MASTER'S THESIS 2024

Representing and Classifying Diffs of Hierarchically Dependent Queries to a Graph Database

Sean Jentz, Karolina Haara Löfstedt

ISSN 1650-2884
LU-CS-EX: 2024-16

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-16

Representing and Classifying Diffs of Hierarchically Dependent Queries to a Graph Database

Representation och Klassificering av Förändringar i Hierarkiskt Beroende Förfrågningar till en Grafdatabas

Sean Jentz, Karolina Haara Löfstedt

Representing and Classifying Diffs of Hierarchically Dependent Queries to a Graph Database

Sean Jentz
sean.jentz@gmail.com

Karolina Haara Löfstedt
karolina.lofstedt@gmail.com

April 8, 2024

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Anton Berghult, anton.berghult@infrasightlabs.com
Lars Bendix, lars.bendix@cs.lth.se

Examiner: Per Andersson, per.andersson@cs.lth.se

Abstract

This thesis investigates the possibility of a tool that can detect changes to queries and visualize them at a higher level of abstraction than the `git diff` function. Queries in vScope fetch, transform and display data from a graph database. The queries are structured hierarchically and are implemented as interdependent XML-formatted files. These files are contained in Git repositories and change requests are currently reviewed by developers with Git in a text-based format. InfraSight Labs wants to simplify the process around query configuration in order to enable non-developers to take over the task.

We found that we can detect and classify changes to queries and their dependencies with the use of reflection and persisting annotations. This is exemplified with a proof of concept diffing tool. The tool serialized Java objects from XML-files and compared them using reflection, then used annotations provided to the fields to classify the impact of a change.

The conclusion is that it is possible to represent changes made to query files at a higher level of abstraction than the text based `git diff` function. Integrating the tool into the current product is left for future work.

Keywords: Diff Classification, Diff Impact Analysis, Hierarchical Queries, Change Classification, Diff Representation, Hierarchical Dependency Diff

Acknowledgements

We would like to thank the whole team at InfraSight Labs for their enthusiastic participation in our research activities. In particular for the engaging discussions during interviews, our multiple evaluation phases and technical discussions. We would also like to thank our university supervisor, Lars Bendix, for his extensive feedback, guidance, support, and for encouraging us to "Just f****ng do it" when perfectionism was about to take over.

Contents

1 Introduction **7**

2 Background **9**
- 2.1 Context . . . 9
- 2.2 Research Questions . . . 14
- 2.3 Methodology . . . 14
  - 2.3.1 Requirements Phase . . . 15
  - 2.3.2 Design Phase . . . 17
  - 2.3.3 Prototype Phase . . . 18

3 Requirements Phase **19**
- 3.1 Interviews . . . 19
- 3.2 Lo-Fi Mocks . . . 21
- 3.3 Literature Study . . . 22
- 3.4 Evaluation . . . 24

4 Design Phase **27**
- 4.1 Use Cases . . . 27
- 4.2 Technical Design Alternatives . . . 30
  - 4.2.1 Using XML . . . 30
  - 4.2.2 Using Reflection And Annotations . . . 32
- 4.3 Design Proposal . . . 33

5 Prototype **35**
- 5.1 Implementation . . . 35
- 5.2 Evaluation . . . 39

6 Discussion & Related Work **43**
- 6.1 Reflection on Process . . . 43
- 6.2 Threats to Validity . . . 44
- 6.3 Generalizability . . . 45

6.4 Related Work . . . 46
6.4.1 Related Work : Enabling Fuzzy Object Comparison in Modern Programming Platforms through Reflection [3] . . . 46
6.4.2 Related Work: Change Impact Analysis Based on a Taxonomy of Change Types [12] . . . 47
6.4.3 Related Work: Component Configuration Management[9] . . . 48
6.4.4 Related Work: Multi-party authorization and conflict mediation for decentralized configuration management processes [8] . . . 49
6.5 Future Work . . . 49

7 Conclusion **51**

References **53**

Appendix A Questions for semi-structured interviews **57**

Appendix B Requirements specification **59**

Chapter 1
Introduction

This thesis will explore the possibilities of a tool that can detect and classify changes made to hierarchically dependent query data models, and work at a higher level of abstraction than text based diffs. We will refer to these data models as *vScope queries*. vScope queries are a key component of the IT inventory tool vScope. vScope uses probes that integrate with services from Microsoft Azure, Amazon Web Service, Veeam, Docker among others, vScope processes data and stores it in a graph database. The processed data is categorized into distinct asset types such as machines, user accounts, and certificates. Within the graph database, vertices represent specific properties of these assets, while edges establish relationships between them. For instance, a user account and a machine may be linked by sharing the same user. Once data has been inventoried it is presented to the user in various formats such as tables, dashboards, widgets, tracker cases and tags.

vScope queries depend on each other in an hierarchical structure and are stored in XML-formatted file. These files contain descriptive elements such as names, descriptions and presentation format, a unique identifier, some way to fetch data and sometimes an operation to transform the data. vScope queries are created and maintained internally at the company and contained within every new release. The files are stored in Git repositories, and change requests are currently reviewed by developers with Git in a text-based format.

Updating already existing queries or adding new ones involve a full sprint cycle of 3 weeks before being accessible to the end user. Users can also create their own queries locally. At present, newly created vScope queries are contained within the product and released to every installation regardless of relevance to individual users. This approach affects the user experience and makes it harder to navigate in the product as it is cluttered with irrelevant information. To address this issue, InfraSight Labs wants to create a platform for sharing vScope queries, separate from product releases. The platform would initially be used by the company's customer success team, to update and share queries with vScope end users. In the future the platform could be made available to external users such that they could make change requests. InfraSight Labs would still need to audit that process, but the sharing of files would be managed in a peer-to-peer like manner.

The introduction of this platform introduces some configuration management related concerns. It will also require a certain level of technical competence query creation and updates. Given the hierarchical structure of the query files, understanding the potential impact of changes on other dependent queries is necessary. Up until now this process has been the domain of the software developers but the company wants to shift the responsibility for query updates to the customer success team. To allow for this change there is a need to elevate the level of abstraction for viewing changes to vScope queries, moving away from text-based diffs to more user-friendly alternatives. It would facilitate the understanding of what the changes imply before they are accepted or rejected at the publication stage to the central repository. This would also help reduce the risk of wrongfully updating vScope queries which could potentially lead to a break of functionality.

There is a need for a tool that can calculate differences in query files and their dependencies, and work at a higher level of abstraction than text based diff. A number of requirements need to be considered when developing the tool, with respect to query updates, dependent queries and secure handling of queries.

Regarding query updates, the difference between two versions of vScope queries should be presented in a clear and understandable manner. Changes need to be easily understood by end users of vScope rather than software engineers well-versed in tools like Git. This will involve storage in a repository with version control and history. Change requests to the central repository and to vScope installations should provide a change set that is readable to the intended end users, and should consider dependencies. Regarding security, we need to establish which precautions should be considered to ensure secure handling and updating of vScope queries.

Chapter 2
Background

This chapter will provide context and background to give a better understanding of InfraSight Labs and their product, vScope. In the first section, the outcome of an initial case study will provide context and technical background. We will introduce vScope queries, which are key components of vScope, and discuss how they relate to the perceived problems at the company. Then, we will state and motivate the research questions. In the last section, we will describe and discuss our methodology and motivate decisions for how the data collection was done.

2.1 Context

This section will explore the background of InfraSight Labs and their product vScope. vScope queries will be discussed in detail, as well as the problem that initiated this project. We will start by discussing the company and the product. After this, we will analyze the problems that the company is facing, and assess some of the requirements elicited from an initial case study.

InfraSight Labs is a company founded in 2010, based in Malmö. The company has built a product, vScope, which is an IT Inventorying and asset management system. vScope is integrated with various services from which it collects data, which is then modeled in a graph structure and saved as part of the graph database at the heart of the product. The vScope software is installed and hosted on the company's customers own servers, and new versions of it are updated from a remote build pipeline after each 3-week sprint cycle. Thus, no customer data is shared between the different installations. The graph database is then queried to construct various reports which provide insights into a customers IT infrastructure and how their various assets are connected. vScope is used to gather insights for many purposes, whether it be inventorying, tracking of server usage and application versions or user account activity.

The reports generated by vScope consist of what we will call vScope queries. vScope

queries contain queries in the Gremlin language, identifiers, operators, filters and descriptive properties such as names and descriptions. They are strictly hierarchically dependent, as shown in figure 2.1, and are stored as XML-formatted files when not being used. While in use these files are deserialized to Java objects in vScope, which are in turn used when querying the database, when transforming data and when presenting data. End users can create their own vScope queries in vScope, which are identical in the functionality to vScope queries created internally at InfraSight Labs. However, end users cannot share or update vScope Queries outside of their own installation. The vScope queries that are created by InfraSight Labs are stored in a Git repository which is used to build releases after each sprint cycle. This means that building and releasing new queries takes a long time, and that all installations contain all vScope queries in the central repository. The graph of dependencies between vScope queries can be viewed as consisting of many tree structures, where nodes can share child nodes with other trees as displayed in figure 2.2, node 2 and 4 (counted from the left) share child nodes. If one of the child nodes that are shared is modified, then nodes 2 and 4 are considered impacted by this change if the values returned by the vScope queries have been changed in some way.

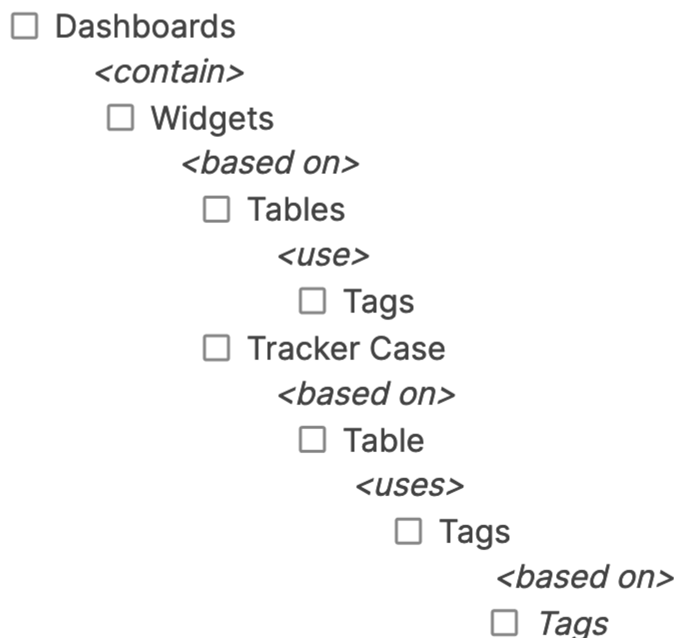

Figure 2.1: The hierarchy of vScope queries.

The software is installed and hosted locally on servers that the customers own, so no data is shared between installations, only vScope queries. The queries can in this context be viewed as empty report shells which include the query to fetch the data that it needs. There is a pipeline that installs new releases of the software after every sprint, which is done automatically with few exceptions. This structure can be seen in figure 2.3. Currently, vScope is built from multiple Git repositories which can be seen in figure 2.4, one of which contains all the default vScope queries from the central repository. The current way of creating and distributing new vScope queries have caused a few challenges that need to be addressed.

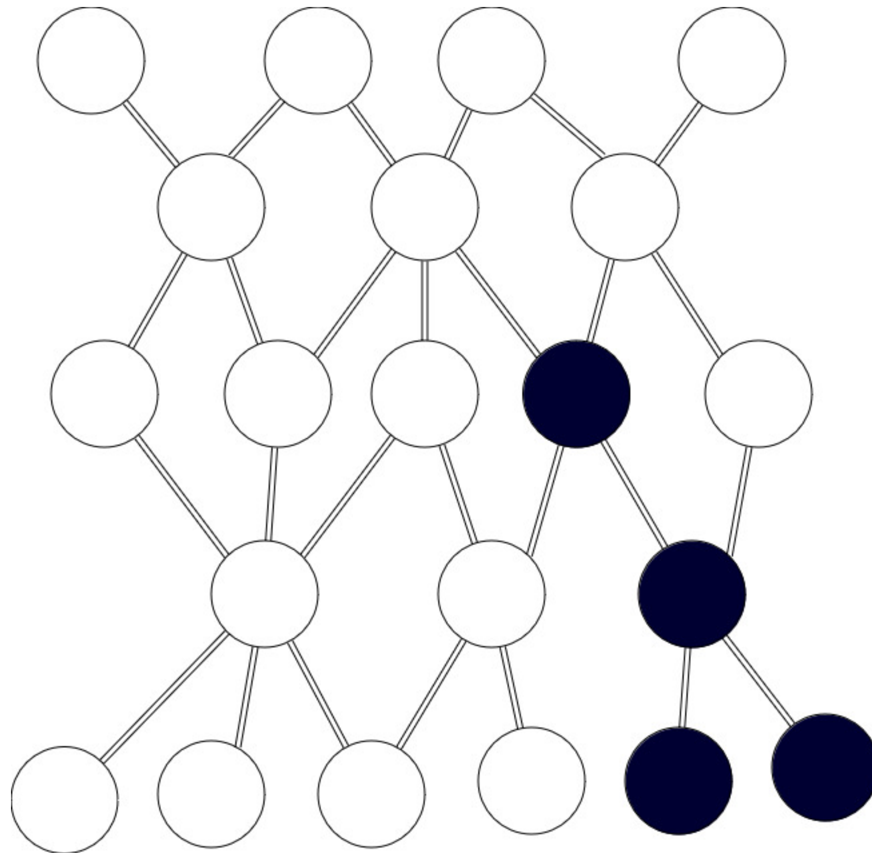

Figure 2.2: An illustration of how vScope Query trees share child nodes

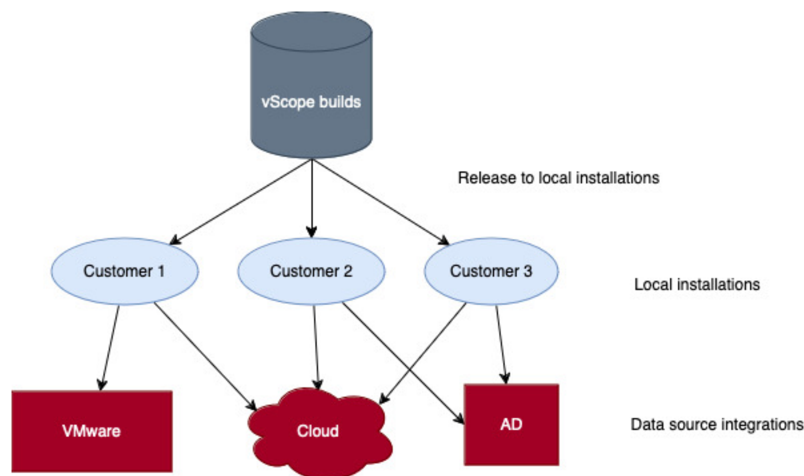

Figure 2.3: How vScope is released, installed and integrated.

End users of vScope have stated that the number of vScope queries available sometimes becomes overwhelming, and that there is a lot to dig through to find what is of interest to their respective companies. There are also a fair share of customers who want to collaborate with each other to produce reports that they have common interest in, for instance municipalities who may have security directives from their government in common that do not

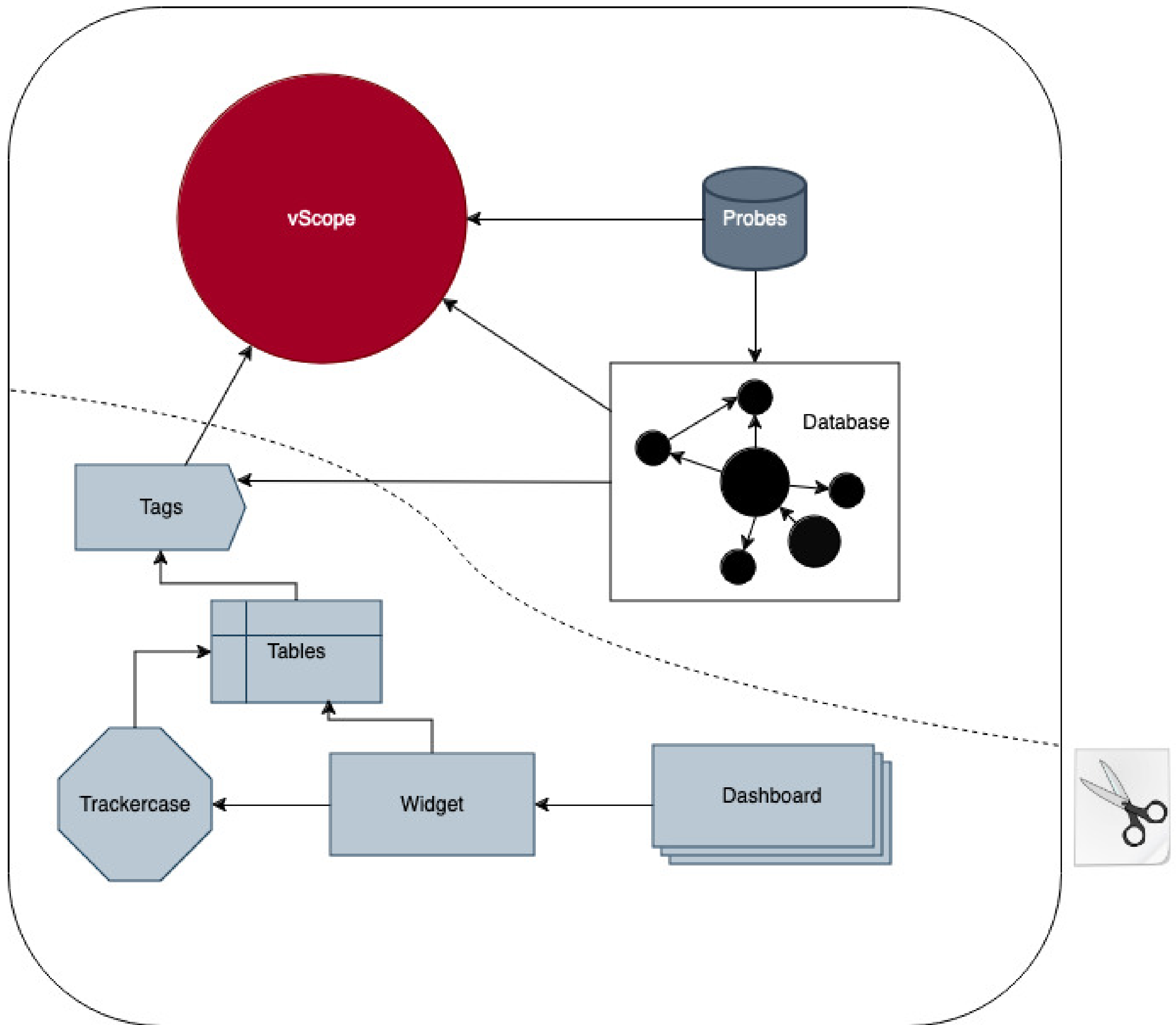

Figure 2.4: The vScope repositories used to build releases. The scissors that cut through the figure and detach the vScope queries from the rest of the repositories aim to visualize the proposed solution to move the queries from the 3 week release cycle and onto a marketplace, or sharespace, platform.

apply to other organizations. Collaboration between organizations is currently not possible without manually copying configuration files between servers, and there are no warnings or features to support such work. There have also been instances where the release cycle causes issues with releasing new vScope queries in a timely manner. An example is a case where having software which used Log4j installed on a machine with older operative systems caused security leaks. vScope queries which could find such instances could easily be built, but in this case a fix release had to be done to push the new vScope query in a timely manner. Thus, the company has an interest in developing a service that is used to maintain and distribute vScope queries. This service would allow end users to search for queries that are of interest to themselves and possibly also contribute to the repository.

Internally at InfraSight Labs, most of the vScope queries are built by the customer success and sales roles due to their immediate contact with end users. These InfraSight employees instead have a better sense of what the users of vScope want to see contributed. Often these employees view the requirement to involve the software development team in changes or additions to the repository of vScope queries as a hindrance. If the vScope query is thought to be useful for other end users then a copy of the related XML files will be sent to one of the developers. The developer would then commit it to the central repository before it is included in the next release. The company considers this process cumbersome, and would like a solution where the customer success and sales employees could maintain and update the vScope query repository without involving the development team. The employees in these roles consider this difficult when working with textual diffs and Git. Since the vScope queries are hierarchically dependent, and multiple instances of higher order vScope queries may be dependent on the same lower order query, there is an expressed desire to also consider these relationships between the vScope queries when reviewing change requests.

InfraSight Labs has as a result of these problems decided to explore the implementation of a marketplace of sorts. The idea is to remove most of, or fully exclude, vScope queries from the current release builds and instead delegate the distribution and release of new queries to its own service. InfraSight Labs would then be intended to create and commit new vScope queries to a central repository from which they can be downloaded by end users of vScope. The vScope Marketplace would include some metadata with each vScope query to categorize them such that end users recognize what vScope queries could be of interest to them.

The marketplace would also ideally serve as a platform for collaboration in the future, allowing external parties to commit new vScope queries to the central repository. This process is intended to involve a validation step by employees at InfraSight Labs, and is intended to be performed by non-developers. Since the intended maintainers are not familiar with Git and it is cumbersome to learn and use for this context, the idea was to explore options for diff tools, possibly something already existing and working with XML files or by building a new prototype. The marketplace is thought to be useful for reducing the number of vScope queries that end users are met with when using vScope, making vScope more user friendly. The marketplace solution would adopt a component architecture, as most installations will use a subset of the central repository rather than the repository in its entirety.

The employees intended to take over this function would then either have to understand how the current workflow functions, using Git and a Git repository with text diffs, or a new tool set would have to be purpose built for users who understand vScope as a tool rather than users who understand version control software. This is particularly problematic when considering the end goal of allowing external end users as they are typically not familiar with using version control tools. Installations of vScope would then include their own version of a repository which would include a subset of the vScope queries available in the central repository, along with vScope queries that the users of the installation have built. Their locally built vScope queries may be dependent on vScope queries fetched from the central repository. In such cases, fetching updates to existing vScope queries would have to involve a step that visualizes changes in a way that is understandable to external end users and the diff to their local repository in its entirety.

The new diff tool will have to fulfill a number of requirements. It should be easy to understand which parts of a vScope query have been modified without having to look at pure text diffs, since a pure text format is not viable for the intended end users of the tool.

As vScope queries are highly dependent on each other, the tool should provide insight into the impact of a change unto other parts of the repository. Furthermore, there are new security concerns when allowing non-technical users, both employees of the company and end users at customer companies, to modify and distribute vScope queries. This prompted us to explore the research questions below.

2.2 Research Questions

1. **How can changes made in a repository be described on a higher level of abstraction than pure text?**
2. **How can the impact of change to an artifact be described when considering other artifacts dependencies upon it?**
3. **Which routines should be put in place to ensure secure handling and updating of artifacts by end users?**

The first and second research questions aim to address the issue with moving the acceptance of change requests to a group of end users who are not familiar with configuration management tools and can not be assumed to have backgrounds as software developers. The goal is that the results of answering these questions will produce a prototype that proves that a diff tool can be built such that these employees understand what has been changed, which parts of the repository are affected by the change, and how the changes modify properties of the vScope queries affected.

The third research question will analyze some of the security concerns which may appear as we work on the project. The goal is to continuously evaluate and analyze security when gathering data for the other questions and to explore what can be automated and which routines regarding validation of new vScope queries should be put in place. We will assess what can be automated and what end users will have to be aware of when viewing changes. As the marketplace platform will be held in context and involves the exchange of query updates between internal and external users, there is a need for ensuring security and avoiding possible avenues for sharing malicious code.

Thus, the overarching goal of this project is to explore a user-friendly diffing tool that enables users who are not familiar with traditional version control tools to understand changes in their vScope query repositories. The primary objectives include creating a prototype diffing tool or a suite of existing tools that facilitate the understanding of query changes, their impact on the repository, and their implications for vScope queries. Additionally, the project seeks to analyze security concerns surrounding query updates, automate validation routines, and ensure security measures are in place.

2.3 Methodology

This section will discuss and motivate our method and the different stages of the data collection. The study is divided into three stages: a requirements phase, a design phase, and a prototype phase. The requirements phase consists of an initial case study, interviews, a

literature study, mock designs and technical meetings. The design phase analyzes use cases and some technical alternatives. The prototype phase consists of a partial implementation without graphical components, and an evaluation phase. The different stages of our chosen methodology, and how the results from each stage are used as input to subsequent stages, are shown in figure 2.5.

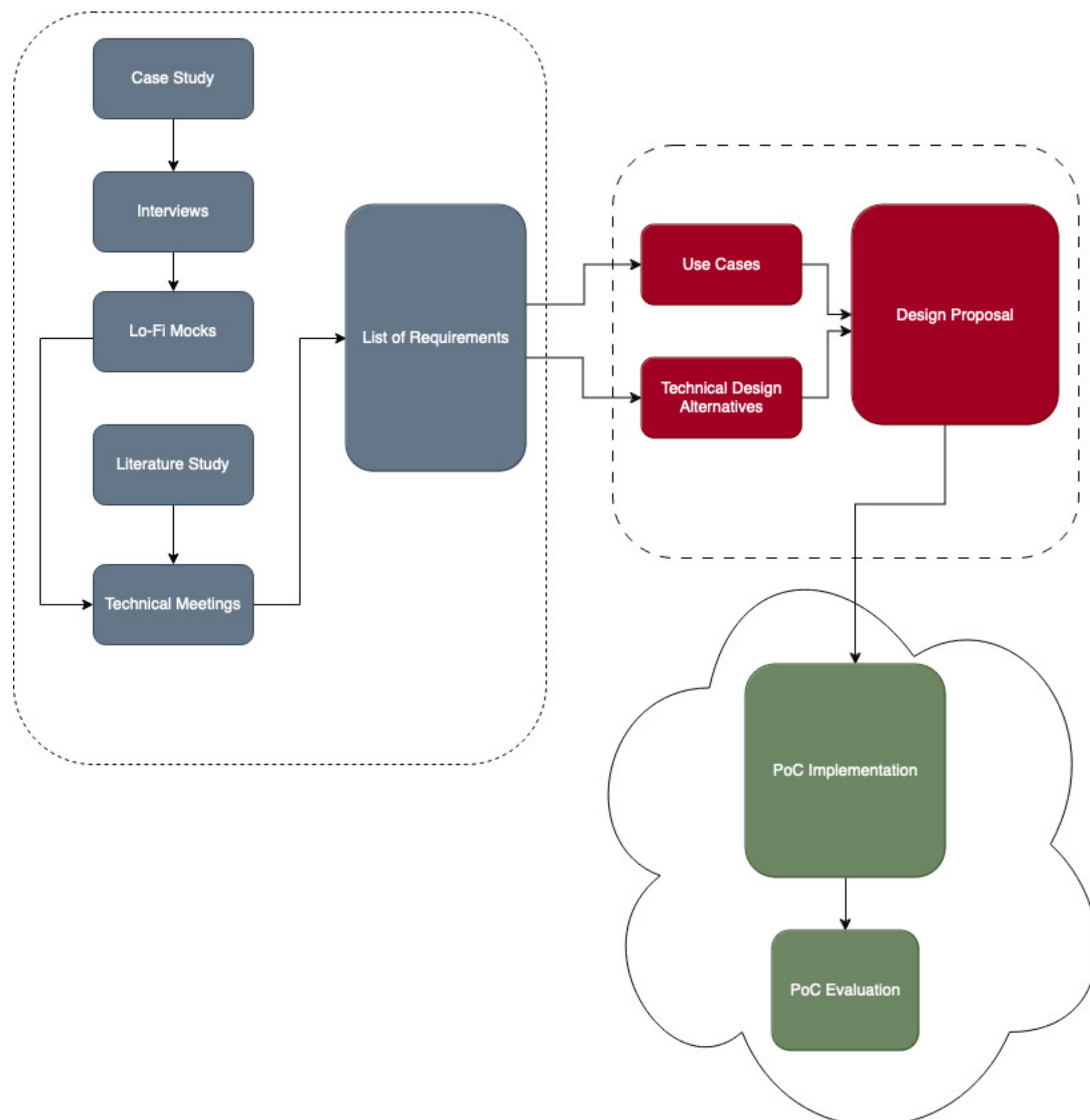

Figure 2.5: Flowchart of the methodology. The different colors (and groupings) indicate the three different stages: blue for the requirements phase, red for the design phase and green for the proof of concept phase.

2.3.1 Requirements Phase

The purpose of the requirements phase was to produce a specification of requirements, both for the Graphical User Interface (GUI) design implementation and for the technical imple-

mentation. This stage focused on data collection in the form of requirements, which were used to provide context when exploring potential solutions. Furthermore, the goal of this stage was mainly to get input from the intended end users and their needs, but also to establish an initial set of technical requirements with regard to the end users requirements. The requirements were used as input for the design and prototype phases.

Case Study

Initially we conducted a case study to gain a better understanding of the initiating problem. The goal of the case study was to provide further context for how and why the customer success and sales team should take over the responsibility for the query repository. We wanted to understand the motivation behind exploring a marketplace platform, and why there was a need to provide a higher level of abstraction for diffs.

We used the case study to identify flaws in the current workflow that potentially could lead to incorrectly updated queries. As stated by Runesson and Höst[10], case studies are fitting to provide context and direction for where to begin with later stages of data collection, as it facilitates the understanding of the organization and the perceived problem. As part of the case study, we conducted an unstructured interview with the company's product owner. The goal with the interview was to assess the context and reasoning behind the marketplace solution, along with challenges regarding the shift of maintainers of the central vScope query repository.

Interviews

To familiarize ourselves with the internal end users and their opinions regarding a marketplace and new responsibility as repository maintainers, we conducted semi-structured interviews. As stated by Runesson and Höst[10], this approach provided flexibility for asking follow-up questions. Another reason for choosing this interviewing structure, was to leverage the product knowledge possessed by the interviewees, encouraging them to share their thoughts and ideas on the product. The proposed shift in responsibilities would lead to new assignments for the customer success and sales team, so we focused on investigating what these employees would need in order to take over the function as maintainers of the vScope query central repository. The interviews touched upon all three research questions, aiming to establish the requirements to comprehensively visualize changes in query files and their dependencies, to reduce the risk of wrongfully updating query files. 8 questions concerning workflows, usability and version control were planned ahead of time, see Appendix A. We interviewed 1 employee in the Customer Success role, and 3 employees working with Sales, which represented the entire sales and customer success team available at the time of the interview phase. Semi-structured interviews can be used to elicit qualitative data, and since the target group for the tool is so small in numbers this was considered preferable to options such as surveys which are often more focused on quantitative data.

Lo-Fi Mocks

The interview phase yielded some partly conflicting results, which lead to the use of low fidelity mockup designs to communicate different ideas for solutions. Low fidelity mockups

are a strong method for eliciting further requirements and ideas for designs and workflows from an end user perspective [1]. The mock-ups aimed to determine how the interviewees' needs could be communicated through some form of interface and what they would need to confidently maintain the central repository. After the lo-fi design discussions, we found requirements which directed the project to a more technical focus, causing us to explore previous studies and literature.

Literature Study

We used literature analysis to explore other attempts to provide context of changes made. The goal was to find other attempts at providing abstractions for diffs, and how interdependent objects that are being modified impact one another. Literature analysis is useful for providing context and to analyze the generalizability of our findings in relation to other work. This phase was necessary to understand the underlying theory and to provide potential avenues for exploration from other related work.

Technical Meetings

The technical meetings that followed sought to establish technical requirements for the tool. From a configuration management standpoint, managing a repository with multiple branches can result in conflicts. If query files are stored accordingly, proper behavior when downloading new query files or receiving a release update needs to be ensured and if data were to be overwritten, there must be the possibility to roll back to an earlier version.

We chose to conduct the technical meetings following a more conversational approach rather than a formal interview structure, since we hoped it could lead to in-depth discussions between us and the developers. We had multiple discussions with the development team regarding the workflow and architecture of a marketplace, and which form of abstraction a diff tool could take. The technical meetings were focused on the requirements for the implementation, with regard to requirements elicited in prior steps of data collection, and included discussions for all research questions. When considering RQ1 the focus was to determine which forms of solutions to explore, particularly if there were any constraints regarding technologies, generalizability or formatting. Regarding RQ2, the meetings aimed to provide insights in determining which queries are impacted by a change. Concerning RQ3, the meetings aimed to refine the broader subject of security to something closer to the company's specific needs.

2.3.2 Design Phase

The purpose of the design phase was to produce a design proposal, aiming at fulfilling the established requirements. This phase was divided into three parts: analysis of use cases, researching technical design alternatives, and finally a design proposal. This phase considered the requirements elicited in the requirements phase, and analyzes various options for solutions which will be used as input for the prototype phase.

To establish design requirements of the tool, and specifically to classify changes and identify potential risks in the current workflow around vScope query updates, use cases were written. Use cases help in identifying how a system should behave upon certain actions, which

made them a suitable approach to assess how the tool should react when changes to different kinds of vScope queries were made.

The research of technical design alternatives included researching approaches that could calculate changes in query files of XML-format and visualize the changes in accordance with our design and technical requirements. We analyzed two options for which format to use when calculating diffs, with the goal to provide a comprehensive analysis of both alternatives.

We then concluded the design phase with a design proposal, where we discussed the technical design alternatives and the use cases. The purpose of the design proposal was to discuss various alternatives we found, and why one option was chosen over another. The design proposal was then used as input for the prototype phase.

2.3.3 Prototype Phase

The last phase consisted of two parts, the implementation of the prototype and an evaluation. The prototype was built around the algorithmic parts of our requirements, and addressed the viability and possibility of solving and possibly answering all three research questions. The prototype was built for providing diffs at an adequate level of abstraction, that can calculate the impact of the suggested change and classify the type of change depending on which properties that had been modified. To establish the viability of the Proof of Concept, it was evaluated against the list of requirements produced in the requirements phase.

The implementation step outlined the steps involved in the algorithm, the interesting decisions made during implementation and the motivation for making certain choices. The purpose of this step was to provide context for how the prototype was implemented.

The final step was to evaluate the prototype, with the purpose to analyze the prototype's ability to fulfill requirements found during the requirements and design phases. The analysis includes discussions with employees at the company, such as to validate the prototype.

Chapter 3
Requirements Phase

This chapter will discuss the outcomes of the data collection phase. The goal is to produce a specification of requirements that will serve as a foundation for the design and implementation of a prototype diffing tool. First, we will discuss the requirements derived from the interviews with end users. Then, we will evaluate the mock designs that were created based on interview findings, and discuss related requirements. Next, we will discuss the outcomes of a literature study of related work and existing diff tools, from which additional requirements will be produced. Finally, the specification of requirements will be evaluated and validated together with the developers at the company, which will yield a requirements specification used in upcoming steps.

3.1 Interviews

This section will discuss and analyze the interviews held at the start of the data collection phase. The main goal with the interviews was to produce requirements focused on what type of information the intended maintainers of the central vScope repository would need for reviewing change requests. Furthermore, design requirements and suggestions for designs were elicited from the interviews as well.

The interviews, for which the questions asked and discussed can be found in the appendix A, started off with an introduction of the thesis project to provide context. This approach aimed to ensure that participants understood the purpose and scope of the research, facilitating more informed responses. A total of 4 interviews were conducted with employees in the Customer Success (1) and Sales roles (3) respectively, which is noteworthy as a small sample size but represents all employees at the company in those roles at the time of interviewing. This is considered representative, as it includes all employees in the roles.

The interviewees described the marketplace service as a platform for vScope query creation and distribution that could be used to independently create and share tailored vScope queries to meet newly received use cases from end users of vScope. The interviewees ex-

pected benefits included time savings, improved customer satisfaction, and enhanced sales effectiveness through the availability of standardized content templates and best practices. Time savings and enhanced sales efficiency are expected to come from shortening the process of creating new vScope queries and delivering them without having to involve the software developers. Furthermore, they could produce more niche vScope queries that may only interest a few end users of vScope without further overwhelming end users who already find the sheer number of vScope queries overwhelming today. These findings are in line with our findings from the case study.

The participants emphasized the importance of a review process for change requests both from internal users, which the case study found would preferably be the employees interviewed here, and for external users. The interviewees unanimously expressed that they had a hard time understanding the XML format that vScope queries are stored in, and that they would need some form of tool to clarify which changes are made and what parts of a specific vScope query has been modified. Some of the interviewees mentioned that they would like to see comments explaining why changes were made to each respective vScope query. Furthermore, when prompted to describe how se3uch a diff could be presented 3 interviewees mentioned a feature called "History and Compare" in vScope, which is used to trace changes in data collected over a set period of time. This feature can be seen in use in figure 3.1. When using this feature, which is only available in table format, values which have been modified during the selected time frame will be highlighted in one of 3 colors. Green if added, lighter gray if deleted, and blue if modified. This yields some directive for RQ1, and provides some of the basis for our later mockups.

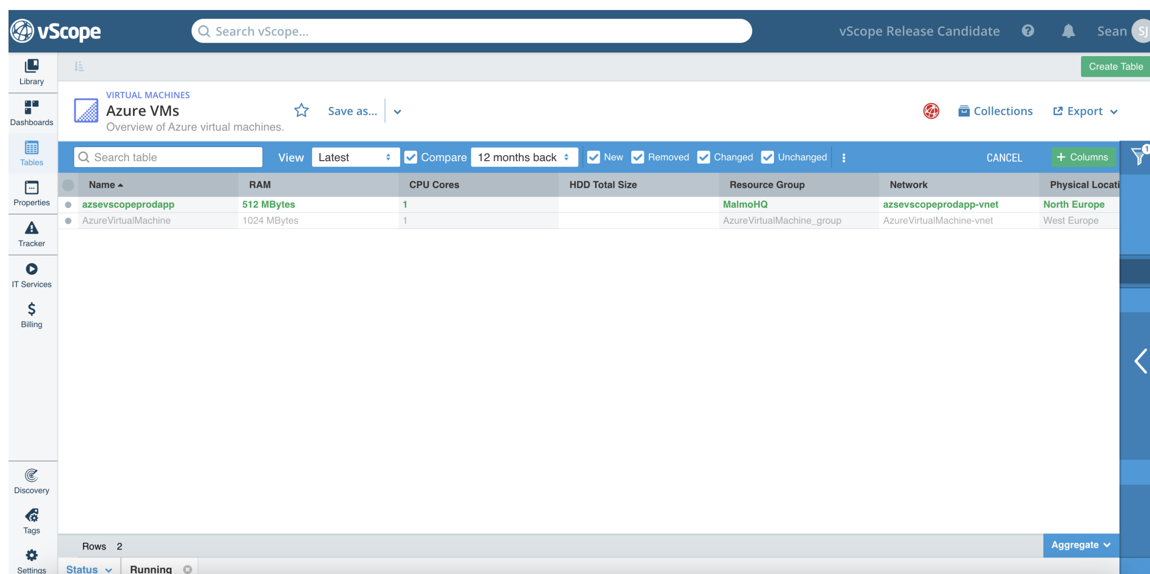

Figure 3.1: The history and compare feature in vScopes table explorer. The first row is new within the scope of the last 12 months and colored green, whereas the second row has been deleted sometime in the last 12 months and is a lighter gray instead.

The follow up question touched upon RQ2, and asked whether or not the interviewees would need to see the vScope queries that are dependent on the modified vScope query, as part of a change request. 3 interviewees responded that they would, while 1 considered it

unnecessary. When asked why, the 3 interviewees who said that they would like to view the dependents stated that they could be affected such that they no longer fulfill their intended function. The interviewee who responded that it was unnecessary stated that they did not think that it would be a problem. Moreover, it was stated by the interviewees that they would want to be notified of dependent vScope queries that may have been affected and that they wanted some form of classification of the change. The classification would help in the validation process, and guiding the maintainer in what to validate. For example, there would be no need to validate dependents if descriptions of a query have been changed. There was no expressed need to look at dependencies that have not been modified, which is expected as dependencies are not affected by their dependents. These responses validate the need to answer RQ 2, and what form of relationships are most important to consider for this specific study.

We also asked about some specific vScope query types, referring to tags. This is due to tags being the lowest form of vScope query, which contain properties hard-coded to how they are stored in the database such that they match with asset definitions set when integrating new data sources with vScope. There are a few different types of tags, where the dynamic tag is the most low-level form of tag that contains the most specific Gremlin query code in the sense that it is supposed to directly match with objects that developers have created manually when building new integration's to new data sources. The interviewees described it as a difficult thing to take over the responsibility for creating these tags, but that they could manage external change requests as it would be unlikely that end-users of vScope outside of the company would be able to create these tags themselves. The interviewees expressed that they want more support in finding when changes to functional components of vScope queries have been modified, such as to avoid harmful or breaking behavior which is difficult to recognize if the reviewer of a change request is not familiar with code.

In summary, the interviews found a number of requirements. Those being the necessity of a clear and comprehensive review process, the need for a tool that displays changes such that the employees in the sales and customer success roles can understand them with change request comments. When considering design requirements it was specified that the design philosophy of the “History and Compare” feature should be followed.

3.2 Lo-Fi Mocks

This section will discuss the requirements found from evaluation of our mock design. The purpose of the low fidelity mockups was to find design requirements that had not been elicited from interviews. We shared and discussed the designs with the company's user experience (UX) designer and product owner, who are the people most involved with making pre-development decisions regarding the design and workflow of company products. The key takeaway from the interviews was that the users wanted a comprehensive and intuitive review process, where they could easily understand what had been changed and what had been affected. The mock designs aimed at communicating these requirements in the context of the product. We were inspired by the current design of the "History and Compare"-view from the product, see figure 3.1, and used the same color coding for changes, updates and deletions to create a design proposal that end users would find familiar. Given the interdependence of queries, modifying one query could potentially impact several others. It was

proposed by the UX designer to create a checklist in the sidebar containing everything that needed reviewing, to avoid overloading the UI. To further improve usability, there could be a gray button at the bottom of that page that turned green and became clickable once the user had reviewed all independent elements of the list. See figure 3.2 for proposed design.

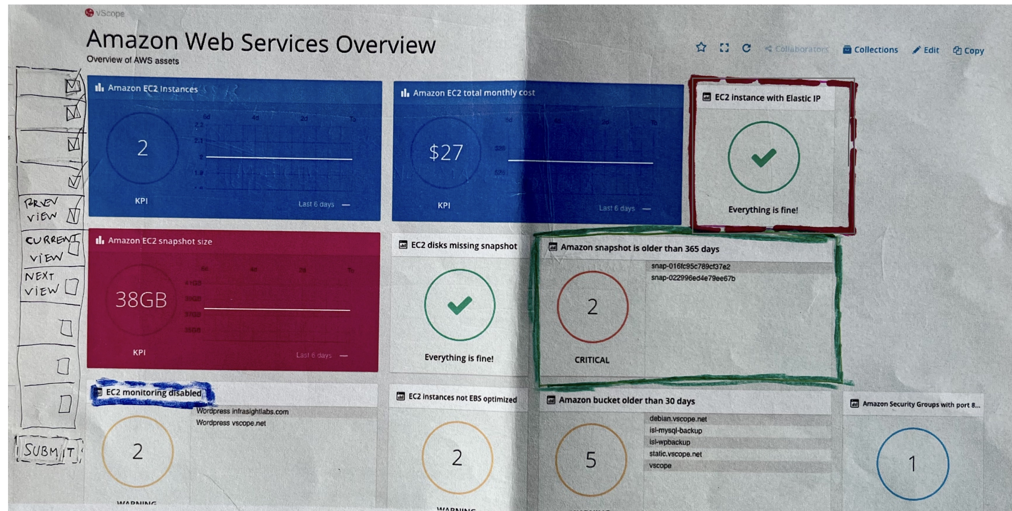

Figure 3.2: Lo-Fi mock design of a dashboard view in vScope. The checklist on the left indicates where the user is currently reviewing in the dependency chain. Blue indicates changed, green indicates added and gray indicates removed.

The purpose of this phase was initially to test our designs against the end users and improve them based on iterative feedback. However, at this stage we noticed how we started to drift away from the focus of our research questions, regarding how to communicate change requests, and instead we spent time on usability and design. Following discussions with the UX-designer and product owner, it was decided that our main contributions would be to retrieve and present the information needed for the auditing process, meaning information detailing what has been changed. Integrating it into the UI and producing a design was decided to be out of scope for this thesis. Consequently we only validated the mock designs with the UX-designer at the company, and not with the interviewees, to collect feedback from someone experienced in design best practices and usability. In summary, this phase reiterated that a better review process for new vScope queries was important, and that the impact on other parts of the vScope query repository should be evaluated and communicated efficiently. Furthermore, this phase emphasized the requirement of producing automated analysis of change requests.

3.3 Literature Study

This section will discuss some of the material found in related literature and how it can be related to this report. We will build off of our findings in the case study, interviews and low-fidelity mockups. The purpose of the literature study was to investigate other attempts at implementing and using XML-based diffing tools to evaluate diffs between versions of a document, and to analyze the findings and implementations in the context of this report.

The goal of these analyses is to produce potential solutions and possible new requirements from literature that have not been found in the other steps but are applicable to this project.

During this phase we were exploring options working with XML-formatted documents along with metadata files when diffing vScope queries. As such, we assessed XML diff tools and change types defined by such tools. The interviews and low-fi mocks suggested displaying the modified properties, some classification and the affected dependents of modified vScope queries could be sufficient for employees to manage change requests. Cuculovic et al[4]. proposed the edit operations insert, delete, move and update. Where insert and delete are considered level 1 complexity operations as opposed to move and update which would be considered level 2 complexity operations. The motivation for the level 2 complexity operations is that instead of describing every change where an XML element is moved or updated as a deletion followed by an insertion of a very similar XML element the move and update operations could provide a more concise and readable diff. This is preferable in the case of this study as many changes will be to pre-existing nodes in vScope queries such as the description field. It is also preferable to provide a more clear and concise diff when communicating changes to the intended maintainers of the central repository since it avoids overwhelming change sets for non-technical employees.

Cuculovic et al[4]. measure the performance of XML diff tools by testing their time and the memory usage when evaluating. They also discuss some of the possible configurations for the tools assessed. The authors compare the XML-diff tools jats-diff, similarity, jndiff, xcc, and xydiff when evaluating diffs on 100-400KB documents. The tests do cause some concerns when considering the number of documents involved in the vScope query repository, which at the time of writing contains just below 5000 documents. Since some of the higher order vScope queries contain hundreds of documents which would have to be diffed with old versions of themselves which suggests a potential diff calculation time of several minutes with the use of the XML diff tools evaluated by Cuculovic et al. Some of the evaluated tools provide support for ordering of XML nodes while others do not, which may be of concern if for instance ordering of columns is considered significant. Whether the ordering of XML nodes is considered significant or not can have a significant impact on the performance of XML diff tools, as unordered comparisons can compare nodes by their paths which avoids unnecessary comparisons[13]. Some vScope queries, such as tables, contain multiple sets of elements where ordering is only significant for some cases. Examples are the order in which columns appear, where ordering is significant, and the order of filters applied, which is considered insignificant.

When considering dependencies between vScope queries we seem to be reliant on separate files that denote the set of dependencies a specific vScope query may have. This is similar to findings from A. Benelallam et al.[2], who explored a graph based representation of Maven central by using the Maven POM-files. In the case of InfraSight Labs this is acceptable since vScope queries can be exported paired with metadata files containing the set of dependencies. When using XML diffing tools discussed in this section these metadata files could be used in conjunction with the tool to map the dependencies and dependents of all vScope queries, which could be used to determine the potentially impacted vScope queries from changes found by the XML diffing tool.

On the topic of security, XML diffing tools do not provide much support other than checking the validity of the XML format. Since modifications to certain XML nodes are inherently more sensitive from a security perspective, this would mean that the maintainers

of the central vScope query repository would need to recognize these fields and the implications of any suggested changes to them. The XML nodes could be labeled as more important from such a perspective, which could be used to provide further context of the change. However, this is reliant on developers recognizing that new fields added to new models of vScope queries can be harmful if modified incorrectly, and adding them to this set of nodes.

We found that some form of classification of changes is needed, both from a security perspective and from the perspective of what impact the change has on the repository. Furthermore, we also found that performance requirements should be investigated.

3.4 Evaluation

This section will discuss the findings from meetings with the development team. The purpose of this phase was to validate the requirements specification together with the developers at the company and update it based on their feedback, aiming to produce a revised list of requirements. The evaluation phase elicited mainly technical requirements that the employees involved in prior steps may not be aware of.

When evaluating already existing diffing tools it had been established that they did not meet the requirements in terms of speed. This prompted the investigation of how to execute the diffing faster. An idea that was discussed during the evaluation meetings was to diff the hashes of two versions of query files, to see if it could shorten the execution time [7]. The theory was that if change requests only modify a few files out of a few hundred it could possibly save time.

Regarding RQ2, and the possibility to view what impact changing one query might have on other queries, we discussed the necessity to store and compare information concerning dependencies. This information is currently contained within metadata files that are paired with each vScope query file. One concern revolves around the interdependence of the vScope query files. The fact that queries are dependent on queries leads to dependency chains that need to be checked each time changes are proposed. An important consideration for the prototype implementation and the requirement for secure query updates is how to manage these interdependencies, and how putting restrictions on changes could possibly simplify the reviewing process. The developers did not consider dependents on modified queries to be unnecessary to validate, suggesting that there may be some need to motivate why validating dependent queries is necessary when introducing the tool to new employees.

During these meetings it was again decided to prioritize the extraction and presentation of necessary change-related information to the end user, rather than incorporating it directly into a design prototype. The future design of the tool would need to follow certain design conventions, and the implementation and testing would be carried out by the UX designer rather than by us. Therefore it was decided to focus on what information would be needed about the change in order to present it to the end user. The information was regarding both the changed query and its dependent queries.

Since external users are expected to be able to suggest changes from anywhere in the world, we cannot assume that we can employ strict long transactions [5]. Strict long transactions here refer to a workflow where the common ancestor of two branches is the branch head of the branch which is being merged into, thus only requiring a direct diff between versions of two file sets. Thus, we will assume that 3-point diffs are necessary to view changes. The

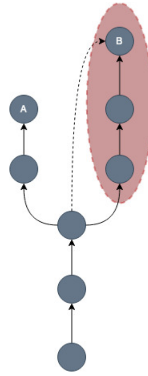

Figure 3.3: A visualization of a 3-point diff from A to B. The diffing elements between A and B are the three nodes circled in red, i.e. the difference between the most recent common ancestor and the latest commit.

`git diff` command works like a 3-point diff, and outputs the difference between the common ancestor, and the most recent commit see 3.3 [6]. The discussions also concluded that a single vScope query and its dependencies get committed at a time, meaning that all changes are treated as a component being committed to a repository. This is due to the complexity of displaying changes in multiple subcomponents of higher order vScope queries and how a change has impacted the repository, such as in the case shown in figure 2.2.

From the above steps we have produced a specification of requirements, see Appendix B. This specification aims to serve as a foundation for the design phase discussed in the next chapter.

Chapter 4

Design Phase

This chapter will discuss the design choices that were considered to fulfill the requirements found in the previous chapter. The goal with the design phase is to conceptualize a design that can be translated into a prototype tool with the required functionality. Use cases catching behaviors that require extra consideration, especially concerning secure handling of queries, will be discussed and motivated. The learnings from the use cases will be used to categorize possible changes to query files. Technical design choices, including libraries and programming considerations, will be discussed and motivated. Finally, we will discuss a design proposal, from which we will build a prototype in the next chapter.

4.1 Use Cases

This section will discuss use cases that helped us express expected behaviour when considering certain design requirements. The use cases serve to explore how the users will likely interact with the tool and the expected outcome of their actions. Together with a predefined set of change classifications, the findings from the use cases will help us in specifying technical design choices for the design proposal of the diff tool.

Prior to formulating the use cases, we defined three different roles: *maintainer*, *administrator* and *end-user*. These roles illustrate users with different privileges, and the privileges affect what the user is allowed to do.

- The *maintainer* is someone at the company's customer success or sales team, or product owner. The maintainer is not part of the development team but rather someone in contact with external users, i.e. customers of vScope.

- The *administrator* is someone that works with source code updates in the company's back end development team. The administrator has the highest privilege possible.

- The *end-user* is a vScope customer, i.e. external user.

We also defined three different types of changes: descriptive, functional and requiresAdminReview. This classification is needed since the changes will affect the vScope query repository in different ways.

- A *functional* change is a change on queries that potentially transform or fetch data, or on queries that are dependent on queries that potentially transform or fetch data. Functional changes may add or remove dependencies to the query.
- A *descriptive* change is a change on the parts of queries that do not transform or fetch data, such as a description. No new dependencies are introduced.
- An *requiresAdminReview* change is a query change that is not allowed without review from an administrator.

The creation of the following use cases will help us in identifying potential risks in the current work flow around vScope query updates. Some scenarios need extra cautiousness and restrictions on who is allowed to do what, which we need to keep in mind when implementing the prototype. The different scenarios will require different kinds of validations or may be prohibited altogether depending on the user's privileges. The classification of changes (descriptive, functional, requiresAdminReview) will be used to annotate vScope query attributes in the upcoming implementation of the prototype tool.

Given the three kinds of changes and the three different roles, the following use cases were created. In each use case the actor is interacting with the marketplace platform which is the platform where future query uploads and audits will take place.

- **Name Change**

  Scenario: Anton, a maintainer, wants to change the name of a tag from "Azure AD" to "Microsoft Entra ID" due to a product name change. He changes the <name>-field in the concerned vScope-query files and uploads a change request with the query bundle to the marketplace.

  Concern: In the vScope query infrastructure, the name is linked to the ID, so the name should not be updated since it could break dependency chains and consequently break functionality in a customer's installation.

  Action Required: It is not permitted to change the ID-field without proper privileges, which are exclusive to developers at InfraSight Labs. Anton's query upload gets rejected.

  Type of change: requiresAdminReview and functional

- **Name Change in Description**

  Scenario: Instead of directly changing the name, Anton updates the name in the <description>-field of the concerned vScope-query files and uploads a change request with the query bundle to the marketplace.

  Concern: The meaning of the description of the tag must stay comprehensible.

Action Required: Validation of the description of the tag. If all looks good, Anton confirms the upload, and the bundle is pushed to the vScope query central repository. If not, Anton is asked to revise the queries and try uploading it again.

Type of change: descriptive

- **Restricted Data Field Update Attempt**

  Scenario: An end-user, i.e., an external vScope user, updates the <data>-field on a dynamic tag, which is a specific vScope query type that fetches data, and uploads a change request with the query bundle to the marketplace.

  Concern: Modifications of dynamic tags could possibly modify the <data>-field, which could lead to the execution of malicious Gremlin code.

  Action Required: It is not permitted to change the <data>-field without proper privileges, which are exclusive to developers at InfraSight Labs. The end-user's vScope query upload gets rejected.

  Type of Change: requiresAdminReview and functional.

- **Table Filter Addition and Sorting Order Change**

  Scenario: An end-user adds a filter to a table and changes the sorting order from "ascending" to "descending" and uploads a change request with the query bundle to the marketplace.

  Concern: The change in sorting order might change the meaning of the table since new dependent queries might be shown in the table. New dependencies are potentially introduced.

  Action Required: Validation of the sorting order change and that it does not change the meaning of the table. Manual check on the dependent queries if introduced. If all looks good, the end-user confirms the query upload, and the bundle is pushed to the Marketplace repository. If not, the user is asked to revise the queries and try uploading it again.

  Type of change: functional.

- **Dashboard Widget Addition**

  Scenario: An end-user adds a widget to a dashboard and uploads a change request with the query bundle to the marketplace.

  Concern: The meaning of the dashboard might have changed upon the newly added widget. At least one new dependency, i.e., the widget, is introduced.

  Action Required: Validation of the added widget and that it does not change the meaning of the dashboard. Manual check on the dependent queries if introduced. If all looks good, the end-user confirms the query upload, and the bundle is pushed to the Marketplace repository. If not, the user is asked to revise the queries and try uploading it again.

  Type of change: functional.

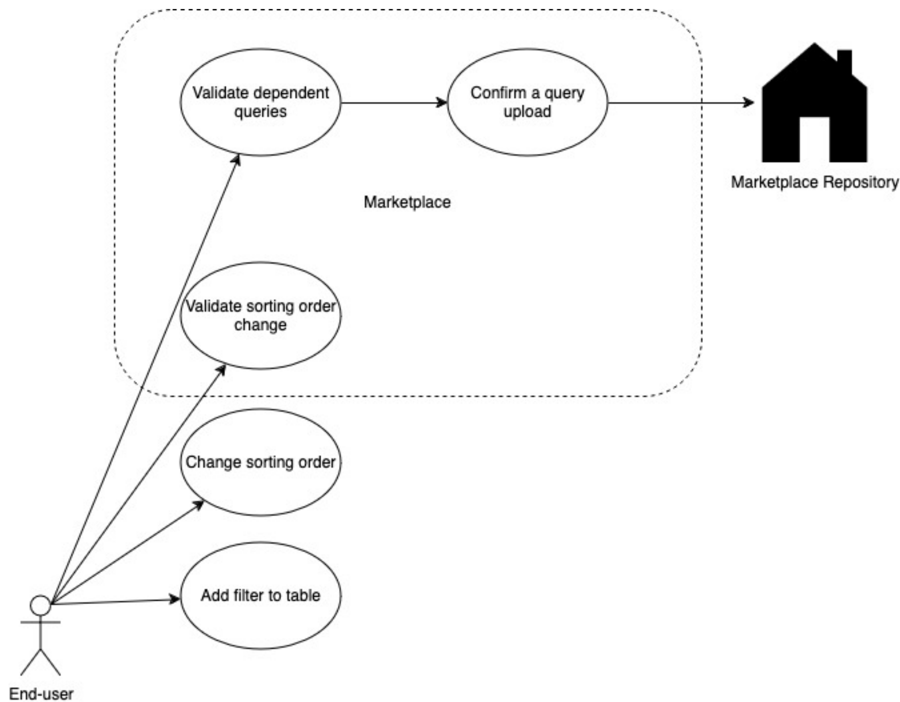

Figure 4.1: An illustration of the use case "Table Filter Addition and Sorting Order Change". Three of the actions are done inside the Marketplace platform.

4.2 Technical Design Alternatives

This section will discuss technical design alternatives and considerations related to the planning and programming of the prototype diff tool. First we will discuss and analyze an approach using XML, followed by an approach that involves using reflection and annotations. Furthermore, we will analyze the implementation of a dependency graph and how it interacts with the two options for analyzing diffs explored in this report. Finally, we will analyze the security implications of the respective solutions.

4.2.1 Using XML

This section will analyze the potential use of XML tools and Java libraries to implement a diff tool for the context of this report. We will discuss the implications of using XML in the context of this project, and analyze the advantages and drawbacks of such an implementation.

This approach was chosen due to vScope queries being stored as XML files in the repository, motivated by the availability of multiple XML diff tools with configurations that could fulfill many of the requirements found in chapter 3.

The tools explored in the literature study did not seem appropriate for our requirements, and we could not find other XML diff tools that could calculate diffs with adequate speed with the configurations we require. We chose to explore a new implementation of an XML diff tool with the specific requirements of this project in focus.

The first requirement to consider is the level of abstraction that diffs are shown at. When

calculating diffs using XML we base it around the inherent tree structure that XML documents can be viewed as, and thus base the diffs around comparing the existing node paths and their values. We use the edit operations insert, delete, move and update as outlined by Cuculovic et al[4]. which was analyzed in the requirements phase.

```
<List>
        <String> Tag 1 </String>
        <String> Tag 2 </String>                    unchanged
        <String> Tag 3 </String>
<\List>
__________________________________

<List>
delete:    <String> Tag 1 </String>
move:      <String> Tag 2 </String>
move:      <String> Tag 3 </String>                 with move, update
insert:    <String> Tag 4 </String>
</List>
__________________________________

<List>
delete:    <String> Tag 1 </String>
insert:    <String> Tag 2 </String>
delete:    <String> Tag 2 </String>
insert:    <String> Tag 3 </String>                 with delete, insert
delete:    <String> Tag 3 </String>
insert:    <String> Tag 4 </String>
</List>
__________________________________
```

Figure 4.2: The difference in diff result when using move, update operations compared to only insert and delete.

We counted the number of unique node types in the vScope query central repository to 123 in total at the time of writing. We would want to categorize these nodes such that we know when a modification to a node indicates that dependent vScope queries are potentially impacted. The purpose of categorizing or classifying nodes in this way is to analyze the impact of changes to the graph of vScope queries, and to facilitate the generation of a list of queries to validate before a change request can be accepted. Such classification can also be used to highlight changes that contain potential security issues as well, such as the XML nodes which contain executable queries in the form of Gremlin code to the graph database. One issue we found with the categorization of node types is that there are instances where nodes that have the same name fulfill different roles. In vScope the queries have two different types of unique identifiers depending on the query type. Most queries have a unique ID node that is essentially just a randomized unique number. The tag type of queries instead use the value contained in the name node as both its display name and unique identifier, and omit the ID note entirely. Changing the name for these tags is a far more significant change as a result, and should therefore be classified differently. As such, these variations when classifying diffs introduce more complexity due to the need to consider more than one node value at a time.

When using XML we defer the mapping of dependencies to metadata files, which are also available in XML format. These metadata files are deserialized and used to generate a dependency graph. The graph is then used to fetch the impacted vScope queries when a change involves nodes requiring the validation of dependent queries. Since dependencies are

hierarchical, it is always the dependents of modified queries we have to validate.

Security concerns would be partially addressed by defining a set of XML nodes that could be modified to execute malicious code, in the same way that we define the set of nodes that trigger a necessity to validate dependent queries. This approach would have the same potential issues as when XML-node names that are shared by different query types have different impact classifications. Furthermore, we assume that nodes which are not categorized impact the dependents on the modified query when modified and that they are not allowed to be modified. This decision was made to avoid occurrences where an unclassified node has a significant impact or risk when modified but is allowed to be changed without warning, which is considered a risk.

4.2.2 Using Reflection And Annotations

This section will analyze the potential use of reflections and annotation properties in the Java language and discuss their use as an alternative to XML based solutions. We explored this option due to finding some concerns with using the XML format in this case, which will be discussed in the design proposal section.

When using reflections and annotations to produce diffs we calculate diffs from Java objects, which is what the XML documents represent in the case of this project. Reflection is a Java language feature that allows a class to reflect upon its own structure and properties at runtime. This capability makes it possible to inspect fields, both private and public, and their values. This method would introduce an interface Diffable that implements a default method for object comparison between two instances of Java classes that implement3 the interface. We also introduce a method that gets all the field values as a hashmap where the keys are the field names and the values are the value of the field. This method stub cannot be implemented as a default method in the interface due to security features in Java that prevent the interface from accessing the private fields of its subtypes, though it is possible to inherit this method from an abstract class which we also introduce in this project. A requirement for this implementation to function properly is that the objects contained in the vScope query fields override the equals method if necessary, such that the object comparison is not made by object reference. The approach of using reflections is valid for comparison of further depth than just the immediate fields in an object, provided we inherit the Diffable interface for such class instances as well.

For this approach we introduce annotations to classify changes as well, as shown in figure 4.3. Annotations are a form of metadata that can provide information about code without affecting its functionality, and allow developers to embed additional information which can be utilized to access information about the source code and how to interact with it. It is commonly used to provide context for frameworks, the runtime environment or the compiler. The annotations are configured to match class fields, to persist during runtime, and to be inheritable.The first annotation introduced was the Functional annotation, which would be used to mark fields which impact dependent queries when modified. Other annotations were Descriptive for descriptive elements, Ignore for some fields that were not relevant, and RequiresAdminReview to highlight changes external users are not allowed to make. When applying these annotations we found that functional elements were those that either fetched data or somehow transformed it, whereas descriptive elements typically contained some text such as names and descriptions. We found that the annotations could be mixed in some cases,

for instance the data fields for certain tags are functional but not permitted to be modified by external users which led to them being annotated with @Functional and @RequiresAdminReview. These annotations would then have to be either inherited or added field by field to new vScope query models. This solution allows for more flexibility when it comes to identical field names that should be treated differently when compared to the solution discussed when using XML, since field annotations can be deferred to the specific class instance instead of its general name.

```
2
3⊕ import java.lang.annotation.ElementType;
8
9  @Retention(RetentionPolicy.RUNTIME)
10 @Target(ElementType.FIELD)
11 @Inherited
12 public @interface Functional {
13 }
14
```

Figure 4.3: An example of an annotation used.

The approach using reflections and annotations does in this case include Java classes which have a method to get dependencies available in order to facilitate the creation of metadata files. This means that we can fetch dependencies directly when creating the dependency graph. The graph still has to be created however, since there are no mappings for dependent vScope queries in the class instances. When diffs are found that include fields annotated as functional, the dependency graph is used to get dependent vScope queries that would have to be validated.

One of the concerns considered for this project is security when receiving change requests. These concerns can be handled to an extent with the use of annotations to denote the fields which are prone to being modified to execute malicious code. We use the annotation RequiresAdminReview to highlight fields which are not permitted to be modified by external users, which could be redefined to instead feature some necessary level of authorization to accept a change request that includes such changes. Users of the diff tool, the maintainers of the vScope query repository, would then have to understand the implications of changes to these fields and the effect of the specific change contained in the request.

Similarly to the XML based solution, we assume the harshest limitations and most significant impact on changes that are not annotated. This means that if a field has no annotations and the annotations used in this project are in place, we assume that changes are to be treated as functional and requiresAdminReview.

4.3 Design Proposal

This section will present and motivate our design proposal for system behavior and the requirements for a prototype diff tool. The design proposal will be used as input for the prototype phase in the next chapter.

As outlined in section 4.1 we will consider three types of changes, descriptive, functional and requiresAdminReview. The change operations are add, delete and update. We also define 3 change types: Functional, Descriptive and RequiresAdminReview. While there may be a need to model the change types with further granularity or levels of change in the future, this was not found during the requirements phase. It is simple to add more change types or to provide some level of administrative access required to accept changes if it is found to be required in the future.

When comparing the two technical design alternatives, we found that using reflections and annotations was preferable over the use of XML. This is due to the need for classification of changes based on class implementations rather than XML node names. When we explored the use of XML based diffs, we found that node names should be classified differently depending on their query type. After discussing the two options with the development team, it was found preferable to allow developers to define how a new vScope query class would be diffed and how their fields would be classified at the time of implementation, rather than by the use of predetermined XML node names. This does introduce more boilerplate code, but was considered less complex by the development team.

We propose that a prototype solution should be implemented, which implements the following design choices. The prototype will introduce a few key components. The interface `Diffable` will be used to define the methods required to compare two vScope query instances. The methods will have the following functions: one will fetch the field names and values, another will fetch the field names and their change classification. Finally, a default method that compares an instance of the `Diffable` interface with another object that implements the interface and produces some result. We will also build a dependency graph that maps the dependencies and dependents of vScope queries to produce the set of affected queries from functionally changed queries. We will use the query classes as defined in vScope to deserialize the XML files to Java class instances with the JAXB XML parsing library [1] . The library was chosen due to its use when serializing vScope queries, therefore being familiar to the development team. After this, we will produce a prototype that can take the input of 3 folders containing vScope query files, parse them, diff them and produce a result with the information required for the intended end users to understand the implication of a change request.

[1] `https://eclipse-ee4j.github.io/jaxb-ri/`

Chapter 5
Prototype

This chapter will discuss the implementation and evaluation of our query diffing tool, which will serve as a proof of concept. First, we will motivate the implementation of the tool by describing and motivating how diffs are calculated. Then, we will evaluate the tool with regard to our requirements elicited in previous phases. We will analyze and motivate our test cases, the test results, and how the diff calculation could be improved upon in the future. We will also discuss and analyze the meetings held with the developers during the evaluation of the diffing tool.

5.1 Implementation

This section will discuss the implementation of the prototype tool along with key components and decisions that were made during implementation. The purpose of this is to provide both a general outline of how the code is written, and to provide insight as to where requirements elicited in this project affected the implementation. The underlying structure will be motivated by an explanation of some key components of the tool and with the help of code snippets. The building blocks of the tool include the diffable query interface, the diff of query files and the diff of dependencies. Finally, we will explore some options for how to improve the prototype.

In the diff tool, a number of sequential steps are performed to produce diffs, as seen below. This list of steps was produced as a result of the design proposal from chapter 4 along with the requirements found in chapters 3 and 4.

1. Deserialize three sets of XML files containing vScope query objects and metadata (common ancestor, current version, proposed change set).

2. Generate three dependency graphs and evaluate change sets from the common ancestor query set to the other sets.

3. Store dependency graphs in two hashmaps: one for tracking dependencies and another for tracking dependents.
4. Process diffs for every unique identifier found in any of the three file sets:
   - If the object is not null in either set compared.
   - Compare fields and determine operations: insert, delete, update.
   - Handle non-primitive types (sets, lists, and nested query instances) with specific diffing logic.
5. Determine behavior based on object type:
   - Ignore the object if specified.
   - Treat null, primitives, and directly comparable non-primitive types differently.
   - Treat instances of Diffable as separate entities for diffing by adding them to the working set.
   - Fetch change types from annotations used with the field.
6. Produce results containing field name, change type(s), initial value, and new value.

To further detail each step, first the three sets of XML files that contain the vScope query objects and their corresponding metadata are deserialized. The three sets represent a common ancestor, the current version on the head node of the branch that we are comparing with and the proposed change set. This is due to a requirement to allow for a similar workflow to when creating and merging branches in Git, where diffs are produced by the set of changes made since a common ancestor of the two branches. The prototype assumes that these 3 sets are identified correctly, and that it should only calculate diffs. This step is necessary since we cannot use reflection directly on the XML files, thus we instantiate the Java objects that they represent.

After instantiating the vScope query class instances we generate the three dependency graphs, and evaluate the change sets from the common ancestor query set to the other sets. The dependency graphs are stored in two hashmaps, one which tracks dependencies and another which tracks dependents. The graphs are stored until the result is generated. This step is used to model the relationships between queries, such that the impact of changes can be mapped from the modified query to its dependents. Due to the hierarchical structure of queries, meaning there are no circular dependencies, we chose to return all transitively dependent queries as the set of queries is limited. This may not be necessary in the future, in which case there is the option to only return immediate dependents or dependents within a certain number of steps.

Next we process the diffs for every unique identifier found in any of the three folders. We omit the possibility that a file has been deleted at this stage, as a consequence of not diffing entire repositories. This is due to the design choice to only allow the change request of a single vScope query and its dependencies. A deleted dependency in a change request does not guarantee that the removed dependency is unused elsewhere by other vScope queries. Even if the removed dependency has no other dependents it may still be used, as every query except tags can be viewed in isolation. This means that the deletion of a file is not currently modeled, which we will evaluate in the evaluation section. New files can be introduced however, since

we then know that it did not exist before the change request. This will yield a diff where the new file is viewed as a new class instance where every non-null value is inserted. When we compare fields the operations are insert, delete and update. An insert is when a field has the value null in the base query, and not null after a change. Delete is the opposite, where a value that was not null is set to null. The update operation is used when a field is modified but is not null in either the base or changed query.

When comparing fields there were some significant decisions to make during implementation. One such decision is how to interact with non-primitive types or enums. In this case the vScope queries do not have non-primitive types that have to be compared beyond immediate equality with three exceptions. Sets, Lists and a specific query type called Widget which is sometimes present directly in dashboards instead of using references to identifiers. Sets and Lists are both collections of elements, where the elements are primitive types in our case. This means that when we diff these types we can simply check for the presence of elements. Ordering of elements can also be considered, yielding the three operations insert, delete, and optionally update. In the case of the Widget query type occurring as an immediate element in a Dashboard rather than as a reference we chose to check if elements are an instance of the Diffable interface.In this case we ignore the Widget instance contained in the Dashboard and treat it as a separate Diffable instance with its own ID and entry in the dependency graph. This is only possible since it has a unique identifier as can be seen in the Diffable interface. This behavior does introduce more complexity, but provides the flexibility in diffing that is required to meet the company's needs. The added complexity stems from having 6 possible scenarios when diffing, as listed below.

- The object should be ignored. This was used to experiment with some values that are not visible to external end-users not being relevant or reasonable to modify for such users.
- The object is null.
- The object is a primitive, and should be compared directly.
- The object is not a primitive, but should be compared directly. In this case, we need to ensure that the object implements a reasonable equals method, unless we want to check equality based on reference.
- The object is an instance of a collection, where the result is the inserted or deleted elements to the collection.
- The object is an instance of Diffable. In this case, we add the object to the end of the working set that is yet to be diffed.

These scenarios occur due to requirements on the implementation. The ignore case is important in a case where some field should not be accessed, either due to irrelevance to the diff operation or due to the component having to be private. The cases for null checking, comparison between primitives and class instances which should be compared directly with equality are regular comparisons that are the basic components that can be compared in a class. The diffing of collections is not directly made as we typically are not interested to know that the collection in its entirety is not identical, but would instead want to know the

elements that have been changed. An example of this is the columns in a table, where information for which columns have been added or removed provides more precise information than simply stating that the list of columns is modified in its entirety. In the context of how queries are structured currently, this approach to diffing objects and nested objects provide the flexibility to produce diffs at a level of granularity as outlined by the requirements on the tool.

The change types are fetched from the annotations used with the field. It should be noted that fields cannot be accessed directly by an interface or supertype in Java, so in this case we use an abstract class AbstractDiffable to define the methods that provide the implementation which can be inherited and used by all classes that extend it. The getFieldValues method can be seen in figure 5.1, it acts the same as getFieldAnnotations except that the annotations are saved as a set of types. The method getFieldAnnotations is configured to ignore annotations derived from other uses, such as annotations used when deserializing XML documents. The purpose of fetching annotations is that we made the design choice to classify change types by annotating a field with an appropriate impact type that modifying it implies. The purpose for using annotations instead of field names, is that changing a value of a field may have different impact depending on what query type it is, even though the fields have the same name.

```java
@Override
public Map<String, Object> getFieldValues() {
    Map<String, Object> fieldValues = new HashMap<>();
    Field[] fields = this.getClass().getDeclaredFields();

    for (Field field : fields) {
        if (field.getDeclaredAnnotation(Ignore.class) == null) {
            field.setAccessible(true);

            try {
                Object value = field.get(this);
                fieldValues.put(field.getName(), value);
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
        }
    }
    return fieldValues;
}
```

Figure 5.1: An example of an annotation used.

The fields that are found to differ are then used to produce results that contain the field name, the change type or types, the initial value and the new value. The results are stored by ID, so each ID has 2 lists of changes. At this point we created a JSON object of the results which stores all the information from the diff results. The JSON object is used to model the response which would be sent to the frontend of the application, as that is how the backend and frontend parts of the application communicate. This will be evaluated in the evaluation section in a discussion with the development team.

The outline of the Diffable interface can be seen in figure 5.2. It contains all the methods necessary for object instances to be diffable by the algorithm.

```java
public interface IDiffable {

    @XmlElement(name = "id")
    public String getId();

    default Map<String, Object> compareFields(IDiffable other) {

    Map<String, Object> getFieldValues();

    Map<String, String> getFieldChangeType();

}
```

Figure 5.2: An example of an annotation used.

5.2 Evaluation

This section will evaluate the diffing prototype based on the assessed requirements. We will discuss the performance in terms of speed, sufficient information retrieval concerning the change request and security. The takeaways from evaluating the tool with the development team will be addressed and some improvements will be discussed. We will also evaluate the prototype tool with regard to the requirements elicited in previous chapters.

The first aspect tested was the speed at which diffs could be calculated, since it was one of the significant concerns found during the requirements phase. We tested the algorithm's runtime when diffing the most high-level query type currently implemented, dashboards, which typically involve the largest number of files. Dashboards are expected to be one of the most common query types in change requests, and typically consist of 200 queries when including the dependencies. The largest set of files tested consisted of 402 files, which are query files and metadata files in pairs. This set of files was copied into two other sets, which were then modified to represent the initial common ancestor state, a head state and the incoming change request. This models a 3-point diff, which was established as a requirement. When calculating this set the runtime was between 4.2 seconds and 4.9 seconds, which was within acceptable limits for the employees in roles which would use the tool. When using 3 identical sets of files the runtime was not significantly affected, which suggests that the number of changes does not significantly affect the runtime. This result strengthens that the speed requirement, established during the literature study, was met.

In terms of speed, it is worth mentioning potential for improvements. Due to time constraints for the project no parallelization was considered. All computations are done sequentially where there are many instances where some computations could be performed in parallel. Two examples are the deserialization and dependency graph building steps, which may be done more efficiently by using multiple threads. There is also the option to split the diff calculation from the common ancestor to the incoming change, and from the common ancestor to the branch head into two separate tasks which could be performed concurrently. Beyond parallelization, when evaluating the efficiency of the diff program we found that most steps, such as reading files and building the dependency tree, are performed with linear complexity, and depend on the number of queries. The complexity is a bit more difficult to assess when considering the comparison between queries and their number of fields, which proves to not be a problem in our test cases. This is due to the low complexity of the objects used in queries, where the number of queries far exceeds the number of fields compared

per query. In an example where many nested objects would have to be compared, this may not be the case and may result in the number of nested fields being more significant. The total number of queries available in the query repository is currently just below 5000 and is constantly growing with new implementations. As a result of this we theorize that only diffing one query and its dependencies at a time is important from the perspectives of both performance as well as complexity. This is in part due to such comparisons involving fewer potential changes in one request, which provides more clarity for maintainers, but it also ensures that the number of queries diffed at any time is lower.

One of the decisions that were made previously was that the deletion of a query is not modeled. This was discussed during the requirements phase with the development team, as well as when evaluating the prototype. This decision when discussed with the developers was again thought to be important here due to prior experience when discussing the deletion of queries. Since the repository is not representative of how queries are used outside of the main repository, such as in the case of non-shared queries external users have made, we cannot know if a query has no dependents. This makes deletion unsafe unless there is some insight into whether the query has external use or not, which is unknown. The developers instead suggest a separate operation, unlisting, that can only be performed by employees at the company. Unlisting a query would not remove it, and if dependents exist then they would still be able to fetch data. Unlisting would however avoid new queries from using the unlisted query, which is thought to be sufficient.

Due to time constraints and the decision to focus on calculating diffs and classify changes, we eliminated the GUI from the prototype. As part of the evaluation process with the development team and UX designer, we discussed the possibility of implementing a GUI given the JSON output produced by the tool. It was established that the output included all necessary information about the change request, such as change classification and affected dependencies. Thus, the requirement associated with retrieval and presentation of data was met. Also, the output string leads to a higher level of abstraction of the diff, which was one of our research questions. The developers also thought that it was possible to represent the diff as outlined by mock designs with the help of this result. The workload required for implementation was found reasonable, meaning these requirements were fulfilled. It was established that the result included all necessary information to represent the information specified as necessary in the requirements phase. The developers thought that it was possible to represent the diff as outlined by mock designs with the help of this result. The workload required for implementation was found reasonable, meaning these requirements were thought possible to fulfill.

When addressing the topic of security with this implementation we considered two main topics. First the classification of certain fields that pose a potential threat if modified, and second the use of reflections. Regarding the classification of fields, the development team suggested that simply having one annotation that either denotes a field as cde3§e3 to modify entirely or not is not fully adequate. They suggested that different employees at the company may have different levels of authority for making and accepting change requests. The implementation of varying levels of authority can be modeled easily by assigning the requiresAdminReview annotation a property with some minimum authorization needed. For example an enumeration with values like “ADMINISTRATOR”, “MAINTAINER” or “USER” which can be assigned to users that interact with the diff tool.

Regarding the use of reflection, it has been targeted as a security risk in Java previously,

where improper use of reflections prevent proper encapsulation of classes. Since the fields accessed include those with private modifiers unless the ignore annotation is used, there is the potential to leak private fields for classes that should not be externally accessible if the ignore annotation is not used. Two fields that were potentially at risk were the data field and the matchProperties field, which store arbitrary code segments used to execute queries in the Gremlin language. We instead annotated these fields as illegal to modify, since the code itself is not harmful to leak to external users but may be harmful if allowed to be modified. The Diffable interface does introduce new factors to consider for future work on new query models, since new fields added to query classes have to be assessed from this security perspective. While there is the option to ignore fields by using the ignore annotation, there is a risk when writing new code that this aspect is missed which could lead to exposure of fields which should not be accessible. The proposed solution does not have to provide access to the fields themselves or modify the permissions for reflexive access to the fields, so there is no need to configure the program to allow such access.

One requirement from the evaluation phase remained unfulfilled, namely the creation of a user guide to help review and validate change requests. We discussed the value of teaching new maintainers how to properly use and interpret the tool, and that it would be needed in addition to a clear reviewing process, but due to time limitations we left this requirement for future work. The other technically related requirements, such as the query interface, support for 3-point diff, and disallowing certain changes, were successfully addressed.

Chapter 6
Discussion & Related Work

This chapter will reflect upon and discuss our results and how it relates to other literature. Additionally, we will assess the validity of our research method and results, and discuss the generalizability of our results. Furthermore, we will discuss possible future work that can leverage our findings, outlining some of the work which was considered out of scope for this project but still of interest. Our results will be compared to the results of other work, and we will discuss the implications of these comparisons and our contributions.

6.1 Reflection on Process

In this section we will reflect upon our process. We will discuss a number of things that went as expected, and also the things that did not. We will reflect upon and compare where we ended up with what we wrote in the goal document in the very beginning of the process.

From the beginning we did a good job in documenting findings, articles, thoughts and ideas. Since we spent the first 4 months focusing on data collection, research and prototyping and only started the thesis writing the last 4 weeks it was helpful with carefully documented and organized data. Failing to document our findings would have most certainly resulted in double maintenance like duplication of research efforts or conducting repeated tests unnecessarily. Careful documentation of findings, even though it might not seem valuable at the time of documentation, is definitely a practice we would maintain if we were to conduct the research again.

The focus of the thesis changed a lot over time. In the beginning we simply did not know where we were to end up and there was a need to investigate several paths both briefly and thoroughly before we could make decisions on where to continue. As an example, for research question 3 we had ideas to conduct a security analysis or a penetration test on the marketplace platform, where we would test the current solution for weaknesses. However, after several discussions with mainly the product owner we were asked to focus on the localization of existing weak points in the current work flow around queries, since this was considered more

valuable for the company. Another example is that we spent a fair amount of time researching already existing XML diffing tools, only to find out that they did not meet our established requirements. We wanted to investigate the possibility that already existing tools could do what we needed, and if so, we could have spent more time on integrating the tool into the already existing platform. Even though we ended up not using any of the tools, it led to better insights in how structured data could be diffed which were findings that we could apply when implementing our own tool.

As for the first and second research question, we realized rather early in the process that we could not fully realize a graphical interface and a diff algorithm within the time constraints for the project. Ideally, we would validate mock designs with the sales and customer success employees and we had also planned to conduct workshops with internal end users. These steps were skipped due to time constraints and shift of focus towards programming a solution. We might have been too optimistic in the beginning of the process when defining the scope of the thesis, where we thought that we had time to focus on both user experience related aspects and configuration management related aspects. A reason for this, and what we have learned, is that when talking to different stakeholders they will want different things from a solution and it is difficult to give them all what they ask for. There was constantly the need to find the balance between what we wanted to do, or were asked to do, and what we realistically had time for.

6.2 Threats to Validity

This section will discuss the validity of our study by addressing some threats to validity and limitations of our results.

During the data collection we interviewed only a few internal end users. The number of employees interviewed was small, at a number of 4, but since there are not any more employees in these roles it was not possible to perform more interviews. A small sample size could lead to data bias if the interviewees are not representative of the actual end-users, but since we interviewed all internal end users at the company, they must be considered to be representative. On the other hand, there are the external end users, meaning users of vScope. External end users are typically working in IT and could have a similar domain knowledge as the internal end users, but we have very little knowledge of their needs for a comprehensive reviewing process. Also concerning the data collection we did not collect feedback from the interviewees on the mock designs. The purpose of the designs were to communicate the needs of the interviewees, which we did not clearly establish. Iterative feedback from proposed designs is an important step [11] in data gathering since it shows whether we understood the interviewees' needs, and there is the possibility to tweak or change the designs. Skipping this step is clearly a threat to validity but since the focus of the thesis changed over time, we decided not to spend more time on the visual design.

Also, when conducting interviews there is always the risk of subjective bias. When doing the initial case study and researching the existing product, we started to form ideas on possible solutions for the tool and how the change request could be visualized. We gained technical insights that the interviewees did not possess, which affected the direction of the questions. We noticed during our first interview that the interviewee did not understand our questions, and that we needed to ask very specific ones and provide explanations for why we

asked certain questions. It is possible that we influenced the interviewees in the way that we asked our questions, even though we tried to keep the questions as objective as possible.

As for the diffing tool, from the rather limited tests that we carried out, the results showed that we managed to classify changes and visualize them as an informative text string. However, we only ran tests on one type of query files, Dashboards, but since Dashboards contain all other query files it means that the diffing algorithm works for all query files currently existing. Performance might vary with scalability and since we only ran the tests on a folder with 200 queries (which is the expected max size currently) we cannot say anything about the performance if we were to run the tool on a folder with more query files. In the future, there is the possibility that there will be significantly larger query sizes, which could affect performance. If new query types are introduced there is some programming that needs to be done for the tool to function on these new query types too: the new query files need to comply with the structure of already existing query files, implement the diffable interface and get annotations for classifications of changes. From the tests that we did the result was as expected, but with more time we would have done more testing with larger sample sizes, with some deviating edge cases, which could possibly have affected the performance of the tool.

As for RQ3, we tried to assess measures to improve secure handling of artifacts, aiming at mitigating the risk that end users accept incorrect updates, and improve the chances that they accept correct ones. This is very subjective and different users might need different information communicated to understand the tool. As earlier stated, since we conducted very little user testing, we do not know to what extent our solution is helpful. The implemented method of classification can be used to provide context for how risky a change request may be, but users of the tool would have to be able to understand its implications. As a result, there may be a need for training employees to use the tool.

6.3 Generalizability

This subsection will discuss the generalizability of our results when not considering the specific case of vScope queries.

The solution proposed in this study is relevant when there is a need to classify and provide context for changes to object instances that follow some model, in this case defined as a Java class. The use of reflections and annotations is possible in other programming languages, so the diff algorithm and interfaces are possible to implement in other languages. The idea of using reflexive properties to compare models and metadata to classify the implications of change to a field is reusable, both together and separately.

The problem with delegating versioning of some artifacts to users who are not developers or familiar with version control tools is not unique. While the specific solution is tied to the implementation of vScope queries, the use of reflections and field comparisons has other potential use cases for diffing objects in other contexts. This is exemplified in some of the related work, where similar implementations are discussed but for other models with and contexts. The structure of the artifacts are different depending on the context in which they are applied, where our context involves models for objects that have very clear class definitions and properties. Our findings are generalizable to the extent that artifacts which have clear properties and will receive change requests that need to be verified can be diffed

at a higher level of abstraction with more context provided than pure text-based diffs. The classifications of the impact of a field being modified are tailored to this context, and would likely differ in another context particularly if the artifacts handled are more complex in their structure. The idea of tying classification to some metadata that provides context for its importance, or impact if modified, is reusable in other contexts. Some of these contexts are discussed in some of the related works, though the details of implementation vary.

The use of a dependency graph in the case studied here is generalizable particularly where dependencies are hierarchical in structure. For graphs that involve circular dependencies it may be more difficult to classify the impact of a change, and to capture which dependents are affected. In a highly connected graph with many circular dependencies, this may lead to every change affecting the entirety of the graph which is not useful information. In cases where it cannot be assumed that all transitive dependencies need to be validated, there is the option to only validate dependencies a certain number of steps away from the changed artifact.

6.4 Related Work

In the related work section we will compare and discuss consistencies and discrepancies with other research. We will analyze and discuss the implications of such occurrences. The related works were found by searching for academic articles with the help of keywords relevant for this report. Four articles will be discussed in this section, from the perspective of our project.

6.4.1 Related Work : Enabling Fuzzy Object Comparison in Modern Programming Platforms through Reflection [3]

The paper “Enabling Fuzzy Object Comparison in Modern Programming Platforms through Reflection” by [3] discusses the topic of comparing "fuzzy" objects, which refers to the comparison of objects that may be similar rather than identical. An example of a fuzzy comparison is cases where people may be equivalent in some contexts if their age is in the same range, and they work in the same sector. The authors discuss the use of reflection as a means for accessing fields and comparing for equality given the context where the object is used, rather than deferring the comparison to the object’s implementation itself. The authors’ proposed solution is similar to the one implemented in the context of this project, with the key differences being in the purpose of comparing objects. They use java as a programming language, which is also used in our project.

The authors introduce reflection as a means to dynamically compare objects based on their properties, or fields as we have referred to them. They cover how reflection can be leveraged to implement custom comparisons and provide some examples and possible ways of implementation. Reflection allows runtime introspective access to the code structure, which is then used to provide context for object comparison. Without considering specifics in implementation, this is similar to how we access fields for the purpose of comparison.

Furthermore, the paper discusses practical considerations such as performance overhead associated with using reflection on objects, where the authors consider the benefits of flexibil-

ity of such comparisons outweigh such concerns in many cases. The authors provide context for how to solve issues with cyclic structures as well, which is of concern if comparison is to be done recursively at further depth than the immediate properties of an object or has circular chains of object references.

It is notable that the report is from 2003 and that the Java language has shifted towards more restrictive access when using reflections due to security issues with object encapsulation. This aspect is more thoroughly explored in our project, and our findings indicate that the use of reflections for the purposes of field comparison are still viable. F. Berzal et. al do highlight the importance of proper class modeling such as to avoid circular dependencies from a performance perspective, and we find that it is equally important to consider which fields are accessible during comparisons from a security perspective.

The authors' code examples include a discussion for how object comparisons could weigh the importance of different fields, which is done manually with the use of arrays. In our case, we instead used Annotations to label the impact or importance of various fields. Annotations were introduced in java 1.5, which was released in september 2004, so the authors did not have access to this language feature. The difference in using the authors' way to label or provide context for fields and in using annotations is where the configuration is made. In their case the assignment is done by the class that is doing the comparison, whereas in our case it is deferred to the class implementation of the object which is to be compared.

6.4.2 Related Work: Change Impact Analysis Based on a Taxonomy of Change Types [12]

The focus of the paper is on static change impact analysis, CIA, for object-oriented programs [12]. The motivation behind the research is to understand potential effects that a change in software might have on other parts of the program, and specifically how to improve the precision of the change impact.

In the article, the authors propose a static CIA technique that depends on the change type of the modified entity, and the dependence types between the modified entity and other entities of the program. Change type classification includes classes, methods, and fields and dependence types can be for example interfaces. The change impact analysis is computed by traversing a dependence graph based on the change types of the program's parts.

The effectiveness of the technique is evaluated through a case study where a number of versions of Java source code is analyzed. By using a number of metrics such as precision, recall and F-measure, the change impact is measured.

The authors find that their technique is applicable to the small sized program that was tested in the case study for which it can improve the precision to some extent. Anyhow, the authors can not guarantee it being generally applicable to large-scale programs.

One parallel that can be drawn between the research of the paper and our project is that the change impact analysis is carried out through building and traversing a dependency graph. Their approach is similar to ours, but we omit the change types from the graph. The different change types used in the analysis could be applicable to our diff tool, where we could extend our classifications to contain more types (currently only descriptive, functional and requiresAdminReview). Similar to how the authors do, we could classify the changing of a column containing a tag that is dependent on another tag, to something like ATDoT (Add

Tag Dependent on Tag).

6.4.3 Related Work: Component Configuration Management[9]

In the paper Component Configuration Management M. Larsson and I. Crnkovic[9] explore the topic of dependency management and component configuration management. The goal was to outline possible solutions for issues with version management between components, where the component versions used could cause the system to break. The introduction of a dependency graph is similar to ours, though there are some differences to discuss.

The paper is written in the context of codebases of which subcomponents can be mixed and matched to construct or compose a system or application. It was motivated by a rise in popularity for Component or Plugin-based programming at the time, causing developers to run into issues with changes in dependencies breaking builds. This motivated the authors to further explore the topic of Component Management and the management of Component Dependencies. The authors describe a graph structure of dependencies which is similar to the dependency structure of vScope queries, and explore similar problems to this project.

The authors use personal experience and cited sources to formulate problems and potential solutions for component based software development. Components in the case of this paper refer to binaries, which can be other system components or library dependencies. They propose a dependency model displayed in a sort of dependency browser as part of a solution, which is used to track dependencies to other components in the form of a graph. The graph can be saved as a snapshot when components are modified in order to trace dependencies for a given composition of components. The proposed dependency browser is then used to highlight dependents of a modified component, which can be used for testing, validation and traceability.

The context is fairly different when compared to ours, but the use of a dependency graph and the proposed mock solution for how it could be modeled for maintainers is similar in some ways. They define different levels of compatibility, which are used to categorize the impact of changes in the context of system components which interact with each other. The goal with these classifications is to identify what type of validation is necessary after a change. This is the same goal as the purpose of annotations discussed in our proposed design and implementation though in a different context, suggesting that there are parallels between both problem statements. The main difference lies in the simpler structure of vScope queries when compared to general purpose languages, where we can model change types in a more simple way. One example is their proposed Behavior Compatibility, which requires that characteristics such as performance and resource requirements must be preserved. This could be relevant in the future if vScope queries become more complex or if the maintainers of the repository need further insights regarding performance than what has been discussed in this project.

6.4.4 Related Work: Multi-party authorization and conflict mediation for decentralized configuration management processes [8]

As the title states, the focus of the paper is on a multi party authorization (MPA) and conflict mediation for a more secure configuration management process [8]. The paper addresses the necessity of protecting systems against improper configurations, whether they are the result of a human error or a malicious act.

The proposed solution is a decentralized configuration management process that uses MPA, controlled by a configuration management system (CMS). This combined approach reduces the risk of a single point of failure, i.e. one compromised cryptographic key can result in malicious configurations if one admin on its own can create and deploy configurations. Instead, their decentralized model splits the process into several steps that are performed by multiple parties. However, the result of an MPA process can result in conflicts that must be resolved

Their strategy is showcased by proposing 5 conflict mediation strategies that can be used one by one or in combination. The strategies are iterated until a unary decision is reached, and the result is stored in the CMS TANCS (tamper-resistance and auditable network configuration management system), implemented by the authors. TANCS works on top of a distributed ledger, meaning it uses a peer to peer blockchain technology for storing configurations. As long as the majority of nodes are honest, the system ensures tamper resistance and integrity. The goal with this process is to reject malicious configurations, and accept good configurations when under attack.

The authors argue the benefit with their strategy over majority voting through a couple of examples. Although costly, the authors state that it is beneficial and less expensive than recovering from data leakage or compromised credentials. The downside with the strategy is that it does not provide quick responses.

The paper proposes using a distributed ledger instead of a single public key for verifying and storing configurations. Although the focus of our project has not been on actual IT security connected to vScope queries, but rather on investigating the workflow around them to reduce mistakes in the auditing process, this could probably be applicable to the handling of vScope queries that are shared between users. Using a distributed ledger for storing credentials would mitigate the risk of a single point of failure, as a compromised key could not be used to tamper with the integrity of the queries. Instead there would be the need for a consensus among all involved nodes for changing something like a credential. Also, requiring multiple parties to accept or reject a change request is something that we have looked into, for certain change requests that need extra cautiousness.

6.5 Future Work

This part of the report will discuss some further avenues for exploration on the subject of Configuration Management. These ideas could have been part of this study but were omitted from the scope of the project, or could build upon our findings.

The first, and most obvious, part that should be explored further is the graphical repre-

sentation of diffs in a case similar to what has been explored in this project. The representation of diffs in a more abstract format is interesting to explore due to the domain knowledge required to understand changes sometimes not matching the knowledge required to understand version control tools. In the case of vScope, the intended new maintainers for the query repository are the employees that create the most queries on a day to day basis during contact with new or existing customers of the tool. Despite their understanding of how to create and use queries, they are not able to maintain the repository on their own due to not understanding the java object when represented in XML format on the back side of the tool.

It is also worth mentioning that the format of viewing changes in a context where the object diffed is being used is an interesting topic. The interviews and mock designs suggested processing diffs in the vScope tool, which when building new queries could be highly coupled with the data seen or removed when applying changes. It would be interesting to explore how readable a change in this context is if the person reviewing a change request has to understand the context of for example a filter added that does not affect the data set available to them. Here the challenge would be to understand a context that is not applicable to the reviewer's own context. One idea is that this could potentially be mitigated with comments coupled with the change request that provide an explanation for the change request, but we would not know if that is sufficient.

This project involved strictly hierarchical dependencies, which leads to a relatively simple graph structure to traverse when analyzing the impact of a change. It would be interesting to further explore how a similar problem could be solved if that was not the case, and how an eventual solution could be implemented.

Chapter 7
Conclusion

This thesis has found that it is possible to provide a higher level of abstraction and contextualization when diffing vScope Queries which are in essence Java objects. It was also found that while XML diffing tools provide a lot of the necessary functionality for the use cases explored in this report, they proved insufficient when reviewing the requirements for this project. Specifically, there were issues with performance while it was also necessary to use different configurations for different files to generate the desired diff.

Interviews with internal end-users found a need or preference to see changes in the context of vScope as a tool rather than in text files due to them working primarily as salespeople and customer success managers with mostly non-technical backgrounds, suggesting that non-technical users do not want to view changes to the queries themselves but rather their outcome. This finding does come with a few challenges, such as when modifications to a query don't make sense without data that the reviewer may not have on their own installation.

Since XML-based diffs proved nonviable in this case we instead propose a small but purpose-built program. The program uses reflections of the classes in which vScope Queries are defined to gather their included fields and values in order to diff different versions or instances. The classes may also include annotations to provide context for the implications of a change to a specific field, which provides context regarding the impact of a proposed change. The types of changes are defined as Functional, Descriptive and RequiresAdminReview, and the operations include add, delete and update depending on the data structure. Functional changes are the change type which triggers the need to validate dependent queries, as such changes affect the data fetched or the operations performed on data. The diffing tool thus prompts the user viewing a diff to also consider the dependents further up in the query hierarchy and validate their correctness given the new context. No general way was found to differentiate functional changes that do or do not affect dependents without manual validation.

On the topic of security and the exposure of modifying vScope queries, we discovered that annotating parts of the queries that pose particular risks—primarily Gremlin code—could be beneficial. Highlighting or disallowing changes to such fields should prove sufficient in

preventing malicious code from executing, given proper training and routines surrounding the validation of change requests.

In summary, our findings suggest that usage of reflexive properties along with annotations available in some programming languages can be a good solution to produce more readable diffs and classify them depending on the changes made.

References

[1] Hadi Ali and Micah Lande. Understanding the Roles of Low-fidelity Prototypes in Engineering Design Activity. *ASEE Annual Conference proceedings*, 2019.

[2] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. The Maven Dependency Graph: A Temporal Graph-Based Representation of Maven Central. pages 344–348, 05 2019.

[3] Fernando Berzal, Juan-Carlos Cubero, Nicolás Marín, and Olga Pons. Enabling Fuzzy Object Comparison in Modern Programming Platforms through Reflection. In *Fuzzy Sets and Systems*, pages 419–513, 06 2003.

[4] Milos Cuculovic, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. Semantics to the rescue of document-based XML diff: A JATS case study. *Software: Practice and Experience*, 2022.

[5] Peter Feiler. Configuration Management Models in Commercial Environments, Technical Report SEI-91-TR-7. 01 1991.

[6] Git. `https://git-scm.com/docs/git-diff`,. Accessed on March 22, 2024.

[7] Chariton Karamitas and Athanasios Kehagias. Improving binary diffing speed and accuracy using community detection and locality-sensitive hashing: an empirical study. *Journal of Computer Virology and Hacking Techniques*, 19:1–19, 10 2022.

[8] Holger Kinkelin, Heiko Niedermayer, Marc Müller, and Georg Carle. Multi-party authorization and conflict mediation for decentralized configuration management processes. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 5–8, 2019.

[9] Magnus Larsson and Ivica Crnkovic. Component Configuration Management. 06 2000.

[10] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2008.

[11] Pamela Savage. User interface evaluation in an iterative design process: a comparison of three techniques. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, page 307–308, New York, NY, USA, 1996. Association for Computing Machinery.

[12] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. Change Impact Analysis Based on a Taxonomy of Change Types. In *2010 IEEE 34th Annual Computer Software and Applications Conference*, pages 373–382, 2010.

[13] Sathya Sundaram and Sanjay K. Madria. A change detection system for unordered XML data using a relational model. *Data Knowledge Engineering*, 72:257–284, 2012.

Appendices

Appendix A
Questions for semi-structured interviews

Note that Content is the name for vScope queries internally at the company.

- Kan du berätta lite om din roll/position och dina arbetsuppgifter på företaget?

  Could you tell us about your role and your responsibilities at the company?

- Utifrån din roll på företaget, vad ser du för fördelar med ett Marketplace?

  From the perspective of your role at the company, which benefits do you see with a Marketplace?

- Kan du beskriva processen från att skapa nytt content till att få ut det till kund idag?

  Could you describe the process for creating and releasing new content?

- Hur upplever du processen?

  What do you think about the process?

- Vilka steg skulle du vilja se innan nytt content publiceras?

  Which steps would you like to see before new content is published?

- Hur skiljer sig dessa för olika typer av content?

  How would that differ for various types of content?

- Vad tror du att du skulle behöva för att skapa de olika typerna av taggar?

  What do you think you would need to create the different types of tags?

- Övriga tankar/önskemål?

  Do you have any other thoughts?

Appendix B
Requirements specification

Interviews

- Conform to current design philosophy of "History and Compare"
- See which vScope queries that are dependent on the vScope query that has been modified as part of a change request
- Clear and comprehensive reviewing process
- A higher level of abstraction than text-based diffs.
- Review inside the product

  Which leads to:
- Displaying the modified query, some classification and the affected dependencies

Lo-fi Mocks

- Retrieve and present all necessary data about change request (actually the same as in Interviews)

Literature Study

- Some elements should be considered changed if their ordering is changed
- Some form of automated security risk assessment should be performed
- Some form of automated impact analysis should be performed
- Performance requirements in terms of speed to calculate diffs should be considered.

Evaluation

- Interface for vScope query modifications .
- Some changes should not be allowed to facilitate maintenance (currently no one is allowed to do these changes but in the future it could be depending on what role the user have e.g. developer, maintainer, end-user)
- Some form of information regarding why certain steps in validation are necessary should be available for new maintainers of the vScope query repository.
- Support for 3-point diff is necessary

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-04-04

EXAMENSARBETE

STUDENTER Sean Jentz, Karolina Haara Löfstedt
HANDLEDARE Lars Bendix (LTH)
EXAMINATOR Per Andersson (LTH)

Versionshantering för nybörjare: tydligare, enklare, snabbare

POPULÄRVETENSKAPLIG SAMMANFATTNING **Sean Jentz, Karolina Haara Löfstedt**

Ett skräddarsytt versionshanteringsverktyg som är enkelt att förstå - även om du inte kan Git. Genom att upptäcka och klassificera förändringar i olika versioner av XML-filer blir resultatet ungefär samma som med `git diff`, fast anpassat efter användare med begränsade programmeringskunskaper.

InfraSight Labs produkt vScope samlar in information om kundens IT-miljö och lagrar denna i en grafdatabas. För att presentera datan för användaren används XML-formatterade vScope queries. Som en del i produktutvecklingen behöver dessa queries ändras och uppdateras kontinuerligt.

I nuläget är det tidskrävande att granska ändringsförlag på queries. Detta beror på att de är inbördes beroende, vilket betyder att en ändrad query lär påverka andra queries och alla beroenden behöver kontrolleras innan en ändring godkänns. Dessutom görs granskningen genom versionshanteringsverktyget Git vilket försvårar eller rent av omöjliggör uppgiften för anställda utan programmeringskunskaper.

För att förenkla processen har vi byggt ett verktyg som kan upptäcka och klassificera ändringar i query-filer, genom att använda Java annotationer och reflektioner. Verktyget identifierar vad för typ av ändring som gjorts, vilken fil som ändrats, och vilka andra filer som påverkats av ändringen. Sedan kommuniceras detta på ett sätt som är anpassat efter de tilltänkta slutanvändarna på InfraSight Labs. Jämförelsen görs dessutom snabbare än de redan befintliga XML-jämförelseverktyg som vi testat.

```
@@ -4,13 +4,13 @@
   <data class="string">Here goes some Gremlin code</data>
   <resultName>This is the name that is shown in column head
   <attributes id="2">
-    <entry>
-      <string>unittype</string>
-    </entry>
     <entry>
       <string>convUnitStep</string>
       <int>1</int>
     </entry>
+    <entry>
+       <string>AddedEntry</string>
+    </entry>
   </attributes>
   <matchProperties id="4">
     <entry>
```

```
Added Entry: {<string>AddedEntry</string>}

Removed Entry: {<string>unittype</string>}
```

Figur 1: Förenklat exempel på hur en s.k. diff ser ut i nuläget med Git (svart bakgrund), respektive hur det ser ut i vårt verktyg (vit bakgrund, blå text).