of the description size: top-down issues have a longer description size on average. The average comment size of Fig 5.14 (b) also follows this trend with a greater degree of difference. Fig 5.14 (c) presents a peculiar behavior inspired by Fig 5.13 (b), and Fig 5.14 (d) shows that there typically are no relevant attachments in all of the sets except maybe a few sometimes.

The presence of longer text in the top-down issues are likely because with longer texts comes more frequent occurrences of the search terms used in the query, and therefore becomes a more relevant result. The bottom-up only ranks the architectural relevance from the degree of code changes committed.

We can also note that the shape of the plots tends to have long upper whiskers and many outliers.

5.3.2 RQ3: AK-Issues vs. Non-AK-Issues Results

Now we make two sets: A set of all AK-issues and a set of all non-AK issues. It turns out that there are 919 AK-issues and 1551 non-AK-issues. Now we check how the issue properties differ among the two sets:

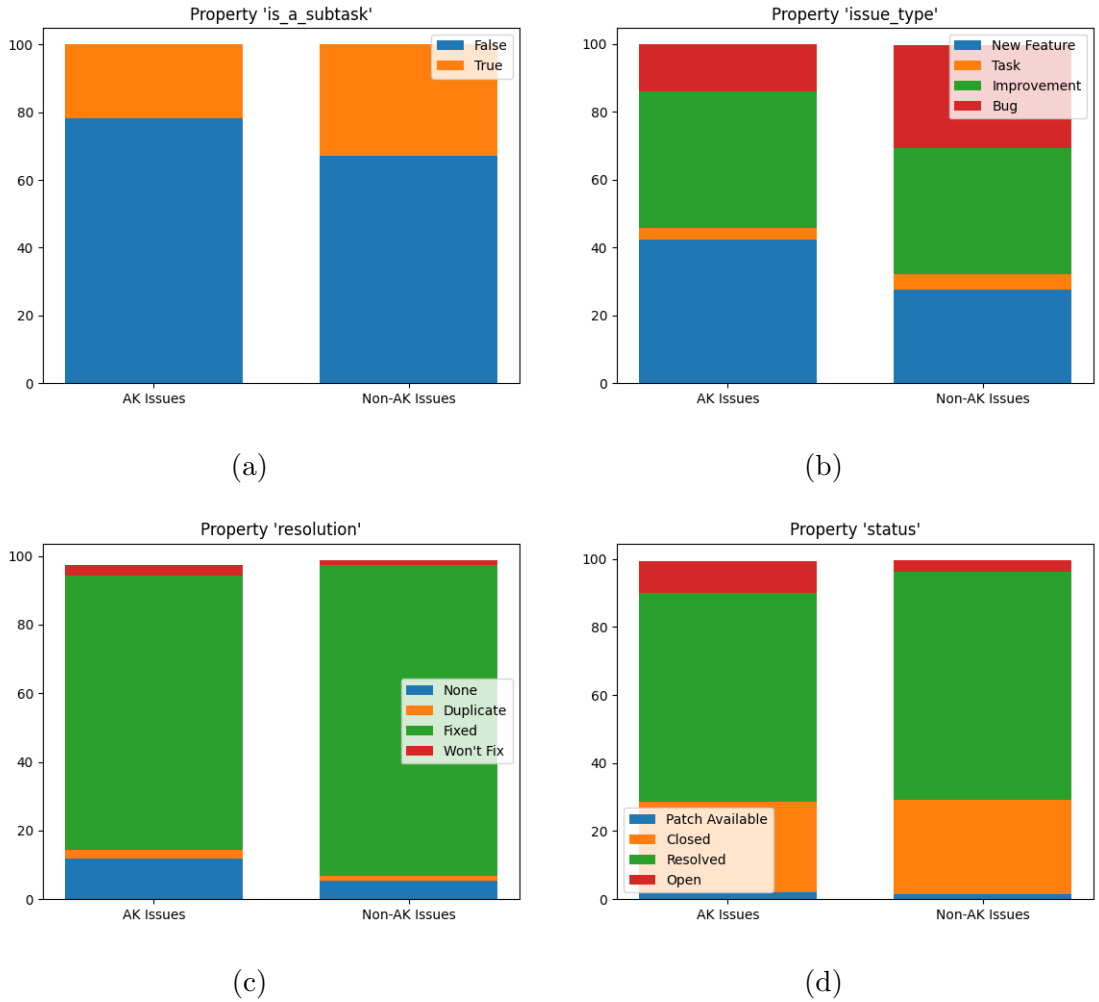


Figure 5.15

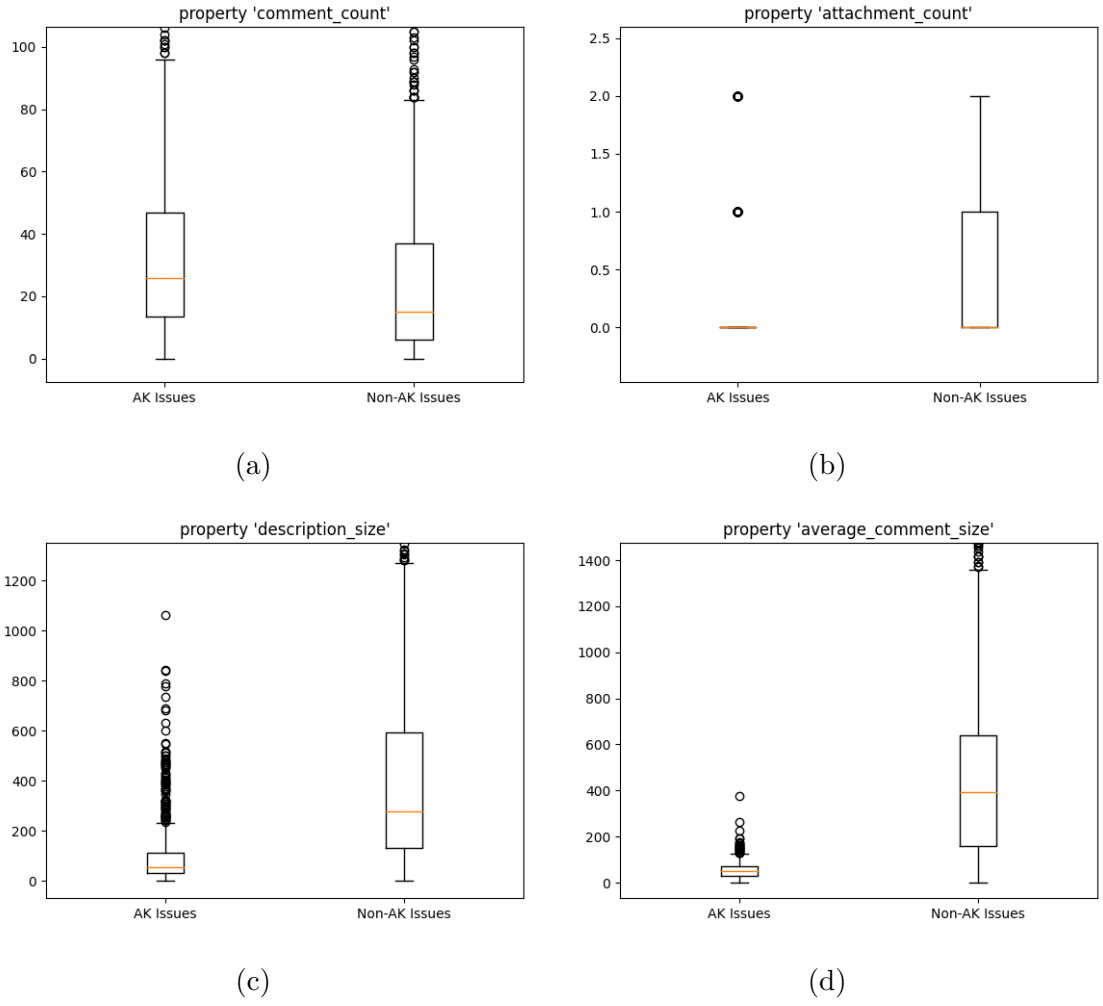
Fig 5.15 is similar to Fig 5.12, but now all the AK issues (formerly divided into top-down, intersected, and bottom-up) are added together and condensed into a stacked chart. The same thing has been done to non-AK-issues. The property values are also expressed as a percentage of the total issues in that set.

In Fig 5.15 (a), we can see AK Issues are less likely to be represented as subtasks. This is an expected result because sub-tasks are intuitively 'smaller' than non-sub-tasks.

The key differences in Fig 5.15 (b) seem to be that AK Issues consist of fewer bugs and more new features than non-AK. Bugs are usually not architecturally related, but rather specific code related, and new features typically involve new component(s) which are architecturally relevant.

In Fig 5.15 (c), the differences are slight, but a smaller percentage of AK Issues are considered fixed, and the non-finished states (None, Duplicate, Won't fix) are modestly increased. This makes sense because non-AK issues contain smaller tasks that are more quickly resolved.

Finally, we have Fig 5.15 (d), where there are no significant differences except for a larger number of open AK issues.



(a) (b)

(c) (d)

Figure 5.16

Fig 5.16 (a) shows a slight preference to a larger amount of total comments for AK issues, but they are not all that different, mean-wise and distribution-wise. Fig 5.16 (b) tells a similar story to Fig 5.14 (d).

Fig 5.16 (c) and (d) can immediately appear anomalous at first glance: In (c), the most distant outlier of the AK issue description size lies comfortably inside the upper whisker of the non-AK distribution. In (d), the most distant outlier lies within the mean of the titan set. This striking result is most likely because issues that contain massive stack-traces copy pasted into them are usually not architecturally relevant. These stack traces inflate the size of the text greatly.

Chapter 6

Discussion

6.1 How effective are top-down and bottom-up approaches in finding architectural issues in issue tracking systems?

Implication for practitioners: Comparing the average performance of the top-down query results to the bottom-up issues shows that the bottom-up issues perform consistently worse in all cases when it comes to the percentage of architectural issues. However, revealed in section 5.3.1 is the fact that the majority of bottom-up issues found are not present in the top-down approach. This is a useful result because users are presented with the possibility of using both top-down and bottom-up approaches to maximize issue AK output.

Implication for researchers: Because the search engine looks effective, researches could look into building alternative queries for it and testing out its resulting performances. As for the bottom-up, finding ways to eliminate false-positives (code that looks architectural but isn't) could also be beneficial because of the bottom-up's ability to find architectural issues that the top-down misses.

6.2 What are the differences in AK characteristics and issue properties between the top-down and bottom-up approaches?

Implication for practitioners: In both of the approaches, we have the fact that the *existence* occurs the most, followed by *property*, and then by *executive*. This means that users can use this particular issue tracking system in order to find many *executive* ADDs. As for deciding which approach to use, the differences in issue properties could be used as a deciding factor: For example, almost all bottom-up issues found are fixed/resolved, while many top down issues are still open or even rejected.

Implication for researchers: The distribution of ADDs in specific projects are shown to be slightly different in most cases. It would probably be useful to look into

more projects and see how they vary, and potentially check if any specific attributes of the projects contain indicators of what ADDs the project's issues contain.

6.3 What are the differences in issue properties between AK issues and non-AK issues?

Implication for practitioners: Because there are a few differences in properties between these two sets of issues, users can use these statistics to increase the chances of confirming that any particular issue is architectural, like if the issue type is not a bug, or the issue type is a new feature.

Implication for researchers: While the majority of the results indicated is not of a degree whether further research might contribute more useful data, there is an area of improvement to work on: because the description/comment sizes is so drastically affected by useless information like stack-traces, a better method of calculating these statistics could be explored.

6.4 Threats to Validity

All the results derived in this study depend on an accurately labeled set of issues. Labeling a large number of issues can be error-prone for a single person due to different biases and/or understandings of what ADDs to look out for and at what point issues are decided to have relevant AK in them, but with the guidance of the supervisor of the project, along with a large sample of data collected, the existing inaccuracies should be minor.

An external threat to validity is most likely the source of the data: a subset of projects in Apache JIRA. Things might be different in other issue tracking software, and we cannot be sure how effective the analysis methods explored here will be in a different issue tracker with different projects.

Another threat is present with the usage of the tools themselves: While the queries used seem to be effective, there are no guarantees that better ones do not exist. Additionally, the bottom-up's calculation of the A2A metric could also contain problems that could be improved upon. In order to more accurately determine which of the approaches are more effective, we likely need a collection of tools for both the top down, and the bottom up, and aggregate the results of their usage, or isolate the top-performing ones.

Chapter 7

Conclusion

This exploratory study aims to research two different approaches in which we can acquire architectural knowledge from issue tracking systems, the effectiveness of these approaches, and then use the collected data to gather information about the AK features of these collected issues to learn better how AK is represented in such issue tracking systems.

First, 1062 issues were classified from the top-down approach, and 1600 were classified in the bottom-up approach. It contributes a valuable dataset of issues that can be used in other works to further this line of study.

The effectiveness of the top-down approach of using a search engine to find issues with AK has been determined to be high. However, this was done with a carefully constructed set of queries, so the performance of this method depends on the quality of the query used. The bottom-up approach did not match the effectiveness of the top-down by itself, but a good fraction of issues it did detect were unique from the AK issues in the top-down approach, including from which projects they originated from in some cases.

Analysis of the AK-characteristics and properties of the issues outputted from the top-down and bottom-up approaches reveal differences in the data that can mostly be reasoned about due to the mechanics of how each approach collects issues, therefore knowledge is gained on how to better collect specific types of issues.

7.0.1 Future Work

The somewhat disjointed set of AK issues that both issue collection approaches produced suggests that evaluating how effective using both approaches can be is a valid research path to take from here.

The methodology applied in this paper can be tested on issues coming from other projects or issue tracking software to understand how generalized these results are across a larger domain.

Future annotation efforts could include the time spent labeling x amount of issues. This detail was not recorded in this project, but this measurement could help pre-

dict how long the annotation process would take.

The accuracy and speed of the labeling process can be improved. Multiple people could be involved in the labeling process. At a larger scale, a web application could be designed that provides tools (bottom-up and/or top-down) to collect issues from issue tracking systems and functionalities that let multiple users tag issues with ADDs under the instruction of a coding book. It leads to each issue having a distribution of possible ADDs. This solution could statistically increase the accuracy of the labeling process and increase the raw amount of issues labeled per unit of time. Even larger and more accurate datasets could be produced in this way.

Bibliography

[1] LS Schrijver. "Research in architecture : hard science or tacit knowledge?" Undefined/Unknown. In: *Doctoral education in schools of architecture across Europe*. Ed. by M Voyatzaki. ENHSA, 2014, pp. 72–75. ISBN: 978-960-9502-15-3.

[2] A. Jansen and J. Bosch. "Software Architecture as a Set of Architectural Design Decisions". In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. 2005, pp. 109–120. DOI: 10.1109/WICSA.2005.61.

[3] Philippe Kruchten, Patricia Lago, and Hans Vliet. "Building Up and Reasoning About Architectural Knowledge". In: vol. 4214. Dec. 2006, pp. 43–58. ISBN: 978-3-540-48819-4. DOI: 10.1007/11921998_8.

[4] Olaf Zimmermann et al. "Managing architectural decision models with dependency relations, integrity constraints, and production rules". In: *Journal of Systems and Software* 82.8 (2009). SI: Architectural Decisions and Rationale, pp. 1249–1267. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2009.01.039. URL: https://www.sciencedirect.com/science/article/pii/S0164121209000181.

[5] Arman Shahbazian et al. "Recovering Architectural Design Decisions". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, pp. 95–9509. DOI: 10.1109/ICSA.2018.00019.

[6] Manoj Bhat et al. "Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach". In: *Software Architecture*. Ed. by Antónia Lopes and Rogério de Lemos. Cham: Springer International Publishing, 2017, pp. 138–154. ISBN: 978-3-319-65831-5.

[7] Mohamed Soliman, Matthias Galster, and Paris Avgeriou. *An Exploratory Study on Architectural Knowledge in Issue Tracking Systems*. 2021. arXiv: 2106.11140 [cs.SE].

[8] Mohamed Soliman et al. "Improving the Search for Architecture Knowledge in Online Developer Communities". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, pp. 186–18609. DOI: 10.1109/ICSA.2018.00028.

[9] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI series in software engineering. Addison-Wesley, 2003. ISBN: 9780321154958. URL: http://books.google.fi/books?id=mdiIu8Kk1WMC.

[10] Remco C. de Boer and Rik Farenhorst. "In Search of 'architectural Knowledge'". In: *Proceedings of the 3rd International Workshop on Sharing and Reusing Architectural Knowledge*. SHARK '08. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 71–78. ISBN: 9781605580388. DOI: 10.1145/1370062.1370080. URL: https://doi-org.proxy-ub.rug.nl/10.1145/1370062.1370080.

[11] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the Study of Software Architecture". In: *SIGSOFT Softw. Eng. Notes* 17.4 (Oct. 1992), pp. 40–52. ISSN: 0163-5948. DOI: 10.1145/141874.141884. URL: https://doi.org/10.1145/141874.141884.

[12] Antony Tang et al. "A comparative study of architecture knowledge management tools". In: *Journal of Systems and Software* 83.3 (2010), pp. 352–370. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2009.08.032. URL: https://www.sciencedirect.com/science/article/pii/S0164121209002295.

[13] *Exploring the effectiveness of search engines for finding architectural knowledge in open source repositories*. https://fse.studenttheses.ub.rug.nl/25813/. Accessed: 2020-09-22.

[14] Philippe Kruchten, Patricia Lago, and Hans Vliet. "Building Up and Reasoning About Architectural Knowledge". In: vol. 4214. Dec. 2006, pp. 43–58. ISBN: 978-3-540-48819-4. DOI: 10.1007/11921998_8.

[15] Shinobu Saito et al. "How Much Undocumented Knowledge is there in Agile Software Development?: Case Study on Industrial Project Using Issue Tracking System and Version Control System". In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. 2017, pp. 194–203. DOI: 10.1109/RE.2017.33.

[16] Alexander Fyodorov. "Connecting Discussions within Issue Tracking Systems". PhD thesis. 2020.

[17] Arjan Dekker. "Exploring Technology Design Decisions in Issue Tracking Systems". PhD thesis. 2021.

Appendix A

Code Book

A.1 General

A.1.1 Recurring architectural changes

While annotating, sometimes one may come across a type of architectural design decision with repeated elements, therefore making a decision on an annotation can be faster if such a change is encountered repeatedly. Examples:

- So the proposal for this ticket is to **move the LRU cache logic completely off heap** (CASSANDRA-7438)

In the above examples, the decision template is: "Move {component} off-heap". In the context of the project(s) we are currently annotating, Java is used, and therefore its virtual machine, JVM. This decision template therefore applies to this technology, so some decision templates are project-specific.

A.2 Existence Decisions (Ontocrisis)

Existence decisions can be split into two categories: Behavioural and Structural. However, further dissection of the content in issue descriptions shows that it is usually hard to separate issues in terms of structure and behaviour - they both are relevant; discussions of structure is impractical without behaviour, and vice versa. Therefore existence decisions are considered as a whole.

A.2.1 Design Specifications

The issue description is structured into different categories similar to how a design document would be specified - problem, solution, rationale, feature list, pros and cons, etc. This is indicative of a new component/system, or a change of an existing component/system, that is of high impact. Examples:

- **The goals, The mechanism, Nice-to-haves, Format and Consumption, Alternate approach, Additional consumption possibility** (CASSANDRA-8844)
- **Introduction, Problem statement, Proposal** (HADOOP-7939)

A.2.2 New Architectural Component(s)

Pieces of text suggesting the addition of a new architectural component, or possibility a set of architectural components. Components can be anything from individual classes (if impactful enough), to entire software solutions used in the project. Examples:

- "This jira provides a new block layer, **Hadoop Distributed Storage Layer (HDSL)**" (HDFS-7240)
- "Below are the key requirements for a **Resource Manager** for Hadoop" (HADOOP-3421)

A.2.3 Configuration descriptions

Oftentimes, a named component is not provided as the proposal, but instead we have various keywords and sentence structure that indicate an existential change worthy enough to qualify as an existential design decision. Examples (red: structure, blue: behaviour):

- "Service-level authorization is the initial checking done by a Hadoop service to find out if a connecting client is a pre-defined user of that service. If not, the connection or service request will be declined."
- "Datanodes should actively manage their local disk so operator intervention is not needed." (HDFS-1312)

A.2.4 Bans or Non-existence

Bans are a subtype of an existence decision, so we won't include it as a separate decision. Descriptions of elements and/or behavior that shall not appear, or otherwise not allowed to do certain things, are thought of as bans. Examples:

- The Hadoop KMS implementation will **not** provide additional ACL to access encrypted files. (HADOOP-10433)

A.3 Property Decisions (Diacrises)

Property decisions are best identified by the appearance of text relating to quality attributes (see part two of bass et al [9]) Examples:

- "More-efficient ExecutorService for improved throughput" [Performance QA] (CASSANDRA-4718)
- "Job state needs to be persisted (RM restarts should not cause jobs to die)" [Availability QA] (HADOOP-3421)
- "This the top jira for webapp security. A design doc/notes of threat-modeling and counter measures will be posted on the wiki." [Security QA] (MAPREDUCE-2858)

A useful indicator of where such quality attributes may appear is sections of the issue description discussing the rationale/pros/cons of the changes.

Another way a property decision might be inferred is when the discussion point is correlated with an obvious modification to a quality attribute, but such properties aren't explicitly mentioned. Examples:

- "We will implement a common token based authentication framework" [Security QA] (HADOOP-9392)
- "we need a cache implementation that uses an eviction algorithm that can better handle non-recurring accesses." [Performance QA] (CASSANDRA-11452)

A.4 Executive Decisions (Pericrises)

Executive decisions involve dealing with issues at the business/enterprise level. Things like process decisions relating to meeting requirements or the addition of libraries/frameworks. Executive decisions are originally split into three further subcategories: process, tool, technology. But we don't find it necessary to distinguish between these, as we have found that there aren't many Executive decisions anyway, and including further resolution in the definition does not help the research process significantly. Examples:

- "The resulting implementation should be able to be used in compliance with different regulation requirements." (HDFS-6134)
- "This JIRA is to include a library in common which adds a o.a.h.net.unix package based on the code from Android (apache 2 license)" (HADOOP-6311)

Appendix B

Search Queries

B.1 Decision Factors

actor* availab* budget* business case* client* concern* conform* consisten* constraint* context* cost* coupl* customer* domain* driver* effort* enterprise* environment* experience* factor* force* function* goal* integrity interop* issue* latenc* maintain* manage* market* modifiab* objective* organization* performance* portab* problem* purpose* qualit* reliab* requirement* reus* safe* scal* scenario* secur* stakeholder* testab* throughput* usab* user* variability limit* time cohesion efficien* bandwidth speed* need* compat* complex* condition* criteria* resource* accura* complet* suitab* complianc* operabl* employabl* modular* analyz* readab* chang* encapsulat* transport* transfer* migrat* mova* replac* adapt* resilienc* irresponsib* stab* toleran* responsib* matur* accountab* vulnerab* trustworth* verif* protect* certificat* law* flexib* configur* convent* accessib* useful* learn* understand*

B.2 Reusable Solutions

action* adapt* alloc* alternativ* approach* asynch* audit* authentic* authoriz* balanc* ballot* beat bridg* broker* cach* capabilit* certificat* chain* challeng* characteristic* checkpoint* choice* cloud composite concrete concurren* confident* connect* credential* decorat* deliver* detect* dual* echo encapsulat* encrypt* esb event* expos* facade factor* FIFO filter* flyweight* framework* function* handl* heartbeat* intermedia* layer* layoff* lazy load lock* mandator* measure* mechanism* memento middleware minut* monitor* mvc observ* offer* opinion* option* orchestrat* outbound* parallel passwords pattern* peer* period* piggybacking ping pipe* platform*

point* pool principle* priorit* processor* profil* protect* protocol* prototyp* provid* proxy publish* recover* redundan* refactor* removal replicat* resist* restart restraint* revok* rollback* routine* runtime sanity* schedul* sensor* separat* session* shadow* singleton soa solution* spare* sparrow* specification* stamp* standard* state stor* strap strateg* subscrib* suppl* support* synch* tactic* task* technique* technolog* tier* timer* timestamp* tool* trail transaction* uml unoccupied* view* visit* vot* wizard* worker*

B.3 Components and Connectors

access* allocat* application* architecture* artifact* attribute* behav* broker* call* cluster* communicat* component* compos* concept* connect* consist* construct* consum* contain* control* coordinat* core criteria* data database* decompos* depend* design* diagram* dynamic element* engine* entit* event* exchang* exist* external filter* function* hardware* independ* information infrastructure input* instance* integr* interac* internal item* job* layer* link* load* logic* machin* memor* messag* model* modul* node* operat* outcom* output* part* peer* platform* port* process* produc* program* project* propert* provid* publish* read* relat* request* resourc* respon* scope separate server* service* shar* source* stor* structur* subscrib* support* system* target* transaction* trigger* runtime realtime network* thread* parallel notif* distribut* backend* frontend* central* persist* queue* concurren* middleware* provid* suppl*

B.4 Rationale

advantag* alternativ* appropriate assum* benefit* better best caus* choic* choos* complex* condition* critical decid* decision* eas* evaluat* hard* quick* rational* reason* risk* simpl* strong* tradeoff weak* rational* disadvantag* comparison* pros cons good differen* slow* lightweight overkill* recommend* suggest* propos* outperform* important* versus vs contrast* distinct* fast* heav* boost* drawback* option*