Towards Understanding and Analyzing Rationale in Commit Messages using a Knowledge Graph Approach

Mouna Dhaouadi  
DIRO  
Université de Montréal  
Montréal, Canada  
mouna.dhaouadi@umontreal.ca

Bentley James Oakes  
DIRO, Université de Montréal  
GIGL, Polytechnique Montréal  
Montréal, Canada  
bentley.oakes@polymtl.ca

Michalis Famelis  
DIRO  
Université de Montréal  
Montréal, Canada  
famelis@iro.umontreal.ca

Abstract—Extracting rationale information from commit messages allows developers to better understand a system and its past development. Here we present our ongoing work on the Kantara end-to-end rationale reconstruction pipeline to a) structure rationale information in an ontologically-based knowledge graph, b) extract and classify this information from commits, and c) produce analysis reports and visualizations for developers. We also present our work on creating a labelled dataset for our running example of the Out-of-Memory component of the Linux kernel. This dataset is used as ground truth for our evaluation of NLP classification techniques which show promising results, especially the multi-classification technique XGBoost.

Index Terms—rationale structuring, rationale extraction, Natural Language Processing, Linux, ontology, dataset, openCAESAR

I. INTRODUCTION

The development of software systems involves constant decision-making at various levels of abstraction and degrees of importance. For every such decision, the system developer has a rationale, i.e., the why reasoning that explains their decision. Rationale may be more or less explicitly articulated and is often left implicit or unrecorded. Regardless, it is a very useful piece of information for the development team, as it can be used to learn from mistakes or reuse solution patterns [1]. Many researchers have thus attempted to extract rationale and to help developers leverage it [2]–[5].

In this article, we focus on the rationale expressed in developer commit messages submitted to version control systems such as Git. Tian et al. have found that a quality of a “good” commit message is that it contains rationale information [6]. Such information can help other developers understand and contextualize the proposed changes with respect to other project commits. Crucially, it also allows other stakeholders to assess the quality of the proposed changes and how well the changes fit with the project requirements. Over time, these messages help build a shared understanding of the design and behavior of the software system among the development team.

The primary challenge to understanding rationale is that rationale information in commit messages is not typically expressed in a structured representation or model, and instead is embedded in natural language text. Therefore, identifying and understanding the rationale depends on deep prior knowledge of the project context. This increases the mental effort required to understand a commit, its key decisions, and their rationale. From the point of view of maintaining a shared understanding, this makes it harder to on-board new contributors, and makes it difficult to assess the quality of the provided rationale of each new commit. Finally, it makes it difficult to establish and maintain traceability with decisions and rationale information elsewhere in the project.

Here, we present our results towards extracting models of rationale from commit messages using the Kantara framework for end-to-end rationale reconstruction [7]. Kantara defines an information inference component to structure and extract rationale information, and an analysis interface to understand the extracted rationale. We apply a proof-of-concept demonstration of Kantara to a set of 180 commits from a Linux kernel subsystem. The main contribution is showcasing the practicality of end-to-end rationale extraction and analysis for a real world software project, and highlighting the challenges.

In Section II we explain our running example of the OOM-Killer subsystem in the Linux kernel, our approach to labelling this subsystem’s commits with the rationale information, and discuss our insights on how kernel developers explain their commits. The extraction of rationale information from commits is presented in Section III, where Natural Language Processing (NLP) techniques are used to automatically classify commit sentences. From our ground truth dataset, we quantitatively evaluate the performance of our classifiers. Section IV provides our contribution on the structuring of this rationale information in an ontologically-based knowledge graph [8], along with our efforts on representing this graph as an ontology in the openCAESAR framework [9] to leverage ontological semantics for inferencing. Section V discusses the analysis of the rationale information once it is placed in the ontological format. We present a query operating on this graph to extract valuable information about commit sentences, and an interactive visualization allowing a user to better understand how rationale is presented in commits by different authors.
In Section VI, we overview related work, while Section VII concludes with a discussion on challenges we faced and the future work to address them.

II. RUNNING EXAMPLE: LINUX SUBSYSTEM

This section will discuss our running example of the Out-of-Memory (OOM) memory management subsystem in the Linux kernel. First, the kernel itself and relevant development information is presented. Then, we discuss the OOM subsystem itself. Finally, we discuss our ongoing work on labelling rationale information in the commits of the OOM subsystem.

A. Linux Kernel and Development

Since its creation by Linus Torvalds in 1991, the Linux kernel has grown to run on devices ranging from mobile phones and tablets to supercomputers. This wide-spread adoption is partially due to its open-source and collaborative development.

The main development channel for Linux is the Linux Kernel Mailing List (LKML)\(^1\). The LKML contains email threads concerning every aspect of Linux development, including bug reports and potential patches. For example, this email\(^2\) is a request for comment (RFC) on a potential patch to be applied to the kernel. Once the patches are accepted by a subsystem maintainer, they are passed through the maintainer hierarchy until the patches arrive at Linus Torvalds himself to be applied and released as a new version of Linux.

To maintain a high degree of quality in the Linux kernel, there are requirements for patch submission\(^3\). Among these, the subject line for the email containing the patch must fit a particular standard. This patch email must also contain a few lines or paragraphs describing the motivation/rationale behind the patch, and the impact on the kernel the patch will have. Traceability information is provided in multiple ways, such as a) patches are encouraged to add explicit links to LKML discussions in the patch descriptions, and b) patches must have a summary phrase (such as “oom: give the dying task a higher priority”) which allows for tracing this path across commits and LKML discussions. Linux kernel commit messages are thus a comprehensive repository of decision/rationale information to evaluate our approach.

B. Out-of-Memory Killer (OOM-Killer) Subsystem

The OOM-Killer kernel subsystem frees up memory when tasks have requested all available memory, preventing the system from crashing due to the lack of available memory. The OOM-Killer performs two primary tasks: a) it selects a task to kill, then b) it forces that task (the OOM victim) to release its memory and exit. The selection process is particularly relevant for our examination, as there are rich discussions about the best way to select the task to kill. For example, one code change was to reduce the chance of selecting kernel tasks specifically, later reversed in favor of a uniform selection process.

This OOM approach has been controversial since its origin in 1998, as some developers reason that the system should be allowed to crash in the presence of faulty software, or express concern that desirable user processes (such as a windowing environment) may be selected for killing.

C. Labelling Dataset for OOM Commits

To evaluate our Kantara framework (Section III), we require a form of ground truth to measure our extraction approaches. For this purpose, we are systematically creating a dataset of labelled commit messages for the OOM subsystem indicating the rationale information present in each commit sentence [10].

1) Commit Pre-processing: To create our data set, we selected 180 commits from the commit history of the OOM-killer file\(^4\). We did not include any merge commits, i.e., commits whose messages start with Merge tag. We pre-processed the messages of these commits: For each message, we removed the meta-data at the end of the message (e.g., information like Signed-off-by and Suggested-by were removed). We also removed URLs, references to other resources, and call traces using regular expressions. Afterwards, we split the message into sentences and kept only sentences with more than 3 characters and that are not source code. We identified the source code in the message body using heuristics like identifying keywords or symbols such as git, $cd or $echo. These keywords came from manually investigating the data.

2) Sentence Labelling Procedure: We performed six iterations of piloting rounds and consolidation meetings during which the three annotators (a PhD student, a post-doctoral researcher, and a professor) considered 38 randomly-chosen commits in total (which they annotated independently). Finally, we reached a consensus regarding the set of labels to use and our understanding of each label as shown in Table I. We conducted the labelling by batches where if two annotators said yes to a label, then this was taken as the consensus. During the labelling process, Fleiss Kappa averaged 0.69 for eight rounds (so far). This indicates strong agreement considering the subjective nature of rationale [1].

3) Sentence Classification: Our rounds of labelling and consensus-building led us to further insights on how to classify each sentence. Consequently, we have updated our representation of rationale from what we had previously reported in our earlier work [7]. In particular, as we now examine commits at a sentence-level, each sentence is given multiple

<table>
<thead>
<tr>
<th>Label</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision</td>
<td>An action or a change that has been made, including a description of the patch behaviour</td>
</tr>
<tr>
<td>Rationale</td>
<td>Reason for a decision or value judgment</td>
</tr>
<tr>
<td>Supporting Facts</td>
<td>A narration of facts used to support a decision</td>
</tr>
<tr>
<td>Inapplicable</td>
<td>Pre-processing error or bad sentences (i.e., does not contain English sentences)</td>
</tr>
</tbody>
</table>

This table shows the codebook used to label the rationale information in the commits of the OOM subsystem.

\(^1\)Various archives exist for the LKML such as https://lkml.org/.
\(^2\)https://lore.kernel.org/all/20100527180431.GP13035@uudg.org/
\(^3\)https://www.kernel.org/doc/html/latest/process/submitting-patches.html
\(^4\)https://github.com/torvalds/linux/commits/master/mm/oom_kill.c accessed on 2023-01-12
An example commit with labelled sentences from our dataset

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Labelling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mm, oom: base root bonus on current usage</strong></td>
<td>Decision</td>
</tr>
<tr>
<td>A 3% of system memory bonus is sometimes too excessive in comparison to other processes.</td>
<td>Supporting Facts, Rationale</td>
</tr>
<tr>
<td>With commit a6383f4271b (&quot;oom: badness heuristic rewrite&quot;), the OOM killer tries to avoid killing privileged tasks by subtracting 3% of overall memory (system or cgroup) from their per-task consumption.</td>
<td>Supporting Facts</td>
</tr>
<tr>
<td>But as a result, all root tasks that consume less than 3% of overall memory are considered equal, and so it only takes 33+ privileged tasks pushing the system out of memory for the OOM killer to do something stupid and kill dladent or other root-owned processes.</td>
<td>Supporting Facts</td>
</tr>
<tr>
<td>For example, on a 32G machine it can’t tell the difference between the 1M agetty and the 10G fork bomb member.</td>
<td>Supporting Facts</td>
</tr>
<tr>
<td>The changelog describes this 3% boost as the equivalent to the global overcommit limit being 3% higher for privileged tasks, but this is not the same as discounting 3% of overall memory from <em>every</em> privileged task individually during OOM selection.</td>
<td>Supporting Facts</td>
</tr>
<tr>
<td>Replace the 3% of system memory bonus with a 3% of current memory usage bonus.</td>
<td>Supporting Facts</td>
</tr>
<tr>
<td>By giving root tasks a bonus that is proportional to their actual size, they remain comparable even when relatively small.</td>
<td>Supporting Facts</td>
</tr>
<tr>
<td>In the example above, the OOM killer will discount the 1M agetty’s 256 badness points down to 179, and the 10G fork bomb’s 262144 points down to 183500 points and make the right choice, instead of discounting both to 0 and killing agetty because it’s first in the task list.</td>
<td>Supporting Facts</td>
</tr>
</tbody>
</table>

Table II: An example commit with labelled sentences from our dataset.

Fig. 1: Distribution of the sentences in our OOM dataset

interpretation is that this indicates the subjective nature of rationale. Another possibility is that rationale is present in these sentences to motivate decisions and to provide value judgments on the existing state of the system as discussed in supporting facts. Examples of these multi-label sentences are found in Table II.

III. EXTRACTING RATIONALE INFORMATION

This section will discuss a key part of the Kantara framework: the classification of sentences with respect to whether they contain rationale, decisions, or supporting facts. We discuss our approaches and provide an evaluation of our approach on our Linux subsystem dataset discussed in Section II-C.

Considering the subjective nature of the rationale information and the notable overlap between the different categories (Fig. 1), we investigated two classification approaches: binary classification (we do not consider the overlap), and multi-label classification (we consider the overlap). We selected the TF-IDF vectorizer to embed the commit messages. The vectors were then input into various ML models. We used the sklearn library for the models implementation and kept the default parameters values. The models were trained and evaluated using 10-fold cross-validation. That is, the data was randomly

---

3Note that the colouring shown in Figure 1 for each category is also reflected throughout other tables and figures in this article.

TABLE III: Binary classification evaluation

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression</td>
<td>0.72</td>
<td>0.75</td>
<td>0.20</td>
<td>0.30</td>
<td>0.69</td>
<td>0.20</td>
<td>0.03</td>
<td>0.05</td>
<td>0.70</td>
<td>0.75</td>
<td>0.13</td>
<td>0.21</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>0.78</td>
<td>0.71</td>
<td>0.61</td>
<td>0.64</td>
<td>0.68</td>
<td>0.51</td>
<td>0.57</td>
<td>0.53</td>
<td>0.71</td>
<td>0.59</td>
<td>0.46</td>
<td>0.51</td>
</tr>
<tr>
<td>SVM</td>
<td>0.73</td>
<td>0.65</td>
<td>0.22</td>
<td>0.33</td>
<td>0.69</td>
<td>0.20</td>
<td>0.03</td>
<td>0.04</td>
<td>0.71</td>
<td>0.75</td>
<td>0.17</td>
<td>0.27</td>
</tr>
</tbody>
</table>

A. Binary classification

In binary classification problems, any of the samples from the dataset takes only one label out of two classes. We therefore consider the subset of the OOM dataset with the sentences that were labelled with only one label. Specifically, we ended up with 307 decision sentences, 94 rationale sentences and 356 supporting facts sentences. Since the data is imbalanced, we apply under-sampling to the majority classes and consider only 100 decision sentences and 100 rationale sentences. We then trained different classifiers considering one label at a time (i.e., when trying to classify decision sentences, rationale and supporting facts sentences we labelled as negative).

Table III reports the classification results of the widely-used binary classification models: Logistic Regression, Decision Tree and Support Vector Machines (SVM) [11]. We report the Accuracy, Precision, Recall and F1-score evaluation metrics.

From Table III, we extract four insights. First, the overall recall across the binary classification techniques is low. Second, the Decision Tree algorithm gave the overall best results. Third, the classification of the Decision sentences was more successful than the classification of the Supporting Facts sentences. Finally, the performance of the three classifiers when classifying Rationale sentences was rather low.

B. Multi-label classification

The multi-label classification is the most natural approach that applies to our problem as any sample from the dataset can be labelled with more than one label. In this experiment, we considered 1151 sentences (distributively shown in Fig. 1) and we tried widely-used models for multi-label classification including the eXtreme Gradient Boosting (XGBoost) [12]. We report the micro-averaged evaluation results over the three categories in Table IV. Results indicate that the XGBoost classifier gave the overall best performance.

We split the dataset to 60% train set and 30% test set. We train XGBoost on the train set and test it on the test set. We report its performance on the three categories in Table IV. Results indicate that the best classification results were for the Decision label, the second best were for the Supporting Facts. The Rationale classification results were the worst.

C. Summary

These results indicate that the most challenging task is rationale classification, possibly due to the overlap of categories in the messages themselves, as seen in Figure 1.

TABLE IV: Multi-label classification evaluation

<table>
<thead>
<tr>
<th>Model</th>
<th>Precision*</th>
<th>Recall*</th>
<th>F1-score*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest</td>
<td>0.75</td>
<td>0.51</td>
<td>0.60</td>
</tr>
<tr>
<td>XGBoost</td>
<td>0.67</td>
<td>0.60</td>
<td>0.63</td>
</tr>
<tr>
<td>KNN</td>
<td>0.60</td>
<td>0.50</td>
<td>0.54</td>
</tr>
</tbody>
</table>

* Micro-averaged

We find that the Decision Tree and XGBoost techniques are most promising, but further investigation is needed to provide satisfactory results.

IV. Structuring Rationale Information

This section will discuss our contribution on structuring commit rationale information in an ontological-based manner.

A. Decision/Rationale (DR) Graph

In our previous work [7] we defined a graph structure to explicitly represent commits, their labelling, and their relationships. For example, we related conflicting or similar commits together and represented the rationale for each commit as free text. Our current work is investigating the structure and classification of the rationale for one commit; we thus omit those inter-commit relationships here. Instead, from our insights from our labelling procedure (Section II-C), we now focus on rationale information at the sentence level where sentences can have multiple labels.

Table II shows the example commit in textual form with sentences colored according to their label, while Figure 2 represents that commit as a DR graph. A commit has the usual information like the commit hash and the date, and each commit is linked to its author and the individual sentences within the commit text. Each Sentence at the bottom of Figure 2 has a link to the nextSentence (to retain the structure of the commit), and are multi-typed with each label. Section IV-C explains how these classifications are inferred automatically through ontological semantics, which aids the analysis and reporting functionality of the Kantara framework.

B. Ontological Representation

The DR graph described in the previous section explicitly captures concepts and their relation. This makes it natural to opt for a representation of this DR graph using an ontological basis, as ontologies represent “a shared understanding of a domain” [13].
approach to assign these subtypes to individual sentences through inferencing discussed in Section IV-C.

The advantages are that a) subtypes of sentences can be automatically inferred by the ontological reasoner, and b) sentences can be labelled with multiple classification types. While this could be adequately captured using a standard meta-modelling approach, we found that this ontology-based approach simplified our analyses and more accurately captured the meaning of our labelling.

C. Inferencing

An advantage of storing the commit and rationale information in an ontology is that ontologies naturally form a graph structure (a "knowledge graph" or "knowledge base") suitable for inference and querying [8]. We leverage this inferencing capability of ontologies to perform additional reasoning on our data set. From the Rosetta editor, tasks are available to run the Pellet ontological reasoner to perform this inferencing and check the consistency of the vocabulary and descriptions.

An example of this reasoning in the rationale vocabulary is found in Listing 2. The rule defines an antecedent/precedent rule where if \( s \) is a sentence and it has a classification of \( ct \), where that \( ct \) is Rationale, then the sentence \( s \) should be inferred to be a \( \text{RationaleSentence} \). While this is a straightforward rule, the benefit is that it simplifies the further analysis and reporting of which sentences contain rationale. That is, the analysis and reporting steps will not have to follow extra edges to determine if a sentence is classified as having rationale or not.

Listing 2: Classification rules for Sentences and Commits

```java
concept DecisionSentence < Sentence
concept RationaleSentence < Sentence
concept SupportingFactSentence < Sentence
rule RationaleSentenceInfer {
    Sentence (s) & hasClassification (s, ct) & Rationale (ct) -> RationaleSentence (s)
}
concept CommitWithRationale = Commit {
    restricts base:contains to min 1 RationaleSentence
}
```

The \text{restricts} inference rule in Listing 2 goes farther with a similar typing action. Here, a commit that \text{contains} at least one \text{RationaleSentence} is a \text{CommitWithRationale}. Again, this simplifies analysis and reporting by removing the need to resolve links during those phases. For instance, without this inferencing, each analysis and report would have to determine if a commit contains a sentence with a classification type of rationale. Instead, this inferencing allows us to directly query for whether the commit is a \text{CommitWithRationale}.

Section V presents an analysis query and visualization which we have developed within the openCAESAR framework to better understand the rationale information. This query directly relies on these inferences.
V. Analysis

This section discusses the last step of the Kantara framework: analysis and visualization of the extracted rationale information. The intention of Kantara is to provide the user with insights into the presence of rationale in the commit sentences. This can be used to increase the quality of commit messages. For example, the patch submission process could require a certain percentage of the commit to contain rationale. Or, this could be used to identify authors or subsystems with insufficient rationale, so that interventions could be made.

A. Graph Queries

We employ the openCAESAR framework to build the rationale ontology, such that Kantara can produce OML description models through text generation which conform to this vocabulary (Section IV-B). Then, we can execute SPARQL queries on these description models to obtain results. For a developer to better understand the rationale information present in our DR graph, we have created queries such as querying for a list of authors and their commits, and querying for those sentences classified as DecisionSentence, RationaleSentence, or SupportingFactSentence.

Listing 3 demonstrates a more comprehensive SPARQL query (with omitted PREFIXes) which can be executed within the openCAESAR framework to return a JSON result containing commits, their authors, and whether each commit contains rationale. This table also contains the text and classification type(s) for each sentence in that commit.

Note that this query directly relies on the inference rules discussed in Section IV-C. With the typing provided by the inference rules, the query can be more concise. This motivates our choice of using an ontological approach for storing this rationale information.

B. Visualization

The results of the query in Listing 3 can be inspected in tabular form but we developed Kantara to also support interactive visualization, e.g., to display authors, their commits, and the sentences within those commits. The intention is for developers to use such visualizations to understand the rationale information present in each commit.

Figure 4 shows a screenshot of a particular visualization that centres committers’ practices. In the example shown we focus on OOM developer Michel Lespinasse, who has authored one commit (c25) without rationale, and two commits (c26, c27) with rationale. The interactivity of the visualization allows the user to collapse or expand the children of nodes to focus on certain authors or commits. The visualization runs within a web browser and is built using Javascript and the D3 library, modified from an example provided in openCAESAR.

Authors are the left-most node in Figure 4 where the color indicates whether their commits contain rationale or not. Each commit (c25, c26, etc.) is shown to the immediate right. Finally, the sentences making up that commit are on the right-hand side, with the same coloring as in Figure 1 and Table II to indicate the classification of the sentence. In Figure 4, all

Listing 3: Query for commits and labelled sentences

1 SELECT ?author ?commit_id ?order ?text
2 (BOUND(?hasRationale) AS ?isCommitWithRationale)
3 (BOUND(?rct) AS ?isSentenceRationale)
4 (BOUND(?dct) AS ?isSentenceDecision)
5 (BOUND(?sct) AS ?isSentenceSupporting)
6 WHERE {
7 ?commit a rationale:Commit .
8 ?commit baseV:hasIdentifier ?commit_id .
9 ?commit rationale:hasAuthor ?author_id .
11 ?commit baseV:contains ?s .
12 ?s rationale:text ?text .
13 ?s rationale:sentenceOrder ?order
14 OPTIONAL { ?commit ?hasRationale rationale:CommitWithRationale .
15 }}
16 OPTIONAL { ?s ?rct rationale:RationaleSentence .
17 }}
18 OPTIONAL { ?s ?dct rationale:DecisionSentence .
19 }}
20 OPTIONAL { ?s ?sct rationale:SupportingFactSentence .
21 }}
22 ORDER BY ?commit_id ?order
seven sentences are labelled as decisions, and two are also labelled as rationale.

This visualization provides insight in committers’ rationale documentation practices in these commits. It lets a developer understand the presence of decisions, rationale, and supporting facts written by others, exposing commonalities in a set of commits. It also highlights committers whose commits do not provide rationale. This could help the community to improve its rationale documentation practices, by alerting maintainers to commits with insufficient rationale information.

VI. RELATED WORK

We divide related work into two categories: extraction of rationale and representations of rationale.

A. Extracting Rationale

Sharma et al. employed a heuristics-based approach to try to unearth rationale from Python Email Archives [14]. In [15], the authors propose ASGAR, a semantic grammar-based approach to automatically capture and structure design rationale. These works differ from ours as we propose an NLP-based approach.

Other researchers have employed machine learning (ML) techniques to extract rationale. In [16], the authors tried to automatically identify decisions from textual artifacts. They created a labelled ground truth from the Hibernate developer mailing list and tried various classification models and configurations. Similarly, in [17], the authors analyzed Apache commit messages to study the impact of message quality on software defect proneness. As a quality indicator, the authors considered the binary presence of rationale information and they manually labelled the commits as containing or not motivation/rationale information. These works employ binary classification, thus our work differs by proposing a rationale representation with different multi-classifications (Table I).

ML techniques have been employed in an attempt to solve the capture problem (i.e., the large effort required to capture the rationale manually and adhere to a specific representation) [18]. They aim at bridging the gap with the unused existing rationale representation models such as Burge’s rationale model [1]. For example, in [19], the authors propose annotating chat messages that contain rationale by capturing five rationale elements adapted from the IBIS model [20]. In [5], the authors focus on designing an algorithm to extract and structure design rationale from design documents based on the ISAL model [21]. Rogers et al. investigated the usage of ontology and linguistic features to identifying rationale from Chrome bug reports [22]. Lester et al. investigate evolutionary algorithms for optimal features to extract and classify design rationale from Chrome bug reports and design discussion transcripts [23], and Mathur tried to improve Lester’s classification results for the rationale with new features and algorithms [24]. These model-based classification approaches differ from the data-driven categorization we employ in this work.

B. Representing Rationale

Recently, other rationale representations have been proposed. AlSafwan et al. proposed a model with 15 categories to represent rationale in code commits after conducting interviews and a survey [25]. Hesse et al. proposed a documentation model for decision knowledge built upon the results investigating the comments to 260 issue reports from the Firefox project [26]. Kleebaum et al. built upon Hesse’s documentation model and proposed ConDec tools to support requirements engineers in documenting and exploiting decision knowledge for change impact analysis [27]. ConDec tools build up and visualize a knowledge graph consisting of knowledge elements and trace links. Soliman et al. worked on an empirically-grounded ontology for architecture knowledge from StackOverflow [28]. Their work also supports automating architecture knowledge capturing and proposing solutions to effectively search for relevant architectural information in developer communities. Bhat et al. proposed AdeX, a framework to extract and reuse architecture knowledge and to recommend alternative architectural solutions that could be considered during architectural design making [29]. The authors propose a static and a dynamic architectural knowledge models. In AdeX, the rationale of a decision refers to the quality attribute the decision addresses. AdeX automatically identifies architectural elements in design decisions. These architectural elements are identified using concepts captured in a publicly available cross-domain ontology.

The related work differs from our in two main ways. First, none of these works focus on representing and managing the rationale specifically for the Linux kernel. Second, we provide a knowledge graph-based approach and pipeline using the state-of-the-art openCAESAR tool for constructing and querying ontologies.
VII. CONCLUSION AND FUTURE WORK

This article has presented the Kantara framework for structuring and extracting the rationale found in the commits of developers. We have presented our new classification of rationale information, focusing at the sentence-level and now including rationale, decision, and supporting fact labels. We discussed approaches to classify each sentence using NLP approaches and a brief evaluation. Finally, we described how this rationale information is represented using an ontologically-based knowledge graph in the openCAESAR framework [9], along with the analysis and reporting capabilities in Kantara.

Challenges and Future Work

a) Challenge 1: Expanding the Rationale Ontology: Our categorization of components of the rationale of commits was data-driven where we came to a consensus after several discussions during labelling II-C. Our model here focuses on sentence-level labelling to evaluate our Kantara framework and the NLP classification. In the future, we plan to extend our representation to include all the components of the rationale of commit messages (e.g. Goal, Need, Benefit, etc. [25]) and other previous rationale representations (Section VI).

b) Challenge 2: Subjective labeling: It is possible that our manual labelling process has introduced unintentional bias. To address this, the three authors labelled independently. A Fleiss kappa of 0.69 indicates a high reliability of our labelling. In the future, we are exploring avenues such as: a) including kernel developers themselves to validate our labels, or b) determining if large language models (LLMs) can provide suitable performance for this labelling task.

c) Challenge 3: Classification performance: As shown in Section III, the performance of the classification approaches was rather average. While it may not be possible to achieve perfect performance due to the subjective nature of this classification, we are attempting to improve these metrics. Concrete steps include a) applying hyper-parameter tuning to all classifiers to find optimal hyper-parameters, b) expanding our labelled data set of commit messages, and c) investigating more advanced classification architectures such as Bidirectional Long Short-Term Memory (Bi-LSTM) [30] or pre-trained language models, e.g., BERT [31]

d) Challenge 4: Generality: The last challenge we wish to address is the threat to validity of generalizing this classification approach and our insights. Section II describes how the Linux kernel and the OOM subsystem have a particular development culture which emphasizes detailed commit messages. This culture may not be present in other projects, such that developers are not encouraged or required to produce commits with such detailed rationale information. In future work, we will investigate this by applying our Kantara approach to other codebases.

ACKNOWLEDGMENT

This research work is partially funded by the Fonds de Recherche du Québec – Nature et Technologies (FRQNT) Doctoral Research Scholarship (B2X).

REFERENCES


