university of groningen

faculty of science and engineering

Mining and Analysis of Cost-related Decisions in Cloud Infrastructures

Massimiliano Berardi
Rareș-Dorian Boza
Matei-Tudor Penca

University of Groningen

Mining and Analysis of Cost-related Decisions in Cloud Infrastructures

Bachelor's Thesis

To fulfill the requirements for the degree of
Bachelor of Science in Computing Science
at University of Groningen under the supervision of
Dr. Vasilios Andrikopoulos (Computer Science, University of Groningen)
and
Dr. Daniel Feitosa (Computer Science, University of Groningen)

Massimiliano Berardi (s4063384)
Rareș-Dorian Boza (s3938964)
Matei-Tudor Penca (s4039696)

July 15, 2022

Contents

Abstract

The rapid development of cloud computing and infrastructure allowed for tools such as cloud orchestrators to become a key aspect in the computing industry. By allowing developers to manage infrastructure as code certain advantages such as traceability of decisions in machine-readable format become possible. This study explores the use of mining software repositories techniques to extract commits and issue tracker data from repositories hosted on GitHub that utilize Terraform as their cloud orchestrator. More specifically, we analyzed relevant commits and issues with the goal of extracting insights into how developers manage their infrastructure with respect to operation expenses, and constructed a taxonomy of actions discussed to this effect. Finally, we showcased how natural language processing techniques such as topic modeling can provide further insights, and potentially help the filtering process. The sentiment analysis experiment emphasized its low efficacy on software engineering data, highlighting areas for improvement. The conclusion of our exploratory research illustrated the existence of cost-related discussion between application developers, while also putting emphasis on the potential for future studies.

Acknowledgments

We would like to thank our supervisors, Vasilios Andrikopoulos and Daniel Feitosa for their invaluable guidance and feedback throughout the process of our bachelor project.

1 Introduction And Motivation

Cloud computing has become one of the fastest-growing models of computation in the information technology industry. The reason for this ascension stems from its capability of offering on-demand availability of services such as data storage, computing servers, and networking. The acquired agility provided by cloud technology facilitates the empowerment of companies and individuals alike, allowing development at a faster pace [1]. All hardware, storage, network and maintenance requirements needed for the services above are managed by cloud infrastructures. The cloud tools required to create a complete cloud computing system fall under the infrastructure category. With the number of providers increasing over the years, these tools have diversified, expanded, and specialized [2]. Thus, choosing the optimal technologies catering to the use-case at hand has become a crucial task in today's industry.

To ease this endeavor, tools called *cloud orchestrators* have emerged as commercial products. With their help, developers can deploy their projects with little user interaction [2]. This process is facilitated by a *descriptor file*, also referred to as an *artifact*, that provides the infrastructure with the steps necessary to undertake deployment. Compared to outdated in-house infrastructure, commercial cloud orchestrators enable companies to delegate deployment logistics at the expense of a service cost. Service providers, in turn, deal with the direct costs and logistics of deploying computational hardware, managing a data center, general maintenance and upgrades as well as power consumption [3].

Cloud orchestrators allow the integration, monitoring, and provisioning of *infrastructure as code (IaC)*. Consequently, the advantages offered by code management through version control systems and issue trackers are replicated at the level of infrastructure administration. As such, all the billing related to the cloud infrastructure of an application, such as running or deployment costs, can be managed through code. In turn, this creates interest in the software engineering research community, as it opens new ways of analyzing how application developers deal with cost management in small or large projects. However, having access to open-source projects that make use of infrastructure as code is not enough to allow researchers to assess whether there are systemic documented decisions with regard to cost-management. This is because of the vast amount of possibly relevant information that is stored in version control systems.

To aid that, this research project makes use of another field within Computing Science that has developed at a fast pace in recent years, that of *Mining Software Repositories (MSR)*. This area focuses on extracting and analysing data from software repositories, in order to gain relevant insights into the current status of software development [4]. As discussed further in Section 2.4, while there are many studies that use MSR techniques, there are no works that utilize these techniques to investigate cloud orchestrators or cost management. For this reason, we conducted our research in this field as to uncover its potential worth to academia and industry.

Lastly, as this is an exploratory study, the project focuses on gaining as much insight as possible from the collected data. To achieve that, we turned to natural language processing techniques as we interact with a lot of textual data. The first approach taken was to use topic modeling to validate our filtering methods and find potential improvements for future research. The second approach was to use sentiment analysis to assess whether it is possible to extract any other extra insights that were not found during the manual labeling.

1.1 Research Question

The lack of directly related research highlighted above, but also in Section 2, motivated us to conduct this exploratory study that investigates cost management documented in software repositories that make use of cloud orchestrators. As a result, the research question that we aim to answer is:

Can we extract cost management insights based on cloud orchestrator artifacts and VCS data using MSR techniques?

Consequently, the main question can be split into the following sub-questions:

Q1. What repositories should be mined, and how can we achieve that?

Q2. How can we create a reusable and sanitized data set?

Q3. How can we classify the extracted cost-related information?

Q4. What kind of insights can be gained from the extracted information?

1.2 Research Outcomes

By answering the questions above, we highlight the unexplored potential of analyzing cost-related decisions using concepts such as infrastructure as code and cloud orchestrators. Moreover, we have created a robust taxonomy that can easily be expanded, and an overview of the distribution of the cost-related decisions.

In terms of deliverables, we have created a well-written and documented code base, accompanied by a reusable data set. The code contains all relevant scripts from mining the required data, to processing it and visualizing it. The data set comes in JSON (raw data) and CSV (labeled data) formats which can be used for further analysis or serve as a machine learning training set.

In order for this paper to serve as a backbone in this domain, we have looked into potential ways of improving the insight extraction process by taking into account natural language processing techniques. The results of these techniques can also improve the filtering and labeling tasks.

1.3 Distribution of Work

The nature of our research required a team of three people to work together in order to achieve the required results. As such, all three of us contributed in equal amounts to the majority of the parts. Nonetheless, we each had separate sections where we conducted most of the work by ourselves, with occasional feedback from our teammates. The distribution of work can be seen below:

- Extraction and filtering of data: equally distributed between all team members.
- Analysis and labeling of data: equally distributed between all team members.
- Topic modeling on commits: Rares-Dorian Boza
- Topic modeling on issues: Massimiliano Berardi
- Sentiment analysis on commits and issues: Matei-Tudor Penca

The distribution of sections for the manuscript follows as such:

- Matei-Tudor Penca: Chapters 1 and 5, Sections 2.3, 2.5 (Sentiment analysis), 3.1, 3.7, 4.5.
- Rares-Dorian Boza: Sections 2.4, 3.2, 3.3, 3.6, 4.3 and 4.4
- Massimiliano Berardi: Sections 2.1, 2.2, 2.5 (Topic Modeling), 3.4, 3.5, 4.1 and 4.2

1.4 Thesis Outline

In Chapter 2, we present background information and relevant scientific works that are related to the core concepts of our work. Chapter 3 contains the study design and execution of the research, while Chapter 4 discusses the results obtained. Lastly, Chapter 5 contains the conclusions and potential future works that can follow from this research.

2 Background Literature

This chapter provides background information about the core concepts of our research. In Sections 2.1 and 2.2 we go over the notions of cloud economics and cloud orchestrators, respectively, showcasing why they are important and what other works discuss about them. Sections 2.3 and 2.4 explain what version control systems and mining software repositories are, and how they help us extract information. Lastly, in Section 2.5 we go over the theory behind the natural language processing techniques we use, and showcase related studies that use these techniques in similar subjects.

2.1 Cloud Economics

Cloud expenses can span into many areas of interest. When thinking about improving infrastructure, computational instances and general storage are the first two upgrades that come to mind. An instance is a type of computational machine that can be either physical or virtual in nature. It can scale in its prowess with the more powerful machines reaching a price point suitable only for large operations [5]. The location of the instance itself can also drive its price up or down, the developer has the choice between physical proximity for better networking needs or economical convenience [6]. The storage associated with the instance has to be consonant to the operational needs [5]. In this regard, read and write speed, size and scalability are all characteristics to take into consideration that can heavily influence the final cost of the chosen solution.

Less obvious economical choices delve into networking and provider conditions as well as specific billing plans. Networking can regard Virtual Private Network (VPN) usage or Network Address Translation (NAT) gateways. As an example, Amazon Web Services (AWS)[1] instances might have costs associated with accessing the public internet from a subnet. Being responsible with their use or getting rid of them altogether might reduce costs. Selecting a provider that offers the best plan for the expected usage is also an important responsibility. In this case, bigger names in the industry will inspire more reliability but they might expect a more generous industry standard pricing [7]. When it comes to billing plans, a development or testing environment might benefit from an on-demand business model where the detailed usage of a resource is billed. As an example, Amazon DynamoDB offers a per-request billing plan that would be prohibitively expensive for a production environment given the expected traffic [2]. In a development environment instead, the number of requests needed to fine-tune the final use will cost less than a full-fledged plan. It is important to keep these potential cost sinks in mind when we delve into the inner workings of cloud orchestrators.

2.2 Cloud Orchestrators

Cloud orchestrators are platforms meant to create an abstraction layer over the various cloud services' APIs, with the intent of evening out all the differences among the various providers [2]. They make use of infrastructure descriptors, also known as artifacts, to autonomously deploy their infrastructure [2]. These artifacts come in different languages, such as *Hashicorp Configuration Language*[3] (HCL) and *YAML Ain't Markup Language*[4] (YAML), and are fundamentally instructions for the deployment process. This means that, effectively, a cloud orchestrator that makes use of an updated artifact file can deploy the desired infrastructure without any user interaction.

There are many characteristics that determine the worth and maturity of a cloud orchestrator among the many commercial and open-source solutions available. The most notable ones, as shown in [2], are the availability of an open-source version, cloud services compatibility, interface accessibility and the maturity of their API. In this regard, two main contenders are taken into consideration given their large popularity and market-share, Terraform and Cloudify [2]. The former, while not providing a graphical interface, offers a large range of

[1]https://aws.amazon.com/
[2]https://aws.amazon.com/dynamodb/pricing/
[3]https://github.com/hashicorp/hcl
[4]https://yaml.org/

supported cloud providers and a distinctly measured edge in computational performance and network traffic. Cloudify, on the other hand, is only marginally behind in terms of performance but most notably has some of its features locked behind a paywall [8], potentially turning away many small time and open-source developers. Conversely, Cloudify is known for its more approachable graphical interface as well as its maintenance features and the many run-time information that it is capable of providing.

Until now, any attempt at formalizing and analyzing cloud orchestrators has been aimed at their Infrastructure-as-a-Service (IaaS) and Software-as-a-Service (SaaS) aspects of their design, tackling issues such as resources scheduling, provisioning and deployment [9]. A challenging characteristic to evaluate instead, is their cost management capabilities. There are no papers that directly address cost-related topics in relation to cloud orchestrators, but works like [3] touches on the impact that cloudification can have on the financial aspect of a business. Moreover, the paper also discusses the potential hidden costs of merging to a cloud infrastructure.

2.3 Version Control Systems

Conducting research on the subject means somehow having access to a large variety of information involving the development process of projects that employ cloud orchestrators artifacts. This kind of information can be found in Version Control Systems (VCS). These tackle the need for a platform where users can collaborate on a project in an organized manner [10]. Every change to a project's codebase is registered and applied through a commit, which also involves the respective commit message with the purpose of giving a description to the applied changes. On top of version control systems, software engineering teams also make use of issue trackers for their discussions regarding the development process. An issue tracker is a message board styled system that allows user to report issues, bring up potential improvements or simply discuss a project direction and main features.

There are many VCS available both in the proprietary and open-source market, as well as many VCS hosting facilities. Among the many VCS solutions, some of the most popular ones are Git[5], CVS[6], Mercurial[7]. Git is the most used VCS solution given its long standing use for the Linux kernel development. The most popular Git hosting service is owned by Microsoft and it is called GitHub[8], known to be the top ranking VCS service by userbase [11]. Other VCS hosting solutions are GitLab[9], Bitbucket[10], Launchpad[11], SourceForge[12]. These exhibit varying characteristics such as a paid or free business model, centralized, decentralized or federated distribution and attached services such as internal or third party issue tracker integration.

2.4 Mining Software Repositories

In order to search and extract as much information as possible that may prove relevant to our research question, we have looked into previous Mining Software Repositories studies. Past research employed MSR techniques successfully for varying purposes. For example, exploratory work analyzed the usage of diagram creation tools such as PlantUML or BPMN [12] [13]. Others [11] [14] focused on identifying popular energy-saving practices, as well as documenting power management awareness decisions. In terms of developer consciousness revolving around software engineering concepts, this is also emphasized by examining discussions regarding more abstract concepts such as technical debt [15]. Finally, applying MSR procedures is also relevant when building code data sets for machine learning algorithms [16].

The aforementioned papers, albeit not tackling the mining of financial cost in respect to cloud orchestrators, are still able to provide relevant practices and tools fit for our goal. As such, we will be weighing potential

[5]https://git-scm.com/
[6]http://cvs.nongnu.org/
[7]https://www.mercurial-scm.org/
[8]https://github.com/
[9]https://about.gitlab.com/
[10]https://bitbucket.org/product/
[11]https://launchpad.net/
[12]https://sourceforge.net/

methods of gathering source repositories and filtering public data that have already been tried and tested in previous research. Regardless of what approaches we can consider, the revisionist characteristics allowed in decentralized source code management systems emphasized by Bird et al. [10], must be kept in mind for how they can affect the quality of our results.

As expected from the work outlined in Section 2.3, the most prominently examined VCS by previous MSR research is GitHub. Still, despite pinpointing the code management system, efforts are needed in order to single out only the relevant repositories from the large amount of projects available. There are several ways of undertaking this step. In the case of [14], it has been demonstrated that one can look for external lists of open-source projects that are curated based on a shared characteristics such as being an Android application. These projects are often linked to their respective repositories and can provide a faithful data set of the current development in the field. At the same time, taking a sample percentage of the projects available on GitHub can be done using GHTorrent [17, 12]. However, this heavily reduces the potential number of results if the repository is required to use a specific feature, as in our case. In fact, the GitHub built-in API can be used to further clean the data set of non-relevant repositories [12]. Nevertheless, while many sources point out the restrictions and flaws of using the GitHub API [13, 18, 12], it still remains an effective tool in performing constraint-heavy filtering.

In regards to **commit mining**, GHArchive[13] can be used to query from the total amount of currently archived repositories [11]. There are other tools to consider when undertaking this step such as GraphRepo [18], G-Repo [19], which contains a built-in user interface to ease mining, and MetricMiner [20], library that focuses also on statistical inference of metrics of the results. Others have turned to PyDriller, a library that helps developers reduce the verbosity of mining code, while amassing as many details from commits as possible [16, 21]. PyDriller is effectively a wrapper around GitPython[14], dealing with potential exceptions thrown by the API such as non-UTF-8 file reading, in an attempt to simplify the underlying complexity. The reduction in lines of code promised by the developers makes it approachable for researchers to add their own filtering on top of the commit mining. Furthermore, PyDriller stores the mined commits preemptively as a DomainObject, where all the various data such as the commit message and files changed are lazy-loaded, meaning they are only computed when requested, thus improving the performance. For these reasons, PyDriller is a strong contender as a tool to be used in our research. Moreover, it has been shown [14] that additional filtering can take place to reduce the total number of commits that will need to be manually processed. This is accomplished by composing a list of domain-specific words that should be found in potentially relevant commit messages. Forming such an array of keywords can be aided by past research that have validated their efficiency and suitability in relation to the desired topic.

When it comes to discussions between software engineers revolving around project development, the most suitable platforms of hosting such conversations are issue trackers. Therefore, due to the availability of such data in the case of open-source repositories, mining these services is a tremendously attractive option for MSR research in the interest of understanding design decisions and arising problems. For example, GitHub has a built-in issue tracking management system collaboration that helps users keep track of to-do lists, potential bugs and additions per repository [17]. Besides the usual title and text, developers are allowed to comment, but also assign other users to issues and add labels for categorization [17]. Other similar services are provided also by Jira[15], FogBugz[16] and the Google Issue Tracking[17] system, although mining them will make it increasingly difficult to connect the discussions to the existing repository, without knowing of the link beforehand [15].

In terms of **issue mining**, Bellomo et al. [15] chooses handpicked projects where the trackers are known and access is authorized. Bao et al. [17] demonstrates that using GHTorrent to mine issues from repositories is possible, although the ability of additional filtering is not mentioned or suggested. Other tools that allow for

[13]https://www.gharchive.org/
[14]https://gitpython.readthedocs.io/en/stable/
[15]https://www.atlassian.com/software/jira
[16]https://fogbugz.com/
[17]https://issuetracker.google.com/

the mining of issue trackers are LibVCS4j [22] and SmartSHACK which can collect data from various sources: Jira, Bugzilla, but also GitHub [23]. Similarly, Perceval [24] is a component of the GrimoireLab package that allows for data retrieval from multiple sources in a clean and consistent way. Despite also featuring as a standalone program, Perceval can also be used as a library in Python, which facilitates easier integration with the rest of our mining endeavors as well as any potential analysis and processing of the results. The return type of a Perceval query is in JSON, making it quick to examine the data retrieved.

2.5 Natural Language Processing

Natural Language Processing (NLP) is an established branch of Machine Learning, and Computing Science that employs computational techniques in order to facilitate the comprehension and reproduction of any human linguistic content [25]. Usages of NLP can be quite diverse, ranging from improving human-to-human communications with accurate on-demand translations, to providing hands-free methods of interacting with software through speech recognition. Nevertheless, fitting for our needs and research scope are two NLP methods that can extract desired information from verbose data sets: topic modeling and sentiment analysis.

Topic modeling

Topic modeling is the technique of automatically extracting topics from text data, where a topic is defined as a collection of terms that occur frequently in the documents of the corpus [26]. To clarify the previous definition, a corpus is defined as an unordered set of documents, while documents are an ordered set of terms [26]. The original purpose of topic models is to make use of the semantic structure of unlabeled text to index its content for search-related reasons. In our case, extracting topics from repositories that involve specific contexts, such as cost-related subjects, will potentially suggest keywords that can be employed in future research.

To ensure meaningful results, it is highly advised to review the corpus of the text data and perform a cleaning process taking into consideration the data context [26]. As an example, if the text data that is being analyzed is formatted using any markup language, such as HTML or Markdown, the formatting should be properly removed. In a more general scope, terms that appear in more than 80% or less than 2% of the documents are most likely common terminology, such as connectives or prepositions, or one-off terms that do not bear any weight [26]. Each document of the corpus is then processed into a Bag of Words [27], where order and grammar is discarded in favor of keeping only the terms that are relevant to the research. Repeated occurrences of useless terms will inevitably pollute the end result, after all, topic modeling heavily relies on analyzing repeating patterns of terms. In addition to the previous pre-processing step, a list of stop words can be used to remove terms that we do not want to be taken into consideration. The term selection that forms this list generally does not abide by any particular logic expressed in the text cleaning operation defined above [26]. There are multiple techniques to create topic models, with a large number of variations on their implementation. For our research, we will take into consideration some of the more relevant ones.

Latent Semantic Analysis (LSA), also known as Latent Semantic Indexing (LSI), constructs a matrix that contains the TF-IDF (Term Frequency-Inverse Document Frequency), which represents the weight of each term in each document [26]. Next, the dimension of the matrix is reduced using Singular Value Decomposition. This process reduces the number of unique terms while maintaining the similarity structure among documents [26]. LSA is known to be quick and efficient to implement, but it generally requires a large set of documents to obtain accurate results.

Latent Dirichlet Allocation (LDA) is another example of topic modeling. It employs a generative statistical method for modeling a corpus [28]. This enables LDA to describe each document of the corpus as a probabilistic distribution over latent topics. Similarly, each topic is characterized by a distribution over several words [29]. Therefore, LDA can discover a wide range of subtopics in research areas prone to have various intricacies, depicting such subtleties based on the topic arrays defined. In terms of performance, it competes with LSA while making the overall process more scaleable by allowing the method to be embedded into a more complex model.

The **Hierarchical Dirichlet Process (HDP)** employs a Bayesian non-parametric mixed membership model in its design, featuring each document as a collection of topics that may have disproportionate magnitudes [30]. HDP promises to overcome some of the downsides of LDA, such as its sensitivity to the number of topics expected from the procedure [31]. As, such there is a decrease or increase in the number of topics based on the size of the data that will ultimately alleviate the dangers of underfitting and overfitting respectively.

Having enumerated the topic modeling approaches that are going to be used in this paper, the tools that will allow us to apply them in our research are also crucial in dictating modularity, flexibility and performance. For this, Gensim [32], an open-source Python topic modeling toolkit, has been considered due to its optimized implementations. Gensim offers developers the ability to run LSA, LDA and HDP on large corpora of text, without having to manage the inner works of the specific models themselves [27]. Even so, data pre-processing as a task is still the user's responsibility, and can still heavily affect the outcome of the topic modelers.

Sentiment analysis

Sentiment analysis is a NLP technique that aims to extract a sentiment or opinion from textual data. This process is implemented using machine learning algorithms that use training data to understand and learn how humans express sentiments in text. As this is a classification task, there are many different types of sentiment analysis approaches [33] with varying levels of precision. The simplest version of classification is polarity analysis which attributes either a positive, neutral or negative sentiment to a sequence of sentences as seen in Figure 1. Some models can transform this categorical label onto a numerical scale providing the user with the ability to discern between sentiments such as very positive, positive, very negative, and negative. These types of classifications are relevant in applications such as product reviews [34]. Sentiment analysis has a lot of usages in the social media industry too, as there have been many papers in the past years that apply these techniques on tweets which have become a popular way for customers to express their opinions [35, 36].

There are many techniques inside the machine learning domain that help developers create sentiment analysis tools. Many of these rely on dictionaries that map text features to emotions. This is achieved by splitting the text into words, phrases, or topics which then are translated to a continuous interval using different heuristics. The main machine learning algorithms used in creating a polarity analysis model are support vector machines (SVM), logistic regression, or Naïve Bayes [37].

Sentiment analysis is known to under perform in cases where the data sample contains certain characteristics such as humour, bad grammar, lack of punctuation, and specialized terms that are not part of basic speech [38]. As our research focuses on text extracted from commits and issue trackers, this limitation of sentiment analysis becomes an important factor. This raises questions about the efficiency of applying sentiment analysis on software engineering textual data. Papers such as [39] discuss the limitations of this technique in this regard. The authors explain how the state-of-the-art tools in sentiment analysis do not perform as expected on software engineering data because of the issues described above. They conducted a comparison of different models under which they include an algorithm trained on software engineering data from websites such as StackOverflow[18] and Jira. Their conclusion focused on the poor results returned by the models used, putting emphasis on the fact that there is still room for improvement and exploration on sentiment analysis in software engineering.

As NLP is developing quickly, there are many available models to be used in our research. Currently, the most discussed models are TextBlob [40], Vader [41], Stanza sentiment [42], and SentiStrength-SE [43]. All of these models have been mentioned or used in [39], [44] and [45], and as such are of interest to our research. Both Vader and TextBlob have available python APIs that can be used without having to apply any pre-processing

[18]https://stackoverflow.com/

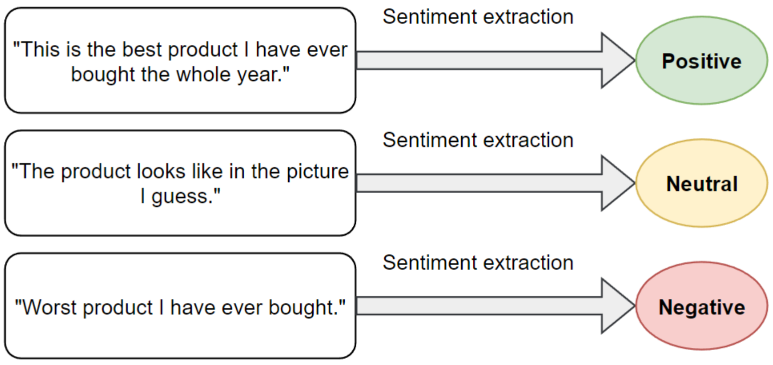

"This is the best product I have ever bought the whole year."
Sentiment extraction
Positive
"The product looks like in the picture I guess."
Sentiment extraction
Neutral
"Worst product I have ever bought."
Sentiment extraction
Negative

Figure 1: Example of sentiment analysis applied on product reviews.

to the data. The result returned is on a continuous scale, meaning the developer can set the thresholds for classifying the text as positive, neutral or negative. Stanza offers a python API that easily allows the developer to add neural pipelines such as sentiment analysis. The only difference is that Stanza splits the input into sentences and attributes a sentiment to each one of them. This in turn requires the developer to create a heuristic in order to assign the sentiment to the initial document. Compared to TextBlob and Vader, Stanza's output cannot split into thresholds, as it can only return three different values, one for each sentiment. One advantage of Stanza is that the API allows the developer to improve the pre-trained model with their own specific training data. Lastly, The SentiStrength-SE model was developed in Java as part of the research conducted in [43], and as such has not received any more updates or improvements in the past years. Moreover, in order to use the model, the developer needs to use the user interface pre-packaged with the application. The disadvantage of this is the fact that the UI requires data pre-processing in order for the sentiment analysis to work. The output of SentiStrength-SE's model contains both the positive and negative scores of the text, allowing the user to create different heuristics for attributing the final sentiment.

Out of all of them, TextBlob and Vader are easier to use as they can analyze a textual document without any pre-processing done. Stanza works differently as it splits the document into sentences and attributes a sentiment to each one of them. This in turn requires the developer to create a heuristic in order to attribute the sentiment to the initial document. Lastly, the SentiStrength-SE model was developed as part of the research conducted in [43], and as such has not received any more updates or improvements in the past years. Its downside comes from having a user interface that requires data pre-processing, which in turn slows the entire experiment.

It is worth noting that in most cases to achieve optimal results in a specific use case, it is recommended to train a model on the data that is being used in that specific project. On the other hand, this approach requires more time to set up a model to be taught, classify the data manually and fine-tune it until it produces the best results.

While the main reason for the usage of sentiment analysis is to uncover insights in the data, this technique could also help improve the filtering or labeling processes. Similar approaches have been applied in related research, which achieves optimal results using the SentiStrength-SE model [46]. This highlights the possible applications that sentiment analysis can have in our research.

3 Study design and execution

This chapter goes over the design decisions that have been made throughout this research, but also how they have been implemented. Section 3.1 explains how we stored and transferred the data we mined between the team members, and the scripts and applications used. Moreover, it highlights the external libraries used for processing the data. Section 3.2 follows the process of extracting and filtering the projects that use cloud orchestrators from GitHub, while Sections 3.3 and 3.4 go more in-depth with how we mined data from the commits and issues that mention cost-related terms. Section 3.5 showcases the manual process of labeling the data, and our logic behind it. Finally Sections 3.6 and 3.7 discuss the methods we adopted to investigate whether it is possible to extract more insights from the data by using techniques such as topic modeling and sentiment analysis.

All the links to the raw data files (JSON), Google Spreadsheets and Python notebooks mentioned throughout this section can be found in the Appendix at the end of the paper. While the notebooks have been run in the Datalore[19] cloud environment, we provide local version of these files that can be re-run to reproduce our experiments.

3.1 Data workflow, storage and processing

This section goes over the decisions taken in order to create a sanitized and reusable data set that can be used for future research in similar domains. Moreover, we will outline any external dependencies to our project's scripts as well as provide reasoning for their usage.

From the beginning of our study, we were aware that our data might not have a well-defined structure and we would have to order it ourselves. Moreover, our study uses concepts from both mining software repositories and machine learning fields. As such, we decided to use JSON for the raw information that we extract for each relevant repository. JSON proved to be a fitting choice because we had to store nested data of integer and text types.

While this format might be good for storage and communication of data, it was not ideal for working on the data when performing analysis or natural language processing experiments. For this reason, we decided to use a more popular format within the machine learning community, Comma Separated Values (CSV). There are many applications that can support CSV format such as Microsoft Excel, but as we had to collaborate on the labeling process, it would have been complicated for each member to have their own local file. To circumvent that, we decided to use Google Spreadsheets in order for all members to have access to the CSV file in real time. For this reason, we have written a Python script to directly write the needed data from our raw JSON files to the spreadsheets, where we would manually label it. Furthermore, this approach can be reversed in order to read the data from the Google Spreadsheets as CSV in our Python scripts, where we could generate statistics and visualizations without requiring a local file.

The implementation provided in Listing 1 heavily relies on the Pandas library, a package that aids the developer in the manipulation of data in Python. The DataFrame component provided by Pandas, allowed us to store our data set in an accessible manner, while also enabling the transcription of information in the previously mentioned CSV format. These features work in unison with the Pygsheets library, which provides a python framework that facilitates the automation loading and reading operations directly into Google Spreadsheets.

This has been a crucial step for our project as, without cloud storage, it would mean that we would have to re-download the CSV file every time a small tweak was made to the Google Spreadsheets by a peer. At the same time, loading the file through scripts (Listing 2) is optional, as after no more updates are done to the CSV, downloading the final version would be advised in the cases where an internet connection is not available all

[19]https://datalore.jetbrains.com/

```python
# authorize API to write inside the Google Spreadsheets
gc = pygsheets.authorize(service_file='commitevalutaion-6a3c4370bf89.json')

# open the google spreadsheet
sh = gc.open('Terraform data analysis')

# select the first sheet
wks = sh[0]

# update the first sheet with df, starting at cell B2.
wks.set_dataframe(commits_pd, (1, 1))
```

Listing 1: Python script for writing JSON data to Google Spreadsheets.

```python
# access the sheets and get the commit sheet
gc = pygsheets.authorize(service_file='commitevalutaion-6a3c4370bf89.json')
sheets = gc.open_by_url("https://docs.google.com/spreadsheets/d/1OYsymuoRJGAXTHQIm...")
commit_sheet = sheets[1]

# get a dictionary of the tables
data_dic = commit_sheet.get_all_records()

pandas_df = pd.DataFrame(data=data_dic)
```

Listing 2: Python script for reading data from Google Spreadsheets into a Pandas data frame.

the time. This combination of using JSON and CSV files that are hosted on cloud services allowed us to work as a team without having to use external version control systems as it was covered by Google Spreadsheets.

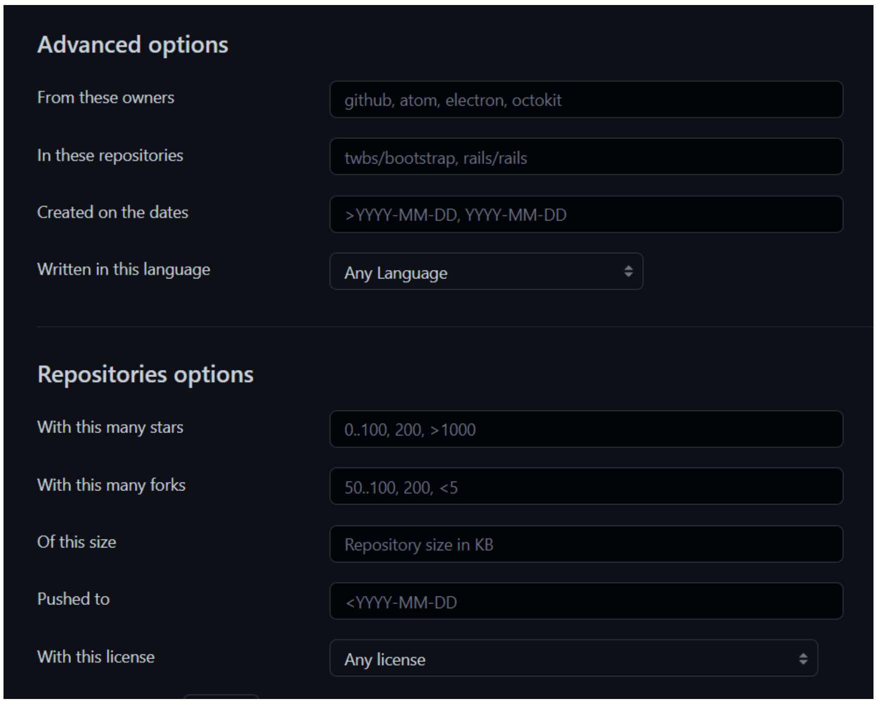
Other Python libraries that we used are BeautifulSoup and Markdown. Both of these libraries help developers clean and extract data from sources that contains HTML or Markdown markup which is not wanted. For data visualization, we have used Matplotlib and Seaborn which are both regarded as state-of-the-art python libraries.

In addition, we have used Jupyter notebooks to run our Python code. These notebooks have been split so that we could run them in parallel to reduce potential run-time. We have also used a cloud environment called Datalore to host and run our notebooks, primarily in order to be able to code together, without any downtime.

3.2 Repository retrieval

In this section, we are describing the steps undertaken in order to retrieve a list of relevant repositories that utilize Terraform as a cloud orchestrator. The reasoning behind the course of actions of our research traversed in order to reach the final data set is also emphasized and explained in depth. In terms of the toolkit used during this process, PyGitHub was chosen, as it implements a simple python wrapper for the GitHub API which is presented in Section 2.4.

Due to the dependence of our research scope on the usage of cloud orchestrators by the examined repositories, we were barred from performing some of the methods outlined in Section 2.4, such as randomly selecting a percentage of archived repositories [17]. Instead, we opted to build a search query that would return us a list



Advanced options

From these owners	github, atom, electron, octokit
In these repositories	twbs/bootstrap, rails/rails
Created on the dates	>YYYY-MM-DD, YYYY-MM-DD
Written in this language	Any Language

Repositories options

With this many stars	0..100, 200, >1000
With this many forks	50..100, 200, <5
Of this size	Repository size in KB
Pushed to	<YYYY-MM-DD
With this license	Any license

Figure 2: Filters of the GitHub advanced search that can be used to influence the repository search

of relevant repositories. As the PyGitHub API, and the GitHub one, are running the same request behind the scenes, the queries have been tested in the former environment before being put into code. Nevertheless, before delving into the steps involving mining lists of projects for Terraform, we have to note the difficulties encountered with the GitHub API, formerly brushed upon in Section 2.4. Despite not being mentioned in any form of documentation, our attempts of building up a query found that not all the available filters of the advanced search affect the end result of repositories. Figure 2 displays the only criteria we could base our search on. Extensions, for example, while being crucial to our data set's integrity, were not queried through the dedicated filter, needing a workaround. Similarly, some filters, such as intervals, did not reliably return the same result, or the correct one. Last but not least, the Rate Limit enforced by the API of 5,000 requests per hour [12] meant that sleep timers had to be incorporated in our scripts, increasing the running time by days. Even so, no other alternative was found, that could offer us the filtering options that this research needed.

With regards to Terraform, an initial query was set up to scrape any repository having the name of the respective cloud orchestrator in any type of documentation. As for GitHub, this meant either the title and description of the actual repository, or the `Readme.md` file linked to the project. Furthermore, a size of at least 100 `KB` was introduced, so that empty projects will be avoided. Lastly, the release year of the commercial product was taken into account, to rule out any potential false-positives that might arise due to other semantic meanings of the word 'terraform'. This resulted in the following entry inside the advanced search:

```
Terraform in:name,description,readme size:>100 created:>=2014-01-01
```

As can be seen in Figure 3, this query returned a total of 46,264 repositories. Subsequent examination on this set concluded that this would not be a viable data set, as Terraform can be easily omitted from the pieces of documentation that are examined by this search. As per Terraform's documentation[20], the cloud orchestrator uses HashiCorp Configuration Language (HCL) for its configuration files. While it is true that this might introduce more false-positives because HCL is used in other instances, they can be easily excluded by checking file extensions. For Terraform, HCL files require specific extensions, namely `.tf` and `.tf.json`. Consequently,

[20]https://www.terraform.io/language/files

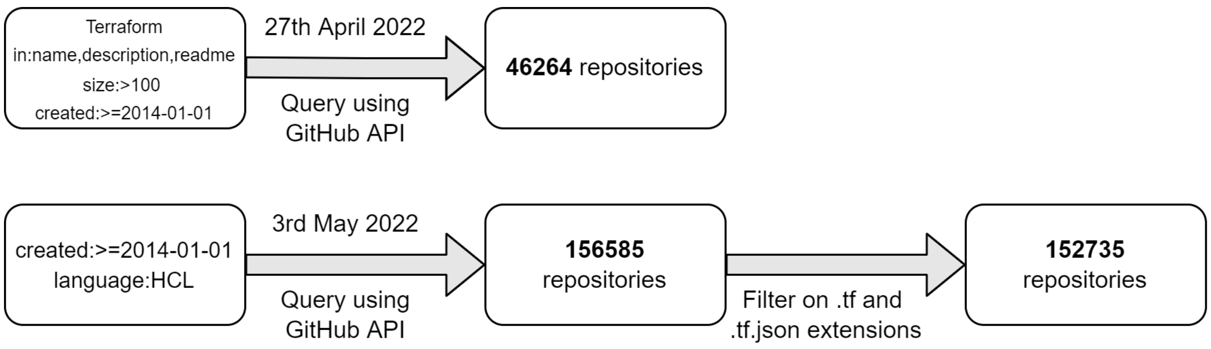


Figure 3: Chronology of our queries, containing the methods used to achieve the results.

we set up a new query applying the language criteria, but also the previously used release date. The minimum size was discarded, upon the speculation that some developers might use repositories only for saving various descriptor files and not entire projects. As such, any changes applied to them with cost in mind would potentially be relevant, but ultimately overlooked. This translates into the following query:

```
created:>=2014-01-01 language:HCL
```

The resulting output of this query is however inconclusive. Running this search is bound to return a different amount of repositories every time despite the filtering being the same. In our efforts to make the data set reproducible, we discovered that running the language query on a specific date does indeed output a reliable number of repositories. This also aids us in circumventing an issue with the PyGitHub implementation, as only the first thousand results are accessible with it. As such, we decide to run this query with the `created` filter changing throughout the interval of 2014–present on a day by day basis. These considerations can be visualized in the code snippet in Listing 3. This process formed a data set of $156,585$ repository links that require further processing, a clear improvement from the past query.

Afterwards, PyGitHub allows us to get all of the contents of the repository recursively, which is used to remove any repository that does not have any file with the expected extension for Terraform. This resulted in $152,735$ repositories, the commits of which required further examination. To save them, we have stored the URLs in a `.txt` file, separated by newlines.

One of the threats to our project's integrity that has to be underlined, is the ability to reconstruct the exact same data set as previously described. Here, some notes are presented regarding uncontrollable external factors that may affect this endeavor. Several repositories might be deleted or privatized. As underlined by Bird et al. [10], revisionist practices might take place that would prevent repositories from being picked up by our query. Furthermore, due to the issues we encountered with intervals, we did not set an end date to our query. However, we did keep track of the dates our searches occurred, as exemplified in Figure 3.

3.3 Commits mining

In this section, we show the process that was followed to extract all the relevant commits from the selected repositories. The core tool that was used is PyDriller, which was presented in Section 2.4. Its functionality is utilized to extract all the commit objects from every repository.

Firstly, the exact process consists of styling a list of financially cost-related keywords, namely: `cheap, expens, cost, efficient, bill, pay`. These will be used during the commit message filtering phase. The keywords are meant to be used with Python matching logic, therefore the root `expens` relates to words that contain it as a prefix, suffix or both, such as `expense`, `expensive` and `inexpensively`. It is worth noting that unlike [14], we do not have precedents to base ourselves onto. As such, the keyword selection is our own and its performance

```
script_urls = []
for year in range(2014, 2023):
    for month in range(1, 13):
        print(f"Scraping month {month} of year {year}")
        for day in range(1, 32):
            date = f"{year}-{month:02d}-{day:02d}"
            try:
                time.sleep(2)  # sleep for search limit
                repos = g.search_repositories(query=f"created:{date} language:HCL")
                for repo in repos:
                    time.sleep(0.2)  # sleep for core limit
                    with open("hclURLs.txt", "a") as file:
                        file.write(f"{repo.clone_url}\n")
                    script_urls.append(repo.clone_url)
            except RateLimitExceededException:
                print("Rate Limit Exception reached!")
            except:
                print(f"Skipping: {date}")
```

Listing 3: Python code for repository mining

Item	Description
Commit hash	Unique identifier of a commit
Commit message	Short description of the changes that have been made within the commit
Commit author	Name of the author of the commit
Creation date	Date of the creation of the commit
List of modified files	Files that have been modified in the commit

Table 1: Data retrieved per commit.

had to be evaluated throughout the research itself.

We then create a list of relevant repositories, the elements of which will contain (Python) dictionaries meant to represent each repository. Each dictionary will hold the repository URL and a list of relevant commits, which, in turn, are commit dictionaries themselves.

We begin by traversing all the repositories' URLs from which we create a `Repository` object with PyDriller. For each commit object in the repository, if a commit message is present and also contains any of the cost-related keywords, it is then taken into consideration. Supposing that the commit has been deemed relevant, every file that was modified is saved and a commit dictionary is created. The contents are described in Table 1.

Out of all these elements, `author` and `creation date` will eventually never be utilised in the following processing steps. Originally, it was our intent to use the `author` field to determine a group of cost-experts when it comes to cloud orchestrators as per [11]. Nevertheless, this did not pan out, partly due to our schedule limitations, but also the limited number of commits per projects. The generous selection of entries in the dictionary was also justified by the extremely lengthy running time of the whole process, as any adjustment would have required an unacceptably extensive additional computation time. Once every commit has been analyzed, a repository dictionary is created and the full listing of said commits is added to it. If the repository dictionary contains an empty commits list, it is discarded as it is clearly not relevant.

While PyDriller accounts for most errors that might appear during the mining process, as noted in Section 2.4, exception handling is still necessary as some of the repositories might no longer be reachable during the re-

```json
{
  "no_of_repos": 1278,
  "repositories": [
    {
      "name": "https://github.com/user/terraform-project.git",
      "commits": [
        {
          "id": "a86d89369aaf5a20c1e4d8415a8a771aa7d12345",
          "msg": "provision a droplet with cheapest price",
          "author": "user",
          "date": "2020-03-29 16:02:32+11:00",
          "modified_files": [
            "main.tf",
            "outputs.tf",
            "variable.tf"
          ]
        }
      ]
    },
    ...
  ]
}
```

Listing 4: JSON file containing the commits results (anonymized example)

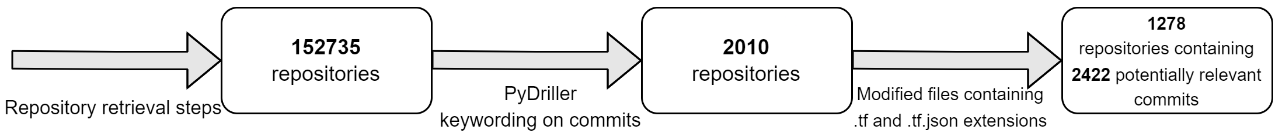

Repository retrieval steps
152735 repositories
PyDriller keywording on commits
2010 repositories
Modified files containing .tf and .tf.json extensions
1278 repositories containing 2422 potentially relevant commits

Figure 4: Chronology of the commit extraction process

trieval process or might have been deleted in the meantime. In this case, a generic exception is caught and the current URL is stored in a separate text file for later examination to determine its availability. The overall process is illustrated in Figure 4.

The commit extraction process, as a whole, has been rather time consuming. Given our schedule constraints, we created four different Datalore notebooks with the same exact mining script, but with different URL list slices. In spite of the workload division, it took roughly four days to finalize the commit extraction. The retrieved data was eventually converted in JSON format and unified in a single JSON file. The final step of the operation consisted in further filtering the results to only take into account commits that actively modify `.tf` and `.tf.json` files. To achieve this, we simply iterated through all the commits previously extracted and eliminated the ones that do not interact with such files. The figures that we obtained from the extraction process are exactly 2,422 relevant commits out of 1,278 repositories. We can see in Listing 4 a visualization of the JSON file containing our results. Nevertheless, the data set made public through this research, found in the repository mentioned in Appendix A, has been altered to remove any personally identifiable information.

An intermediary step that is performed regarding commits is the marking of forks. As the commit hash was retrieved, which is unique between the various entries, we were able to spot forked ones simply by finding duplicates. They were not discarded however, but rather marked accordingly as will be discussed further in Section 4.1. This uncovered 387 commits that belong to forks repositories.

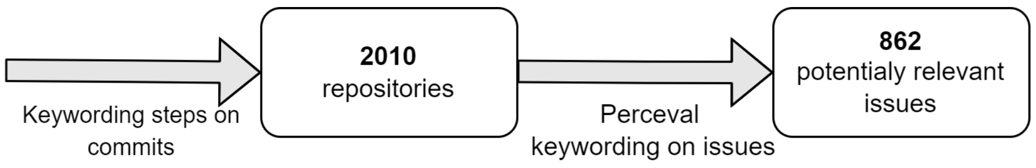

Keywording steps on commits
2010 repositories
Perceval keywording on issues
862 potentialy relevant issues

Figure 5: Chronology of the issue extraction process

3.4 Issue trackers mining

In this section, we discuss the process that was followed to extract all the relevant issues from the selected repositories. The main tool used in this section is Perceval from GrimoireLab, previously presented in Section 2.4. Perceval is part of the GrimoireLab toolkit and facilitates the mining of issue trackers.

Before looking into our main approach, it is worth mentioning our first attempt for recovering relevant issues, which has been scrapped after analyzing the poor results that it yielded. We noticed that some of the commit messages were referencing issues using their characteristic notation of the # symbol, followed by some numbers. We decided to manually find every instance of issue references and we saved the corresponding URL for later analysis. Unfortunately, every single occurrence turned out to be a pull request which are completely out of the scope of our research.

In our final approach, found in Figure 5, the process begins with iterating through all the repositories' URLs that were previously extracted during the commit mining. These are repositories that are known to contain cost-related keywords regardless of their relevance. We then provided to Perceval the repository owner username, the repository name, both extracted from the URL, and a GitHub API token. What Perceval then returned was a list of `Issue` objects that contain every single detail pertaining the issue. It was at this point that its content had to be evaluated to determine its relevance. If the title, body or any of the comments contained one of the cost-related keywords presented in Section 3.3, then the issue was deemed of interest to our research and a dictionary to represent it was created. The data saved in this dictionary is described in Table 2 and Table 3. Not all entries might necessarily be present depending on the specific issue. `comments_data` contains different subcategories of which only the relevant ones have been listed. It is also worth mentioning that most of the dictionary entries were never used during analysis. Nevertheless, once a Perceval request is performed, there is no disadvantage in picking as many elements as possible. Therefore, we thought of saving as much content that could be useful in future research. It has to be noted, however, that as above, the data set available in our repository found in Appendix A has been altered to remove personally identifiable information.

As Perceval was also including pull requests in the source of issue trackers that were mined, we collected them initially with the intent of performing further analysis on them. However, this effort was quickly dropped due to the challenges that pull requests presented in manual labeling. For instance, the keyword that vetted the pull request for saving, was usually found inside a tremendously large changelog. Furthermore, there were several instances of bots updating packages in repositories, that triggered multiple pull requests with the same message across most of the examined projects. Last but not least, Perceval is affected by the GitHub API rate limit, thus requiring timeouts between requests to prevent it from being exceeded. Should the threshold have been reached, the repositories the query failed on were saved and reexamined after the initial process.

We obtained exactly 862 relevant issues requiring manual classification.

3.5 Labeling

This section covers our work organization and structure concerning the manual classification of both the potentially relevant commits and issues. Envisioning the level of categorization that our data required, it became

Column	Description
Title	The issue title in string format
HTML_URL	The address to the specific issue
User	The address to the user's GitHub account
Category	Type of the entry: either pull request or issue
Labels	Any user-defined labels to categorize the issue
Closed_at	Date and time of closing the issue, if applicable
Assignee	User assigned to the issue
Assignees	Users assigned to the issue
Body	The main text of the issue
Comments	The number of comments in the particular issue
Comments_data	Field containing multiple comment objects

Table 2: Metadata information for issue tracker dictionary.

Column	Description
HTML_URL	The address to the specific comment
ID	The unique comment id
User	User details in its own subcategory
Body	The body of the comment in string format

Table 3: Metadata for a comment object instance inside of an issue.

apparent that an in-depth labeling system had to be defined. As such, we devised a taxonomy of our data set, which constitutes a hierarchical classification of entries, in our case commits and issues, into groups or types.

The main need for such categorization comes from the fact that simply characterizing commits and issues as discussing cost-management does not offer the amount of granularity desired. In other words, the research conducted needed to pinpoint the changes involving cost (decrease or increase), as well as explicitly identify, where possible, the level of infrastructure that prompts that alteration. In a similar fashion, even the irrelevant label required its own subcategories. After all, it is important to understand in what circumstances the keyword selection failed.

Despite Sections 3.3 and 3.4 pointing out the wide variety of information that had been extracted regarding specific entries of the data sets, it should be stated that this process did not make use of all details gathered. In terms of commits, we only assessed the commit message provided by the developer. Specific cases where we were in doubt about labeling meant that we also looked at the modified files of the item. When it comes to issues, we found ourselves quickly overwhelmed by the abundance of text contained by title, main body and potential comments that we had to inspect. Therefore, we designed the following regex, to extract exactly twelve words before and after the keywords found in the available issues:

```
r'((?:\w+\W+){{,12}})(\w*{word}\w*)\W*((?:\w+\W+){{,12}})'.format(word=keyword)
```

This way, we were able to inspect the context that the specific keyword appeared in without analyzing the entire issue. If we failed to categorize the issue based on the snippet, the whole text had to be examined. Still, this drastically lowered the amount of time needed to label the aforementioned data set.

Building the taxonomy also uncovered its fair share of challenges. For starters, we had no prior knowledge of what both of the data sets (issues and commits) contained regarding discussions. Therefore, to get an idea

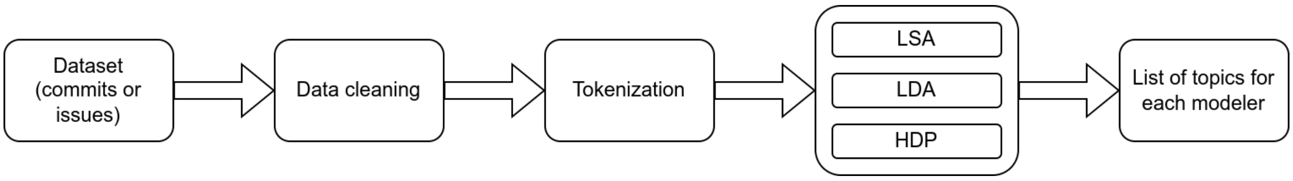


Figure 6: Visualization of the topic modeling process

```python
def prepare_document(doc):
    # Remove MarkDown and HTML
    clean_doc = markdown(doc)
    clean_doc = ''.join(BeautifulSoup(clean_doc).findAll(text=True))

    tokens = [token.to_dict()[0]["lemma"]
                for token in nlp_pipeline(clean_doc).iter_tokens()
                if token.to_dict()[0]["upos"] in UPOS and not token.to_dict()[0]["text"]
                                                        in STOPWORDS
          ]
    return tokens
```

Listing 5: Corpus Preparation helper function ran for each document of the data set

about what our labels could be, we first had to individually go through a sample of the entries. As such, the taxonomy presented in Section 4.1 had been constantly evolving along with the progress of the labeling process. We would individually take on a different sample of pre-determined size. After trying to describe the entries as in depth as possible, labels would be decided upon in a meeting and the examined samples would be updated to the new set. Following these decisions, we would continue classifying entries independently, scheduling new discussions should one of us propose changes to the taxonomy. Due to time constraints and the extensive amount of text we had to traverse, we estimate that each of the researchers categorized a third of the data sets. In case of uncertainty, the specific items were marked and a label for them was discerned in a final gathering. In the event any of the parties did not agree with a verdict, it was settled with a two to one vote.

Following the process portrayed above, but also by utilizing the taxonomy presented in Section 4.1, we have marked 538 commits out of 2,422 as discussing cost, and 206 issues out of 862 respectively.

3.6 Topic modeling

This section covers the methods we have used to involve topic modeling in our research, as well as how we reached the results to be described in Section 4.4. Initially, we started running experiments concerning topic modeling due to our motivation of shortening the manual labeling process that was required with issue trackers. However, our results at the time were not very successful. So we discarded the possibility of using topic modeling for reducing our workload rather quickly. Still, after the analysis was finished, we were left with a topic modeling script and a data set of cost-related discussions. Therefore, we continued this endeavor in order to uncover areas of interest or potential keywords that can be used in further research.

The process we followed is summarized by Figure 6. The first step of this process involved data cleaning of the corpus used, exemplified in Listing 5. For both commits and issues, we progressed in the same manner. As GitHub content contains a mixture of Markdown and HTML code scattered throughout the text that we mined, we had to remove the former. For that, we made use of two third-party python libraries: Markdown and BeautifulSoup4. The former is used to convert any markdown syntax to HTML, so that the latter can clean both at once. We also devise a list of stop words which is partly borrowed from the Gensim library. During this process, we have also taken the liberty of adding additional words on top of the the existing set, so that we can further fine-tune the results.

Only after plain text is left, we can now use more advanced tools to convert human language text into lists of sentences and words. The preparation of each individual document part of the corpus relies heavily on the Stanza neural network NLP pipeline. The modules that we make use of are as follows:

1. Tokenizer: Segments each document into sentences containing lists of tokens. These tokens contain underlying information regarding the syntactic word(s) they describe.

2. Part-of-Speech & Morphological Features: Tasked with labeling each word with its respecting Universal Part-of-Speech tag. In our code, we account for nouns (proper or otherwise, although specifically mentioned), adjectives, adverbs and verbs. If constructs happen to be of an unmentioned morphological type, we simply discard them from our analysis.

3. Lemmatization: Converts each word back to its lemma form. For example, conjugated forms of verbs are returned to its infinitive form. At the same time, indefinite articles are brought to its simpler form ‘a’, disregarding speech conventions. This is especially useful when calling helper functions such as the Gensim `doc2bow`, that suggest such steps to improve the validity of the bag of words creation.

Once the corpus is ready, we convert it into a Gensim Dictionary, a bag of words, and also build a TF-IDF model. These are used as arguments for the LSI, LDA, and HDP helper functions provided by the aforementioned library. As not much is known regarding the structural nature of the examined data, we could not weigh the potential advantages of a single approach. Therefore, we settled for attempting to retrieve information from all three methods. Each respective model can be queried to retrieve a number of topics, as seen in Figure 6. The requested value can also have an impact on the quality of the results that we get, so after a few tests we settled for 100, taking into account the size of our corpus. Finally, the various identified topics can now be inspected and discussed upon.

3.7 Sentiment analysis

This section follows the design and execution of the sentiment analysis experiments that have been implemented after the manual labeling of the data. As mentioned in Section 2.5, in order to obtain the best possible results, it is advised to manually train a model from scratch, or build on top of an existent one. Considering the time constraints and focus of this project, we have decided to compare available pre-trained sentiment polarity models in order to assess whether automating or improving the labeling process can be done using these algorithms. After doing preliminary research, we have decided to use three modern models that are considered to be state-of-the-art technologies according to the scientific works presented in the previous chapter. These models are TextBlob, Vader, and Stanza, all three of which provide well-documented APIs and resources for developers to use. We decided to also use the older SentiStrength-SE model, as discussed in the previous chapter, because it was used and cited in many scientific works focusing on sentiment analysis in software engineering. In the rest of this section we will go in-depth with how we applied each of these models on our data set. A visualization of the sentiment analysis experiments can be seen in Figure 7.

The population sizes for commits and issue trackers differed, but given our time constraints, we decided to only label the first 100 commits and issues, that contain cost management information, as listed in CSV. The task of labeling the commits was easier to manage by only one person as the text of a commit is usually short. At the same time, the context of a commit is reduced, resulting in many cases where even the manual labeling was uncertain. For issues, we followed a similar approach to the cost classification, where we split the workload into the three members of the team. Generally, issues have both a body and comments so there is more context to use for defining an overall sentiment. However, in cases of ambiguity we decided the label as a team.

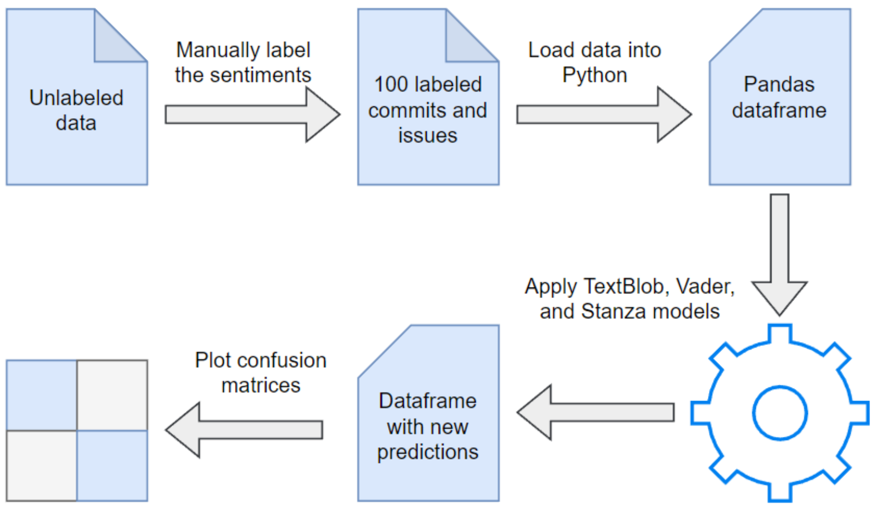

Unlabeled data
Manually label the sentiments
100 labeled commits and issues
Load data into Python
Pandas dataframe
Apply TextBlob, Vader, and Stanza models
Dataframe with new predictions
Plot confusion matrices

Figure 7: Visualization of the sentiment analysis process.

```
# Loading the original json for the complete body
with open('terraform_issues_updated.json', 'r') as f:
  original = json.load(f)

# Filter the title and body of the issues/PRs
issue_bodies = []
for repo in original["repositories"]:
    # For each issue in the repository
    for issue in repo["issues"]:
        for index, row in issues_df.iterrows():
            if issue["html_url"] == row["html_url"]:
                row["body"] = ""
                # Merge title, body and comments and save them in the old field.
                if issue["title"] is not None:
                    row["body"] += issue["title"]
                if issue["body"] is not None:
                    row["body"] += "\n" + issue["body"]
                for comment in issue['comments_data']:
                    row["body"] += "\n" + comment["body"]
```

Listing 6: Python code for merging the title, body and comments of an issue into one text.

The general workflow of the experiments was almost the same for both commits and issues, the only difference being in the data preparation stage. For issues we had to merge the title, body and comments into one body of text so we could apply the model (Listing 6). Commits on the other hand, only have one possible commit text message so applying the API was straight forward.

The next step, which was the same for issues and commits, included cleaning the data of any Markdown or HTML (Listing 7). This was achieved using two third-party python libraries: Markdown and BeautifulSoup4, similarly to the case of topic modeling discussed above.

For the polarity classification, both TextBlob and Vader have similar APIs which allowed for a quick and easy

```python
# Remove any Markdown or HTML embeddings
for index, row in issues_df.iterrows():
    row["body"] = markdown(row["body"])
    row["body"] = ''.join(BeautifulSoup(row["body"]).findAll(text=True))
```

Listing 7: Python code for cleaning the data off Markdown and HTML.

```python
# Loop through the data and apply sentiment analysis on each body of text.
for index, row in issues_df.iterrows():
    polarity = TextBlob(row["body"]).sentiment.polarity

    # I use the default thresholds given in the TextBlob documentation
    if polarity < 0:
        sentiment =  "Negative"
    elif polarity == 0:
        sentiment = "Neutral"
    else:
        sentiment = "Positive"

    row["Predicted SE"] = sentiment
```

Listing 8: Sentiment analysis polarity on TextBlob.

```python
# Loop through the data and apply sentiment analysis on each body of text.
analyzer = SentimentIntensityAnalyzer()
for index, row in issues_df.iterrows():
    sentiment_dict = analyzer.polarity_scores(row["body"])

    # I use the default thresholds given in the Vader documentation
    if sentiment_dict['compound'] >= 0.05 :
        overall_sentiment = "Positive"
    elif sentiment_dict['compound'] <= - 0.05 :
        overall_sentiment = "Negative"
    else :
        overall_sentiment = "Neutral"

    row["Predicted SE"] = overall_sentiment
```

Listing 9: Sentiment analysis polarity on Vader.

```
# Loop through the data and apply sentiment analysis on each body of text.
for index, row in issues_df.iterrows():
    doc = nlp(row["body"])
    # Initialize the document sentiment
    sentiment = 0
    size = len(doc.sentences)
    # Sum the sentiments
    for i, sentence in enumerate(doc.sentences):
        sentiment += sentence.sentiment
    # Divide and round the sum of sentiments over the number of sentences
    sentiment = sentiment / size
    sentiment = round(sentiment, 0)

    if sentiment == 0:
        sa = "Negative"
    elif sentiment == 1:
        sa = "Neutral"
    else:
        sa = "Positive"

    row["Predicted SE"] = sa
```

Listing 10: Sentiment analysis polarity on Stanza.

implementation as shown in Listings 8 and 9. Both APIs return a result on a continuous scale meaning there is room for fine-tuning the model by setting the threshold for each sentiment. As this requires time to test different values, we decided to use the recommended values by the API's documentation.

Stanza required a different approach when predicting a sentiment for either a commit or issue text. As mentioned in the background section, this model splits a textual document in sentences and attributes each one of them a sentiment. While this can improve the level of the analysis and the quality of the prediction, for our use case we wanted one sentiment for the whole text as it would simplify the process. In order to achieve that, we decided to add the sentiments, which Stanza already returns as integers, and divide by the total number of sentences in the document (Listing 10). While this approach works, it is not the most desirable one, as it influences the fact that Stanza will mostly predict neutral or negative sentiments because of the generalization created by our heuristic. This is one design choice that can be improved in future research as more advanced heuristics that do not induce bias could exist. Furthermore, even though Stanza returns the predicted sentiment as an integer, the developer cannot set any thresholds for the outputs which can only be 0 for negative, 1 for neutral and 2 for positive.

The final model that we tested was SentiStrength-SE, which as the name implies, has been built on top of SentiStrength, an already popular sentiment analysis model in Java. The model does not provide the user with an API, but comes packaged as an executable JAR file with a user interface. The disadvantage of this model is that the data workflow that the other models used above cannot be reused. The reason for this is that the analysis tool only receives data in its own specific format. In order to transform our current data into the required model, we would have had to write a new script that could automate the process. Because of this required process and the sub optimal results that will be discussed in the results section, we have decided to drop the usage of SentiStrenght-SE from our experiment.

Lastly, in order to assess the efficiency of these models, we have made usage of popular Python libraries such as sklearn and seaborn to be able to generate and visualize confusion matrices. With the help of these we can express the accuracy of the model while also getting insights into the way the algorithm predicts the sentiments.

4 Results

This chapter discusses the results that followed from the work done in Chapter 3. Section 4.1 presents the taxonomy of cost-related information developed during the labeling process as discussed in Section 3.5. Sections 4.2 and 4.3 go over the insights discovered from this process of labeling the commits and issues, while Sections 4.4 and 4.5 discuss the results of the topic modeling and sentiment analysis experiments.

Throughout this section, we use plenty of visualizations to help explain our findings. All the scripts and data needed to recreate these visualizations, and the output files themselves, can be accessed through the link in Appendix A.

4.1 Taxonomy

This section defines the various labels used in building the taxonomy established for our data set.

Subsequent to the efforts described in Section 3.5, our final taxonomy is split in two main labels: `cost` and `other`. We begin by defining how `cost` is divided in subcategories and their relative meaning and relevance. The three main sub-labels are: `saving`, `awareness`, and `increase`.

Saving refers to any situation where a direct cost cutting action took place, or in other words, when there is a defined and confirmed monetary advantage or cost saving maneuver. Awareness refers to any instance where a cost conscious decision is taken. This means that various options are weighed given their pros and cons or features are introduced to monitor cost. In short, any situation where cost is discussed and acknowledged, but saving money is not necessarily the main focus, is marked as such. Increase is quite self explanatory, as we needed a label which encompasses any necessary cost increase discussed or declared. A description of the taxonomy labels can be seen in Table 4:

Label	**Description**
Alert	Refers to billing alarms enforcing an upper threshold on costs
Instance	Computational instances, such as the Amazon AWS t2.micro
Storage	Storage solutions, such as Amazon gp2, as well as storage practices
Domain	Domain Name System and IP addresses costs
Area	Server or instance geographical location related costs
Provider	Comparison between service providers such as Amazon, Azure, Google, and others
Feature	Miscellaneous label that can contain logging, load balancers or usage of third party libraries
Billing_mode	Type of billing plan being used, such as on-demand for development or normal plan for production
Cluster	When developers mention two or more instances working together and the derivative costs
Policy	General rules implementation to prevent excessive charges
Unknown	Situations where cost management happens, although the commit message is not descriptive enough
Networking	Standalone expresses anything related to networking not covered by other labels in its hierarchy
NAT	Network Address Translation gateways have costs that are often discussed
VPN	Local access to a remote machine might require a VPN or VPC with its related costs

Table 4: Description of cost-related labels.

To mark the irrelevant commits or issue, we use the label called `other`. This facilitates in the discovery of cost-unrelated situations picked by our keyword selection in order to potentially give insights on further filtering.

We have a selection of sub-labels that account for various basic operations carried out by the developers:

- `addition`
- `removal`
- `bug_fix`
- `refactor`
- `performance`

The context for these sub-labels can be found in Table 5:

Label	**Description**
Tags	Changes to the Terraform tag system, used to reference specific modules and elements
Modules	Referring to a Terraform module that consist of several configuration files in a single directory
Payload	Related to transmission of data and capacity of a payload
Vars	Variables, their name might be picked up by the keyword selection
Flags	Similar to variables but these are of boolean type
Cost_calculator	Although cost-related, our research does not discuss cost calculator development
Billing	Variables or changes that refer to setting up billing accounts
Docs	Changes to documentation
Unknown	Commit message or issue is not descriptive enough
False_positive	Keywords being found in names, homonyms in services, phrases or just usage of foreign languages

Table 5: Description of non-relevant (other) labels.

During the manual labeling process, we delimited the various labels using a dash. An example of a cost-related situation that implies saving money by using a cheaper instance will be categorized as `cost-saving-instance`. This ensures that splitting the label on the dash can return us all the elements in their hierarchical order. Several labels can be used for each entry in case more than one concept has to be expressed. This is done by enumerating them with commas. Consequently, this means that the amount of labels obtained during the manual process does not correspond to the number of entries in our data sets.

When it comes to forks, their labeling has also been taken into consideration. In this case, we simply use the same taxonomy explored above with the sole exception that the root, `cost` or `other`, is swapped with `fork`. This way, it is still possible to apply some sort of categorization so that we can provide additional statistics regarding this type of repository. On the other hand, issues are not affected, as they have no forks. Therefore, this labeling is relevant only for commits. We can observe a visualization of our taxonomy in Figure 8.

4.2 Commits analysis

This section will describe the results obtained from the commits mining process described in Section 3.3 along any insights deducted from our results. Additionally, the manual labeling process discussed in Section 3.5, gave us figures regarding the occurrence of the cost-related keywords as well as the labels from our taxonomy, which will be explored.

When it comes to keywords, `cost` is unarguably the most popular one with 297 occurrences within cost-related commits. Its popularity was expected given the scope of our research and the way it fits syntactically within sentences to describe cost-related scenarios. In stark contrast, `efficient` did not have a single occurrence among the 538 keyword matches throughout the entire analysis. While surprising at first, this result is quite logical given the general syntactical use of the keyword to describe performance-related situations more than monetary ones. The keyword `pay` was also seldom present, with only 23 occurrences. What is worth noting

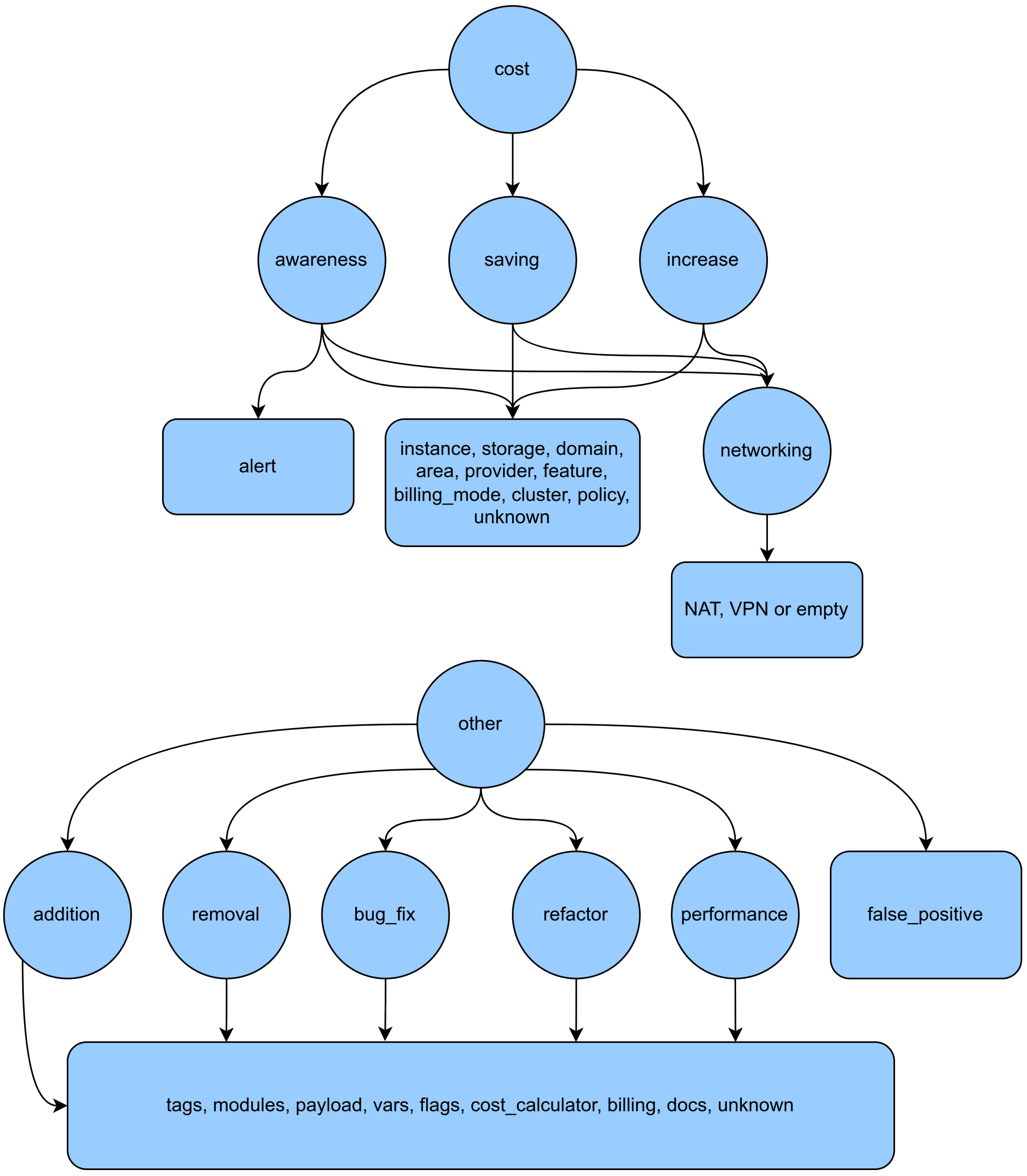

cost
awareness
saving
increase
alert
instance, storage, domain, area, provider, feature, billing_mode, cluster, policy, unknown
networking
NAT, VPN or empty
other
addition
removal
bug_fix
refactor
performance
false_positive
tags, modules, payload, vars, flags, cost_calculator, billing, docs, unknown

Figure 8: Visual representation of the taxonomy

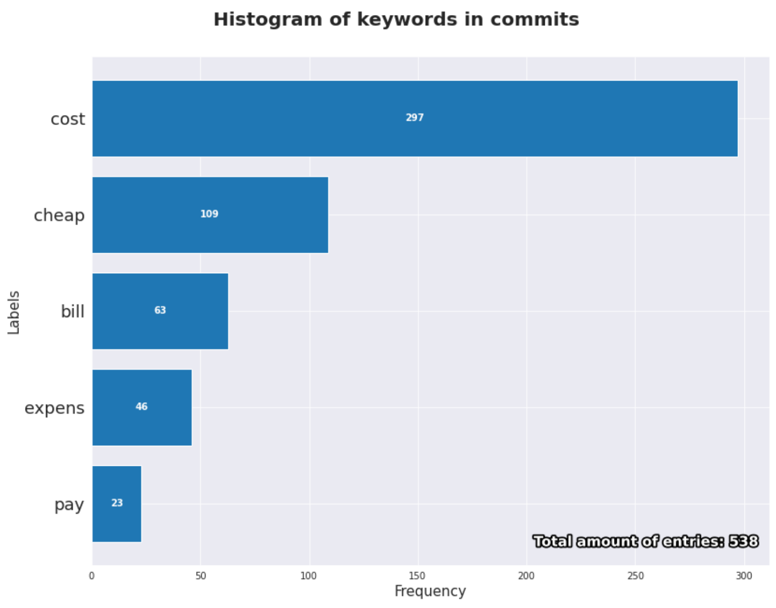


Figure 9: Number of commits containing the keyword at least once

though, is the disproportionate amount of false positives associated with it because of its affinity to the word `payload` and service names. The keyword `bill` also occurred in several false positives given its unfortunate proximity to proper names that might appear colloquially (e.g. “Bill Smith”), or in e-mail addresses, as well as module names and billing account variables. Nonetheless, it was not associated with as many false positives as `pay`. We can observe a visualization of the keywords’ popularity in Figure 9.

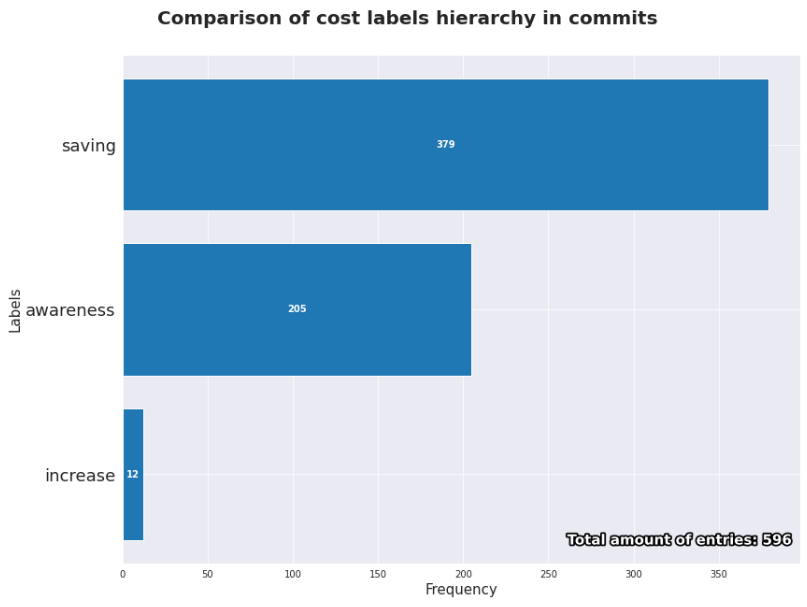


Figure 10: Number of commits labeled within the cost hierarchy

Regarding to the taxonomy labels, we start by looking at the hierarchy rooted in `cost`. As we can derive from Figure 10, `saving` is the most popular label with 379 occurrences. The changes in a commit are summarized in a message that is meant to be succinct in nature. We assume that more in-depth discussion on the matter happens somewhere else and the end result of said discussion is performed with a commit. It is only logical

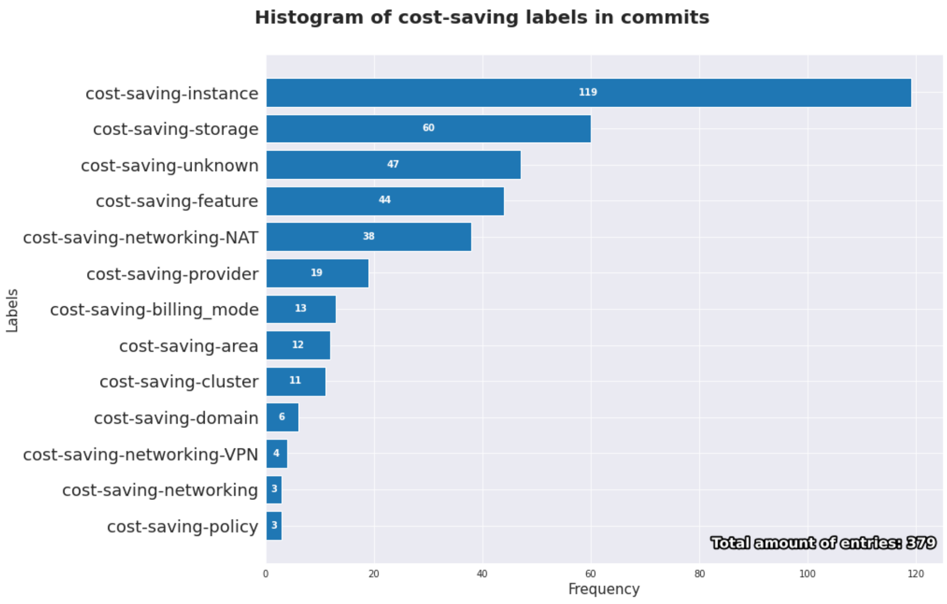


Figure 11: Number of commits labeled with cost-saving

that positive improvements are found within commits, and `saving` implies a definite cost-saving measure being implemented. For `awareness` we have 205 occurrences. This is a balanced amount and it generally involves developers attempting to estimate costs in the modified cloud infrastructure as well as add preventive measures to manage cost dependent components and functionalities. The label `increase` is rather disappointing with only 12 occurrences out of the 596 total. We cannot derive any particular insight given such few occurrences. We can only assume that it is rather uncommon to discuss and apply cost increasing measures.

We can now take a look at the specific sub-categories for `cost-saving`, as shown in Figure 11. The most common one is `instance` with 119 occurrences. This is no surprise to us. After all, balancing an entire project on a suitable instance is vital to avoid unnecessary expenses on unused resources. Additionally, changing instance type is some of the easiest settings to manipulate, where the name of the instance is simply swapped for another one. The second most popular one is `storage` with 60 occurrences. In the same fashion as `instance`, `storage` follows the same logic where a suitable storage solution is selected to avoid wasted resources. Right below we have `unknown`, which, as stated in Section 3.5, covers any situation where the developer declares implementing cost-saving measures that are indiscernible to us given their technicality or lack of description. With 47 occurrences, we can only say that a less time-constrained analysis might have reduced this figure to a lower amount.

For `cost-awareness`, we can see our results in Figure 12. The most common sub-category is `instance` with 49 occurrences. We can assume that speculating about future upgrades and scaling of the project infrastructure comes naturally in the `awareness` sub-label. The sub-category `alert` is the second most popular with 45 occurrences. In this case, discussing the implementation of alerts to prevent excessive expenses as well as their thresholds comes logically in the `awareness` sub-label.

With regards to `cost-increase`, we can see from Figure 13 that we have single digit occurrences of its sub-categories. Any assumption taken from these figures would be beyond any speculation and rather unsubstantial in content, therefore we cannot derive any conclusion from this specific sub-label.

As mentioned in Section 3.5, we also categorized all the irrelevant commits under the `other` label. We can see a visualization of its top five most relevant sub-categories in Figure 14. For clarity, the total amount of 1,040

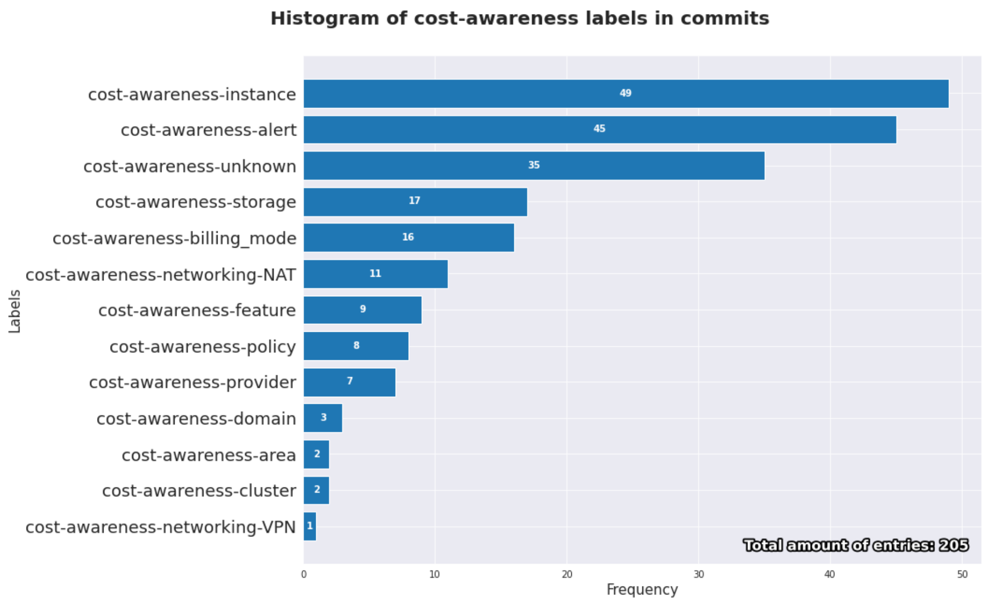


Figure 12: Number of commits labeled with cost-awareness

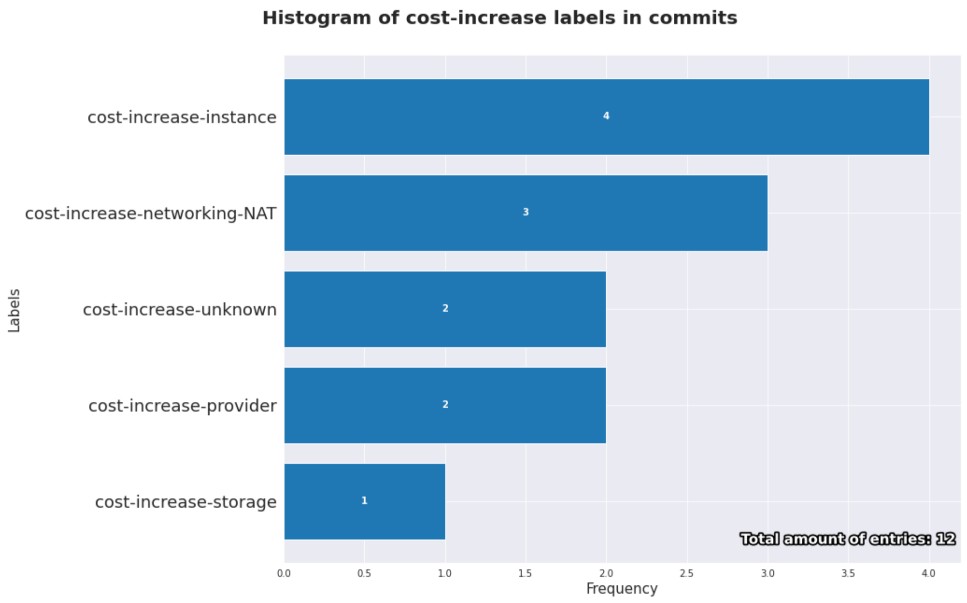


Figure 13: Number of commits labeled with cost-increase

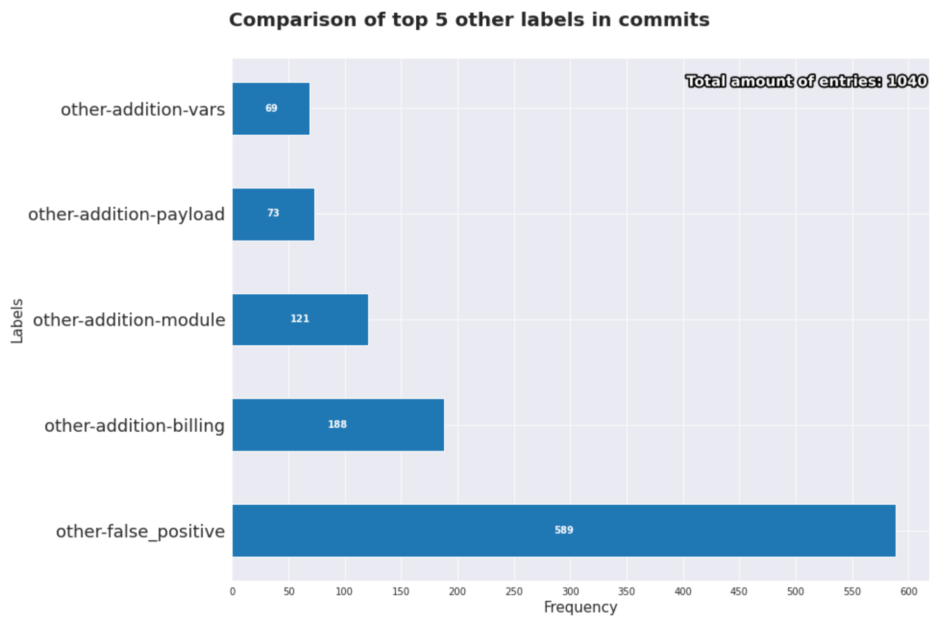


Figure 14: Top 5 ‘other’ labels in commits

is only for the top five, the complete amount is exactly 1,531. As we can see, `false_positive` has the largest amount of occurrences with 589 instances. This shows the sheer amount of situations where our keywords are matched without any context whatsoever and no other categorization can be applied to determine in what field this occurs. Right after, we have `addition-billing`, which, as stated before, matches our pattern of the keyword `bill` often mismatching. In this particular instance, this sub-category represents all the times a billing address is added.

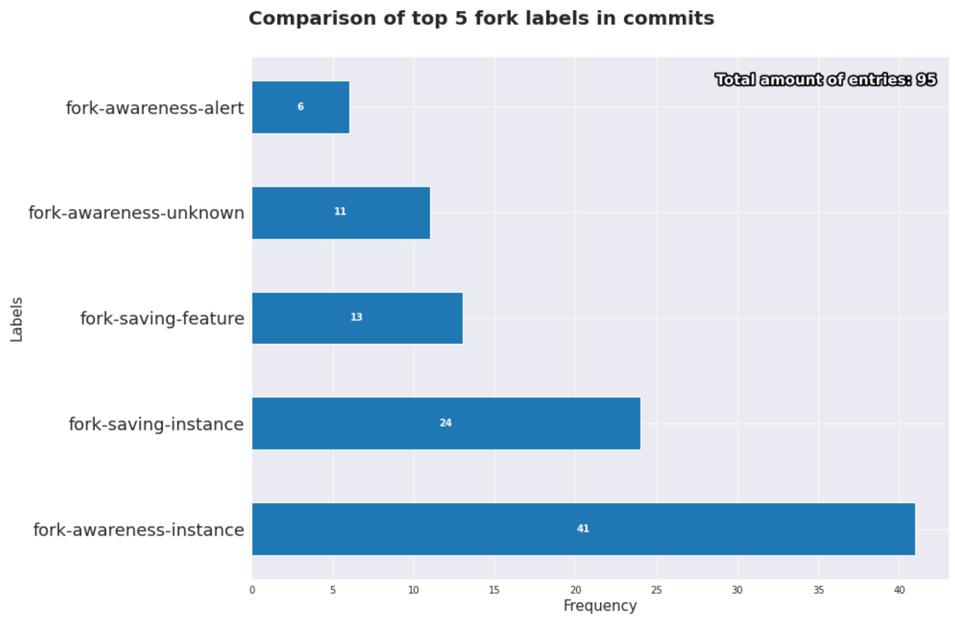


Figure 15: Top 5 ‘fork’ labels in commits

In our analysis we found a total of 124 forked commits discussing cost-management out of the initial 387. We also styled a visualization of the top five cost-related fork labels present in commits, which, as stated

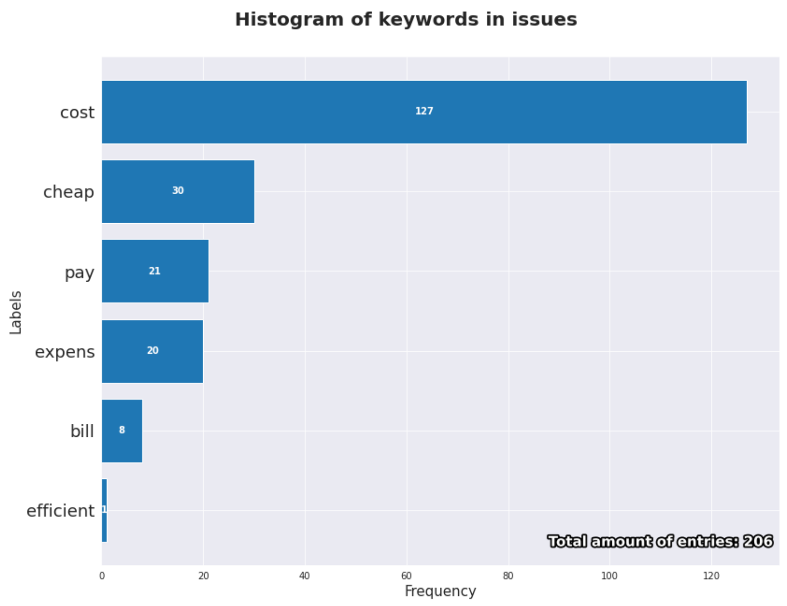


Figure 16: Number of issues containing the keyword at least once

in Section 3.5, is a taxonomy applicable only to this data set. It should be noted that the aforementioned histogram displays only 95 of them, for improving clarity. As we can see in Figure 15, the most common fork sub-category is `fork-awareness-instance` immediately followed by `fork-saving-instance`. The only fact that we can derive is that commonly forked repositories most-likely contain some underlying changes regarding cost-management. Still, we cannot possibly discern whether these changes influence forking popularity.

4.3 Issue tracker analysis

This section discusses the results of the labeling process performed on issue trackers. In addition, where possible, we will also point out contrasts and similarities between the values shown in the case of issues and commits.

Taking a look at keyword frequency for the vetted issues in Figure 16, we notice that `cost` is once again the most common, with 127 occurrences. `efficient` returns the least amount of results, largely appearing in the context of performance. The keywords `pay` and `bill`, while also returning some positives entries, have also accounted for most false positives. We assume this is because most of these words have been extracted from code snippets in which variables are built with them.

By examining the labels attached to `cost` in Figure 17, we can tell that `awareness` is brought up more in the entries we have examined. As opposed to commits, where `saving` was predominant, one can attribute this to the fact that issues offer more of a platform for discussion, rather than a monologue describing actions. In this way, developers are more prone to discuss the way their infrastructure affects their cost, rather than develop concrete methods that can incur savings. The frequency of `increase` is once again minimal, most likely due to lack of documentation.

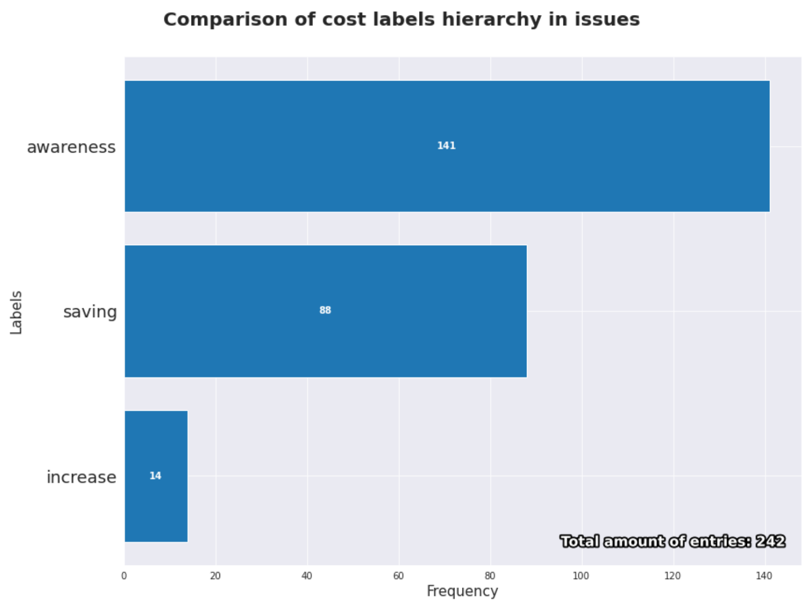


Figure 17: Number of issues labeled within the cost hierarchy

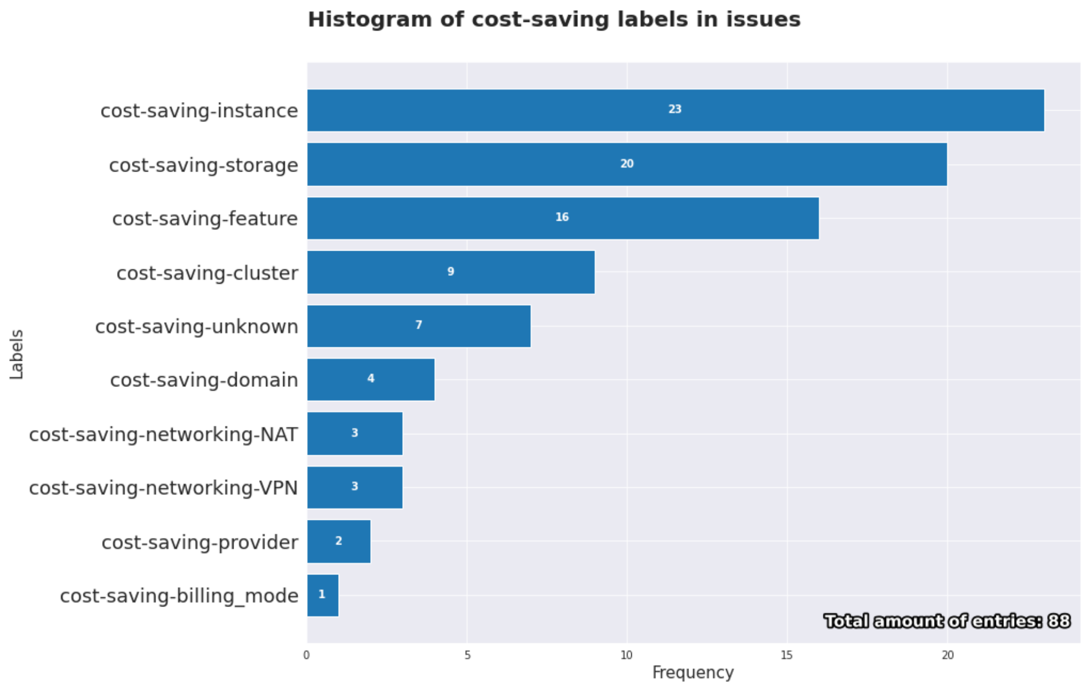


Figure 18: Number of issues labeled with cost-increase

Figure 18 uncovers similar results about `saving` when compared to its commit `cost-saving` counterpart. The labels of `instance` and `storage` are the two most popular, which hint at the fact that these might be the most accessible places where spending can be decreased. Due to more information being found in discussions, the `unknown` tag is lower in ranking when it comes to issues. This allowed us to better pinpoint the level of cloud infrastructure where cost-saving happens.

In the case of `cost-awareness`, we can observe its values in Figure 19. This time, `feature` is the most popular sub-category with 31 occurrences. As stated in Section 3.5, `feature` describes any kind of measure or action taken outside of the other labels. Its affinity with `awareness` is to be expected as developers tend to implement

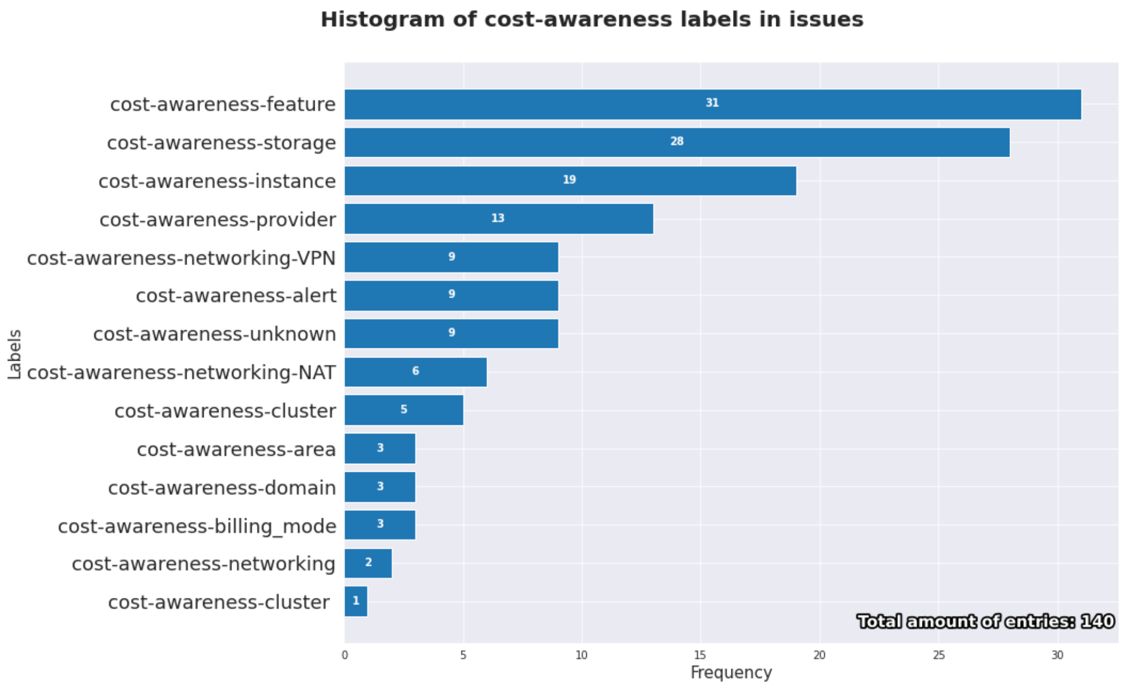


Figure 19: Number of issues labeled with cost-increase

context specific operations that might not fall into our set of labels. Its prominence within this label is also to be expected, given that issues allow for detail about more miscellaneous parts of the project, also due to their larger size. The next two subcategories are `storage` and `instance`. In many scenarios, developers tend to discuss or question their current infrastructure solutions, so issues regarding instances and storage will often consider future upgrades or potential downgrades. As opposed to the results uncovered by commits, we notice a significant decrease in the amount of billing alerts spotted. We presume that is because the addition of alarms does not require excessive effort or consideration, and it does not imply a major design choice.

In terms of `cost-increase` and `other`, the results are akin to those presented in Section 4.2. `false-positive` is still dominant with the latter, as the authors display more code snippets, output logs and use idioms such as ‘at the expense of’ and ‘pay attention to’. The bar plots for these particular labels can be seen in Appendix B.

4.4 Topic modeling

In this section, we illustrate the results obtained while performing topic modeling on the data sets containing relevant commits and issues respectively. We highlight the efficacy of the methods presented in Section 3.6, in addition to how well they correlate with the outcomes illustrated in Section 4.2 and 4.3. Lastly, we will emphasize what can be improved in this department to allow for better fine-tuned results in future research.

In order to assess our work in this sector, we are going to utilize the Gensim `print_topics()` helper function. This grants us the ability to inspect ten words out of the first four most-prevalent topics. Such a step is repeated for every used mode (LSI, LDA, HDP) on both data sets. By these means, we will be able to analyze word frequency, but also the weight on the entry in the actual topic. Furthermore, an intersection of words found across all topics will be computed, as a way of drawing insight from entries consistently present through models.

Topic Modeling for Commits

In Figure 20, one can observe some of the topics composed by the LSI model. In this case, the weights can be either negative or positive, describing how representative a specific word is of the topic it has been associated with. Due to the popularity of the keyword `cost`, we assume that a specific grouping has been created around it, illustrated by the first topic. The fact that it is mentioned in so many entries in our data set, means that the context surrounding it can also differ significantly. This is reflected by other words relating to multiple concepts such as `instance`, `nat` and `alert`, thus supporting this theory. Other topics marked `cost` as unrepresentative focusing on cloud infrastructure parts that we have also identified in Section 3.5. In this manner, we reinforce the presence of alerts through the weights attributed to `billing`, `alert`, `alarm` in the second topic. Furthermore, `instance` is also present, together with network elements such as `nat` and `gateway`. It is also interesting to note, that the last topics reinforce the presence of `reduce` and `cheap` around `instance`, supporting its popularity in relation to cost saving.

When it comes to LDA, as can be noted in Figure 21, the weights are much lower in terms of maximum values. Previously mentioned network related elements like `nat`, `gateway`, but also new entries `vpc`, `vm` are scattered throughout multiple topics. The same trend is applicable with `instance` with a mention of an instance type in the form of `t3`. The `AWS` provider features many of its components such as its key management system and load balancers (`ELB`, `ALB`). Similarly, a few of the mining keywords make an appearance, validating them in relation to cost-management: `cheap`, `expense`, `cost`. Ultimately, there is not much individuality that can be noticed in these topics, explaining the much lower weight values, as cloud infrastructure components are intermingled. However, the presence of these concepts in commits relevant to cost management is very much emphasized.

Finally, by computing the intersection of words across methods, we obtain the following word list:

```
NAT, VPC, account, add, alarm, change, cluster, cost, default, destroy, dynamodb,
enable, gateway, good, instance, move, network, price, save, size, spot, storage,
                       subnet, table, terraform, volume
```

We can see many relevant keywords such as `NAT, cost, gateway, instance, network, subnet` that are recognizable terminology when cross-referencing with the devised taxonomy. Their presence confirms that our text selection has affinity with the scope of our experiment. Furthermore, possible associations can be observed within the words, such as `NAT, VPC, subnet, gateway` relating to networking, `cluster, instance, spot` relating to instances and `volume, storage, size, dynamodb` relating to storage. At the same time, `save`, `size` are entries which display potential in being used for keywording steps in future work. This is because, while not pointing out to specific modules of the cloud infrastructure, cost management seems to encourage saving. Similarly, `size` seems to be the trade-off for optimal cost-efficiency when it comes to storage but also virtual machines and instances.

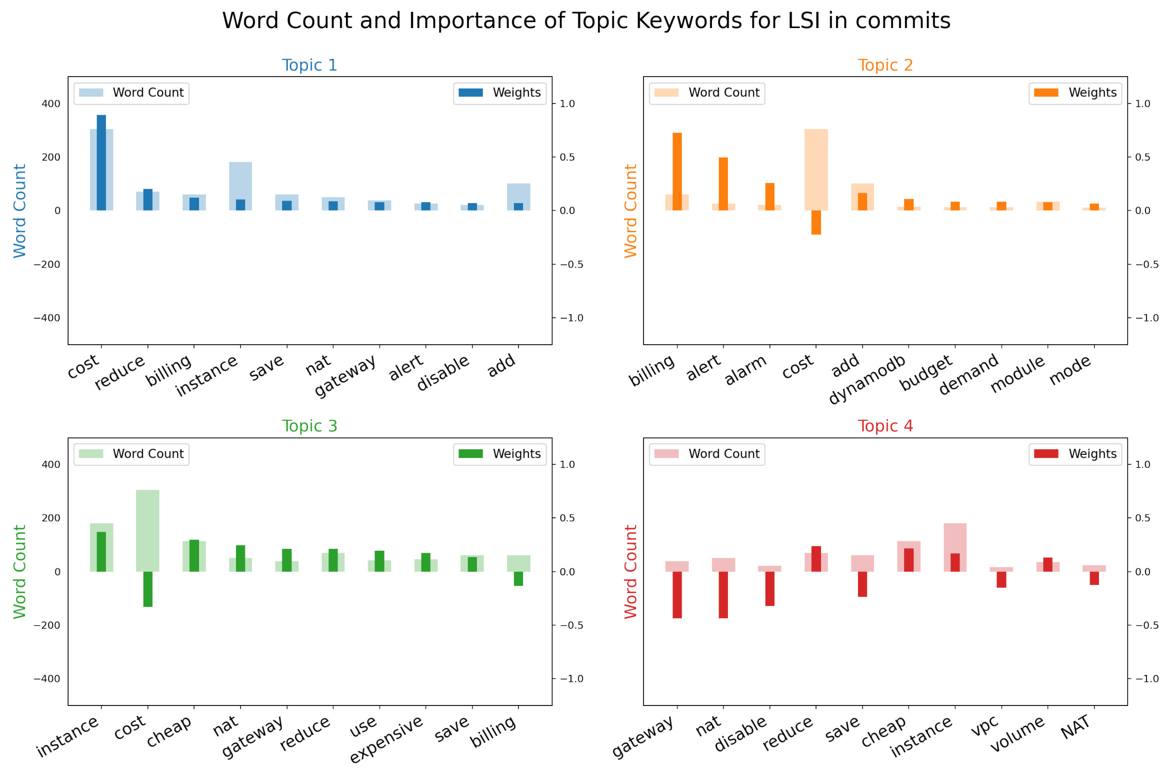


Figure 20: Visualization of the first 4 topics from applying LSI on the commit data set

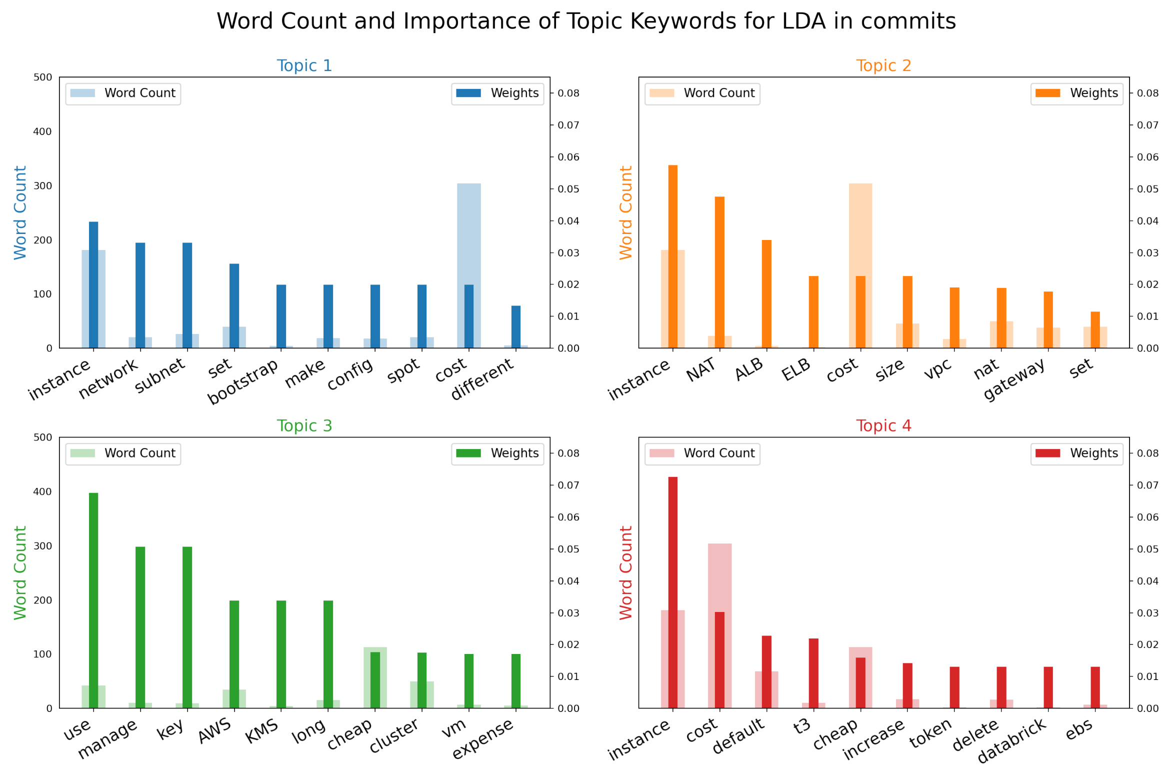


Figure 21: Visualization of the first 4 topics from applying LDA on the commit data set

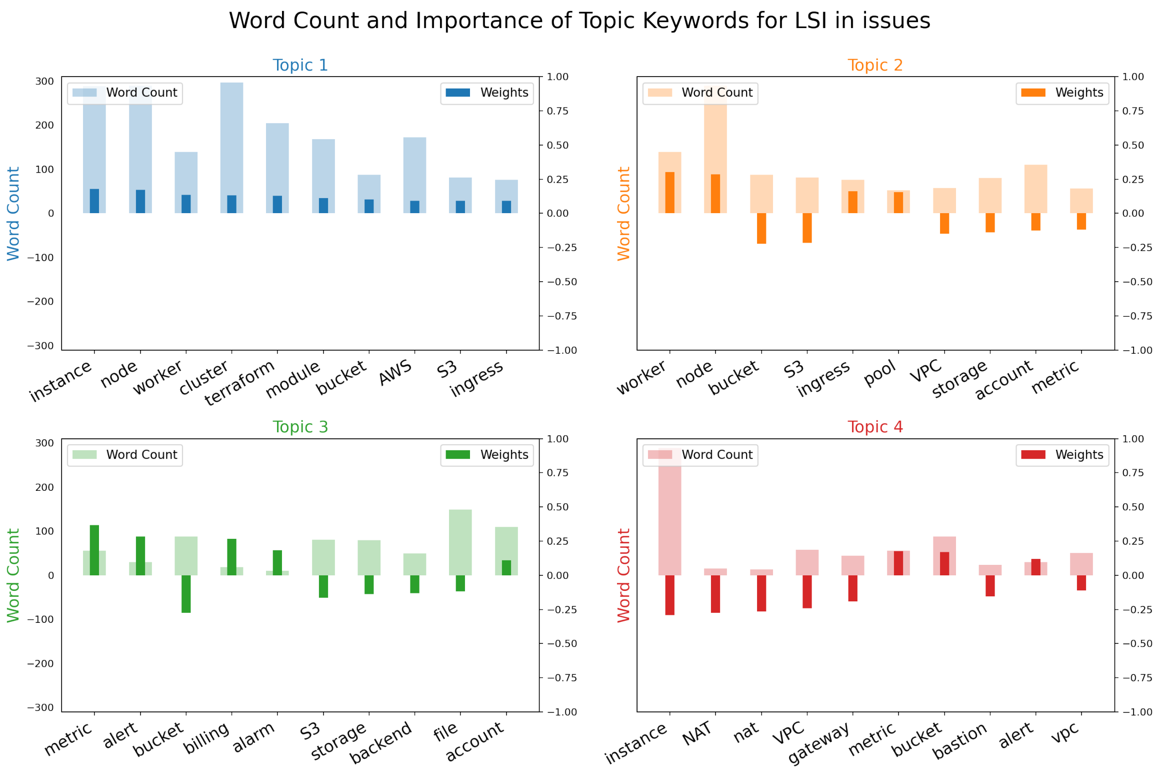


Figure 22: Visualization of the first 4 topics from applying LSI on the issue data set

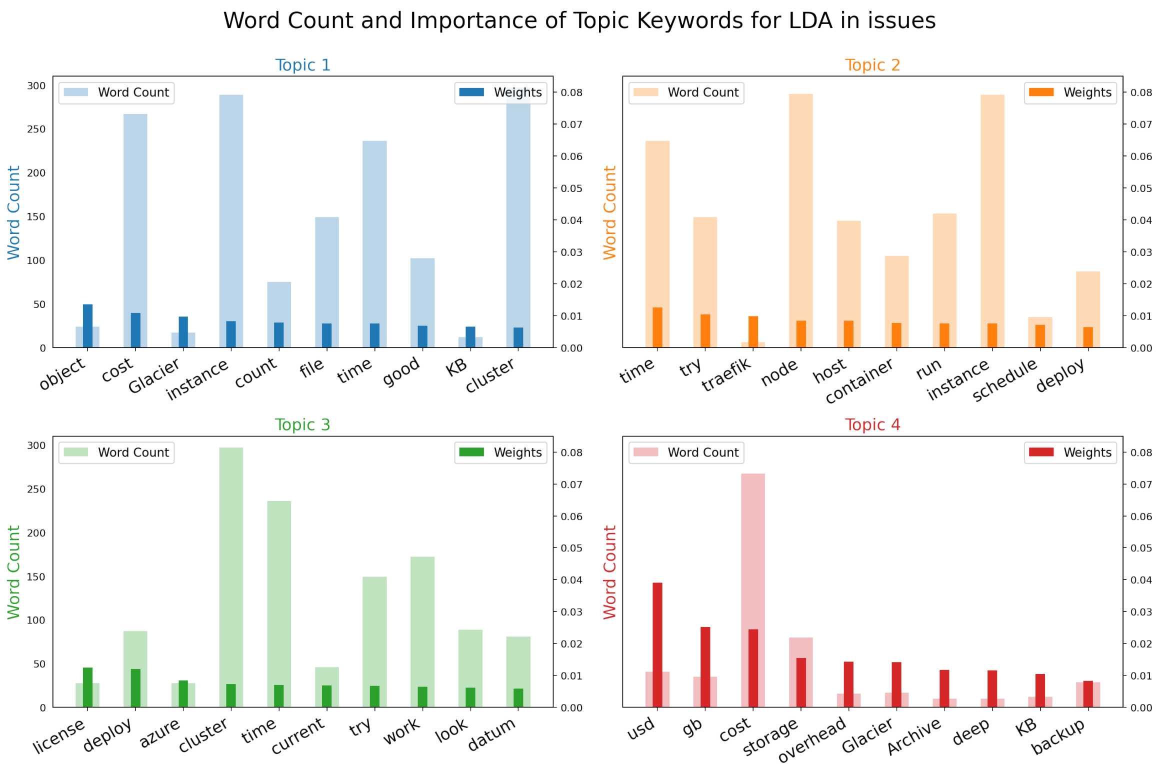


Figure 23: Visualization of the first 4 topics from applying LDA on the issue data set

Topic Modeling for Issues

Regarding issues, we start by analyzing the plots presented in Figure 22. The prevalence of the word cost is emphasized featuring alongside instances, network elements (`nat`, `gateway`), and threshold notifications (`alert`, `billing`). The actions inferred by the first topic are inconclusive, as features can be both disabled and added. Nevertheless, words such as `reduce` and `save` appear once again alongside `cost`, underlying future keywording prospects, but also the intent of developers. In the second topic, `network`, `node` and `pool` immediately draw attention to the use of clusters, although cost-related terms seem to be absent in the grouping. This might happen because, as issues tend to be large, discussions steer from one topic to another. A single comment or paragraph might have prompted us to label it as relevant to cost management. Seeing this, such subtleties can be unclear to the topic modeler, due to the small magnitude compared to the whole document. Similarly, a topic regarding storage (`bucket`) does not have any financial implications, although `metric` can refer to request rates, memory utilization, which ultimately bring cost into debate. Lastly, alarm related issues are singled out, containing recognizable words similar to the commit data set.

LDA-assigned topics can be visualized in Figure 23. In the nature of the topics illustrated by the topic modeling ran on the commits data set, the weights of the words are much lower than LSI. There is also not much categorization done per topics, as many elements are present across multiple of them. As such, we can only point out areas of interest that seem to be picked up by this method. Most notably, entries relating to `storage` are prevalent. These include memory size (`KB`, `GB`), `Glacier` which is an AWS product, and terminology that includes actions and tools (`Archive`, `backup`, `container`). In regards to potential improvements to our keywording lists during mining, the word `usd` was uncovered in one of the topics. Given that we have not considered using specific currency in our efforts, it may be feasible to do so in future research in order to pinpoint intricate financial discussion regarding cloud orchestrators.

By running the intersection on the topic modeling methods used on our issue data set, the result came up to be much more extensive:

```
Azure, Jun, May, account, bastion, bucket, cluster, container, control, count,
device, disk, dropbox, endpoint, error, etcd, exception_class, file, fortigate, gb,
host, image, ingress, instance, ip, launch, log, machine, metric, module, monitoring,
node, plane, pool, price, quota, rancher, request, runner, secret, spot, stop,
terraform, type, usd, volume, vpc, worker, x, zone
```

In this list of words, we are able to identify areas that we have mentioned previously: instances, clusters and storage. This serves to reinforce the popular labels identified in our taxonomy. We restate the possibility of using financial terms such as `price` and `quota` for identification of relevant dialogue around cost management in the future. In this context, `log`, referring to logging, is a frequent feature debated for removal to save on cost. This infers its popularity among issues, also supported by the prevalence of the `feature` label in the issue results in Section 4.3.

Future improvements to topic modeling

Regardless of the data sets we used our topic modeling scripts on, some issues with our approach resurfaced. Largely due to our time constraints, we scaled back on our attempts to fine-tune the process. As it can be seen in the intersection set composed for the two data sets, there is plenty of noise present in the form of dates, variables or case-sensitive repetitions. This demands a more scrutinized data cleaning process, in which additional checks are involved. For instance, lowercasing text can aid in reducing the number of repetitions. Regex can be employed to remove links and e-mail addresses, as well as dates. Unfortunately, typos might be trickier to deal with and the same is applicable to names. Despite us identifying proper nouns, we cannot remove names outright, as this would also get rid of certain providers, products or libraries that might add to the quality of the topic modeling. This is where a more curated stop word list can come into play, which can single out frequent names as well as irrelevant or misspelt words. Finally, in the case of large discussions in which cost-management is only a sideline, cutting the document to only contain the relevant paragraph can be considered. For these reasons, the HDP model rendered subpar results, that although are not explicitly discussed here, they can be viewed in the Appendix B.

4.5 Sentiment analysis

This section will showcase the results of the sentiment analysis experiments described in Section 3.7 while discussing the extracted insights. The goal of the experiments is to see whether polarity analysis is a viable option to speed-up, and automate the filtering and labeling process, or extract more insights from the commits and issues data. One way to determine if any of these models are good for our use case, is to analyse their accuracy and confusion matrices. The rest of this section will compare the results for all the three models tested starting with the issues, followed by the commits.

Sentiment analysis results for issue trackers

The first model tested was TextBlob which returned an accuracy score of 41%, meaning it correctly identified 41 out of the 100 issues that were manually labeled by the team. To get a better understanding of how the algorithm predicted, we can use a confusion matrix which shows us the distribution of the labels. Keeping in mind that the ideal scenario is the one in which the diagonal of the matrix holds all the prediction, looking at the matrix for TextBlob (Figure 24) we can see that it guessed positive too many times.

The second model used was Vader, which also returned an accuracy of 41%, accompanied by the confusion matrix in Figure 24. The algorithm followed the same flawed trend of falsely labeling the commits as positive too many times.

Lastly, we tested the Stanza model which had the small tweaks explained in the previous section such that it would predict the sentiment for the combined title, body and comments of the issue. The model returned an accuracy of only 26%, which as seen in the confusion matrix in Figure 24, is caused by the fact that it wrongly guessed the neutral label too many times.

Looking at the results, we can assume that Vader and TextBlob use similar algorithms and training data to understand emotion in text. This is reinforced by the fact that their accuracy are the same, but also their trends of guessing. On the other hand, Stanza seems to be using different data or different algorithms.

Sentiment analysis results for commits

Initially we planned to use NLP techniques mostly on issue trackers as these contain more text which results in more context for the models to work with. Because of the fact that the sentiment analysis scripts we have programmed could be easily modified to work on commits, we decided to include them in the experiments.

TextBlob returned an accuracy of 50% for the commits that we manually labeled. This was unexpected, but as shown in Figure 25, the way it predicted the label changed to guessing mostly neutral, which in turn increased the overall accuracy score. This makes sense as most of the manual labels were neutral or positive.

Vader's results were similar to what we expected, as the accuracy dropped to 38% on the texts of the commits. The confusion matrix in Figure 25 shows that the trend also changed to guessing mostly neutral labels, but Vader had substantially more wrong guesses for the negative and positive sentiments.

Stanza also increased in accuracy, reaching 32%, while still being the worst performing model. Compared to TextBlob and Vader, Stanza predicted mostly negative and neutral labels as shown in Figure 25. The reason for this behavior has to do with the way our code averaged the sentiment for each sentence in the body of text and attributed one overall feeling to the whole issue. Our current algorithm is more likely to always attribute either a neutral or negative sentiment also because of the rounding that we do in order to transform the output from integers to the three possible sentiments.

Results discussion

Because of the time constraints that we had, our comparison of the sentiment analysis models has been rather shallow. Most studies that focus on machine learning and model training take into account different metrics such as precision, recall, or F-scores when determining the efficiency of a model. For the scope of our project we determined that those are not needed as our goal was not to improve the efficiency of these models, but to determine whether using a pre-trained model is useful to uncover trends or insights in our data.

Taking into account the results shown in the previous paragraphs, we can say with certainty that the tested models are not efficient enough to reveal any well-grounded insights from the data. That said, certain trends where identified as result of the analysis. Some examples are the fact that negative labels are being related to networking concepts such as NAT gates, or positive sentiments being strongly related to developers removing costly features from their application. Moreover, in most cases, the commits that were relevant from a cost-management point of view were either expressing a positive or negative sentiment rather than a neutral. Unfortunately, due to the low accuracy scores, and the inaccurate prediction patterns that all models have we cannot completely trust these trends as they are. Using sentiment analysis in this form could potentially showcase some insights, but with no way of confirming that these trends are not just coincidences. Moreover, neither the filtering process or the manual labeling seem to benefit in any way from the results of our experiments.

While the pre-trained models have failed to produce optimal results, it should be noted again that the task of predicting a sentiment in a software engineering body of text is not an easy task. This idea has been discussed in different research papers, and one solution is to train a model on specific data. One of these models is SentiStrength-SE, but during our tests, the results were sub-optimal. Moreover, the process of using it was time consuming because of the data processing step. However, this leaves room for training a new model, that works with popular data science libraries such as Pandas, and uses state-of-the-art machine learning algorithms in Python.

A different approach could include taking a look at sentiment analysis from a different angle. Instead of using polarity analysis, we could look at a more specific classification task where we can attribute different tags such as `cost-increase` or `cost-decrease`. A more ambitious approach would be to test whether using natural language processing analysis could determine if a sentence talks about cost in a context that would interest us.

Our sentiment analysis experiments were short, and without any positive results, but they showcased the potential that natural language processing can have in our use case, but also in similar ones.

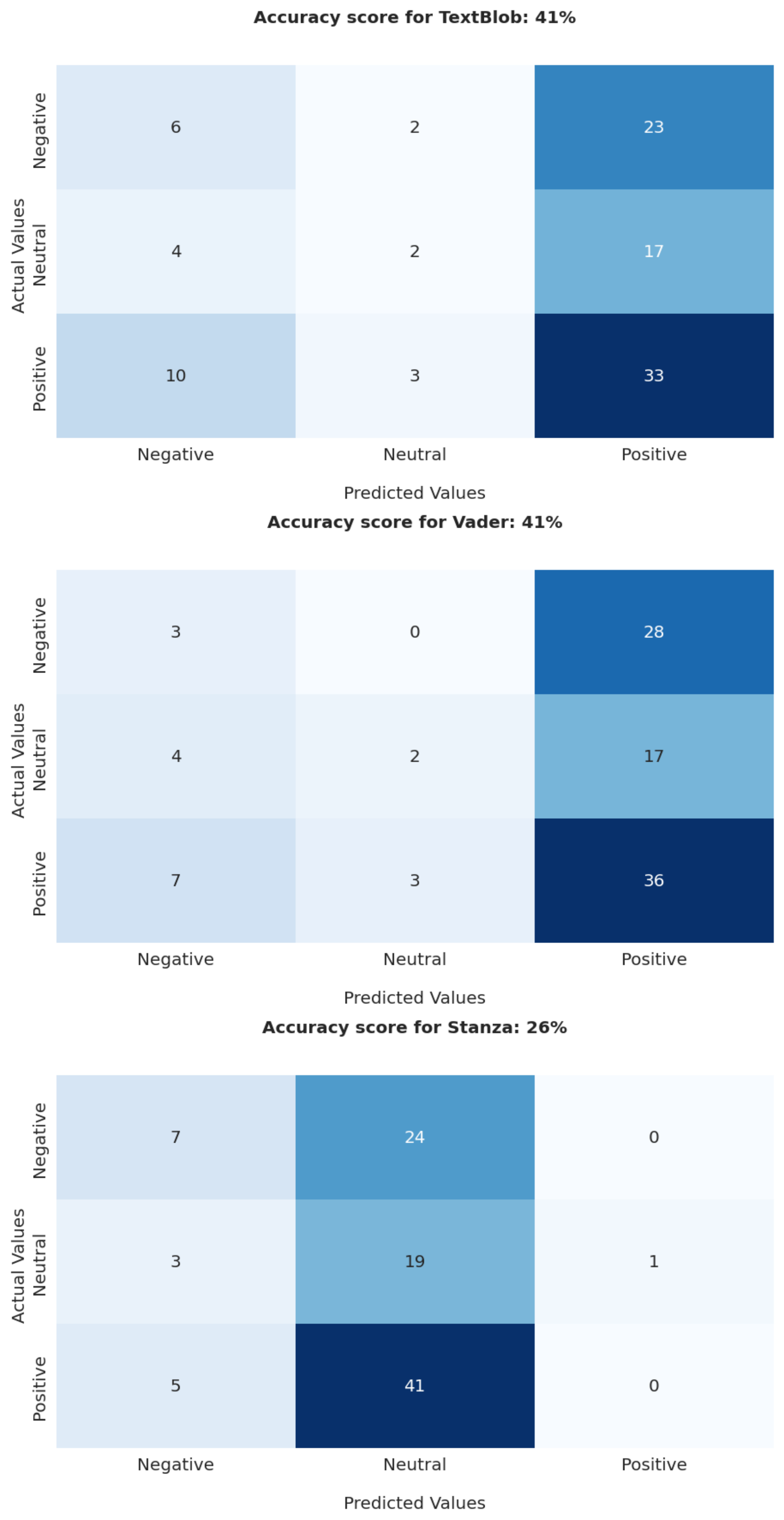


Figure 24: Confusion matrices for the three models used on issue tracker data

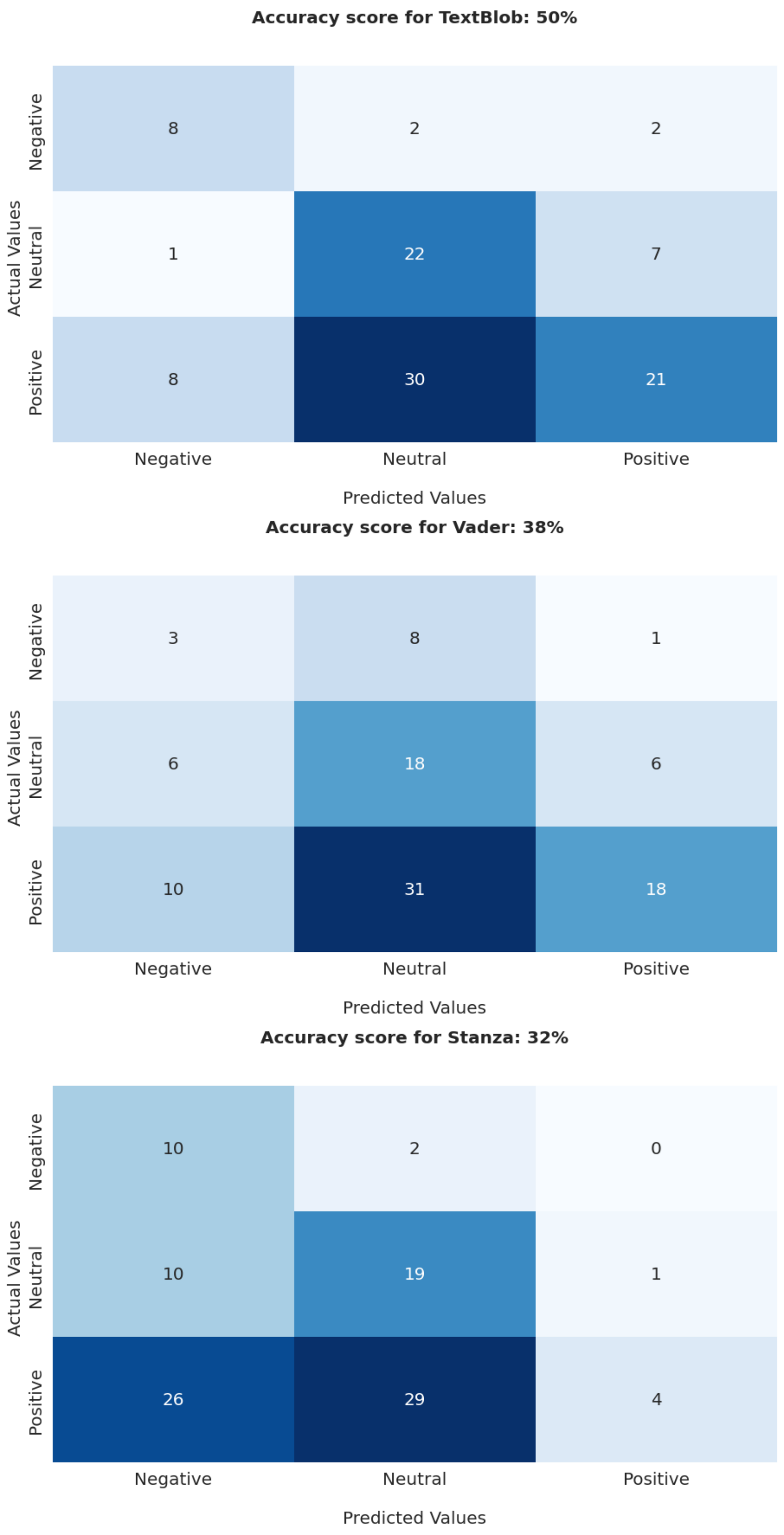


Figure 25: Confusion matrices for the three models used on commits data

5 Conclusion

Through this exploratory work, we tried to determine whether it is feasible to mine and extract information about cost in open-source projects that use Terraform as their cloud orchestrator. Because of the nature of our project, and the lack of previous relevant works, our research question is:

Can we extract cost management insights based on cloud orchestrator artifacts and VCS data using MSR techniques?

Section 5.1 summarizes the work conducted in this paper, while Section 5.2 connects the results and answers produced by our work with the research question stated in the previous chapter. Finally, Section 5.3 goes more in-depth with potential follow-up research.

5.1 Summary of Main Contributions

The first step in our research was to understand, and create a way of retrieving open-source repositories that use Terraform for cloud orchestration purposes. To achieve that, we made use of GitHub's advanced search API which allowed us to query and filter links to repositories. To automate the process and save the results in a reliable way, we have also used PyGitHub, a python wrapper, that allowed us to extract these links to a text file that can be re-used in other scripts. Section 3.2 goes in-depth with all versions of the queries used, but the final solution has been to look for all repositories containing `.hcl`, and `.tf` files. Moreover, we decided to run the query on all repositories that have been uploaded since 2014, when Terraform was officially launched. This process resulted in $152,735$ repositories.

The next step was to extract information from these repositories that we now had links to. We decided to begin by mining commits as there are more relevant works using this MSR technique. The tool used was PyDriller which provides an easy to use API in python. To filter the repositories that discuss cost in their commits messages, we have created a list of keywords. The list contains of the following word roots as keywords: `cheap`, `expens`, `cost`, `efficient`, `bill`, and `pay`, which were added at a later time. These keywords are meant to be used with Python's matching logic, meaning any word that contains these keywords will be flagged as relevant. Section 3.3 describes how this process allowed us to reduce the number of relevant repositories to $1,287$ which amounts to $2,422$ commits.

The extraction of issue tracker information is not as common as its commit counter-part, so using PyDriller for this task was not possible. Fortunately, we were able to use Perceval, another Python library, to extract data such as the title, body, and comments of an issue. Section 3.4 explains the downsides of issue extraction, which have to do with GitHub's API rate limit, and how we circumvented that by adding manual waiting times. Our initial approach was to make use of the relevant commits that have been labeled, in order to extract their linked issues. This idea did not prove useful as most of the linked issues were in reality pull requests, which are treated similarly to issues by GitHub. Our second approach was much simpler as we just filtered out all the pull requests and only accepted the issues that contained the keywords mentioned in the previous chapter. This resulted in 862 relevant issues for us to manually look into.

It is worth noting that while executing the work described in the paragraphs above, we were also tasked with saving all this data into a reusable form. For that reason, we decided to use JSON format as it would enable the data to be easily shared between applications and researchers. While JSON was a good option for the raw data, we decided to use CSV format for the relevant entries that needed labeling. This also allowed us to use software such as Google Spreadsheets which helped the workflow of the team. Both the files and code are available and can be found in Appendix A.

Section 3.5 explains the process used to label the commits and issues that have been filtered in the previous steps. To be able to extract as many insights from the data, we created a taxonomy that can be visualized in Figure 8. In the hierarchy of labels, we put `cost`, and `other` at the top level. The former identifies commits or issues that talk about cost management, and are of interest to our research. On the other hand, `other` is attributed to data that contains the selected keywords, but in a different context. To increase the granularity, we have split the `cost` label into `awareness`, `saving`, and `increase`, to better understand which one of these is most prominent in the software engineering community. The last level in the hierarchy deals with the exact reason for mentioning the cost. Some examples are `instance`, or `storage` which as the names imply, refer to a cost change that had to do with the storage or server instance used for the specific application. A similar approach was applied to the `other` label which is described in-depth is Section 3.5. At the end of this process, we ended up with a well-defined classification of all the data that was manually processed, as discussed in Section 4.1. The visualization and statistics created in Sections 4.2 and 4.3 allowed us to better understand which elements in the cloud infrastructure are the most expensive and discussed within the community.

Sections 3.6 and 3.7 explore the applications of natural language processing techniques such as topic modeling and sentiment analysis on the labeled data. The goal of these experiments was to discover any other additional insights that were overlooked in the previous sections. For topic modeling, we have attempted several approaches on the data sets containing commit messages and issue bodies alike. By utilizing several external libraries, we were able to rid the corpus of any HTML and Markdown formatting code. Afterwards, the Stanza neural network NLP pipeline allowed us to tokenize the various documents, in order to later format them into Gensim Dictionaries, bag of words and TF-IDF models. As a result, the Gensim library gave us access to implementations of models such as Latent Dirichlet Allocation (LDA), Hierarchical Dirichlet Process (HDP), and Latent Semantic Analysis (LSA), which we applied on the pre-processed corpus. This resulted in each modeler returning its own topics composed of word lists, but also an intersection of terms common among all three methods as discussed in Section 4.4.

For sentiment analysis, we have decided to not build our own model, but instead compare four existing pre-trained models. We still needed a test data set, so we manually labeled 100 commits and 100 issues that were classified as being cost-relevant. Next, we first supplied the data to three modern sentiment analysis algorithms that were not specifically trained for software engineering tasks. These models are TextBlob, Vader, and Stanza sentiment, which provided us with python APIs. We also tested the SentiStrength-SE model which was created as part of a research conducted on sentiment analysis in software engineering. Because of the older machine learning algorithms used, the model underperformed compared to the other three. For this reason, we decided to drop this model from further experimentation.

5.2 Discussion of Main Findings

Our work summarized above demonstrated how we can leverage the GitHub API and its Python wrappers to extract information about open-source repositories that contain Terraform configuration files. Moreover, Sections 3.3 and 3.4 showcased how some simple filtering methods allowed us to narrow down the data from $153,421$ repositories to only $2,422$ commits and 862 issues. This work helped us answer our first sub-question posed in the introduction, which asked how can we select and retrieve the repositories needed for this study.

Our second sub-question revolved around how we can label the cost-related information that we have extracted in the step prior to this. Section 4.1 explains how we have created a taxonomy that allowed us to manually label each commit and issue to determine their relevancy. As a result, we further filtered down the cost-relevant data to 538 commits and 206 issues. As such, while answering this question we created a reusable hierarchy that can be further used and improved.

As mentioned in the summary above, while conducting our work, one of our focuses was how the data was stored and accessed. By storing it as JSON and CSV, we have also managed to answer our third sub-question that focused on creating a reusable and sanitized data set.

In order to answer our final sub-question, we looked into ways in which we can extract insights from the labeled data that we have created. As shown in Sections 4.2 and 4.3, our first approach was to create visualizations and statistics using the hierarchy of labels that we have set up. Using these, we gained a better understanding of which cloud infrastructures are most expensive or used. Furthermore, we were able to discern certain behaviors in software development that have to do with cost management. Some examples consist of developers comparing the infrastructures they use, and their associated costs, or having separate services for the deployment and production of GitHub branches. We were also able to see that there is a distinction between performed actions (commits) and discussions (issue trackers). In the former, developers mention more about saving cost rather than bringing awareness about it, while in issues the trend is reversed. Another interesting insight we extracted is that developers rarely mention anything about increasing the costs of their infrastructure.

Because gaining information from statistics and visualization is something that requires manual analysis, we decided to also test techniques such as topic modeling and sentiment analysis to uncover any underlying insights. Section 4.4 discusses our attempts at uncovering topics from the analysis of commit and issue bodies. The obtained word lists show promising results, by confirming the levels of cloud infrastructure where cost management happens. It also provided potential additions to the keywording process that might take place in future research. Nevertheless, there is still plenty of room for improvement, as the irrelevancy of some entries in the topics demand more fine-tuning. On the other hand, Section 4.5 showed that using pre-trained sentiment analysis models does not produce optimal results. This means that without training a personalized model for our task, we cannot use this natural language processing technique to get a better understanding of our data.

Finally, looking at our main research question, and taking into account the above-mentioned results to our sub-questions, we can say with certainty that it is possible to extract and analyse cost-management information from repositories that use Terraform configuration files. In addition, we showed that there is a lot of insights to be gained from analyzing the data that is stored inside version control systems and issue trackers.

5.3 Future Work

Our work served as an exploratory study on whether the extraction and analysis of cost management is possible when focusing on repositories using cloud orchestrators. As a consequence of the lack of research in this domain, we did not have any related studies to base our filtering processes on. As such, the list of keywords we have used to extract repositories that mention cost can be improved by adding or removing entries.

Throughout the labeling process, we have had many instances where we were not able to classify the commit or issue within our taxonomy. As a result, we were forced to introduce the `unknown` label. A future improvement would be to develop the taxonomy in such a way that this label is no longer needed.

Even though we tried to make use of natural language processing techniques, the analysis step could also be improved. Having better data cleaning, and more data entries could help improve the topic modeling algorithms used, which in turn would increase the quality of the results. When it comes to sentiment analysis, it could be interesting to use the data we have created to train a model that instead of doing polarity analysis, does a more specific classification for our use case. This could then potentially extract more insights about cost management and improve the labeling process.

Bibliography

[1] T. DeStefano, R. Kneller, and J. Timmis, "Cloud computing and firm growth," *SSRN Electronic Journal*, 2020.

[2] L. R. de Carvalho and A. P. F. de Araujo, "Performance comparison of terraform and cloudify as multi-cloud orchestrators," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, IEEE, May 2020.

[3] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch, "How to adapt applications for the cloud environment," *Computing*, vol. 95, pp. 493–535, Dec. 2012.

[4] W. Poncin, A. Serebrenik, and M. v. d. Brand, "Process mining software repositories," in *2011 15th European Conference on Software Maintenance and Reengineering*, pp. 5–14, 2011.

[5] G. Juve, E. Deelman, G. B. Berriman, B. P. Berman, and P. Maechling, "An evaluation of the cost and performance of scientific workflows on amazon EC2," *Journal of Grid Computing*, vol. 10, pp. 5–21, Mar. 2012.

[6] N. Ekwe-Ekwe and A. Barker, "Location, location, location: Exploring amazon EC2 spot instance pricing across geographical regions," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, IEEE, May 2018.

[7] B. Gupta, P. Mittal, and T. Mufti, "A review on amazon web service (AWS), microsoft azure &amp: google cloud platform (GCP) services," in *Proceedings of the 2nd International Conference on ICT for Digital, Smart, and Sustainable Development, ICIDSSD 2020, 27-28 February 2020, Jamia Hamdard, New Delhi, India*, EAI, 2021.

[8] J. Kovács and P. Kacsuk, "Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures," *Journal of Grid Computing*, vol. 16, pp. 19–37, Nov. 2017.

[9] K. Bousselmi, Z. Brahmi, and M. M. Gammoudi, "Cloud services orchestration: A comparative study of existing approaches," in *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, IEEE, May 2014.

[10] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, IEEE, May 2009.

[11] I. Moura, G. Pinto, F. Ebert, and F. Castor, "Mining energy-aware commits," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, IEEE, May 2015.

[12] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez, "The quest for open source projects that use UML," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ACM, Oct. 2016.

[13] T. S. Heinze, V. Stefanko, and W. Amme, "Mining BPMN processes on GitHub for tool validation and development," in *Enterprise, Business-Process and Information Systems Modeling*, pp. 193–208, Springer International Publishing, 2020.

[14] L. Bao, D. Lo, X. Xia, X. Wang, and C. Tian, "How android app developers manage power consumption?," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ACM, May 2016.

[15] S. Bellomo, R. L. Nord, I. Ozkaya, and M. Popeck, "Got technical debt?," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ACM, May 2016.

[16] H. Sellik, O. van Paridon, G. Gousios, and M. Aniche, “Learning off-by-one mistakes: An empirical study,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE, May 2021.

[17] M. Ortu, T. Hall, M. Marchesi, R. Tonelli, D. Bowes, and G. Destefanis, “Mining communication patterns in software development,” in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, ACM, Oct. 2018.

[18] A. Serban, M. Bruntink, and J. Visser, “Graphrepo: Fast exploration in software repository mining,” *arXiv preprint arXiv:2008.04884*, 2020.

[19] S. Romano, M. Caulo, M. Buompastore, L. Guerra, A. Mounsif, M. Telesca, M. T. Baldassarre, and G. Scanniello, “G-repo: a tool to support MSR studies on GitHub,” in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, Mar. 2021.

[20] F. Z. Sokol, M. F. Aniche, and M. A. Gerosa, “MetricMiner: Supporting researchers in mining software repositories,” in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, Sept. 2013.

[21] D. Spadini, M. Aniche, and A. Bacchelli, “PyDriller: Python framework for mining software repositories,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, Oct. 2018.

[22] M. Steinbeck, “Mining version control systems and issue trackers with LibVCS4j,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, Feb. 2020.

[23] A. Trautsch, F. Trautsch, S. Herbold, B. Ledel, and J. Grabowski, “The SmartSHARK ecosystem for software repository mining,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ACM, June 2020.

[24] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, “Perceval,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ACM, May 2018.

[25] J. Hirschberg and C. D. Manning, “Advances in natural language processing,” *Science*, vol. 349, pp. 261–266, July 2015.

[26] T.-H. Chen, S. W. Thomas, and A. E. Hassan, “A survey on the use of topic models when mining software repositories,” *Empirical Software Engineering*, vol. 21, pp. 1843–1919, Sept. 2015.

[27] B. V. Barde and A. M. Bainwad, “An overview of topic modeling methods and tools,” in *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, IEEE, June 2017.

[28] H. Jelodar, Y. Wang, C. Yuan, X. Feng, X. Jiang, Y. Li, and L. Zhao, “Latent dirichlet allocation (LDA) and topic modeling: models, applications, a survey,” *Multimedia Tools and Applications*, vol. 78, pp. 15169–15211, Nov. 2018.

[29] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[30] S. Williamson, C. Wang, K. A. Heller, and D. M. Blei, “The ibp compound dirichlet process and its application to focused topic modeling,” in *ICML*, 2010.

[31] A. M. Dai and A. J. Storkey, “The supervised hierarchical dirichlet process,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, pp. 243–255, Feb. 2015.

[32] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, (Valletta, Malta), pp. 45–50, ELRA, May 2010. http://is.muni.cz/publication/884893/en.

[33] A. Qazi, R. G. Raj, G. Hardaker, and C. Standing, "A systematic literature review on opinion types and sentiment analysis techniques," *Internet Research*, vol. 27, no. 3, p. 608–630, 2017.

[34] S. Kumar, M. Yadava, and P. P. Roy, "Fusion of eeg response and sentiment analysis of products review to predict customer satisfaction," *Information Fusion*, vol. 52, p. 41–52, 2019.

[35] M. M. Mostafa, "An emotional polarity analysis of consumers' airline service tweets," *Social Network Analysis and Mining*, vol. 3, no. 3, p. 635–649, 2013.

[36] A. C. Lima, L. N. de Castro, and J. M. Corchado, "A polarity analysis framework for twitter messages," *Applied Mathematics and Computation*, vol. 270, p. 756–767, 2015.

[37] R. Feldman, "Techniques and applications for sentiment analysis," *Communications of the ACM*, vol. 56, no. 4, p. 82–89, 2013.

[38] R. Badlani, N. Asnani, and M. Rai, "An ensemble of humour, sarcasm, and hate speechfor sentiment classification in online reviews," *Proceedings of the 5th Workshop on Noisy User-generated Text (W-NUT 2019)*, 2019.

[39] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering," *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[40] S. Loria, "textblob documentation," *Release 0.15*, vol. 2, 2018.

[41] C. Hutto and E. Gilbert, "Vader: A parsimonious rule-based model for sentiment analysis of social media text," *Proceedings of the International AAAI Conference on Web and Social Media*, vol. 8, pp. 216–225, May 2014.

[42] P. Qi, Y. Zhang, Y. Zhang, J. Bolton, and C. D. Manning, "Stanza: A python natural language processing toolkit for many human languages," *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2020.

[43] M. R. Islam and M. F. Zibran, "Sentistrength-se: Exploiting domain specificity for improved sentiment analysis in software engineering text," *Journal of Systems and Software*, vol. 145, p. 125–146, 2018.

[44] V. Bonta, N. Kumaresh, and N. Janardhan, "A comprehensive study on lexicon based approaches for sentiment analysis," *Asian Journal of Computer Science and Technology*, vol. 8, no. S2, p. 1–6, 2019.

[45] S. Sohangir, N. Petty, and D. Wang, "Financial sentiment lexicon analysis," *2018 IEEE 12th International Conference on Semantic Computing (ICSC)*, 2018.

[46] M. R. Islam and M. F. Zibran, "Leveraging automated sentiment analysis in software engineering," *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017.

Appendix - Supplementary Material

A Links

- GitHub Repository - In this repository, one can find all the technical content related to our thesis. This includes various Jupyter notebooks to perform repository mining, topic modeling and sentiment analysis. Furthermore, the scripts to build bar plots and other graphs are available. The acquired data sets for both issues and commits are also saved for inspection, albeit some personal information has been removed. The data sets include the raw results from mining (in JSON or TXT format depending on the context), and the labeled entries in CSV format. (https://github.com/Max593/Mining-and-Analysis-of-Cost-related-Decisions-in-Cloud-Infrastructures)

B Additional visualizations

In this section we showcase figures that we decided to not include in our Result discussion.

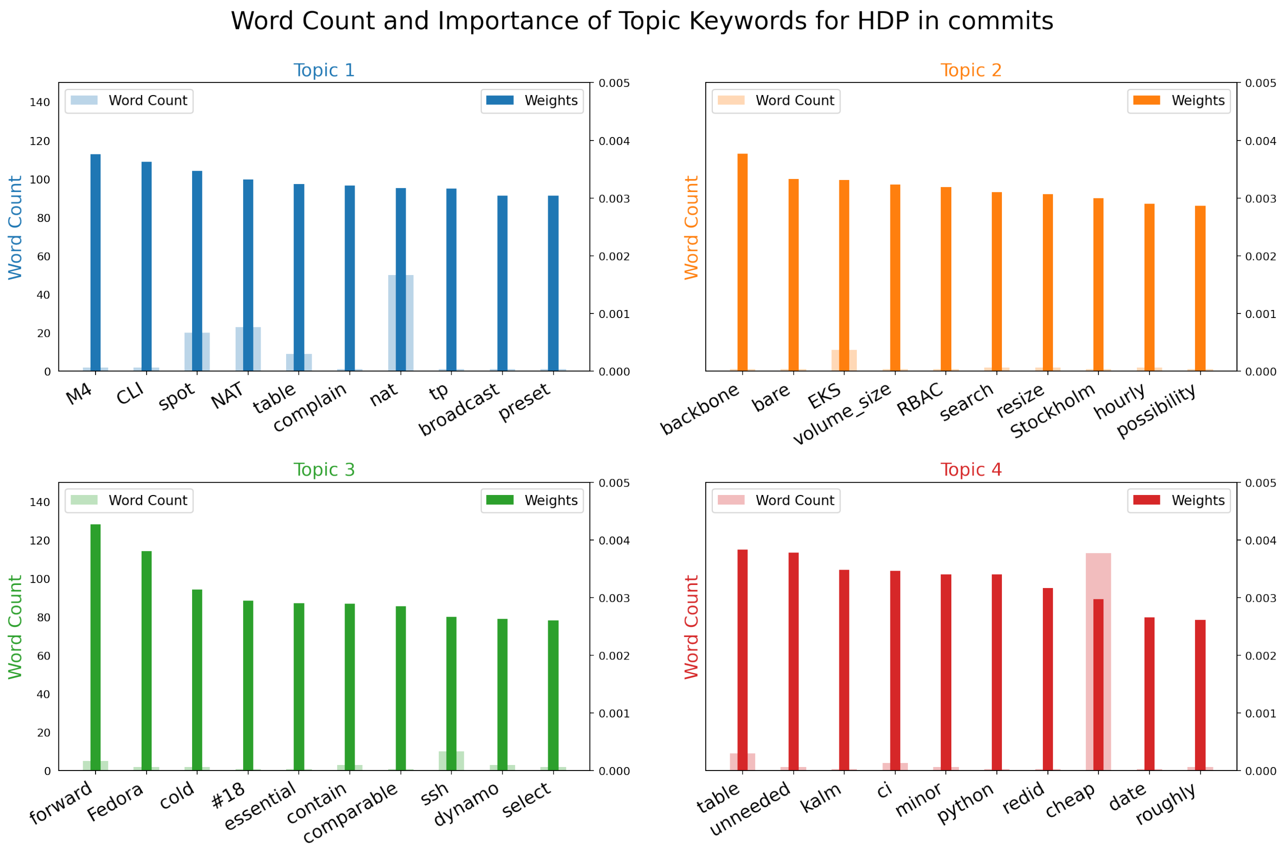


Figure 26: HDP topic visualization for commits

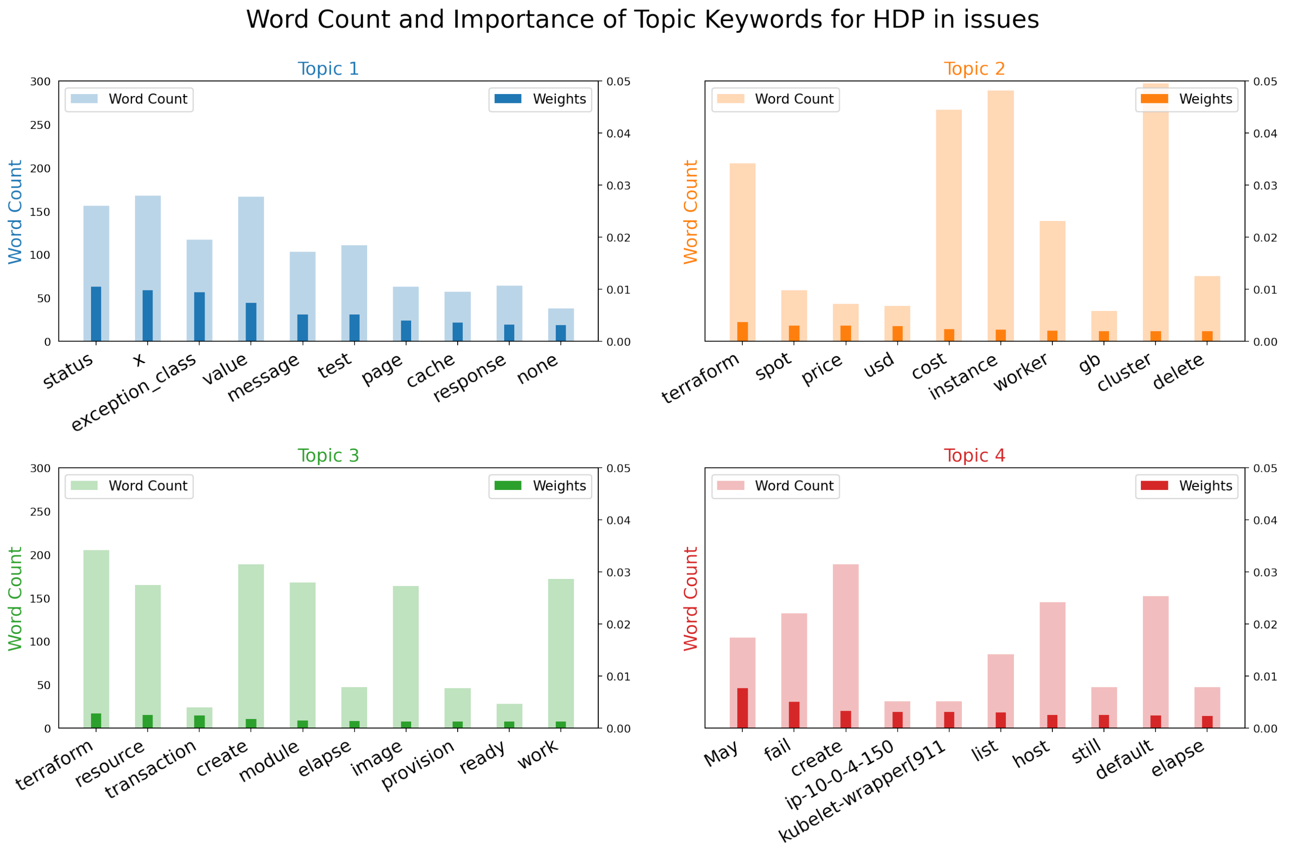


Figure 27: HDP topic visualization for issues

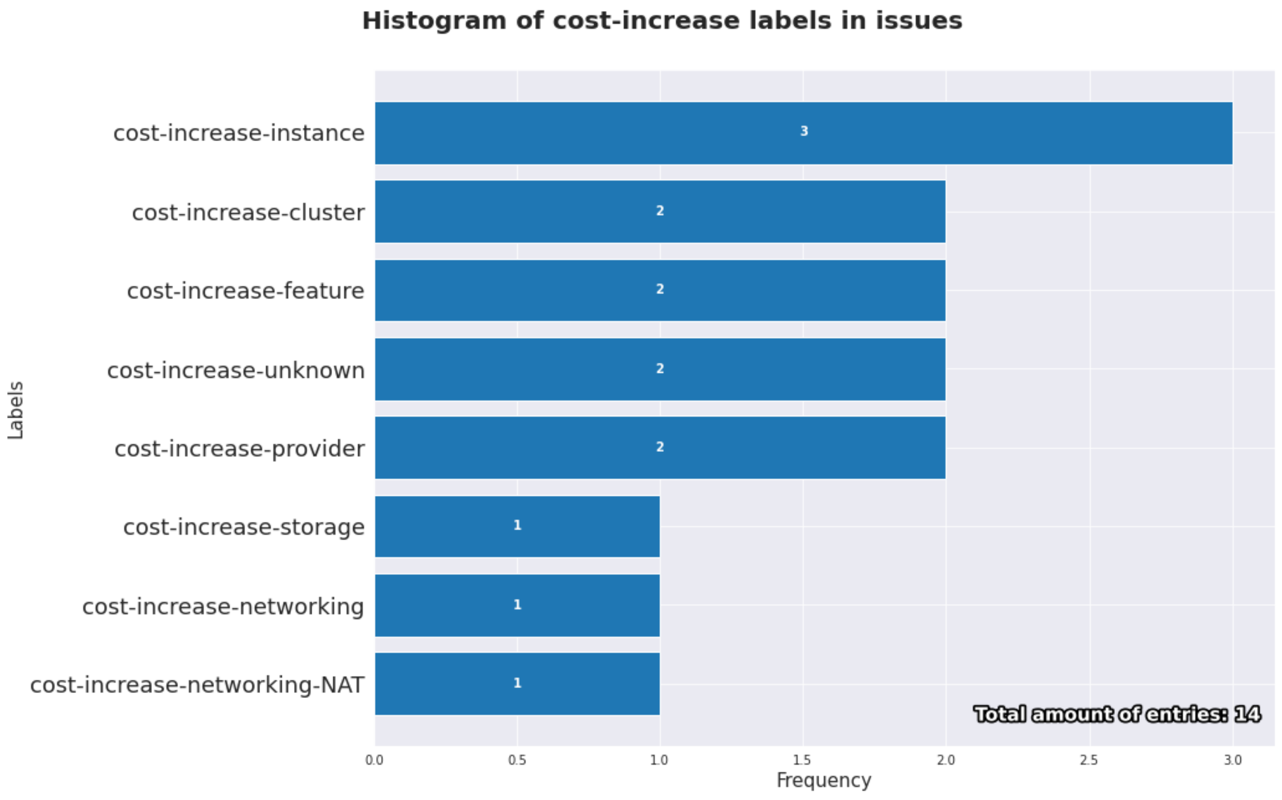


Figure 28: Issue increase

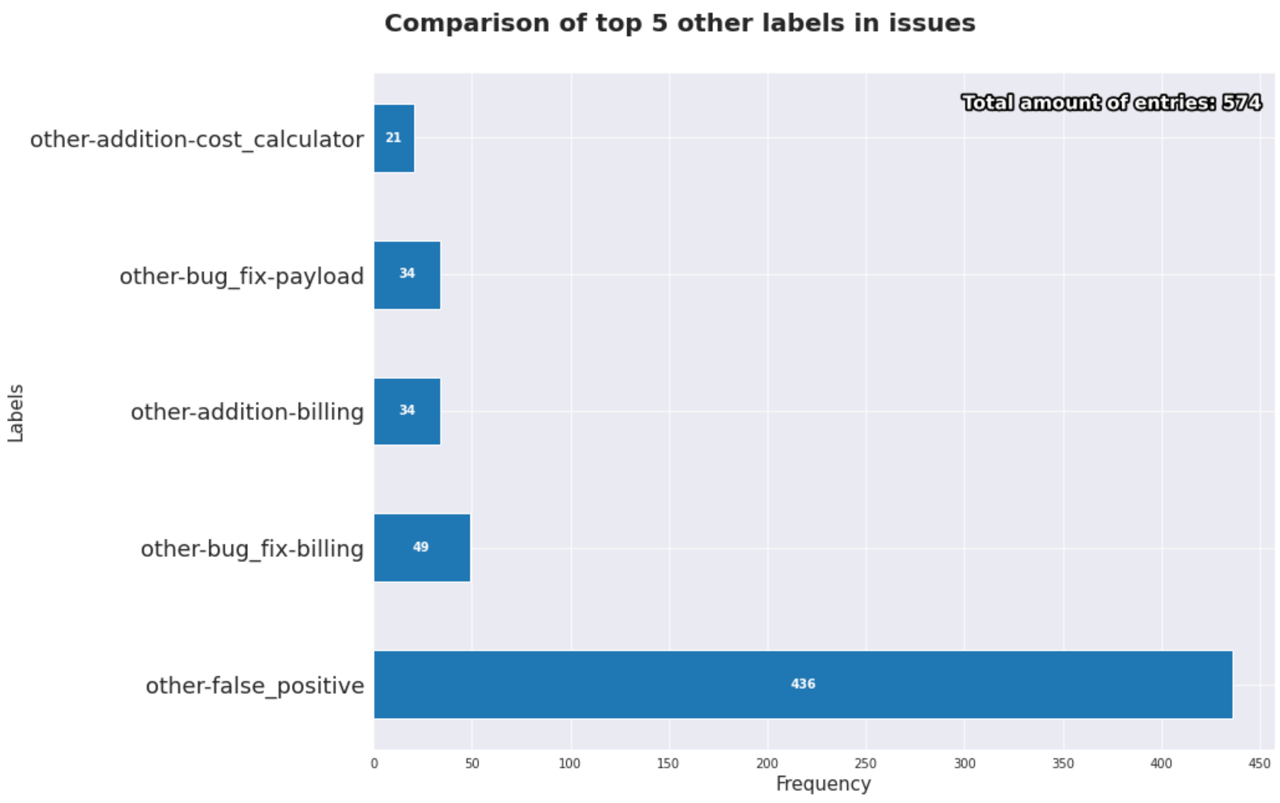


Figure 29: Top 5 ‘other’ labels in issues