UNIVERSITY OF DUISBURG-ESSEN
FACULTY OF ENGINEERING SCIENCES
DEPARTMENT OF COMPUTER SCIENCE AND APPLIED COGNITIVE SCIENCE

Master Thesis

Investigation of Collaboration and Communication Patterns in Software Projects using Social Network Analysis

Christoph Zils
Matriculation Number: 3034735

UNIVERSITÄT
DUISBURG
ESSEN

Computer Science and Applied Cognitive Science
Faculty of Engineering Sciences
Universität Duisburg-Essen

June 7, 2021

Supervisors:
Dr. Tobias Hecking
Prof. Dr. H. U. Hoppe
Prof. Dr. S. Stieglitz

Declaration of Authorship

I hereby declare that this thesis was exclusively made by myself and that I have used no other sources and aids other than those cited.

C. Zils

Duisburg, 4th of June 2021

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Structure . . . 2

2 Foundations **5**
- 2.1 Open Source Software Development . . . 5
- 2.2 Git . . . 6
- 2.3 Community Smell Patterns . . . 8
  - 2.3.1 Social Debt . . . 8
  - 2.3.2 Formalized Patterns . . . 9
- 2.4 Provenance . . . 11
  - 2.4.1 Definition . . . 12
  - 2.4.2 Provenance Data Model . . . 12
  - 2.4.3 Git & GitLab Provenance Models . . . 15
- 2.5 Graph Databases . . . 21
  - 2.5.1 Neo4j . . . 22
  - 2.5.2 Cypher Queries . . . 22
  - 2.5.3 prov2neo & neo4r . . . 23
- 2.6 Network Analysis . . . 24
  - 2.6.1 Provenance Graph Analytics . . . 24
  - 2.6.2 Social Network Analytics . . . 25

3 Concept and Implementation **29**
- 3.1 Concept . . . 29
  - 3.1.1 Extracting & Importing Data . . . 29
  - 3.1.2 Transformation Operations . . . 31
  - 3.1.3 Using Queries to find Community Smells . . . 33
  - 3.1.4 Comparison with Randomized Graphs . . . 35
- 3.2 Implementation . . . 35
  - 3.2.1 Data Cleaning . . . 35
  - 3.2.2 Folding and Transformation . . . 36
  - 3.2.3 Extensions . . . 41
  - 3.2.4 Network Randomization . . . 43
  - 3.2.5 Developer Analysis . . . 43
  - 3.2.6 Network Analysis . . . 44

4 Evaluation **45**
- 4.1 Graph Properties . . . 45
- 4.2 Community Smells . . . 45
- 4.3 Developer Characterization . . . 48
- 4.4 Randomized Networks . . . 49
- 4.5 Network Analysis . . . 50

5 Discussion **51**
5.1 Community Smell Patterns 51
5.2 Developers 53
5.3 Performance 53

6 Conclusion **57**

Bibliography **59**

List of Figures

2.1 Representation of the *Organizational Silo* community smell pattern [Tamburri et al., 2017] 9
2.2 Representation of the *Lone Wolf* community smell pattern [Tamburri et al., 2017] 10
2.3 Representation of the *Black-cloud* community smell pattern [Tamburri et al., 2017] 11
2.4 Representation of the *Bottleneck* community smell pattern [Tamburri et al., 2017] 12
2.5 Provenance core structures and their relations [Moreau and Missier, 2013] 13
2.6 Representation of provenance graph elements with their relations and attributes. 14
2.7 Model of adding a new file to a project in Git. 16
2.8 Model of modifying an existing file in a project in Git. 17
2.9 Model of deleting a file from a project in Git. 18
2.10 Model of creating or annotating a commit in GitLab. 19
2.11 Model of creating or annotating an issue in GitLab. 20
2.12 Model of creating or annotating a merge request in GitLab. 21
2.13 Exemplary representation of a relationship between the two neighbors Alice and Bob. 22

3.1 Pipeline from extracting data from GitLab to transforming and importing communication and collaboration relations into provenance graphs in Neo4j. 31
3.2 Desired connections between users to be created via transformations. . . . 32
3.3 Left: Simple representation of collaboration of two users by file.
Right: Extended representation with activity including time component. . 32
3.4 Path to be queried to create the communication network for all GitLab web resources. 33
3.5 Matrix multiplications to set up the collaboration matrix. User (U), commits (C), file versions (FV) and files (F) adjacency matrices are needed. . 33
3.6 Operationalization of the Community Smell Patterns for the *Lone Wolf* (left) and *Operational Silo* (right). 34
3.7 Structure of the communication graph to identify the *Black-cloud effect*. 34
3.8 Scheme of the extracted provenance graph within *Neo4j*. Nodes are grouped according to their affiliation to Git commit or GitLab web resource model. 36
3.9 Required agents, activities and entities to establish collaboration relations between authors and committers. Established connections between users are represented by dotted lines. 37
3.10 Matrix multiplications to calculate the desired user file matrix 37
3.11 Flawed approach to the creation of the communication network. Connections between annotators are missing. 38
3.12 Part-wise calculation of the communication network with subsequent union. Established connections between users are represented by dotted lines. . . 39

3.13 Cutout of the network regarding users (blue) and commits (pink) before and after importing relations. Communication shown in red, collaboration in green. 39

4.1 Identified community smell patterns for IGitt 46
4.2 Identified community smell patterns for ASE 47
4.3 Correlation diagrams of the full IGitt project for *Lone Wolf* and *Organizational Silo* patterns. 48
4.4 IGitt non-smell users 49
4.5 IGitt smell users 49

List of Tables

2.1 Overview of Git Commit and GitLab Resource Models used in this work. 15

3.1 The researched time windows of considered open source projects. 42

4.1 Overview of the selected projects to be analyzed in this work. Data is portrayed at the time of extraction. 45
4.2 Comparison of community smells from authentic and randomized full IGitt networks. 49
4.3 Comparison of community smells from authentic and randomized first time window of ASE networks. 50
4.4 Computed centralities for communication networks of researched projects and conditions. 50

1 Introduction

In this first chapter, the basic motivation of the master thesis is explained and the resulting research questions are presented. In addition, the context in which this thesis was written is described as well as the possible applications for the solution developed. Finally, the structure of the thesis is described for each chapter.

1.1 Motivation

Growing complexity and size of software projects greatly challenge developers and project managers. Due to this trend, the approach to development is changing. Project management changes from static development models to agile development methods, which aims to reduce the formal effort of project management for both smaller and larger software projects [Jorgensen, 2019]. Software projects of each size nowadays contain distributed teams from different countries and continents that work together, making project management more complex for project managers and developers. To deal with these challenges, teams often use version control systems that allow artifacts to be recorded during development and reverted to previous states if necessary and additionally contain tools for project management such as forums and Kanban boards. This does not only apply to professionally organized teams, but also to communities that promote publicly accessible code, better known as *open source*.

Open source software (OSS) development reconciles both agile methods and distributed work. In OSS projects, interested parties can view any source code and the current development status of a software. They are free to participate in the community by giving feedback on problems or joining the development themselves. This results in flat hierarchies in OSS projects and minimized effort in project management [Crowston and Howison, 2006].

Because of this organizational structure, problematic behavior by users who collaborate on the project but do not communicate effectively with others can occur. Tamburri et al. describe such negatively afflicted patterns as *community smell patterns* [Tamburri et al., 2017]. These patterns, according to them, can cause serious consequences for the entire project in the form of *social debt*. Consequences of this debt can be increased effort due to source code revisions or high turnover of developers in the project due to dissatisfaction. They have explored several patterns that can become critical to an OSS project [Tamburri et al., 2015].

Finding these patterns can become too computationally intensive as the size of the software projects increases. Provenance graphs offer one way of abstraction thereby.

Provenance graphs map the complete history of all files and activities manipulated by users and can describe the history of software projects [Moreau and Missier, 2013]. Details about the changes to the source code are discarded, resulting in an abstracted and stripped-down view of the project which can significantly reduce required computations to find sub-graph patterns such as community smells.

Therefore, this master thesis aims at analyzing the development of software projects with respect to community smell patterns using provenance graphs. For this purpose, a workflow will be set up combining several tools for data extraction, transformation and analysis. As an example, several projects will be extracted from the version control system GitLab via the tool *GitLab2PROV* and analyzed with respect to four community smell patterns. The following research question is to be answered through this approach:

RQ1: How can community smell patterns in communication and collaboration of open source developers be identified using provenance graphs from a version control system?

Since no research about a combination of provenance graphs and the analysis of community smell patterns was found after an extensive search, this approach offers an innovation in the context of the investigation of human aspects in software engineering. It shall be extended by an analysis of users and networks created from provenance graphs, so that a prediction of possible community smells can be made based on a network analysis.

RQ2: Which community smell patterns identified in provenance graphs can be predicted by user or network characteristics?

This work is developed in cooperation with the German Aerospace Center (DLR). DLR is the German research center in the fields of aerospace, energy, transport and digitization. Organized in different institutes and their respective departments, this work is to be located in the area of Intelligent and Distributed Systems of the Institute of Software Technology. DLR uses the GitLab platform to manage in-house software projects and in 2020 developed a tool to extract provenance graphs from GitLab [Schreiber and de Boer, 2020]. The examined data therefore originates from this platform and the implemented approach can be used to uncover software projects of DLR with regard to negatively afflicted patterns in development.

1.2 Structure

Following this introduction, this work is structured into five consecutive chapters.

Chapter 2 In the second chapter of this master thesis, theoretical foundations regarding open source development are introduced. Based on this, certain characteristics of open source developer networks are addressed and explored in greater depth. Lastly, used tools and methods to analyze those networks are presented.

Chapter 3 The concept of transforming the provenance graphs into collaboration and communication networks and the subsequent network analysis to identify negative patterns is presented. The resulting implementation of the concept is described and measures for the solution of occurring problems are outlined. In addition, the stakeholders in identified patterns are examined and a randomization of the networks to verify the results is performed.

Chapter 4 Results found through the implementation are listed. The focus here is on network structures and specifics within the organization of the projects studied. Furthermore, the actors in patterns found are described with regard to their activities.

Chapter 5 In the fifth chapter, previously mentioned results are discussed and compared with other studies. Unexpected results are addressed, and possible backgrounds are explored. Additionally, the performance of used algorithms is evaluated, and possible improvements are suggested.

Chapter 6 At last, the thesis is considered in the perspective of the objectives achieved and remaining open questions. Limitations are described and relevant questions for future work are summarized.

2 Foundations

In this section, all the fundamentals important for understanding are to be sufficiently explained and clearly presented. It reviews the definition and classification of Open Software and its software development process. Open Source Software Development is considered the initiator and main user of version management platforms, of which Git is the most popular at the time of this work. Git as a platform is therefore briefly introduced with respect to the value for OSS projects and particularly with regard to the inherent structure. Within the structure of Git, negatively afflicted patterns can be found in the communication and collaboration of users, so-called *community smell patterns*. The emergence and consequences of these patterns are highlighted and currently known patterns are presented in a formalized way. To find these patterns, underlying graphs must be considered and deeply analyzed. As a basis for this work, provenance graphs are used, which abstractly represent the development process in software projects. The tools used for the extraction of these provenance graphs for the platform Git and especially its adaptation GitLab are described. Subsequently, tools are listed that are used to find community smell patterns within the networks and to characterize the networks themselves. This includes provenance data and social network analysis methods, which are used to analyze the topology of graphs and make graphs comparable in the big picture.

2.1 Open Source Software Development

Open source software (OSS) is a widely used term that appears even before the year 2000 [Raymond, 1999]. It originates from the early hacker scene around 1970 and the free software movement of the 1980s [Bretthauer, 2001], whose goal was to establish freely available software as the standard and to be allowed to copy and modify existing software. Since that time, the value and approach has changed significantly. Today, the term stands for an approach in which software projects are brought to life from the ground up with an open, collaborative mindset. In doing so, interested parties are encouraged to participate, to share openly and to develop in a community-oriented manner [Fitzgerald, 2006]. In contrast to monolithic development by publishers and the resulting complete dependence on users, users themselves can share improvements and enhancements and put them up for discussion. This type of development encourages a growing number of participants to peer review code from other developers, thereby increasing the speed of releases of software versions [Feller and Fitzgerald, 2001]. Due to their size, speed and sometimes volatile developer community, it is challenging for project leaders and participants themselves to steer the open source communities and their software projects. This issue appears stronger as collaboration in OSS lacks organizational forms compared to traditional collaboration in software development

[Hars and Ou, 2002]. In order to better map and manage this process, open source software development projects have been and are being analyzed by the scientific community [Schreiber and Zylka, 2020]).

Researching OSS structures Analyzing these communities has shown that participants might never have cooperated or communicated with one another, which could be problematic since social relationships between collaborators propel the entire development process [Yang et al., 2013]. In particular, developer-developer connections from previous projects contribute to success in future projects, which emphasizes the relevance of connections between developers in OSS development [Antwerp, 2010]. Communication relations between developers are mandatory, as projects do not present patterns of traditional project management [Peterson, 2013], but as self-organizing, collaborating social networks [Madey et al., 2002]. However, it was proposed that these communication networks in OSS communities should have high centrality and low density [Wu and Goh, 2009]. High centrality brings an increase in development activity and increases popularity and thus attracts users. In contrast, high density would have a negative effect on project activity and popularity. These findings are in contrast with the behavior of developers when fixing bugs [Bernardi et al., 2018]. When investigating collaboration in OSS projects, it became apparent that most bugs are fixed by developers who are not responsible for creating them. These developers have a lower communication rate compared to other developers in the same project, from which the authors conclude that the level of communication between bug-fixing developers should be increased to reduce the total number of necessary fixes in a software project.
Although researching OSS communities is highly controversial in the scientific community, it is clear from many sources that to strengthen ties between developers and support OSS development, version management systems such as *Git* are a proven and prominent instrument. Platforms based on Git that provide communication and project control capabilities are today's standard for OSS communities.

2.2 Git

Git is a free and open source based distributed version control system to support software projects of different sizes [Chacon and Straub, 2014]. Git projects are always structured by a tree. When the project is created, the root or *initial commit* is generated. From the root, all files branch off with relations, resulting in individual branches with multiple leaves when files are changed. This structure makes a Git project a directed, acyclic graph through which the addition, modification, or deletion of individual leaves can be captured. These leaves are representations of files but also *commits* that link the files and their respective file versions.

Commits capture all changes made by users within a repository. They capture information about changes, such as an author's name, the ID of the previous commit, a log message, and a *diff* that describes the differences between changed files. Commits are permanently stored in the project. Developers have the possibility to retrieve the state of the project at any time when a commit is made. Additionally, the project can be rolled back to a previous commit at any time if necessary.

Merge requests Software development is non-linear in Git. By branching and later merging their own development branches, individual users can rapidly suggest changes in the development of a project. These branches are stored locally, which keeps Git very fast and scalable. Dedicated work is done in branches until the responsible developer decides that the branch is in a state where it adds value to the project. They can make a merge request to make changes available to the *main branch* and thus permanently change the main branch. Depending on the hierarchy of the project, this merge request must be accepted by an authorized developer. Merges, like any other change within the project, are captured and mapped by commits.

GitLab GitLab is a web administration for version management of software projects, which is based on Git. In addition to all Git functions, GitLab extends the collaboration through Git with functions in the area of software development. This includes project management tools such as Kanban boards and discussion forums. In addition, users are provided with an issue tracking system, which is typically used to point out bugs or suggest improvements that the community can discuss. Although GitLab offers free hosting of private and public repositories, the focus of this work is exclusively on public repositories, which can be queried via API. By hosting public and free repositories, GitLab attracts OSS projects and the resulting open source community.

Supporting open source software development through Git The version control system Git and systems based on it such as GitHub or GitLab offer many advantages to OSS development. Users can set up their own projects for free and invite others to collaborate with them or join other developer communities to support other people's projects. OSS projects thereby rely on voluntary work and benefit from low organizational costs [Jarczyk et al., 2014]. GitHub, as a collaboration platform, offers a transparent development environment that, through its pull-request based workflow, provides an easy way for users to commit, review, and manage source code [Kalliamvakou et al., 2015]. This is true for typical open source projects as well as commercial projects that are adapting workflows on GitHub and increasingly rely on reduced communication, more independent work, and self-organization. Transparency and the typical workflow can thereby help to promote collaboration and to utilize code and knowledge beyond individual projects [Kalliamvakou et al., 2015]. According to McDonald and Goggins, the success of GitHub is due in particular to leading users who understand the importance of community participation in projects and want to strengthen it [McDonald and Goggins, 2013]. They also found that increased levels of participation are due to the features and usability of the development platform that enable entry and work in OSS development. Features that are supportive are, for example, (1) the possibility to edit own approaches of cloned projects by branching and forking, (2) to be able to integrate commits quickly into the running project, (3) to automate processes in projects or to have them managed by bots, (4) discussion forums, issues, and merge requests to trigger communication between users, (5) reuse of code from other projects, and (6) interpret the platform as a social network between users through *follow* and *watch* options to receive updates about favored projects [Perez-Riverol et al., 2016].

2.3 Community Smell Patterns

Although software development processes are largely supported by Git, asynchronous or distributed work via version control systems can also give rise to negative patterns [Tamburri et al., 2015]. This is particularly critical since software engineering success depends on the well-being of the developer community [Keyes, 2011] and additionally the intensity of code smells is influenced by emerging community smells [Palomba et al., 2021]. A proven means to ensure well-being can be changes in organizational structure [Tamburri et al., 2013b]. Because organizational structures are complex, multi-layered constellations of people and artifacts designed to achieve goals [Pugh et al., 1969], interventions in their design must be deliberate. Before interventions, the structure must first be explored and operationalized. According to Conway's law [Conway, 1968], organizational structures, especially communication patterns within a community, can be traced through the software developed. This can be represented by sociograms, which consist of nodes (e.g., people, artifacts) and relations between the nodes [Jones, 2001]. Edges can thereby represent either social relations between people or technical relations between people and artifacts. These relations also reflect the structure of a software development community, which is a specified form of organizational structure.
When investigating the organizational structures of open source development communities, they found negatively afflicted anti-patterns in a high number of communities, which can have a lasting impact on the efficiency of software development projects [Tamburri et al., 2017]. Referring to code smells, functioning but poorly structured code that needs to be reworked [Fowler et al., 1999], they created the notion of *community smells* which resemble functioning but poorly structured communities. Community smells exist in software communities with sub-optimal organizational and socio-technical characteristics [Tamburri et al., 2017]. Using a combined study of qualitative and quantitative methods, they examined multiple communities with respect to four community smell patterns they proposed. In the following, it will be briefly explained which problems arise from the occurrence of community smells and how they can be identified on the basis of their formalization.

2.3.1 Social Debt

The term *technical debt* is associated in computer science with the possible consequences of poor technical implementation [Cunningham, 1992]. This mainly refers to additional effort incurred to make changes or revisions to poorly written software. Analogous to the notion of technical debt, the notion of *social debt* [Tamburri et al., 2013a] was coined to refer to the additional costs incurred to correct a poor development within a community. This refers to additional costs for social and organizational tasks that enable smooth development and execution of software development projects. In software engineering, social debt can be used as a term to classify a community in terms of its ability to solve problems in software development. Examining whether a current organizational layout is performant can thereby help uncover social debt [Cusick and Prasad, 2006]. Since social debt is caused by anti-patterns, these must be identified in order to intervene promptly with mitigating measures and to reduce or prevent phenomena such as social debt.

2.3.2 Formalized Patterns

Although a majority of community smell patterns have already been defined [Tamburri et al., 2015], the focus is on the four patterns *Lone Wolf*, *Operational Silo*, *Black-cloud* and *Bottleneck* [Tamburri et al., 2017] as they additionally have been formalized. While the first two patterns are composed of communication and collaboration relations, the last two patterns are formed by communication relations only. Therefore, for the formalization of the patterns, the premise that $G_m = (V_m, E_m)$ is the communication graph of a project, while $G_c = (V_c, E_c)$ represents the collaboration graph is important. Additionally, the transitive closure of the edge set E_m is expressed as E_m^*.

Organizational Silo The organizational silo effect describes sub-communities that have become distant from and independent of their development community. As a result of this separation, they not only increase their workload due to a lack of consultation, but also develop tunnel vision with a focus on communication and collaboration with close community members and a derogatory opinion towards more distant community members [Tamburri et al., 2015]. In general, this pattern consists of three project members, as shown in Figure 2.1. While developer 1 works on the same code as developer 2, they do not communicate with each other. Developer 2, unlike Developer 1, communicates with at least one other developer from the project who does not belong to the same sub-community as 1. Developer 1 thus reflects the organizational silo in this pattern.

$$(v_1, v_2) | v_1, v_2 \in V_c, (v_1, v_2) \notin E_m^*$$

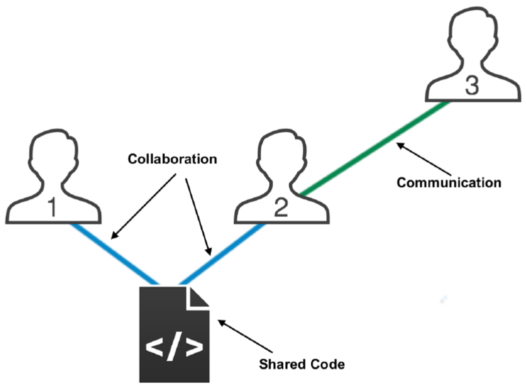

Figure 2.1: Representation of the *Organizational Silo* community smell pattern [Tamburri et al., 2017]

Lone Wolf The *Lone Wolf* effect refers to a structure where a dyad of developers collaborates with each other and does not communicate directly with each other. This favors negative effects such as code duplication and free-riding of individual developers. To prevent these effects, the dyad of collaborating developers that communicate with a common stakeholder but not with each other should be identified. Figure 2.2 shows this constellation with a missing communication link between the *Lone Wolves*. Since this pattern is similar to the *Organizational Silo*, finding both patterns simultaneously is highly likely. At the same time, therefore, the negative consequences of both patterns can occur, which is particularly critical to observe. In the example shown below, developers 1 and 2 resemble a *Lone Wolf* dyad.

$$\{(v_1, v_2)|v_1, v_2 \in V_c, (v_1, v_2) \in E_c(v_1, v_2) \notin E_m^*\}$$

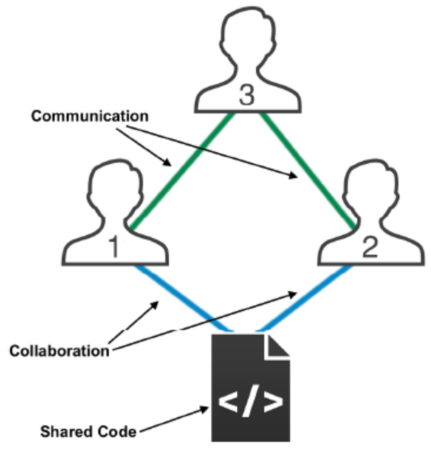

Figure 2.2: Representation of the *Lone Wolf* community smell pattern [Tamburri et al., 2017]

Black Cloud The *Black Cloud* effect is based solely on communication relations that lead to negative social interactions within a software development community. The lack of knowledge exchange between community members leads to gaps in knowledge management. These foster a climate of uninformedness and impose time and mental costs in reconciling the discrepancies between sub-communities [Tamburri et al., 2013b]. The *Black Cloud* effect, like the previously mentioned smells, provides for a division of the community through emergent selfish behavior. Future communication between sub-communities or individual developers is thus impaired. Finding the *Black Cloud* effect depends on cluster identification with respect to communication in the project. In the *CodeFace4Smells* tool [Tamburri et al., 2017], they draw on the *CodeFace* tool developed by Siemens, which is used to analyze software projects. In *CodeFace* there is an unspecified algorithm for clustering the vertices, which is also used for *Black Cloud*. After clustering, the sets of pairs that connect the isolated sub-communities are determined. Communication between the two developers of a pair of nodes does not take place regularly, but with large temporal pauses. For this reason, data from different points in time must be analyzed. Each node pair found that meets these criteria represents an instance of the *Black Cloud* pattern as shown with developers 3 and 4 in figure 2.3.

$$\{(v_1, v_2)|v_1, v_2 \in V_m, (v_1, v_2) \in E_m, \forall i, j(((v_1 \in p_i \wedge v_2 \in p_j) \Rightarrow i \neq j) \wedge \forall v_x, v_y((v_x \in p_i \wedge v_y \in p_j \wedge (v_x, v_y) \in E_m) \Rightarrow v_x = v_1 \wedge v_y = v_2))\}$$

with $P = \{p_1, ..., p_k\}$ as mutually exclusive, completely exhaustive partition of V_m induced by the clustering algorithm.

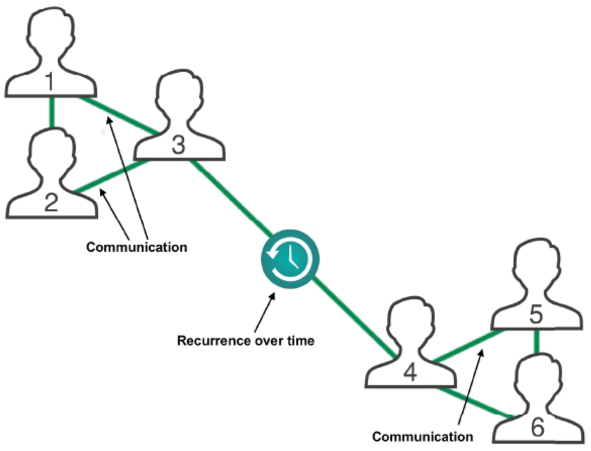

Figure 2.3: Representation of the *Black-cloud* community smell pattern [Tamburri et al., 2017]

Bottleneck To find the *bottleneck* effect, clusters within the network are compared again. Community members who are the only members of their sub-community communicating with at least two other sub-communities are considered *Bottleneck*. Referring to Figure 2.4, Developer 3 forms the bottleneck in communication and prevents more pronounced information flow between other developers. As a gatekeeper, it influences the speed of exchange and thus of the entire project by acting as the single point of contact for any formal interaction between sub-communities. For projects with a highly formal organizational structure, this pattern can bring strong implications as there is no parallel communication path to bypass the *Bottleneck*. Possible consequences are long times between change proposals and their executions, as well as lack of knowledge about information channels due to the gatekeeper's position as a unique boundary spanner between different sub-communities [Tamburri et al., 2013b][Tamburri et al., 2015].

$$\{v|v \in V_m, \exists i(v \in p_i \land v_x(v_x \in p_i \Rightarrow v = v_x))\} \cup \{v|v \in V_m, \exists v_x, i, j(v \in p_i \land v_x \in p_j \land (v, v_x) \in E_m \land \forall v_y, v_z((v_y \in p_i \land v_z \in p_j \land (v_y, v_z) \in E_m) \Rightarrow v_y = v)\}$$

2.4 Provenance

This section briefly introduces the concept of provenance and mainly focuses on electronic provenance and the generalized provenance data model. The elements of the data model are crucial for understanding the transformation process of the provenance graph into communication and collaboration models, which are necessary to find community smell patterns within the project under study. Since this work focuses exclusively on data from GitLab, both the basic tool for extracting data from Git and the resulting extension, GitLab2PROV, are presented as Git provenance models.

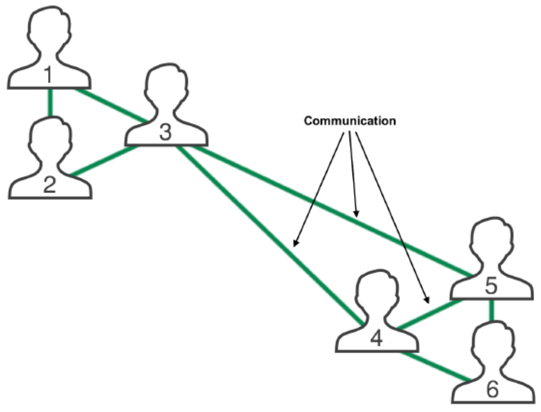

Figure 2.4: Representation of the *Bottleneck* community smell pattern [Tamburri et al., 2017]

2.4.1 Definition

Coming from the French term “provenir”, which translates to “to come from”, provenance is a concept, that is used for valuable items and mostly precious works of art [Feigenbaum et al., 2012]. While art lovers attach great importance to provenance in being able to tell the story of an object from earlier epochs, the greatest value for the art market lies in proving the authenticity of art objects through information about the artists, restorers and previous owners.

2.4.2 Provenance Data Model

The concept of provenance was considered and adapted for computer-based data [Moreau et al., 2008]. With the assembly of documents and financial transactions in mind, the focus here is on helping users, regulators, and reviewers to verify data and their origin resulting in trust towards data. This is especially necessary if derivations of these documents were created in the past. Recording the provenance of said documents helps to simplify and accelerate the verification process.
As stated by Moreau et al. for the W3C [Moreau et al., 2008],

> *”provenance describes the use and production of* ***entities*** *by* ***activities****, which may be influenced in various ways by* ***agents.****”*

Following this principle, the *provenance data model* [Moreau and Missier, 2013] introduced by the World Wide Web Consortium (W3C) aims to standardize the recording and exchange of provenance information. The standard includes the three types of activities, entities and agents and the associated relations. Through these components, the graphical representation for recording provenance is enabled in the graph-based data model as displayed in figure 2.5.

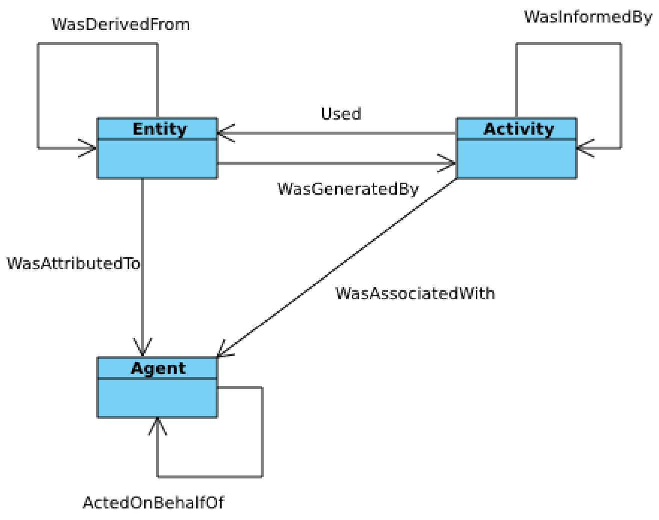

Figure 2.5: Provenance core structures and their relations [Moreau and Missier, 2013]

Provenance Types Utilizing the previously mentioned types and their relationships, every derivative of a document can be traced back to its origin since the provenance graph of a document will always be a directed acyclic graph. In these directed acyclic graphs, the types depend on each other as follows:

- **Entities** An entity can represent a physical, digital, or other type of thing that has several specified properties. An entity does not have to be real or material, it can also be imaginary or immaterial. In graphical representation, entities are shown as yellow colored ovals in provenance graphs.
  Common examples of entities are documents, a state, or files within a file system. Abstract concepts, such as ideas, can also be represented by entities.

- **Activities** An activity always interacts with or influences an entity. The entity can be changed, consumed, transformed, used, moved or even generated by the activity. The activity takes place over a period of time, which is represented by a start and an end time as attributes. The graphical representation is done by a violet blue colored rectangle.

- **Agents** Agents are the originators of activities and thus simultaneously responsible for the resulting consequences on entities. Depending on the participation in an activity or responsibility for an entity, the agent is assigned a role within the provenance graph. For example, they can become the originator of an entity or the modifier of an entity via an activity. The agent is graphically represented by an orange pentagon.

Each of the previously mentioned elements is also characterized by multiple attributes. For each agent, activity and entity there is a unique identifier `id`. Further `attributes` can be found depending on the considered network and can be defined when displaying the said provenance graph. Activities are a special feature, since they are characterized by a start and end time. By the specific attributes `startTime` and `endTime` the time is represented in a usual DateTime format. An exemplary representation of these attributes can be seen in figure 2.6.

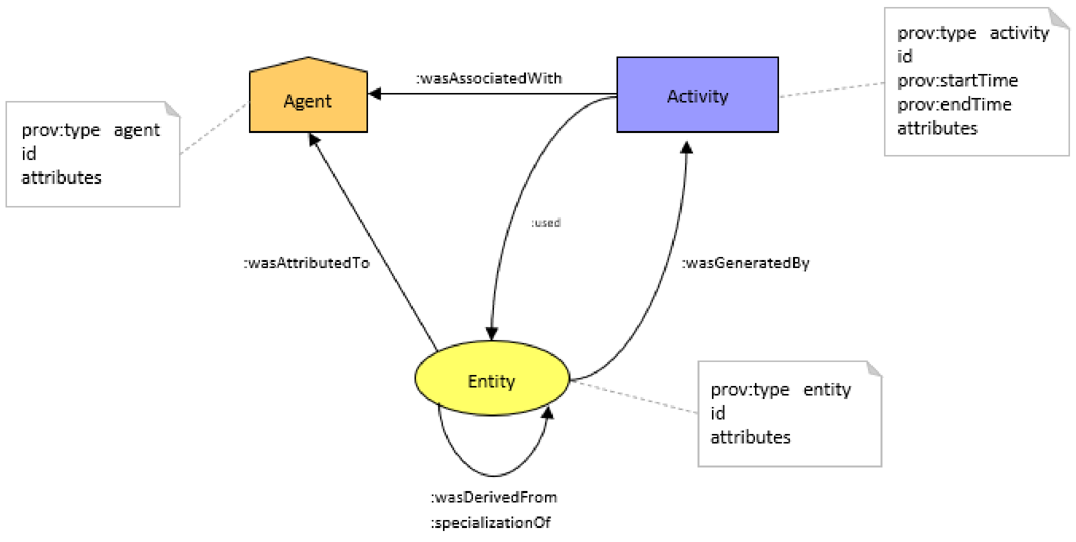

Figure 2.6: Representation of provenance graph elements with their relations and attributes.

Provenance Relations In addition, agents, activities and entities are linked to each other by relationships. The nodes of a provenance graph are connected by semantically named directed edges that represent the relationships between the nodes. What is special here is that instead of running from cause to effect, the edges are oriented in the opposite direction. Thus, starting from a concrete node of the graph, the edges can be used in statements about the antecedents of the node and therefore showing the process of creation. The standard defines a set of relations for this purpose, as well as a scheme that specifies which relations may be used between which nodes. In the following, the relations relevant for this thesis are briefly defined.

- **wasDerivedFrom** If an entity is modified by an activity, a derivative of this entity is created. Both entities are connected by the `wasDerivedFrom` relation, where the edge of the derivative points to the original.

- **specializationOf** Not included in the original provenance data model and therefore missing in figure 2.5 is the relation between two entities. The first entity marks the original version, whereas the second entity is, as the name suggests, a specialization of this first entity. It contains all the information of the first entity and extends its scope by adding further information.

- **used** Usage of an entity by an activity is shown by a `used` relation. The activity has not used the entity before and therefore could not be influenced by the entity. It should be noted that an activity cannot use the same entity more than once. The relation always points from the activity to the utilized entity.

- **wasAttributedTo** An entity created after the execution of an activity is assigned to the agent that initiated this activity. The relation `wasAttributedTo` expresses the direct relationship between agent and entity, where the edge always runs from the entity to the responsible agent.

- **wasGeneratedBy** If an entity is created as part of an activity, the entity is generated by the activity. From this follows the relation `wasGeneratedBy`, which indicates by which activity the entity was generated. The relation always points from the generated entity to the parent activity.
- **wasInformedBy** The exchange of two activities regarding a common entity takes place via the relation `wasInformedBy`. This is true if one of the activities was responsible for the generation of the entity and the second activity utilizes this entity in the further process.
- **wasAssociatedWith** An agent that performs an activity is associated with it. This is represented in the form of the relation `wasAssociatedWith`, starting from the triggered activity and ending with the responsible agent.

Combining all the previously defined parts resulted in the provenance data model, which now provides the basic structure for the representation and extraction of provenance graphs. Introducing this model fits into a phase of switching from monolithic applications with their own standards and data formats towards standardized and centralized formats, making it possible to use the same data types for multiple applications.

2.4.3 Git & GitLab Provenance Models

One possible application of the provenance data model is the representation of collaboration and interaction in software version control systems. Therefore, the adaptation of the provenance data model for the extraction of provenance information from the most prominent version control system *Git* is described in detail below. Because this work is based on the provenance graphs of the Git-based platform *GitLab*, the extension by web interfaces and their representation as provenance graphs must be taken up. The models listed in table 2.1 are fundamental to understanding the extraction of provenance graphs from the GitLab platform and play an important role in subsequent convolutional operations.

Git Commit Model	GitLab Web Resource Models
File Addition (Fig. 2.7)	GitLab Commit (Fig. 2.10)
File Modification (Fig. 2.8)	GitLab Issue (Fig. 2.11)
File Deletion (Fig. 2.9)	GitLab Merge Request (Fig. 2.12)

Table 2.1: Overview of Git Commit and GitLab Resource Models used in this work.

Git2PROV One mechanism to ensure provenance of files in software development are version control systems like Git. Although these are mainly used to facilitate collaboration on code and files within projects, in the background they store the history of each file. This explains the utilization of the provenance data model to represent the stored provenance within version control systems. The goal of their work was to achieve increased interoperability between different systems that track provenance information. For this, a representation of the Git version control system is proposed, where changes to files are mapped using the provenance graph [De Nies et al., 2013]. The Git2PROV tool developed in the framework is based on a web service and by entering the URL of a Git repository, the contained provenance information is serialized into a readable format like PROV-JSON or PROV-XML. At the time of Git2PROV release, only openly accessible repositories can be retrieved. The selected accessible repository is temporarily cloned by the tool and a `Git log` command is executed. Resulting output is then mapped with respect to the provenance data model and then returned as a response from the web service in the requested serialization. For the mapping, the Git commit model is considered, where the addition, modification and deletion of a file is represented.

Git file addition Figure 2.7 represents the addition of a new file to a project through a Git commit. The commit itself is represented by the `Commit` activity, which is associated with the `Author` and the `Committer`. Due to the directed, acyclic structure of Git, almost all `Commit` activities have a predecessor called the `Parent Commit`. Only the initial commit has no relation pointing to a previous action. The commit generates two entities when a file is added. The first `File` entity represents the concept of the file itself, while the `File Version` entity represents the initial version of the added file. Thus, the `File Version` is declared as a specialization of the `File` and connected by the corresponding relation. Both entities are attributed to the `Author` of the commit, since it is presumed that the author created the two files included in the commit. Modifications can create additional `File Version` entities that always point to the original `File` entity and have a relation to the previous `File Version` entity.

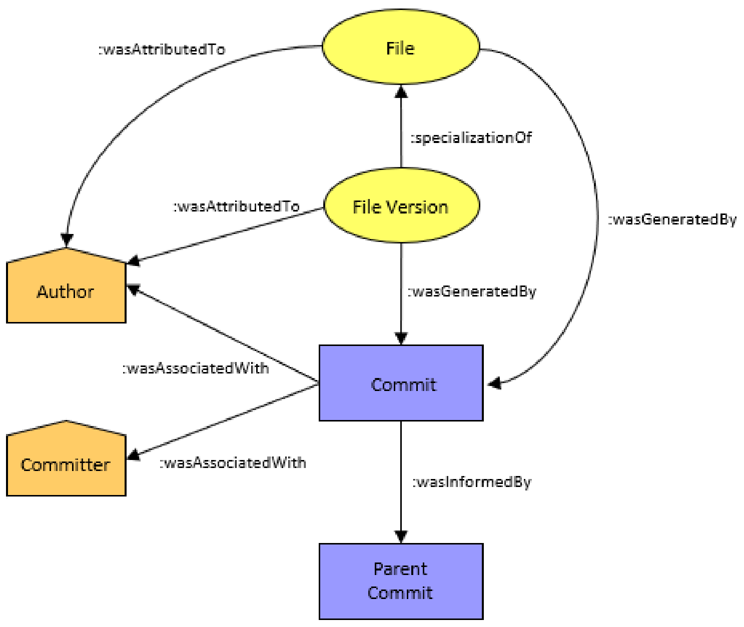

Figure 2.7: Model of adding a new file to a project in Git.

Git file modification The model for modifying an existing file through a Git commit is represented by figure 2.8. The commit is represented by a `Commit` activity as before, and as mentioned before, a new `File Version` entity is created. Thereby, the `File Version` entity created by a previous commit is now declared as a `File Version N-1` entity which is used by the current `Commit` entity. The current `File Version` entity is marked by an appropriate relation as a derivative of its predecessor and at the same time as a specialization of the original `File` entity. Unlike the model of adding a file, here the `File Version` entity is associated with the `Author`.

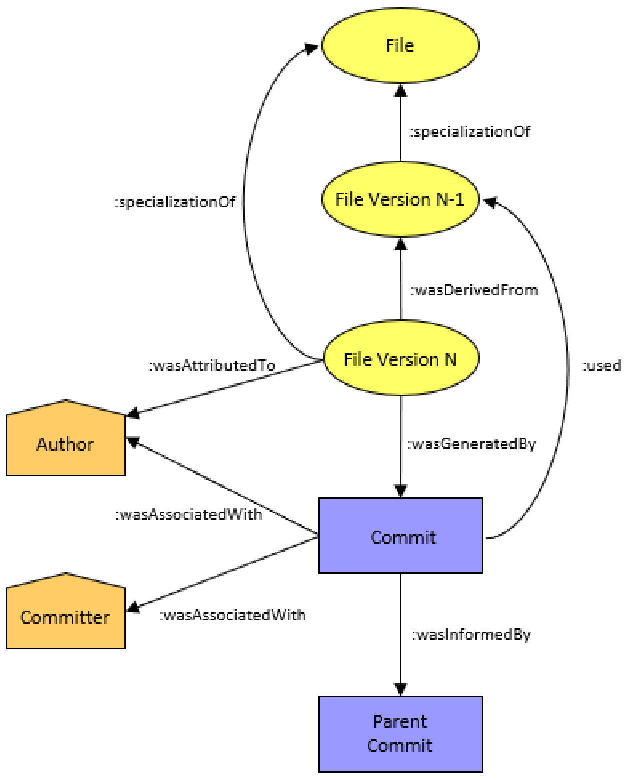

Figure 2.8: Model of modifying an existing file in a project in Git.

Git file deletion Figure 2.9 shows the deletion of a file from a project by a Git commit. The Git commit is represented by the `Commit` activity, which is associated with the `Author` agent and `Committer` agent as responsible users. The `Commit` activity is related to all previous `Parent Commit` activities involving the processed `File` entity. The deletion of a file by a commit is handled by a special `File version`, which is immediately marked as invalid by the `Commit` activity. This `File Version` entity is thus the last descendant of the `File` entity and is marked as a specialization of it.

GitLab2PROV As platforms like GitHub and GitLab extend the version control system with web interfaces, additional features are provided, such as detailed documentation and discussion of commits, issues and merge requests. The data from these functions can be queried via a web API and displayed in provenance graphs as an extension to Git2PROV. For GitHub, this resulted in the GitHub2PROV tool [Packer et al., 2019], where issue tracking and management of merge requests, as well as communication

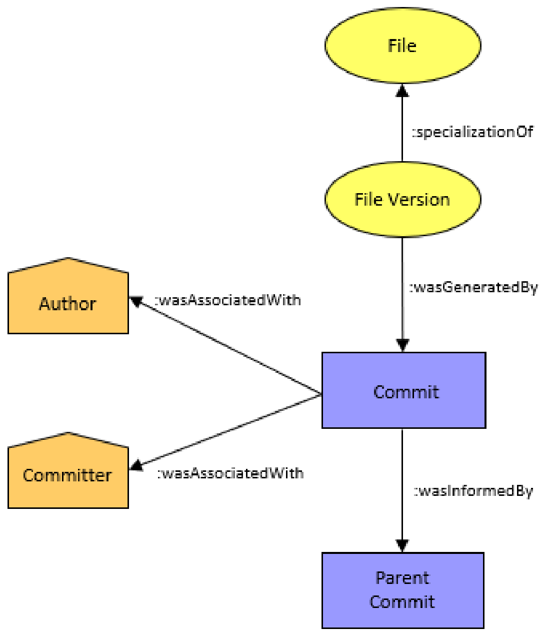

Figure 2.9: Model of deleting a file from a project in Git.

initiated by these functions between users in the project, could be represented as extensions to the provenance graphs created by Git2PROV. Comparable to Git2PROV, this was done through the output of Git log commands issued to the GitHub API. The provenance graphs generated by GitHub2PROV were used to show how the added value of information could also be used in explaining phenomena and questions from the field of project management. Based on this, the GitLab2PROV tool was developed [Schreiber and de Boer, 2020], targeting the GitLab platform. In addition to issue tracking and managing merge requests, GitLab2PROV also records the web interface when commits are created and increases query speed by using an asynchronous HTTP client framework. To represent the retrieval of web resources from GitLab projects, models have been developed to represent the processes for recording actions in GitLab.

GitLab web resources The GitLab web resource models, which record user interactions via GitLab web resources, handle interactions on commits, issues, and merge requests. The commit web resource is used when a commit is created. A committer can tag their own commit with a message, so the supposed changes by that commit can later be tracked by other developers easily. Other users in the project can annotate this commit, which is also recorded by GitLab.
Similarly, in the issue tracking system, a user points out a bug, optimization potential, or improvement, and the community can annotate the issue with comments or reactions displayed by emoticons.
Merge request resources are similar to commits, as a committer can and should describe what changes have been made in their development branch that is about to be merged. Authorized people in the project can now decide whether to accept the merge request or decline it, based on the quality of the submitted solution in the merging branch.

The GitLab platform explicitly recommends using merge requests to start discussions. The web resources mentioned above allow users to discuss and exchange opinions. They record the internal communication within a project on the GitLab platform and thus have a high value in the analysis of social networks in GitLab software projects. These web resources are recorded by the models presented in the following.

GitLab web resource models The various models for recording commit resources, issue resources, merge resources and their respective annotations do not differ significantly in terms of their structure. Therefore, only the model for Git commits (figure 2.10) is explained, but the models for issues (figure 2.11) and merge requests (figure 2.12) are included for the sake of completeness. The selected model deals with two different

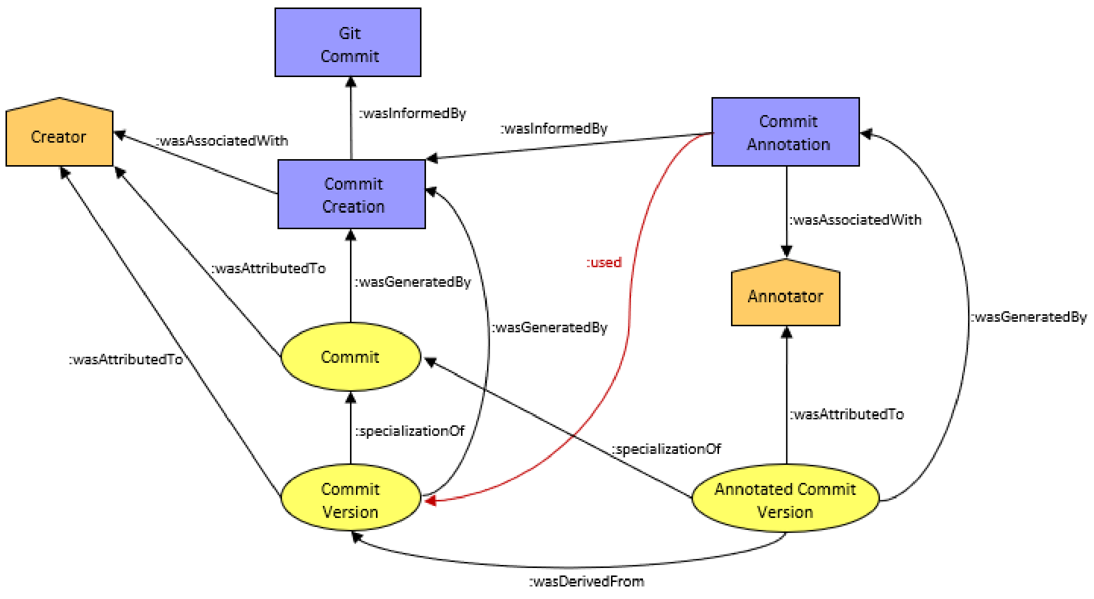

Figure 2.10: Model of creating or annotating a commit in GitLab.

processes in recording web resources: the creation of a web resource and the evolution of the resource over time, after users and the system have annotated and thus initiated events against the resource itself.
When creating a Git commit and the resulting `Commit` activity, a web interface is created for the commit. This interface is represented by the `Commit Creation` activity, which is related to the initiating `Commit` activity. Unlike the Git commit models, the `Creator` agent is associated with the creation of the commit instead of an `Author` agent. This is also attributed as being responsible for the `Commit` web resource and the initial `Commit Version` resource specialized by it. Both the `Commit` resource and the `Commit Version` resource are generated at creation time by the `Commit Creation`.
An annotation of the `Commit Version` resource is described as an event against the web resource. Through this `Commit Annotation` event, a new version of the resource is created that derives from the original `Commit Version`. This new resource and all future `Annotated Commit Version` resources are recorded as specializations of the original `Commit` resource. The executing `Annotator` agent is associated or attributed with the annotation event and the resulting `Annotated Commit Version` resource respectively.

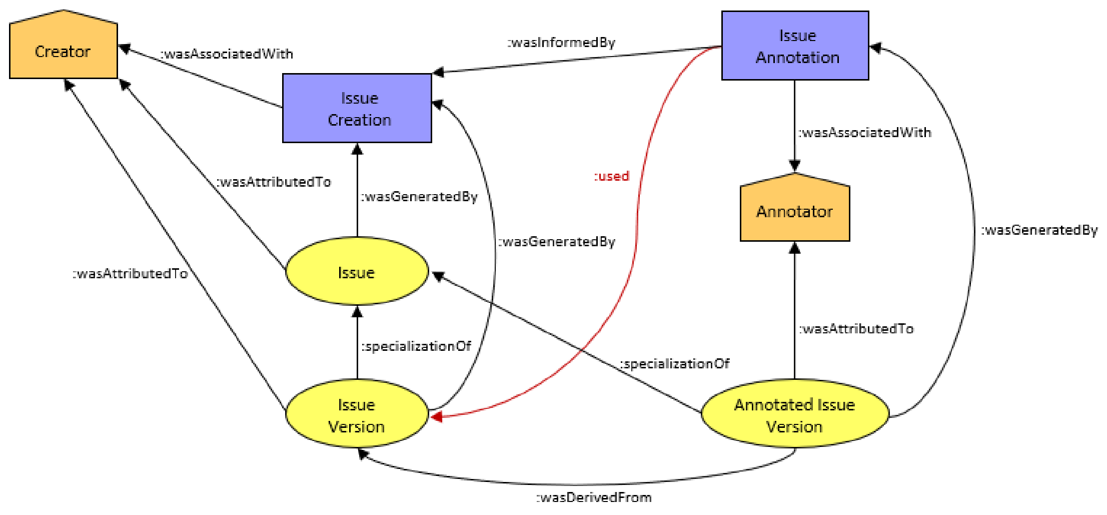

Figure 2.11: Model of creating or annotating an issue in GitLab.

This approach differs for issues and merge requests only with respect to the missing `Commit` event, which is initiated by adding, modifying, or deleting a file.

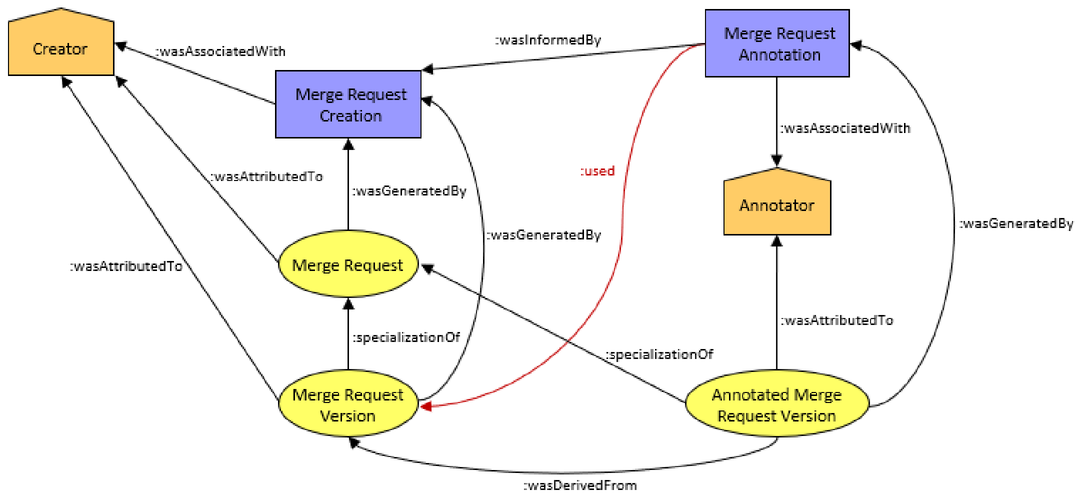

Figure 2.12: Model of creating or annotating a merge request in GitLab.

2.5 Graph Databases

Due to exponentially increasing data volumes, current database solutions are facing major challenges. In most cases, these are based on the principle of relational database management systems, in which data is stored in tables consisting of rows and columns. The standardized query language (SQL) is utilized to query this data. If data is retrieved from multiple tables at the same time, JOIN operations have to be performed by the system, which lead to an increasing computational intensity due to high complexity. In particular, the waiting time between queries and results increases with the amount of data. Relational database management systems are therefore not suitable for structural and topological analyses of large and highly interconnected data sets [Vicknair et al., 2010]. These analyses include biochemical interaction networks, provenance analysis methods, and social network analysis.
To work around these challenges, databases can be used that are not based on the relational database management system and therefore not based on SQL. These can be grouped under the name of NoSQL databases, which also include graph databases. Thereby, the data is not stored in tables, but in a native graph model. Two prominent models in the use of graph databases are the Resource Description Framework model and the Labeled Proerty Graph (LPG) model. The graph database Neo4j used for this work is based on the LPG model. Labeled Property Graphs are characterized by directed multi-graphs whose nodes and relations are labeled [Frisendal, 2016]. Figure 2.13 shows an exemplary representation of an LPG visualizing a relation between the two neighbors Alice and Bob.
As can be seen in the figure, the LPG model consists of two components: Nodes and Edges.

Nodes Nodes can have any number of properties. These consist of a key and the associated value. In the example, the properties `node_ID` and `date_of_birth` are

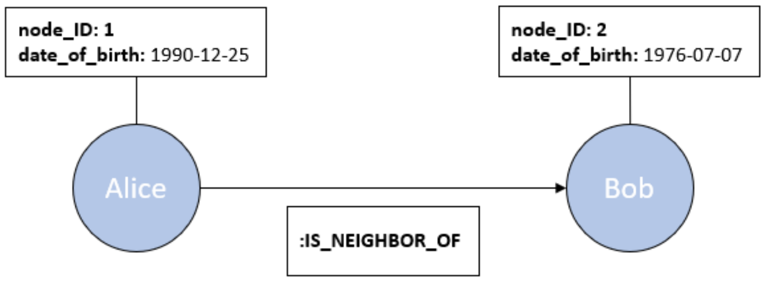

Figure 2.13: Exemplary representation of a relationship between the two neighbors Alice and Bob.

listed with their respective values. Additionally, labels can be assigned to nodes. For example, in the context of Git, labels such as `user` and `file` are possible to identify the respective data types. Within the LPG model, multiple edges can occur between two nodes, hence the model is also described as multi-relational.

Edges Edges always connect only one start and one end node with a given direction. An edge must be distinguished by a label and can additionally contain properties in the form of key value pairs. Alice and Bob are connected by a relation labeled `IS_NEIGHBOR_OF`.

2.5.1 Neo4j

Neo4j is a transactional graph database provided free of charge for non-commercial users under the Gnu General Public License v3. Individual databases can be set up with Neo4j using either dedicated servers or a local instance. Both options offer access via Bolt, HTTP or HTTPS protocols through an integrated interface, allowing queries to be made or data to be added. This is done via `create`, `read`, `update` and `delete` operations optimized for the LPG model.
Since relations are stored separately in Neo4j instead of referencing another table, the speed of certain operations is significantly faster than in relational database systems. Especially for large, directed, acyclic graphs, Neo4j performs significantly better than relational graph databases in the area of pattern matching [Pobiedina et al., 2014]. With regard to the queries, a distinction must thereby be made as to whether and what is filtered in the query. While a query with filtering by node types in Neo4j performs significantly better than a relational graph database, this effect is opposite for filtering by edge types [Hölsch et al., 2017]. When both node and edge types are used as filters in a query, relational and non-relational graph databases perform comparably. For the identification of community smells, the query formulation is important since finding community smells is based on the principle of pattern matching.

2.5.2 Cypher Queries

Queries in Neo4j are formulated using the *Cypher* query language. Cypher was developed by Neo4j Inc. exclusively for its in-house graph database, but was adopted as an open source project by the openCypher Implementation Group in 2015 and has been maintained ever since. As previously mentioned, the operations `create`, `read`, `update` and `delete` (CRUD) can be executed, which have been adapted to the property graph model as part of Cypher. These and other reserved keywords are interpreted by Neo4j and executed according to previously defined scheme. Important keywords in the context of this work are:

CREATE The `CREATE` clause can be used to create individual nodes or edges in a database. A completely new graph can thereby be created or new nodes and edges can be added to an existing graph. Labels and properties can also be defined during creation. For edges, a direction must also be specified, since Neo4j specializes in directed graphs.

MATCH The `MATCH` clause, together with `RETURN`, is probably the most important. `MATCH` initiates the query for patterns within the graph to search for. The patterns searched are specified by arbitrarily long concatenations of nodes and edges. For specification, these nodes and edges can be filtered with respect to their labels or properties, but a query can also be made without any filtering.

WHERE For constraining a `MATCH` query, `WHERE` can be used, to reduce the amount of necessary calculations for a query by specifying conditions.

RETURN This clause can be used to specify which nodes, edges, properties or labels are returned after a query. The representation of the return can also be specified and whether only unique rows should be returned by using the `DISTINCT` operator.

Additionally, there are logical operators like `AND`, `OR` or `NOT` which can partially be combined within queries. An example query for the birth dates of all people named Bob who are neighboring Alice (figure 2.13) is shown below.

```
MATCH (alice)-[:is_neighbor_of]->(bob) RETURN bob.date_of_birth
```

2.5.3 prov2neo & neo4r

To create a database in Neo4j with external data sets and to query data within the database, tools can be used instead of the Neo4j instance with integrated browser. For these tasks *prov2neo* and *neo4r* were employed.

prov2neo Programs like Git2PROV or GitLab2PROV extract files from software repositories and store them as PROV-JSON. In order to make these JSON files usable for an analysis in databases, the prov-db-connector was developed by the German Aerospace Center. The tool allows W3C-PROV documents to be read into databases via an interface. A revised version specialized for Neo4j is called *prov2neo.* prov2neo is characterized by a higher speed when reading PROV documents into the Neo4j graph database and was therefore used in this context.

neo4r Neo4j offers the possibility to analyze graphs by own queries. In addition, the Neo4j internal extension *Graph Data Science Library* provides methods to identify structures within graphs. For this purpose, centralities and similarities of nodes can be calculated, communities can be detected and paths can be found. Even though the library offers some options for action, the integration of many algorithms is missing and the processing of queried data sets is expandable. One tool to address these issues is *neo4r* [Fay, 2019]. With neo4r, a connection is created between Neo4j and an *R* [Team, 2017] instance. Queries can be made in R via a connection object and are sent to Neo4j. The result of the query is then returned as *tibble*, a successor to data frames, which is parsed more deeply by R packages such as *dplyr* or *igraph*. Due to the amount of packages in R that can be used for data and graph analysis, neo4r provides the necessary interface to enable the use of sophisticated algorithms.

2.6 Network Analysis

In the following section, measures of interest for this work are presented for the analysis of networks. First, provenance graph analytics are briefly introduced, which are used to characterize the topology of provenance graphs. Afterwards, measures used in social network analysis are described, which describe properties of the investigated networks and allow comparability between different networks.

2.6.1 Provenance Graph Analytics

Following the principles of provenance networks analytics, the topology of a network should be analysed first in the context of provenance graphs [Huynh et al., 2018]. For this purpose, existing network metrics [Newman, 2010] can be adapted for provenance graphs to highlight their structure. These metrics can be calculated generically by provenance records and thus form the basis for quantitative analysis and comparison with other provenance graphs. In this work, the focus is only on basic properties of provenance graphs, which is why a limited selection of metrics is presented and also used.

A provenance graph is a directed graph $G = (V_G, E_G)$ consisting of a vertex set V_G and an edge set E_G. All vertices in V_G represent the different PROV elements introduced in section XXX: entities, activities and agents. Two vertices can be connected by a directed edge $e = (v_i, v_j) \in E_G$ if a relation in the provenance graph connects this pair of vertices. To capture the quantitative, topological characteristics, these are represented as follows:

- Number of provenance elements in G corresponds to the number of nodes $n = |V|$.
- Number of provenance relations in G corresponds to the number of edges $e = |E|$.

In addition to the nodes and relations, the graph diameter is also interesting for this work, showing how distributed the provenance graph G is. The diameter d_G indicates the longest distance within a graph G, where two nodes u and v are connected by the shortest path $d(u, v)$. This results in the following notation for the distance:

$$d_G = max_{u,v \in V_G} \ d(u, v)$$

2.6.2 Social Network Analytics

Other generic metrics can be found in the area of social network analysis. Social network analysis can be used to understand interactions and social organisation within a network (Wasserman & Faust, 1994). Depending on the semantics of the network under investigation, different analyses can be carried out based on the characteristics of the considered network. Proposed by Dos Santos et al., a network can be classified in terms of collaboration by the properties `density` and *centrality degree* [dos Santos et al., 2011]. In their analysis of open source software developer groups, they try to classify networks in terms of coordination of collaboration within a project by these two properties. Their approach is to map collaboration processes of different levels of collaboration with social network properties. These collaboration processes and their levels were defined in the Collaboration Maturity Model (CollabMM) [Magdaleno et al., 2009], which assumes that communication is an important component but is subordinate to collaboration. As there is a strong separation between communication and collaboration networks in this work, the notion of collaboration according to Dos Santos et al. needs to be critically questioned, especially with regard to the data investigated by them [dos Santos et al., 2011]. Only data from bug trackers and mailing lists were extracted, in which developers discussed the prioritisation of bugs and possible solutions. Therefore, the network measures they use are considered here for communication graphs.

Degree centrality Degree centrality describes the degree of involvement of an actor with other nodes of a network. The centrality can be considered with regard to the entire network (*network centrality*) or the centrality of an individual node compared to other nodes via *node centrality* [Freeman, 1978]. Formally, node centrality is defined by the number of vertices adjacent to a vertex v:

$$C_D(v) = deg(v)$$

Network centrality is determined by *graph centralization.* The graph centralization is determined by the node centrality of all vertices in the network:

$$C_D(G) = \sum_{i=1}^{|V|} [C_D(v*) - C_D(v_i)]$$

With $v*$ as vertex with the highest degree centrality in the graph G. For a normalization of graph centralization, this can be divided by the theoretically maximum achievable score H for graph centralization of the initial graph G. The same attributes as number of vertices and parameters (e.g. considering loop edges) must be used while calculating H. Correspondingly, the normalized graph centralization of G is given by:

$$C_D(G) = \frac{\sum_{i=1}^{|V|} [C_D(v*) - C_D(v_i)]}{H}$$

Betweenness centrality Betweenness centrality can be measured by how often shortest paths between two vertices pass through a vertex v. Vertex v acts as a bridge between the respective vertex pairs and its betweenness can be used to quantify how much control it has in social network communication [Freeman, 1977]. The betweenness centrality of a vertex can be described as follows:

$$C_B(v) = \sum_{i \neq j \neq v \in V} \frac{\sigma_{ij}(v)}{\sigma_{ij}}$$

Where σ_{ij} maps the total number of shortest paths between i and j, while $\sigma_{ij}(v)$ maps the shortest paths between i and j that pass through v (Brandes, 2001). Similar to degree centrality, betweenness centrality can also be related to the entire graph. For the normalized variant of the betweenness centrality of the graph G this results in:

$$C_B(G) = \frac{\sum_{i=1}^{|V|}[C_B(v*) - C_B(v_i)]}{H}$$

Closeness centrality In addition to the concepts of density and degree centrality presented above, the influence of closeness centrality on the emergence of community smell patterns has been demonstrated [Almarimi et al., 2020a], which is why it is also considered an informative tool of social network analysis for this work. Closeness centrality assumes a connected graph and measures the average length from the shortest path between vertex v and all other vertices in a graph G [Sodeur, 2019]. The higher the closeness centrality of a vertex, the closer it is to all other vertices:

$$C_C(v) = \frac{1}{\sum_i d(i,v)}$$

with $d(i,v)$ as the distance between the two vertices v and i. Usually, however, the normalized closeness centrality is given, where the formula is multiplied by the number of vertices in the graph N and the initial vertex is subtracted. This allows a comparison between graphs of different sizes. In larger graphs, the subtraction is omitted, resulting in the following formula:

$$C_C(v) = \frac{N}{\sum_i d(i,v)}$$

While for directed graphs the direction of the connection between the vertices must be considered, this distinction is obsolete for undirected graphs. For graph closeness centrality, the same calculation method as for degree centrality and betweenness centrality applies across the entire graph:

$$C_C(G) = \frac{\sum_{i=1}^{|V|}[C_C(v*) - C_C(v_i)]}{H}$$

Proportions of node centralities As another distinguishing criterion, Dos Santos et al., (2011) use proportions to compare network topologies. Unlike network centrality, the consideration of proportions allows further conclusions about the proportion of vertices with high centrality. For this purpose, the vertices with high centrality are compared with the totality of all vertices within graph G:

$$I_V = \left(\frac{V_C}{|V|}\right)$$

Here I_V represents the proportion of the central or mediating vertices. V_C is the number of vertices with centrality equal or greater than the centrality in the whole graph, while $|V|$ is the number of all vertices in the graph. The computation of these proportions can be done for any of the previously presented centralities.

Density In other studies on communication within open source developer communities, density is used for characterization [Wu and Goh, 2009][Ehrlich and Cataldo, 2012]. Density is seen as a strong influencing component on the project success of open source software projects [Wu and Tang, 2007] and indicates the compactness and especially the level of cohesion of a network [Scott, 2002]. A high density is indicative of a close network that is ready to respond quickly to changes. The density of a network is the percentage of existing connections out of all theoretically possible connections:

$$D_G = \frac{2|E|}{|V|(|V|-1)}$$

All the bases described so far will be connected practically in the following sections. Following is the concept and implementation of the tool for identifying and analyzing community smells in provenance graphs.

3 Concept and Implementation

The following section explains the concept of how data is extracted from considered GitLab repositories, then transformed and finally analyzed. Special features of the transformations will be highlighted. Furthermore, the concept describes the approaches to extract community smell patterns and how these identified patterns can be evaluated in the network. The subsequent subsection deals with the implementation of the concept and which problems and modifications occurred during the implementation. After that, improvements and adaptations that add value to the actual implementation are described and how they were realized.

3.1 Concept

The concept is divided into a total of 3 parts. At first, the criteria to choose possible projects is proposed. Then the tools used to extract data from the GitLab repositories and insert it into the Neo4j graph database are outlined. A description of using provenance graphs in Neo4j, is described and which transformation steps must be carried out in order for community smell patterns to be found at all. This process of extracting, importing and transforming provenance graphs can be seen in figure 3.1. In the following, it will be shown how these transformed provenance graphs can be analyzed with respect to community smell patterns. These will then be evaluated with respect to the patterns found compared to randomized graphs.

3.1.1 Extracting & Importing Data

The first step is to determine which repositories should be examined. For this purpose, the list of popular projects on GitLab was observed over a period of several weeks. Each project was checked in terms of size and communication structures between the contributors. The following criteria for the selection could be derived:

- **Size** The selected projects should be of different sizes in order to be able to detect any differences in the analysis of different sized projects in terms of community smell patterns. The number of developers should not fall below a minimum value of 10 as some researched patterns only emerge with a minimum of 7 users. In addition, the number of issues written so far, the number of submitted commits and the number of confirmed merge requests were interesting in order to be able to detect patterns at all. It was important not to select excessively large projects, since the resources for the calculation were limited and the amount of calculations increases exponentially with the increasing size of a project.

- **Communication Behaviour** Communication should mainly take place on the GitLab platform. Some observed projects externalized their communication to other platforms like *Slack* or *Discord* which led to their exclusion from analysis. Since the density of communication was much lower due to this externalization, the likelihood of some community smells increased drastically (compare *Lone Wolf*, *Organizational Silo*), while others became less likely (*Black Cloud*, *Bottleneck*), since they are dependent on communication relations. Therefore, only projects with a majority of internal platform communication were considered.

- **Accessibility** GitLab hosts both open-access and closed source projects. Since GitLab2PROV allows access to openly available projects via access token and the aim of this work is to analyze open source software communities, only such projects were considered.

- **Popularity** As described at the beginning, only projects on GitLab's current favorites list were considered. These are ranked based on hits and ratings from the community and therefore appear to have added value for other users. A higher popularity could in turn attract new developers or users, stimulating communication and collaboration in the project.

Considering all these factors, three projects were selected.

1. **IGitt** is a library that provides access to various Git based version control systems like GitHub or GitLab via a unified python interface. Development began in early 2016, however it appears to have been terminated in 2019.

2. **Freedesktop** provides a platform and software development kit for Linux and Docker-based applications. It is an extension and upgrade of the outdated Flatpak SDK, which in turn acts as a host for Linux applications.

3. **Atomic Simulation Environment** (ASE) is a publicly available simulation environment with which atomic simulations can be set up, carried out, analyzed and visualized.

Extracting data In order to extract data of the selected projects using GitLab2PROV, an access token for the GitLab API must first be created. This token is then fed into GitLab2PROV together with the URL of the respective project or repository, whereupon the project is extracted in the desired format. In this case, the provenance graph was output as PROV-JSON. It should be noted that no uniform time zone is considered, but the time set in the project is adopted during extraction. For later analyses, it is therefore important to adjust the time format, since graph databases are only designed for the formats they specify for representing time.
During the design phase, it was also noticed that merge requests were not captured by GitLab2PROV. Because these are an essential part of communication in software projects, the tool was adapted after consultation with those responsible. In this way, a complete and correct extraction of the data sets could be achieved.

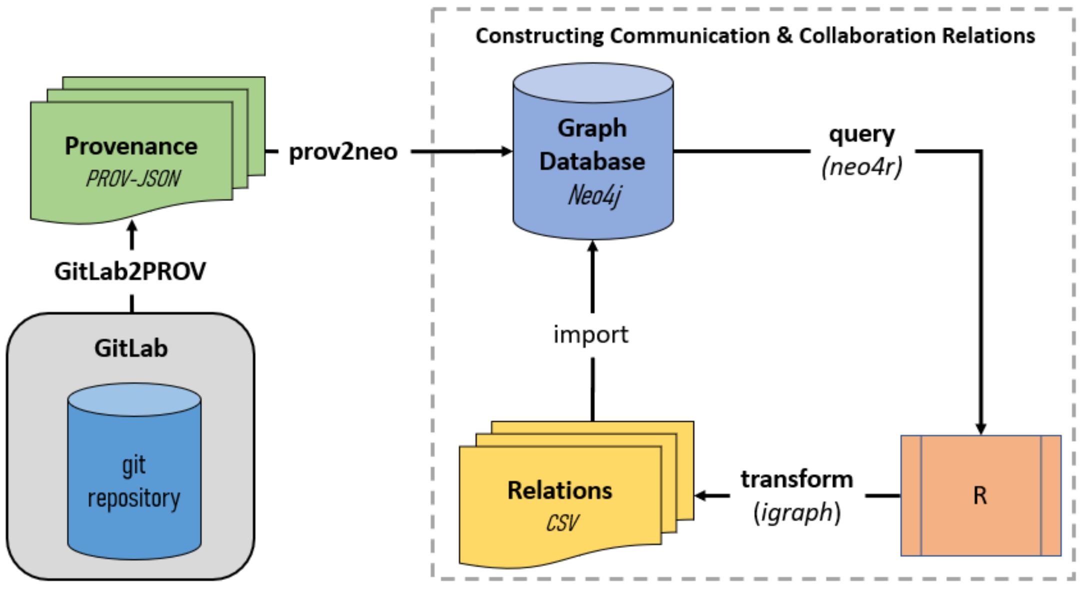

Figure 3.1: Pipeline from extracting data from GitLab to transforming and importing communication and collaboration relations into provenance graphs in Neo4j.

Importing data into Neo4j The next step is loading the resulting JSON file into Neo4j using the prov2neo tool. Like GitLab2PROV, prov2neo is a Python-based application, but it is used to load W3C provenance graphs into Neo4j. After running the tool, a new Neo4j database is created with a name chosen by the user, but it is not displayed in the graphical user interface since it is not known to the application. A new database with the exact same name must be created within the desktop application in order for it to have access to the underlying database containing the provenance graph.

3.1.2 Transformation Operations

Since the researched community smell patterns are defined exclusively by communication and collaboration relations, these connections must be established and added. As can be seen in Figure 3.2, this requires establishing collaboration links between the author of a file and the committers who worked on a derivative of the original file. In addition, the connections among different committers who have created derivatives of the same file must also be connected via the collaboration relation. Similarly, these connections should also be established for the communication network between the creator of a commit, issue, or merge request and annotators who comment on or contribute to the created resource. Only the Git commit models are used to create the collaboration network. For the communication network, the GitLab web resource models are utilized. In order to create the connections and later add them to the provenance graph, some information must first be extracted from the provenance graph and transformed.

Retrieving data via queries The resources needed to transform a collaboration relation can be extracted from the provenance graph by simple queries. Any user who creates the

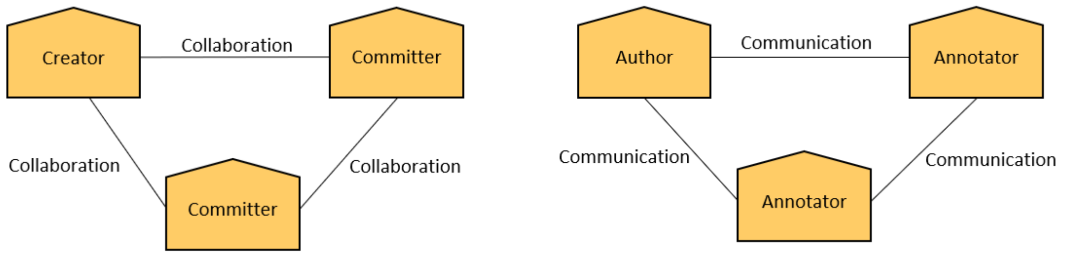

Figure 3.2: Desired connections between users to be created via transformations.

original or a derivative of a file is associated with it via the relation `:wasAssociatedWith`. Therefore, two users who are associated with a common file should also be associated with each other via a direct collaboration relation, as shown in figure 3.3. However, this becomes problematic for time-dependent analyses. Only activities contain the temporal provenance information, which is why they are needed for time-dependent queries. Since activities do not have a direct link to the source file through which the users are connected, the file versions must still be considered, since they point to the common source file. This results in the structure from the right half of figure 3.3, where two users are connected to the source file via an activity and the file version each. Since the shown combination of user, commit, file version and file occurs in all Git commit models, this approach can be applied to all operations of adding, modifying and deleting files in the repository.

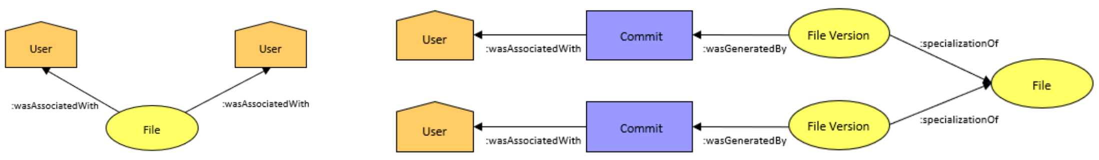

Figure 3.3: Left: Simple representation of collaboration of two users by file.
Right: Extended representation with activity including time component.

The procedure for the communication network is similar, but here only the GitLab web resource models are accessed. In order to access the timestamps of the activities here as well, the path shown in Figure 3.4 including activities must be queried. This path is included in Git commits, issues and merge requests, which is why a common query is sufficient for setting up the network. The entities and activities contained in the paths are captured by the abstracted representation `Creation` and `Annotation`, as otherwise, paths would have to be queried for each model of GitLab web resources. The R package *neo4r* is used to extract the provenance information of the specified paths via queries to the *Neo4j* API. This provenance information is then used to establish relations in the next step.

Transforming PROV data into relations After extracting the provenance information, adjacency matrices are created from it. Each existing connection between, for example, agent and activity is mapped in the matrix with a value of 1 or greater. Cells without

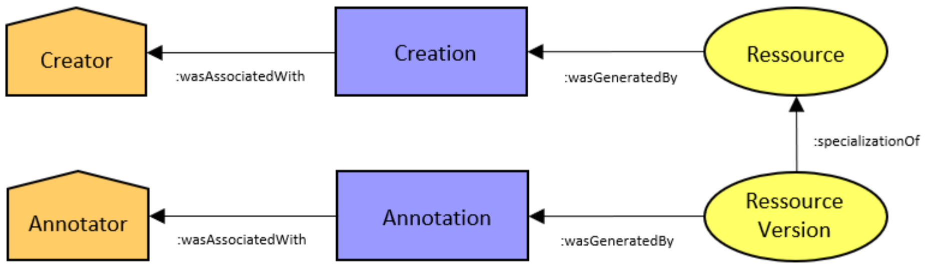

Figure 3.4: Path to be queried to create the communication network for all GitLab web resources.

adjacency, i.e. with a value of 0, are ignored to give the matrix dimensions with which they can be easily multiplied. As can be seen in Figure 3.5, all matrices are multiplied with each other step by step. The final matrix reflects which users have worked on which files. A matrix multiplication of this user-file matrix with its transposed matrix creates the user-user matrix and thus the precursor to the communication network.

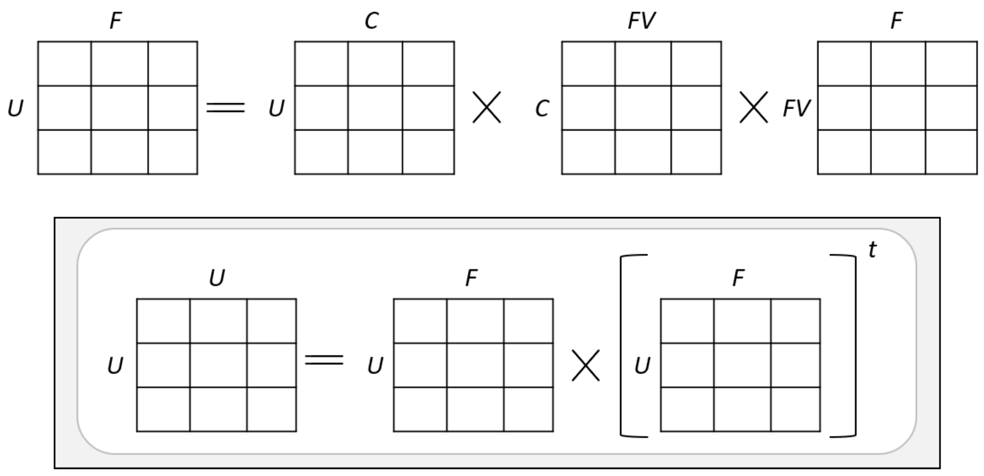

Figure 3.5: Matrix multiplications to set up the collaboration matrix. User (U), commits (C), file versions (FV) and files (F) adjacency matrices are needed.

Since the user-user matrix is square, it can be interpreted as an adjacency matrix. The adjacency matrix simultaneously maps the connections within the graph of the communication network. These connections can therefore be reimported into *Neo4j* as edges between the users involved in the provenance graph.

3.1.3 Using Queries to find Community Smells

For the community smell patterns *Lone Wolf* and *Organizational Silo*, the queries can simply be adopted as shown in figure 3.6. For this, only the communication and collaboration relations between the individual users need to be depicted. In the case of *Organizational Silo*, it must also be noted that the two collaborating developers are not in the same community. Therefore, the community membership is first determined by calculating the modularity and comparing whether both developers are part of the same community. When formulating the queries, care must be taken that the requested structures are returned as the result. With *Lone Wolf*, the dyads of the two collaborating users are considered, while with *Organizational Silo* only the collaborating user who does not communicate is considered. The queries according to both patterns are then sent via *neo4r* to the *Neo4j* API and the returns are viewed.

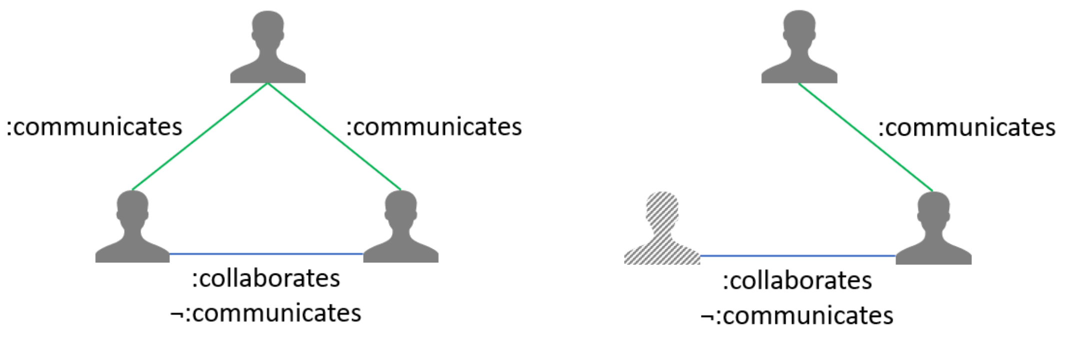

Figure 3.6: Operationalization of the Community Smell Patterns for the *Lone Wolf* (left) and *Operational Silo* (right).

The two developers, who connect two otherwise unrelated communities, form the *Black-cloud*. Due to the comparability of the representation in Figure 3.7 with the theory of *weak ties* presented by Granovetter [Granovetter, 1973], the analysis of the community-connecting edges seems to be appropriate. The Girvan-Newman algorithm is therefore used to determine whether there are edges connecting two otherwise separate communities. These identified edges should then provide information on whether recurring communication with longer periods of silence exists.

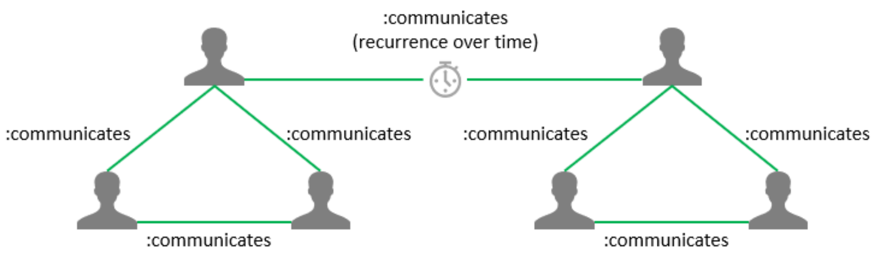

Figure 3.7: Structure of the communication graph to identify the *Black-cloud effect*.

For the last proposed pattern, *Bottleneck*, only users who are the only members of their community interacting with at least two other sub-communities are considered. If this restriction of the only user communicating externally is removed, a clique percolation with *k=3* can be applied. The users who are simultaneously contained in two cliques thus form the respective bottlenecks.

3.1.4 Comparison with Randomized Graphs

To compare the results and check whether these patterns arise only by chance, they are compared with randomised networks. For better comparability, the randomised networks must have the same dimensions as the initial network. For this purpose, derivatives of the original network are created in which the edges between the individual nodes are rearranged. Here, the degree of the nodes should be preserved and connections should be randomised so that they remain connected to the same types of node pairs. A connection that previously connected a user to a commit should also connect a user to a commit after randomisation.

3.2 Implementation

The following section explains the practical implementation of previously explained concepts. The focus is on the work steps after extraction. For a better understanding, commands used and exemplary graphics are shown to illustrate the procedure. Difficulties encountered and their solutions are described and then extensions for the developed implementation are presented. Afterwards, it is explained how user properties are extracted from identified patterns and how randomization is used for comparability with randomized networks.

3.2.1 Data Cleaning

After extracting the provenance graphs from the repository with *GitLab2PROV* and transferring the PROV JSON file to Neo4j via *prov2neo*, the next step is to look closely at the data and clean it. In Neo4j, cypher queries like

```
CALL db.schema.visualization()
```

can be carried out for this purpose, which display the schema of the provenance graph (Figure 3.8). This made possible, for example, to discover that merge requests with an earlier version of *GitLab2PROV* were not captured. In addition, it can be used to verify the structure of the entire graph. With the extension *APOC*, meta-information of the individual nodes, their attributes and the relations can be queried. Through these two methods, a meta-analysis of the graph was performed. It was noticed that in one case multiple timestamps were stored within an activity, which could lead to errors in future analyses, as timestamps were important components of the queries. This activity was removed together with all relations after closer examination, as only one user was associated with this activity and other attached nodes. Collaboration and communication graphs were not altered by this.

Although timestamps in nodes from different projects in GitLab did not contain a uniform format, they were unified in a later version of *GitLab2PROV*. Formatting the timestamps thus became obsolete.

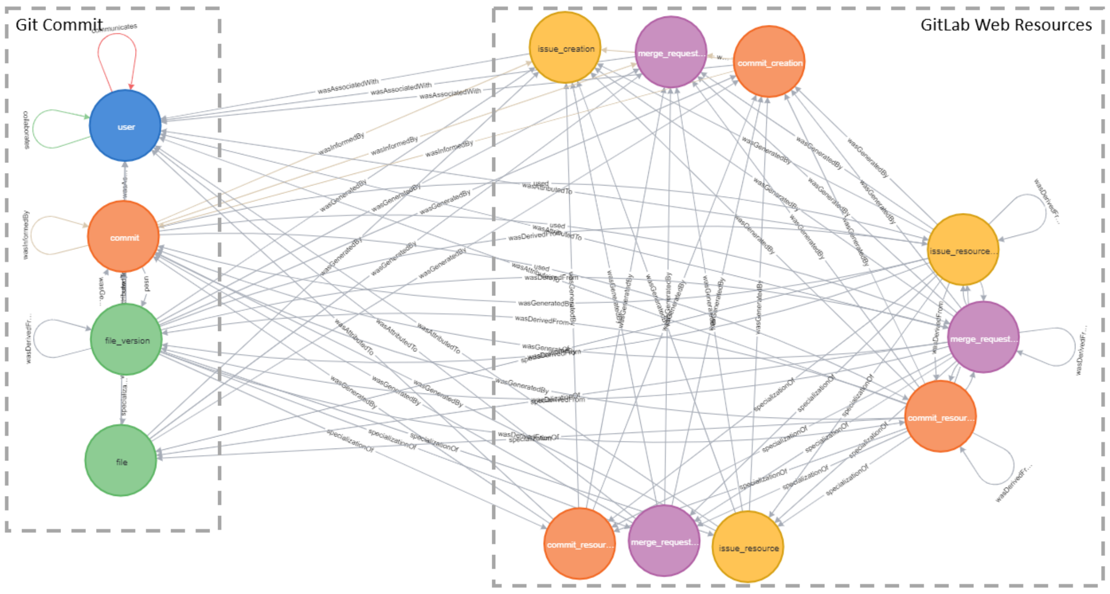

Figure 3.8: Scheme of the extracted provenance graph within *Neo4j*. Nodes are grouped according to their affiliation to Git commit or GitLab web resource model.

3.2.2 Folding and Transformation

Since the extracted provenance graphs do not contain connections of the types communication or collaboration, these must first be created via folding of individual graph components. Therefore, the first step was to analyze and extract the necessary components to perform this folding. Subsequently, the connections are created via matrix multiplications of the extracted components and returned to the original provenance graph.

Collaboration Extraction For collaboration, connections have been created where users have created new file versions of a file via commits. This is basically true for committers but also for authors, since every time a new file is created in Git, a primary file version of that file is also created directly. This enables to find the authors and user groups of committers within the network who have collaborated with each other (see figure 3.9). Since authors and committers are treated the same and both belong to the *user* type within the Provenance graph, they can be queried with the other previously mentioned components using the following query:

```
MATCH p=(u:user)-[:wasAssociatedWith]-(c:commit)-[:wasGeneratedBy]
        -(v:file_version)-[:specializationOf]-(f:file)
RETURN DISTINCT u.id, c.id, v.id, f.id
```

This applies to the Git commit (figure 2.7), file modification (figure 2.8) and file deletion (figure 2.9) models, due to a similar structure and because all requested elements and connections are included in each of the extracted Git provenance graphs. The filtering by edges visible in the query is solely for better visualization. A distinction of the edge types was not made, if possible, as Neo4j performs better when filtering only by node

types.
The result of this query via the neo4r to the API interface of Neo4j was a data frame with the respective IDs of the users, commits, files and file versions. For each possible connection, the entire path was returned, resulting in duplicates, which could be filtered out using the keyword *DISTINCT*.

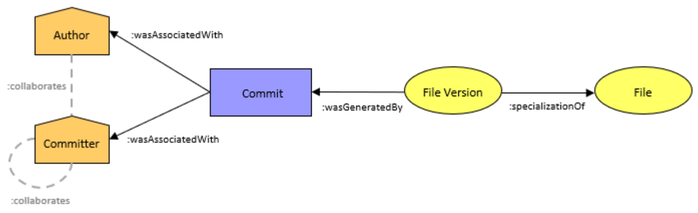

Figure 3.9: Required agents, activities and entities to establish collaboration relations between authors and committers. Established connections between users are represented by dotted lines.

Collaboration Folding After extraction, matrix multiplication is used to determine which user worked on which file. For this the connections between user and commit, commit and file version, file version and file are represented as matrices, by taking the respective two columns from the data frame and representing them as adjacency matrix. These adjacency matrices are reduced to the columns and rows in which there is at least one connection. This changes the dimension of the matrices to the actual contained number of for example users and commits. After this has been done for all connections, they can now be multiplied together step by step. Finally, this results in a user-file matrix that shows which user worked on which file. Figure 3.10 shows the described approach graphically.

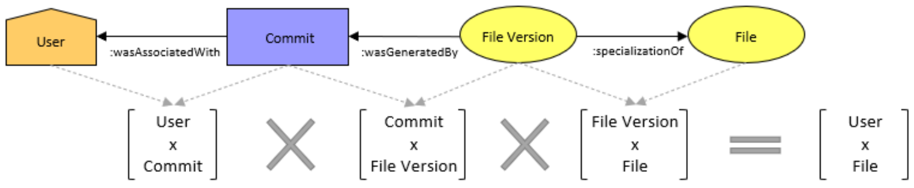

Figure 3.10: Matrix multiplications to calculate the desired user file matrix

In order to make visible which users have collaborated with each other via these files, this user file matrix must be multiplied with a transposed version of the user-file matrix in the next step. This results in a final matrix that does not distinguish between authors and committers, but still shows all their collaborations. To reduce the resulting matrix, the diagonal is set to 0 and the multiplicity of the respective edges is set to a maximum value of 1, since multiple edges provide no added value and only require increased computational effort.
Since no data can be transmitted directly to the Neo4j API via neo4r, the resulting matrix is transformed to a graph and the edges are exported as edge list to a CSV file. This CSV file can be read locally into Neo4j and a command is used to insert connections between collaborating users.

```
LOAD CSV WITH HEADERS FROM "file:///collaboration_edges.csv" AS row
MATCH (u1:user {id:row.from_id}), (u2:user {id:row.to_id})
CREATE (u1)-[:collaborates]->(u2)
```

Communication Extraction Similar to extracting components to generate connections for collaboration, the components are also queried for communication. However, it should be noted here that due to the structure of the provenance graphs, a distinction must be made between creators and annotators, even though they are from the common group of *users*.

```
MATCH p=(creator:user)-[:wasAssociatedWith]-(creation)<-
        [:wasGeneratedBy]-(resource)-[:specializationOf]-(version)-
        [:wasGeneratedBy]-(annotation)-[:wasAssociatedWith]-
        (annotator:user)
RETURN creator.id, creation.id, resource.id, version.id,
       annotation.id, annotator.id
```

As described before, the IDs of the individual actors, activities and entities are returned in the form of a data frame after performing a query via neo4r.

Communication Folding The distinction between creators and annotators changes the folding process compared to the folding regarding collaboration. As shown in Figure 3.11, a similar approach ignores the edges of the communication between annotators. In addition, it is likely that the dimensions of the final creator-annotator matrix do not correspond to those of a square matrix, making adequate mapping as a graph impossible without error.

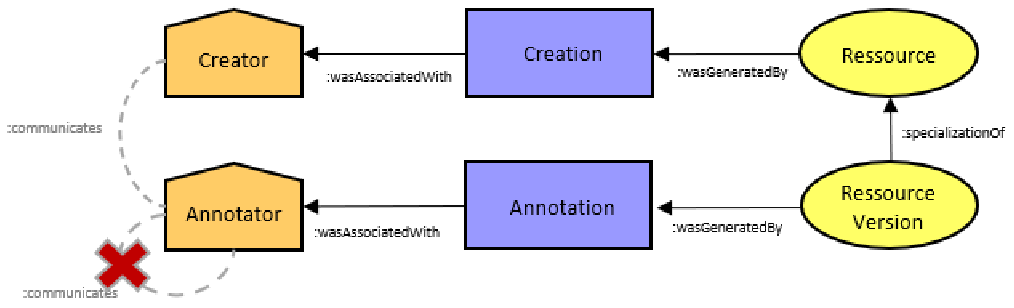

Figure 3.11: Flawed approach to the creation of the communication network. Connections between annotators are missing.

Therefore, the communication network is created in two parts and then merged (see Figure 3.12). The respective resources (commit resource, issue resource, merge request resource) are chosen as the common anchor point, since the structure of the GitLab web resource models is similar except for the naming and irrelevant components. The two parts of the communication network can be calculated via matrix multiplications of the adjacency matrices as described before. Afterwards, a union is performed by adding the

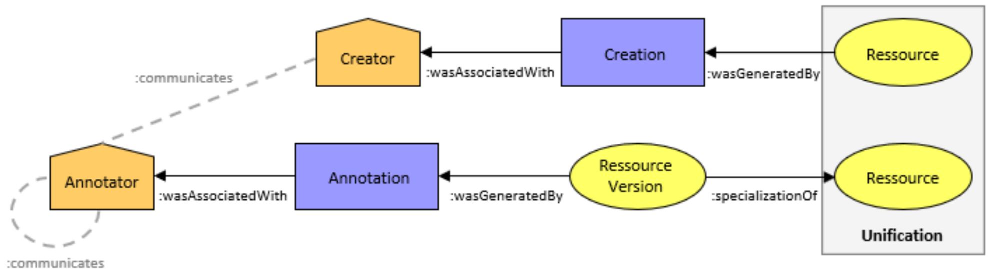

Figure 3.12: Part-wise calculation of the communication network with subsequent union. Established connections between users are represented by dotted lines.

columns of the resource creator matrix to the resource annotator matrix.
Because duplicates are very likely, they are removed and then the merged matrix is multiplied by a transposed version of this matrix to obtain an author annotator matrix. The final matrix is reduced in terms of diagonals and multiplicity of edges as in the collaboration matrix. Subsequently, the matrix is transformed into a graph and the respective communication edges are read into Neo4j via a CSV file.

```
LOAD CSV WITH HEADERS FROM "file:///communication_edges.csv" AS row
MATCH (u1:user {id:row.from_id}), (u2:user {id:row.to_id})
CREATE (u1)-[:communicates]->(u2)
```

As can be seen from the query and Figure 3.13, the new connections between users are fed into the graph with directions. This is an application-dependent requirement when creating new edges for Neo4j, but later queries regarding community smell patterns will query all edges regardless of their direction, so no further interference will result from this.

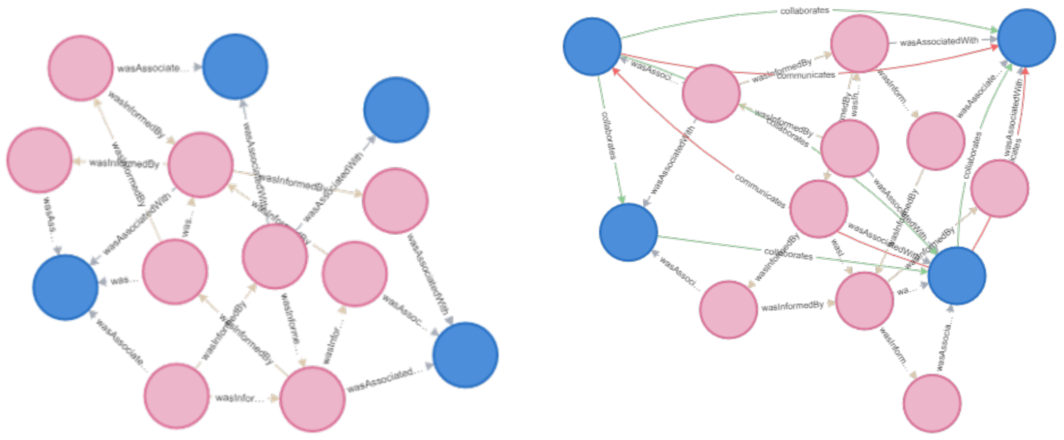

Figure 3.13: Cutout of the network regarding users (blue) and commits (pink) before and after importing relations. Communication shown in red, collaboration in green.

Query Execution On the graph, extended to include communication and collaboration connections, the Cypher queries can now be run to identify developers who fit into one of these patterns. The *Lone Wolf* pattern identifies dyads of users who collaborate but do not communicate with each other. The query identifies all distinct users, which may be at position *u* or position *u2* within the pattern. Therefore, the two IDs returned by the following query are checked again for duplicates and merged if applicable. Additionally, the number of dyads is retrieved using a query that returns the count of all found *Lone Wolf* patterns.

```
MATCH p=(u:user)-[:collaborates]-(u2:user), (u3:user)
WHERE (u)-[:communicates]-(u3)-[:communicates]-(u2)
AND NOT (u)-[:communicates]-(u2) RETURN DISTINCT u.id, u2.id
```

For the Organizational Silo pattern, the Louvain modularity must still be calculated before execution. Basically, Tamburri et al. point out that only the modularity has to be calculated, but no exact procedure is described [Tamburri et al., 2017]. Due to the high speed and availability within the Graph Data Science Library, Louvain modularity was chosen. It should also be mentioned that the Graph Data Science Library cannot be accessed via a query from an external location, such as neo4r. Therefore, each time a network is analyzed, it is necessary to start this calculation manually, which prevents complete automation.

```
MATCH p=(u:user)-[:collaborates]-(u2:user), (u3:user)
WHERE (u2)-[:communicates]-(u3)
AND NOT (u)-[:communicates]-(u2)
AND NOT (u)-[:communicates]-(u3)
AND NOT u.louvain = u3.louvain
RETURN DISTINCT u.id, u2.id
```

After returning the IDs of the users involved, a distinction must be made as to which users are being considered as *Organizational Silo*, in contrast to the *Lone Wolf* pattern. Here it is the user *u* that can be described as an *Organizational Silo* and should therefore be counted alone. User *u2* will be considered for further analysis later on. In addition, it is recorded how often the identified users could be found in the respective pattern.

Analyses without queries As mentioned in section 3.1.3, the *Black-cloud* pattern can be revealed by using clustering algorithms. Since the structure of the *Black Cloud* pattern consists of isolated sub-communities, which are only connected by a relation, the use of the Girvan-Newman algorithm seemed appropriate [Girvan and Newman, 2002]. Edges within the graph are gradually deleted and remaining connected components represent individual communities. Since there is no native implementation in Neo4j to perform the Girvan-Newman algorithm and an implementation based on a cypher query is cumbersome, an alternative was chosen. For the pattern we are looking for, only communication relations are important and the communication network has already been calculated in *R*. Therefore, with the help of the package *igraph* [Csárdi and Nepusz, 2006], the communities were calculated based on the edge betweenness. This uses is an implementation of the Girvan-Newman algorithm, in which the group membership is determined

according to modularity and the relations between communities that act as bridges. The group membership and the respective bridges should be used to identify users who represent the *black-cloud* effect between two otherwise unrelated communities.
For the identification of *Bottlenecks*, a similar approach as for *Black Cloud* was chosen. By accessing the already created communication network, the obvious solution was a *Clique Percolation* with the factor *k=3*. The formulation of this pattern established by Tamburri et al. was softened by the fact that there could not be only one user maintaining contacts with other sub-communities, but several. Thus, only one user from community A might communicate with community B, while another user from community A might interact exclusively with community C. Since there is no integration of the *clique percolation method* in Neo4j, this operation shall also be performed via *R*. Using the package *Clique Percolation* and the necessary dependencies, the graph is first converted into the format *qgraph*. From this unweighted graph, fully networked components with 3 nodes are to be analysed. With the command

```
clique_percolation_k3 <-
cpAlgorithm(communication_graph, k = 3, mode = "unweighted")
```

this analysis was triggered. After several attempts of running the analysis, despite using a small communication graph to be analyzed, the algorithm did not come to any result, whereupon this analysis was aborted.

3.2.3 Extensions

When analyzing networks, it has become standard practice to examine several time windows. Since the previous approach is based exclusively on the entire time period of the extracted data set, it considers several iterations during development and thus different cycles within the project, this standard approach is missing. Additionally, during the development it became evident that hardware can be a strong limiting factor, which is why a method was developed that also leads to success with low performance computers.

Time windows Proposed by Zeini et al. is the analysis of time windows with fixed size [Zeini et al., 2012]. These depend on the nature of the network and are calculated using the clique percolation method. Due to limited resources in the implementation of this approach, no result was obtained for the networks studied, so an alternative had to be found. Instead of fixed time windows, the focus was on data for the publication of new release versions. For this purpose, the two phases between the most recent releases of ASE and IGitt were selected in each case. Due to the lack of information about Freedesktop releases, time windows were selected that correspond to the use of an underlying framework on which Freedesktop is built. If the framework was updated, this was considered a fixed point in time when analyzing the data. The selected time frames for the 3 projects are therefore derived as follows:

IGitt	Freedesktop	ASE
22/01/2016 - 01/06/2017	29/03/2019 - 11/12/2019	16/12/2019 - 08/08/2020
01/06/2017 - 07/08/2017	11/12/2019 - 21/02/2020	08/08/2020 - 18/01/2021

Table 3.1: The researched time windows of considered open source projects.

Interval In the first condition of querying communication and collaboration networks, no distinction is made with respect to time windows or intervals, but the entire crawled dataset is considered. This makes it possible that users collaborated on files at a very early stage of the project and later became inactive. Users who joined the same project later and manipulate the same files are counted as collaborators of the now inactive users, even though they can never communicate with each other. This supports the formation of *Organizational Silo* and *Lone Wolf* patterns. To counteract this, an approach was tried in which only collaborations and communications within certain intervals were evaluated. For example, a collaboration is only counted if a developer has written a commit and another developer has worked on the same file within a predefined period of time. This also applies to communication about the creation of commits, issues and merge requests and their annotations. The specified time period was set to 14 days in this approach. Unfortunately, this approach could not be reconciled with batch processing introduced in the next paragraph, which was used to capture time windows and large data sets. An example query is as follows:

```
MATCH p=(u:user)-[]-(c:commit)-[:wasGeneratedBy]-(v:file_version)
       -[]-(f:file)-[]-(:file_version)-[:wasGeneratedBy]-(c2:commit)
WHERE datetime() + duration.inDays(c.‘prov:endTime‘, c2.‘prov:endTime‘)
      < datetime() + duration("P14D")
RETURN DISTINCT u.id, c.id, v.id, f.id
```

The query determines the time window between two commits and then compares it to a 14-day period. If the time window of the commits was larger than 14 days, it was ignored for further analysis.

Scalability When querying large data sets, calculation errors occurred due to limited hardware. These errors were related to the amount of memory available, which was limited to a total capacity of 8 gigabytes for this work. If this overflowed despite swapping and paging processes, either interruptions occurred in the computation or the program aborted with a segmentation fault, causing the entire R instance was interrupted. In this case, results stored in the buffer were also deleted. To overcome these challenges, a new approach had to be designed and implemented. The principle of batch processing was used to divide the data sets into smaller, more easily computable parts. Thereby, the individual data sets were analyzed in terms of their entirety and short time windows were selected based on the density and the total time period considered. These short time windows were then iteratively sampled and the communication and collaboration relations were determined individually by convolutions. Subsequently, these relations were cached and relations computed in the next batch were added piece wise. Thus, the entire dataset was computed incrementally and later merged so that it could be imported into Neo4j and queries could be executed.

3.2.4 Network Randomization

To determine whether the patterns found are purely random or truly due to organizational structure, the number of community smells identified is compared to community smells in randomized networks. The randomized networks are generated from the respective original provenance graphs. For this purpose, the connections between the individual elements are re-sorted from the provenance graphs. In a first experiment, these connections should be reordered by the *R* package *VertexSort* with a *degree preserving sorting* algorithm. Despite parallelized computation, the algorithm did not run to completion, so a custom adapted method had to be developed. Previously generated adjacency matrices describing the connections of the provenance graphs are reordered by randomly reordering the columns of the matrices.

```
userCommitRandom <-
    userCommit [,sample(1:ncol(userCommit), ncol(userCommit))]
```

This achieves a previously targeted degree preserving algorithm and extremely reduces the computational cost. Through the subsequent procedure, as exemplified in Figure 3.10, strong randomization is achieved in the calculation of the communication and collaboration relations for the randomized networks due to the multiple matrix multiplications. For an automated generation of the randomized networks and the subsequent retrieval of the community smells, the condition of *louvain* modularity had to be waived for the *Organizational Silo*. This was previously calculated using the *Graph Data Science Library* integrated in Neo4j. However, with *neo4r* as an interface, direct queries to compute modularity via the *Graph Data Science Library* cannot be executed. Thus skipping the modularity, 10 randomized networks could be automatically created for each network under study, from which the community smells were subsequently determined via previously described queries. From these randomized networks, all found community smells were recorded, and the average number was calculated in order to compare them with the identified patterns from the original graphs.

3.2.5 Developer Analysis

To identify key developers and to compare developers who are in smell patterns with developers who are not in any pattern, they were examined in terms of their activity characteristics. The behavior of the developers was recorded in terms of the number of their activities in the project via queries and these were tabulated in a data frame. The activities include commits, issue creations, merge requests and their annotations summarized. For the smell pattern developers, the number of smell patterns they are in is also counted. In addition, the average number of file versions and the average number of contributors who worked on the same file were queried for the files on which these developers worked. Subsequently, any correlations between the queried characteristics were determined via the package *corrgram* using a correlation matrix. The comparison of the developers in and outside of community smell patterns should be determined over mean value comparisons of the characteristics. For this purpose, the group of non smell users had to be reduced, since a large part did not write or publish a commit. Due to the different numbers, neither mean value comparisons nor other quantitative, statistical comparisons were possible. Therefore, the numbers of the affected developers and users were only compared superficially.

3.2.6 Network Analysis

In the last step, the studied communication networks were analyzed with respect to the metrics presented in section 2.6.2. The *R* packages *igraph* and *dplyr* were used to analyze the networks and normalize the values in each calculation for comparability of networks of different sizes. Loops within the networks were ignored, and all networks were interpreted as undirected, since creating the communication graph over the adjacency matrices produced a directed network.

4 Evaluation

In this section, the results of the analysis are presented. First, the topology of the examined graphs is briefly described and then the main part of the analysis regarding the identified community smells is described. After that, the results for the developer characterization are described. Finally, the results for the randomized networks and the network analysis of the considered projects follow.

4.1 Graph Properties

In terms of size, the projects for the following approach differ greatly, as the computational effort grows exponentially with a linear increase of nodes. Table 4.1 shows the numbers of the individual node types of the projects. The size of the projects increases by about 200,000 nodes each. Since a multi-relational network is present, the number

Node type	IGitt	Freedesktop	ASE
Users	49	317	650
Files	494	3.198	16.464
File Versions	1.978	19.936	110.423
Commits	417	5.580	18.243
Issues	147	1.178	859
Merge Requests	270	4.598	2.301
Annotations	11.310	173.494	238.310
Total #nodes	14.665	208.301	387.250
Total #edges	44.893	627.156	4.623.292

Table 4.1: Overview of the selected projects to be analyzed in this work. Data is portrayed at the time of extraction.

of edges exceeds the number of nodes many times over. ASE, in particular, is an overly dense network, which can be observed by the approximately 12-fold amount of edges.

4.2 Community Smells

After identifying the four different Community Smell patterns, the results for the analysis are very different. Due to problems that occurred during the analysis, some results could not be recorded.

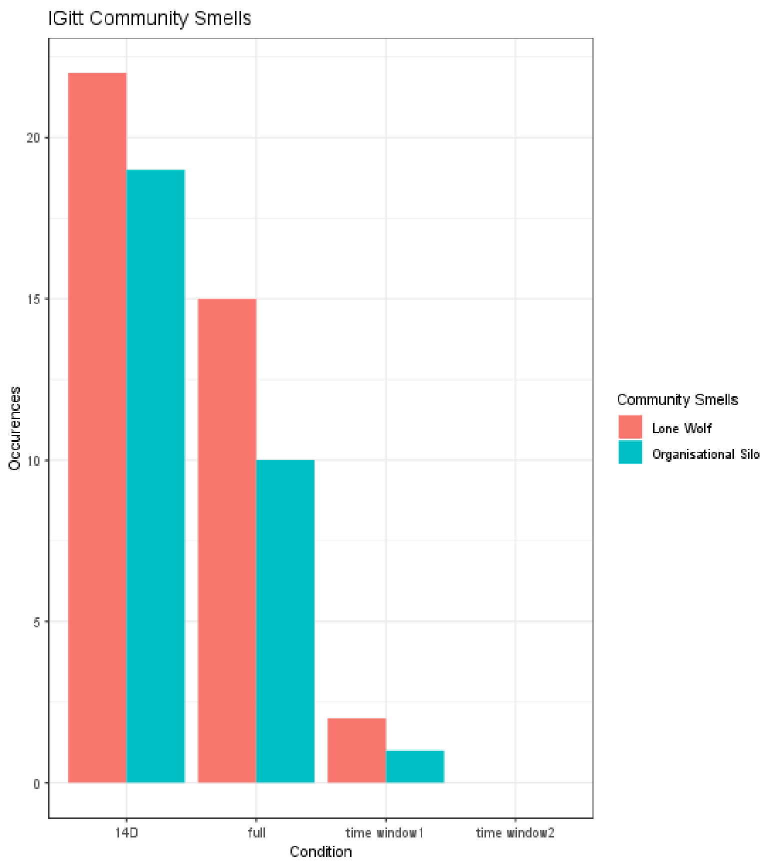

Figure 4.1: Identified community smell patterns for IGitt

IGitt In the IGitt project (Figure 4.1), most of the smells are found in the queried 14-day time window. For both the *Lone Wolf* and *Organizational Silo* patterns, the identified number exceeds that of the analysis over the entire project period. The distinct *Lone Wolves* (n=24) account for a share of about 49% within the interval. The rate of *Silos* (n=19) is lower, with a share of 38.9% for the same condition. Considering the whole network, the amount of *Wolves* (n=15) is 30.6% and that of *Silos* (n=10) is 20.4%. For the two time windows, the number of patterns found is significantly lower. In the first time window, only a total of 2 dyads of users are uncovered within a frame of 17 months (4.1%), which make up the *Lone Wolf* pattern, while for the *Organizational Silo* only a single developer could be identified (2%). This set of constellations is again undercut in the second time window and not a single smell pattern could be identified by queries. The time window here is much smaller and covers only a little more than two full months. For the *Black Cloud* pattern, not even a single occurrence could be detected for any of the conditions. This also results for the *Bottleneck* pattern, since the algorithm did not finish during the analysis.

ASE The project ASE 4.2 is characterized by a much higher number of nodes and a denser network of relations. Due to these characteristics, the project could neither be analyzed with regard to the entire graph nor the 14-day interval. Due to this limitation,

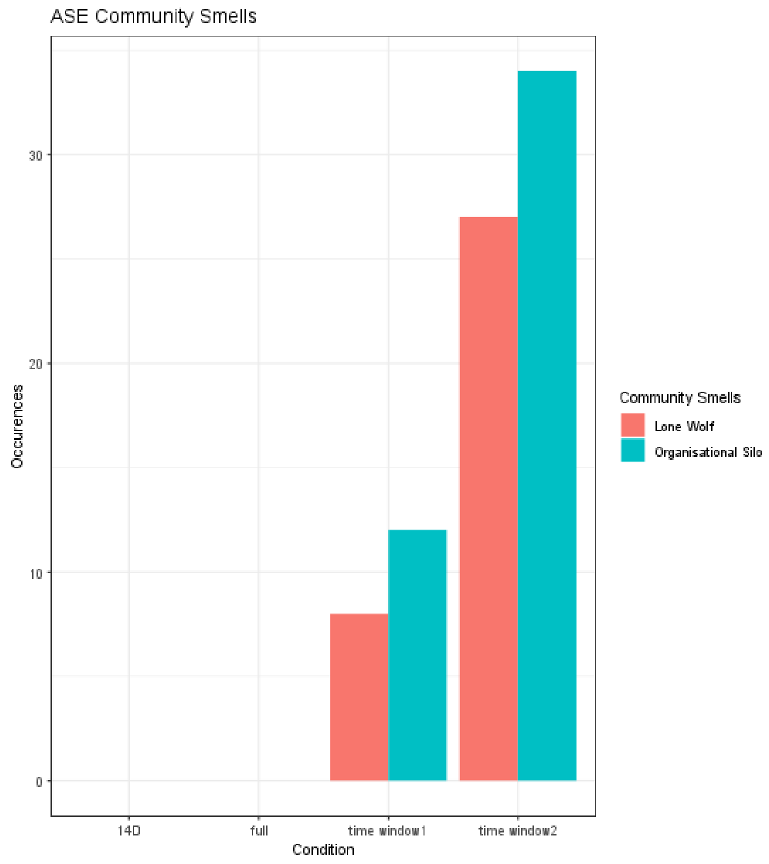

Figure 4.2: Identified community smell patterns for ASE

no instances of *Lone Wolf* and *Black Cloud* patterns could be found. In the first time window, a total of 8 dyads and thus a total of 8 *Lone Wolves* are found in a period of 8 months. Here, one developer in particular is conspicuous by its activities in a total of 36 instances of this pattern. For the *Organizational Silo*, the number of identified developers is slightly higher at 12. In the second time window, which covers just over 5 months, the numbers for the two patterns are significantly higher. The dyad of *Lone Wolves* reaches a value of 27, describing 34 developers. The same number of developers is also found for the *Silos*, although the respective developers uncovered are not congruent for both patterns. Also included in this time window is the previously mentioned single developer, which is represented by 36 instances in the first time window and 8 instances of the *Lone Wolf* pattern in the second time window. As with the IGitt project, the *Black Cloud* pattern has not been identified in any of the conditions examined. This is also true for the *Bottleneck* Pattern due to the previously mentioned reason.

Even though the analysis for the Freedesktop project was not successful in terms of community smells, it did yield results in terms of project structure. Freedesktop is controlled by a few administrators who are the only ones in the project with the ability to accept merge requests. They accept requests from other developers only at certain times in batches. This includes especially commits and comments, which are written automatically by bots. Bots appear both in the community smell patterns found and outside them. Within the community smell patterns, they take the role of the communicating developer in most cases, and in only one instance is a bot itself considered a *Organizational Silo*.

4.3 Developer Characterization

Checking whether certain behavioral patterns of developers correspond to the number of smell patterns was achieved by calculating correlations. Figure 4.4 shows an example of the correlogram for the complete IGitt project with respect to the two smells examined. Based on the rows and columns with the smells, it is visible that the effect size does not exceed .3 and thus the lower limit for a small effect is not reached. Since the values for all other conditions and also for the ASE project do not reach this limit, the developer characterizations for these cases are not shown.

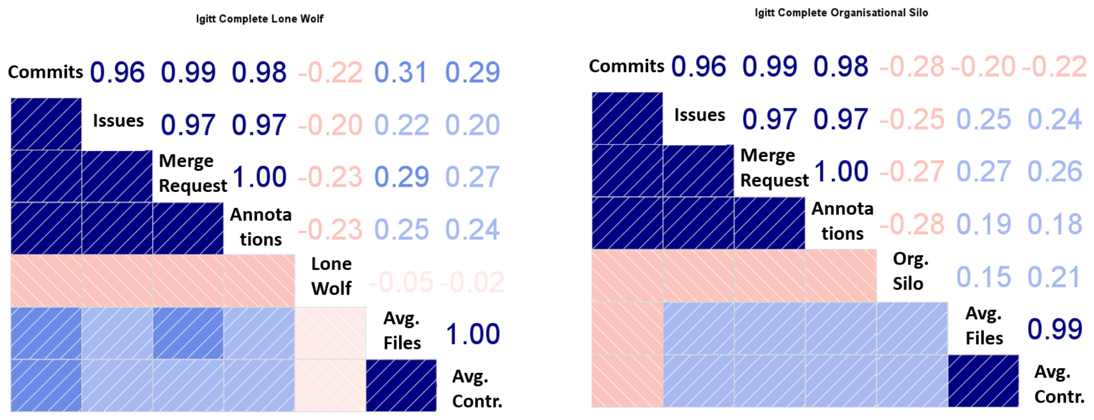

Figure 4.3: Correlation diagrams of the full IGitt project for *Lone Wolf* and *Organizational Silo* patterns.

Regarding the comparison between smell developers and developers who do not appear in any pattern, the queried values are comparable. Figure 4.4 shows the activities of the non-smell users. The average values for commits (C) are 890.34, for issues (I) 30.87, merge requests (M) 6.75 and annotations (A) 17.91. For the smell user (figure 4.5) the values for commits are 370.48, issues 13.5, merge requests 4.23 and annotations 7.5. The number of activities for the smell developers is thus on average much lower than the activities of the non-smell developers. The discrepancy is only smaller for issues with a factor of 1.44 than for the other activities, which differ by a factor of more than two.

C	I	M	A
170	10	97	3.815
55	39	32	2.222
7	4	11	756
6	1	3	257
4	0	0	45
3	0	0	8
1	0	0	19
1	0	0	1

Figure 4.4: IGitt non-smell users

C	I	M	A
157	40	78	3.498
9	0	0	82
9	0	0	147
5	1	5	223
3	2	3	147
3	11	3	289
3	0	0	22
3	1	7	337
2	0	0	6
2	0	0	27
2	0	0	14
2	0	0	12
1	0	1	14

Figure 4.5: IGitt smell users

4.4 Randomized Networks

Retrieving community smells from randomized networks compared to the authentic network consistently yields significantly higher results. For both the *Lone Wolf* and *Organizational Silo* patterns, the mean and total scores for networks from randomly generated provenance graphs are significantly higher for the IGitt project (see table 4.2). For *Organizational Silo*, the number of smell-tainted users is also 50% higher for the randomized network than for the original network. A comparable distribution of community smells is also true for the other conditions studied for IGitt.

	#User	Mean	Sum	Rand. #User	Rand. Mean	Rand. Sum
Lone Wolf	15	1.73	26	15	4.74	67
Org. Silo	10	1.8	18	15	4.93	74

Table 4.2: Comparison of community smells from authentic and randomized full IGitt networks.

For the project ASE (table 4.3) the differences in the values are even more significant. The number of users in smell patterns reaches a 637.5% higher value for the randomized graphs (n=51) than for the initial graph (n=8) for the *Lone Wolf* pattern. The situation is similar for the *Organizational Silo* with an increase of 608% (n=73) compared to the original value (n=12). Due to this high number of users, the total number of community smells for both patterns is also much higher for the randomized graphs. Especially *Organizational Silo* reaches a factor of over 1100% compared to the original graph and is thus significant. Striking is the higher mean value for *Lone Wolf* from the authentic graph (m=5.75) compared to the randomized graph (m=3.955). This is due to an outlier value that far exceeds the mean value in this case. For the second time window of the project ASE, the considered values are comparable.

	#User	Mean	Sum	Rand. #User	Rand. Mean	Rand. Sum
Lone Wolf	8	5.75	46	51	3.955	201
Org. Silo	12	1.92	23	73	3.66	267

Table 4.3: Comparison of community smells from authentic and randomized first time window of ASE networks.

4.5 Network Analysis

Through social network analysis tools, the results shown in Table 4.4 were retrieved. The values of the centralities refer to normalized computed graphs. For IGitt, there are strong changes in terms of centralities and density compared to time windows. While these score low, the diameter of the network (d=4) is twice as high as in the time windows studied. For the time windows, a shift in density and closeness to higher values is evident. Meanwhile, the values for degree and betweenness centrality decrease and converge to the level of the complete investigated network. Values for the ASE project remain consistently low except for degree centrality, which is in a medium range. All values are constant over both time windows.

Project	Condition	Density	Degree	Betweenness	Closeness	Diameter
IGitt	Full	0.195	0.295	0.027	0.077	4
	Time W 1	0.509	0.600	0.325	0.692	2
	Time W 2	0.746	0.311	0.090	0.816	2
ASE	Time W 1	0.059	0.439	0.072	0.029	6
	Time W 2	0.075	0.424	0.035	0.031	6

Table 4.4: Computed centralities for communication networks of researched projects and conditions.

5 Discussion

In the following sections, the results described above are discussed. Thereby, the identified community smell patterns are first compared across the investigated conditions. To explain the occurrence of different numbers of patterns, the centrality measures suggested by literature and the structure of the project itself as well as the activities of individual developers are considered and put into context. The following section deals with the results of developer characterization and the realization that bots also manifest themselves in community smell patterns. The last section deals with the observed performance of the solution presented here and how problems can be solved efficiently when working with provenance graphs and graph databases.

5.1 Community Smell Patterns

Based on the present results, only limited statements can be made about all community smell patterns studied. Here, the amount of occurring smell patterns when querying a 14-day interval in the project IGitt stands out. Although this method was originally intended to reduce the number of *Lone Wolf* and *Organizational Silo* patterns, the number increases compared to the entire observation of a graph. When considering community smells following the template of Tamburri et al., there is generally no distinction between whether developers also communicate to that exact collaboration when they collaborate [Tamburri et al., 2017]. With their formalization, they assume that all collaboration and communication occurs with respect to the same resource. This can be explained by sub-communities that are more likely to work together as a team and therefore treat the same subjects. However, in the field of OSS, it may increasingly happen that developers participate in leaps and bounds in files or project communications that do not belong to their original sub-community, which additionally might have a negative effect on software quality [Foucault et al., 2015]. So it is possible that developers work together on a file and communicate about something entirely unrelated, resulting in many smell patterns not being found. Especially over the consideration of longer periods of time, the probability of finding smell patterns is reduced, since such processes may occur more frequently over time. It has to be considered critically, which time periods should be analyzed and if longer time periods impede the detection of actually existing smells. For this, further studies need to be conducted and the definition of collaboration and communication needs to be narrowed down.
Also striking is the vanishingly small number of smell patterns within the examined time windows of IGitt compared to the rest of the graph. Considering the centrality measures, it can be seen here that especially density and closeness centrality have a strong influence on the formation of community smell patterns, as hypothesized by Almarimi et al. [Almarimi et al., 2020a]. Based on the structure of all patterns studied, it can be said that formalization finds especially missing communication relations. Therefore, when there is a high density and closeness in the communication network,

it is not surprising that the number of identified patterns is lower than in networks with low density. Here, it is also worth mentioning the low diameter, indicating that the short connections provide a fast flow of information that seems to counteract community smell patterns. While this low number of found patterns is explained to be fundamentally positive, as community smells create increased social debt [Tamburri et al., 2015], a low amount of found patterns could be critical for project success. Densely interconnected networks therefore require little effort in the processing of software or project management errors, but can lose out on project success due to the lack of centralization [Crowston and Howison, 2005]. More research should therefore be done to determine the proportion of community smells within a project group rather than the proportion of community smells in the project as a whole. Due to the structure of the projects studied, where the communication network consisted of only one community and isolated nodes, such a distinction could not be made. The fact that there is only one community suggests how decentralized both projects are organized. This is also reflected in the low values for degree centrality. Therefore, when looking at projects, the context of the project must also be considered. Is the project organized institutionally? Does the project have strict guidelines regarding collaboration and communication? Does the project have an unofficial, i.e. recreational, background? A unified analysis of community smells without considering the project background can therefore lead to misinterpretable results.
Also, the attachment of the community smells to individual developers or dyads of developers is to be criticized. During the analysis it was noticed that some maintainers commented or annotated almost every issue or merge request. This approach makes developers feel noticed, and interactions of this kind help to bind new and old developers to a project [Dominic et al., 2020]. The resulting communication links greatly increase the number of identified patterns according to Tamburri et al. [Tamburri et al., 2017]. Thus, the formulation of community smell patterns should be reconsidered in the sense that collaborating developers must be considered independently of a communicating user. If a pattern exists in the network where two users collaborate and do not communicate with each other or externally, it will not be detected by the current formalized community smell patterns.

For the different distribution of the community smell patterns *Lone Wolf* and *Organizational Silo*, the activity of some developers can be cited. Due to few active maintainers in the project IGitt who commented or annotated issues and merge request, communication relations to individual contributing developers were established. Because communicative maintainers communicated with almost all active developers, a dense network of communication relations could be created. This primarily reduces the occurrence of the *Organizational Silo* pattern, since that pattern is conditioned by only a single communication relation. In contrast, this approach supports the emergence of *Lone Wolf* patterns, since the collaborating developers always have at least one communication relation to the outside but not necessarily to the collaboration partner due to the activity of the maintainers. Thus, the role of maintainers here can reinforce some community smell patterns and reduce other smell patterns. With respect to the formalization of community smell patterns, therefore, central, communicative developers must be given special consideration.

The organization of the projects had a demonstrable influence on the number of community smell patterns found. Compared to the randomized networks, not only is the number of smell-afflicted users much lower, but also the total number of patterns, suggesting that the organization, even if not professional and only done through the members of the project itself [Crowston et al., 2007], has a significant impact on the emergence of community smell patterns.

5.2 Developers

When examining the developers with community smells, no correlation was found between the number of their respective activities and the occurrence of smell patterns. Nor do they differ significantly from developers who have authored at least one commit and do not find themselves in any pattern. Therefore, from the data obtained, no correlations can be found between communication and collaboration behavior and participation in smell patterns.

Strongly noticeable when looking more closely at users in and outside of community smell patterns was the participation of bots within the project. Bots are supposed to support the projects in interacting with new users, or they take over maintenance tasks, such as updating files from other projects. In some cases, bots should remind users to document their work. Due to these interactions, communication and collaboration relations between authentic developers and bots emerge, which are perceived as community smells. In several cases, it became apparent during the analysis that bots were found in both *Lone Wolf* and *Organizational Silo* patterns. Considering the previously described problem with communicative users preventing community smells, automated responses from bots could be responsible for reducing the number of negatively afflicted patterns. Depending on the characteristics of the bots, both increases and decreases in identified patterns are possible. In the analyzed networks, the number of bots that found themselves within a community smell pattern but were not themselves identified as *Lone Wolf* or *Silo* was significantly higher. This is related to the type of bots used. While bots, with the task of project communication, maintained many communication relations with the distinct developers, maintenance bots were only responsible for a small part of files that were updated at times. The bots considered in the Freedesktop project, on the other hand, were more feature-rich and could review commits from users and provide feedback, so revisiting this or similar projects has further insight into the impact sophisticated bots have on the number of patterns. As the number of bots increases in all social networks, including developer networks (Ferrara et al., 2016), and they are not considered in automated community smell detection methods [Almarimi et al., 2020b][Palomba and Tamburri, 2021], the results may be heavily skewed by bots. Therefore, it seems appropriate to filter bots when extracting network data. Filtering bots from the extracted network beforehand shrinks the size of the dataset, which can lead to better performance.

5.3 Performance

After retrospectively reviewing the performance of the developed solution, some challenges became obvious. These will now be critically reviewed and discussed with optimization options.

Hardware limitations Regarding the main limitation of memory, which has influenced this work especially in terms of time spent and restructuring of large parts of the code, the simplest and obvious solution would be to increase the available hardware if possible. However, since it can be assumed that the fixed implementation of this solution within an organization comes with a less limited amount of memory, the existing implementation including batch processing can suffice. Overall, batch processing significantly affected the queries and calculations in terms of time. While previous queries within the graph of the ASE project sometimes resulted in waiting times of several minutes, batch processing allowed smaller calculation blocks to be processed in seconds. Cumulatively, the query and calculation time of the batches with the consecutive merging of the individual subnetworks fell below the necessary computation time for an extensive query in the project. This change was also present in smaller projects, however, not to the same extent as in larger projects.

Cartesian products Since the computation time scales exponentially with increasing graph size, the graph to be examined should be considered in more detail in advance. Although the solution designed here has been adapted and tailored to the Neo4j graph database, there are critical components thereby. In particular, due to the structure of the formalized community smell patterns and the resulting queries, Cartesian products occur in the results of many queries. Cartesian products [Vizing, 1963] are created by queries that connect two or more disconnected patterns. With a higher number of disconnected patterns within a query, a Cartesian product is created over all parts. This creates large amounts of data that slow down the processing of the query. For this reason, queries with disconnected patterns should be avoided at best. Cartesian products negate a major advantage of the Neo4j graph database, namely the reduction of complexity due to differentiation in terms of labels and properties but also the restriction of the search space due to chosen conditions.

Alternative graph databases To speed up queries of large graphs, it is therefore debated whether the query structure of the community smells can be adapted. As suggested by Vicknair et al., orienting to a different graph database may also be beneficial if the topology and characteristics of the graph under investigation allow it [Vicknair et al., 2010]. Since in community smells patterns with different numbers of users are queried and the speed depends strongly on the structure of a graph, alternative graph databases such as the relational graph databases DEX/Sparksee or OrientDB could help. In case of a change of the database, a change regarding the whole query structure can be considered here as well. In this solution, filtering is done by nodes and their labels and properties, whereas in other graph databases with one list containing edges of all types, the speed of filtering by edges and their inherent information is significantly increased [Hölsch et al., 2017].

Reducing search space Another way to improve performance is a counterintuitive approach where users not involved in smell patterns are excluded in advance. In this case, however, greater care must be taken to ensure that users who, according to the formulation, do not constitute a negative pattern in the project, but who nevertheless passively participate in such patterns, continue to be taken into account. Exemplary the communicating developers are to be called, which are present in the patterns *Lone Wolf* and *Organizational Silo*. For *Black Cloud* and *Bottleneck*, isolated nodes in the communicating network are not of interest, as they are not counted as an independent sub-community and can therefore be excluded. For later comparisons between smell developers and non-smell developers or for social network analyses, a copy of the original graph should nevertheless be retained.

6 Conclusion

This work aimed to design and implement a solution for analyzing negatively afflicted sub-graph patterns, so-called community smell patterns, in provenance graphs of OSS developer communities. By integrating the two external tools *GitLab2PROV* and *prov2neo* in combination with an *R* interface and the graph database Neo4j, the targeted goal of identifying community smell patterns in GitLab projects could be achieved. After facing several challenges, the approach was optimized especially for devices with limited hardware via batch processing. Additionally, options were created to make the developers and the network analyzable with respect to their network dimensions.

Community smell formalization By analyzing three OSS projects and finding serious differences between the individual test conditions and the projects with their individual developer types, it was shown that the formalization of community smell patterns according to the current proposal of Tamburri et al. (2017) could be extended [Tamburri et al., 2017]. The results are strongly influenced by individual above-average communicative participants, leading to misinterpretable insights after an analysis and therefore influencing project management actions based on it. Bots, which interact with developers and project files in an automated way, are also not considered in current analyses of community smell patterns. Due to their increasing numbers, these must be filtered or treated separately in future research. Additionally, project structure of the considered communities must be evaluated in any analysis. While the centrality measures closeness, degree, and betweenness centrality provide a good indication of the project as a whole, for an effective look at community smell patterns and where they occur, individual sub-communities within a project should be examined. Especially larger projects, which fragment in the context of division of labor and specialization, can be corrected in terms of their management. As smell patterns occur, they are capable of introducing countermeasures in order to avoid negative consequences due to social debt.

Automated community smell detection Due to approaches, which use machine learning to find community smells with a hit rate of 77% [Palomba and Tamburri, 2021] and 89% [Almarimi et al., 2020b], respectively, the implementation for finding community smells seems obsolete at first glance. However, since both approaches consider the entire dataset in the analysis, they are more computationally expensive than the solution presented here, which is tailored exclusively to the required data, which is one of the main reasons in using provenance graphs [Huynh et al., 2018].
The implementation presented is therefore intended to be a guideline of the challenges that can arise when working with provenance graphs and graph databases and how these can be decomplicated with sometimes simple modifications. Furthermore, the approach shown demonstrates the possible courses of action that arise in the context of working with provenance graphs.

Provenance Graphs With the increasing complexity of software development and software development processes, these must be supported with various tools to enable a successful project. In the context of this work, it was prototypically shown how provenance graphs from repositories of version control systems can be used by developer communities in OSS to find patterns with potentially negative consequences. This takes up the original idea of Schreiber & de Boer to be able to map software processes via provenance graphs from GitLab and to be able to analyze and optimize these processes [Schreiber and de Boer, 2020]. Due to the novelty of the tool used to extract provenance graphs from this platform, the current research is limited to the analysis of a software project in the context of the Corona Crisis [Sonnekalb et al., 2020][Schreiber, 2020] and is extended by this work. Through the approach, it was shown how mathematical operations can be used to generate and examine relation folds between developers in OSS communities. Despite challenges in the analysis, it was possible to show how the workflow can proceed from extracting a provenance graph from GitLab, importing it into the Neo4j graph database, and then analyzing the network with respect to the community smells framework of Tamburri et al. [Tamburri et al., 2017]. This demonstrates the versatility with which provenance graphs are used in software development. In contrast to approaches that require analyzing the entire software projects with respect to community smells, graphs reduced to provenance data are sufficient to serve this purpose.
Particularly in the context of network randomization, the power of individual components and their respective components in provenance graphs becomes apparent. Simple swaps of relations in adjacency matrices could avoid new sorts of edges via complex algorithms. Nevertheless, due to the structure of the provenance graphs and by multiplying the re-sorted adjacency matrices, a very high degree of randomization could be achieved. This shows another form of versatility of provenance graphs and may be useful for future analysis and research in software development.

Further implications Apart from the critique of the formalization of community smell patterns and the application areas pointed out, this work gives implications which tools can be used for the analysis of provenance graphs. Additionally, the challenges encountered and the resulting approaches to solving them can provide guidance in the development of new analysis tools in the face of limited hardware in the field of software development.

Bibliography

[Almarimi et al., 2020a] Almarimi, N., Ouni, A., Chouchen, M., Saidani, I., and Mkaouer, M. W. (2020a). On the detection of community smells using genetic programming-based ensemble classifier chain. In *Proceedings of the 15th International Conference on Global Software Engineering*, ICGSE '20, pages 43–54. Association for Computing Machinery.

[Almarimi et al., 2020b] Almarimi, N., Ouni, A., and Mkaouer, M. W. (2020b). Learning to detect community smells in open source software projects. *Knowledge-Based Systems*, 204:106201.

[Antwerp, 2010] Antwerp, M. V. (2010). The importance of social network structure in the open source software developer community. In *In The 43rd Hawaii International Conference on System Sciences (HICSS-43*.

[Bernardi et al., 2018] Bernardi, M. L., Canfora, G., Di Lucca, G. A., Di Penta, M., and Distante, D. (2018). The relation between developers' communication and fix-inducing changes: An empirical study. *Journal of Systems and Software*, 140:111–125.

[Bretthauer, 2001] Bretthauer, D. (2001). Open source software: A history. *Published Works*.

[Chacon and Straub, 2014] Chacon, S. and Straub, B. (2014). *Pro Git*. Apress, 2nd ed. edition edition.

[Conway, 1968] Conway, M. (1968). How do committees invent?

[Crowston and Howison, 2005] Crowston, K. and Howison, J. (2005). The social structure of free and open source software development. *First Monday*.

[Crowston and Howison, 2006] Crowston, K. and Howison, J. (2006). Hierarchy and centralization in free and open source software team communications. *Knowledge, Technology & Policy*, 18(4):65–85.

[Crowston et al., 2007] Crowston, K., Li, Q., Wei, K., Eseryel, U. Y., and Howison, J. (2007). Self-organization of teams for free/libre open source software development. *Information and Software Technology*, 49(6):564–575.

[Csárdi and Nepusz, 2006] Csárdi, G. and Nepusz, T. (2006). The igraph software package for complex network research. *undefined*.

[Cunningham, 1992] Cunningham, W. (1992). The WyCash portfolio management system. *OOPSLA '92*.

[Cusick and Prasad, 2006] Cusick, J. and Prasad, A. (2006). A practical management and engineering approach to offshore collaboration. *IEEE Software*, 74:201–269.

[De Nies et al., 2013] De Nies, T., Magliacane, S., Verborgh, R., Coppens, S., Groth, P. T., Mannens, E., and Van de Walle, R. (2013). Git2prov: Exposing version control system content as w3c prov. In *International Semantic Web Conference (Posters & Demos)*, pages 125–128.

[Dominic et al., 2020] Dominic, J., Houser, J., Steinmacher, I., Ritter, C., and Rodeghero, P. (2020). Conversational bot for newcomers onboarding to open source projects. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, pages 46–50. Association for Computing Machinery.

[dos Santos et al., 2011] dos Santos, T. A., de Araujo, R. M., and Magdaleno, A. M. (2011). Bringing out collaboration in software development social networks. In *Proceedings of the 12th International Conference on Product Focused Software Development and Process Improvement - Profes '11*, pages 18–21. ACM Press.

[Ehrlich and Cataldo, 2012] Ehrlich, K. and Cataldo, M. (2012). All-for-one and one-for-all? a multi-level analysis of communication patterns and individual performance in geographically distributed software development. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 945–954.

[Fay, 2019] Fay, C. (2019). neo4r.

[Feigenbaum et al., 2012] Feigenbaum, G., Reist, I., and Reist, I. J. (2012). *Provenance: An alternate history of art.* Getty Publications.

[Feller and Fitzgerald, 2001] Feller, J. and Fitzgerald, B. (2001). *Understanding Open Source Software Development.* AddisonWesley Professional, 1. edition edition.

[Fitzgerald, 2006] Fitzgerald, B. (2006). The transformation of open source software. *MIS Quarterly*, 30(3):587–598. Publisher: Management Information Systems Research Center, University of Minnesota.

[Foucault et al., 2015] Foucault, M., Palyart, M., Blanc, X., Murphy, G. C., and Falleri, J.-R. (2015). Impact of developer turnover on quality in open-source software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 829–841. ACM.

[Fowler et al., 1999] Fowler, M., Beck, K., Brant, J., and Opdyke, W. (1999). *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1. edition edition.

[Freeman, 1977] Freeman, L. (1977). A set of measures of centrality based on betweenness. *Sociometry*, 40:35–41.

[Freeman, 1978] Freeman, L. C. (1978). Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215–239.

[Frisendal, 2016] Frisendal, T. (2016). Graph data modeling for NoSQL and SQL – technics publications.

[Girvan and Newman, 2002] Girvan, M. and Newman, M. E. J. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826.

[Granovetter, 1973] Granovetter, M. S. (1973). The strength of weak ties. *American journal of sociology*, 78(6):1360–1380.

[Hars and Ou, 2002] Hars, A. and Ou, S. (2002). Working for free? motivations for participating in open-source projects. *International Journal of Electronic Commerce*, 6(3):25–39. Publisher: Routledge _eprint: https://doi.org/10.1080/10864415.2002.11044241.

[Hölsch et al., 2017] Hölsch, J., Schmidt, T., and Grossniklaus, M. (2017). On the performance of analytical and pattern matching graph queries in neo4j and a relational database. In *EDBT/ICDT 2017 Joint Conference: 6th International Workshop on Querying Graph Structured Data (GraphQ)*.

[Huynh et al., 2018] Huynh, T. D., Ebden, M., Fischer, J., Roberts, S., and Moreau, L. (2018). Provenance network analytics. *Data Mining and Knowledge Discovery*, 32(3):708–735.

[Jarczyk et al., 2014] Jarczyk, O., Gruszka, B., Jaroszewicz, S., Bukowski, L., and Wierzbicki, A. (2014). GitHub projects. quality analysis of open-source software. In Aiello, L. M. and McFarland, D., editors, *Social Informatics: 6th International Conference, SocInfo 2014, Barcelona, Spain, November 11-13, 2014. Proceedings*, Lecture Notes in Computer Science, pages 80–94. Springer International Publishing.

[Jones, 2001] Jones, D. (2001). Sociometry in team and organisation development. *British Journal of Psychodrama and Sociodrama*, 16(1):10.

[Jorgensen, 2019] Jorgensen, M. (2019). Relationships between project size, agile practices, and successful software development: Results and analysis. *IEEE Software*, 36(2):39–43. Conference Name: IEEE Software.

[Kalliamvakou et al., 2015] Kalliamvakou, E., Damian, D., Blincoe, K., Singer, L., and German, D. M. (2015). Open source-style collaborative development practices in commercial projects using GitHub. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 574–585. ISSN: 1558-1225.

[Keyes, 2011] Keyes, J. (2011). *Social Software Engineering: Development and Collaboration with Social Networking*. CRC Press.

[Madey et al., 2002] Madey, G., Freeh, V. W., and Tynan, R. (2002). The open source software development phenomenon: An analysis based on social network theory. *undefined*.

[Magdaleno et al., 2009] Magdaleno, A. M., De Araujo, R. M., and Borges, M. R. D. S. (2009). A maturity model to promote collaboration in business processes. *International Journal of Business Process Integration and Management*, 4(2):111–123.

[McDonald and Goggins, 2013] McDonald, N. and Goggins, S. (2013). Performance and participation in open source software on GitHub. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, pages 139–144. Association for Computing Machinery.

[Moreau et al., 2008] Moreau, L., Groth, P., Miles, S., Vazquez-Salceda, J., Ibbotson, J., Jiang, S., Munroe, S., Rana, O., Schreiber, A., Tan, V., and Varga, L. (2008). The provenance of electronic data. *Communications of the ACM*, 51(4):52–58.

[Moreau and Missier, 2013] Moreau, L. and Missier, P. (2013). PROV-DM: The PROV data model.

[Newman, 2010] Newman, M. (2010). Measures and metrics. In *Networks*. Oxford University Press.

[Packer et al., 2019] Packer, H. S., Chapman, A., and Carr, L. (2019). Github2prov: provenance for supporting software project management. In *11th International Workshop on Theory and Practice of Provenance (TaPP 2019)*.

[Palomba et al., 2021] Palomba, F., Andrew Tamburri, D., Arcelli Fontana, F., Oliveto, R., Zaidman, A., and Serebrenik, A. (2021). Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE Transactions on Software Engineering*, 47(1):108–129. Conference Name: IEEE Transactions on Software Engineering.

[Palomba and Tamburri, 2021] Palomba, F. and Tamburri, D. A. (2021). Predicting the emergence of community smells using socio-technical metrics: A machine-learning approach. *Journal of Systems and Software*, 171:110847.

[Perez-Riverol et al., 2016] Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., Leprevost, F. d. V., Fufezan, C., Ternent, T., Eglen, S. J., Katz, D. S., Pollard, T. J., Konovalov, A., Flight, R. M., Blin, K., and Vizcaíno, J. A. (2016). Ten simple rules for taking advantage of git and GitHub. *PLOS Computational Biology*, 12(7):e1004947. Publisher: Public Library of Science.

[Peterson, 2013] Peterson, K. (2013). The github open source development process. *url: http://kevinp.me/github-process-research/github-processresearch.pdf (visited on 05/11/2017)*.

[Pobiedina et al., 2014] Pobiedina, N., Rümmele, S., Skritek, S., and Werthner, H. (2014). Benchmarking database systems for graph pattern matching. In *International Conference on Database and Expert Systems Applications*, pages 226–241. Springer.

[Pugh et al., 1969] Pugh, D. S., Hickson, D. J., Hinings, C. R., and Turner, C. (1969). The context of organization structures. *Administrative science quarterly*, pages 91–114.

[Raymond, 1999] Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49.

[Schreiber, 2020] Schreiber, A. (2020). Visualization of contributions to open-source projects. *arXiv:2010.08874 [cs]*.

[Schreiber and de Boer, 2020] Schreiber, A. and de Boer, C. (2020). Modelling knowledge about software processes using provenance graphs and its application to git-based version control systems. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, pages 358–359. Association for Computing Machinery.

[Schreiber and Zylka, 2020] Schreiber, R. R. and Zylka, M. P. (2020). Social network analysis in software development projects: A systematic literature review. *International Journal of Software Engineering and Knowledge Engineering*, 30(3):321–362.

[Scott, 2002] Scott, J. (2002). Social networks: Critical concepts in sociology.

[Sodeur, 2019] Sodeur, W. (2019). Bavelas (1950): Communication patterns in task-oriented groups. In Holzer, B. and Stegbauer, C., editors, *Schlüsselwerke der Netzwerkforschung*, Netzwerkforschung, pages 35–38. Springer Fachmedien.

[Sonnekalb et al., 2020] Sonnekalb, T., Heinze, T. S., Kurnatowski, L. v., Schreiber, A., Gonzalez-Barahona, J. M., and Packer, H. (2020). Towards automated, provenance-driven security audit for git-based repositories: applied to germany's corona-warn-app: vision paper. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Security from Design to Deployment*, SEAD 2020, pages 15–18. Association for Computing Machinery.

[Tamburri et al., 2013a] Tamburri, D. A., Kruchten, P., Lago, P., and van Vliet, H. (2013a). What is social debt in software engineering? In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 93–96.

[Tamburri et al., 2015] Tamburri, D. A., Kruchten, P., Lago, P., and Vliet, H. v. (2015). Social debt in software engineering: insights from industry. *Journal of Internet Services and Applications*, 6(1):10.

[Tamburri et al., 2013b] Tamburri, D. A., Lago, P., and van Vliet, H. (2013b). Uncovering latent social communities in software development. *IEEE Software*, 30(1):29–36. Conference Name: IEEE Software.

[Tamburri et al., 2017] Tamburri, D. A., Palomba, F., and Kazman, R. (2017). Exploring community smells in open-source: An automated approach. *IEEE Transactions on Software Engineering*, 47(3):630–652.

[Team, 2017] Team, R. C. (2017). R: A language and environment for statistical computing.

[Vicknair et al., 2010] Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., and Wilkins, D. (2010). A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference*, pages 1–6.

[Vizing, 1963] Vizing, V. (1963). The cartesian product of graphs. *Vycisl. Sistemy*, 9:30–43.

[Wu and Goh, 2009] Wu, J. and Goh, K. Y. (2009). Evaluating longitudinal success of open source software projects: A social network perspective. In *In Proc. of the 42nd Annual Hawaii International Conference on System Sciences (HICSS*.

[Wu and Tang, 2007] Wu, J. and Tang, Q. (2007). Analysis of survival of open source projects: a social network perspective. *PACIS 2007 Proceedings*.

[Yang et al., 2013] Yang, M.-H., Chen, J. C., Tsai, C.-L., and Chao, H.-Y. (2013). Investigating collaborative commerce system from the perspective of collaborative relationship. *Journal of Electronic Commerce Research*, 14(1):85.

[Zeini et al., 2012] Zeini, S., Göhnert, T., Hoppe, U., and Krempel, L. (2012). The impact of measurement time on subgroup detection in online communities. In *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 389–394.