Bachelor's Thesis

COMPARING ARTIFACT AND COMMUNICATION NETWORKS IN OPEN-SOURCE SOFTWARE PROJECTS

TOBIAS DICK

November 30, 2022

Advisor:
Thomas Bock   Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel   Chair of Software Engineering
Prof. Dr. Jilles Vreeken   Exploratory Data Analysis Group

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

ABSTRACT

Using social network theory to analyze open-source software (OSS) projects is a prevalent research topic in the software engineering domain. Developer networks built from e-mail or issue data have seen widespread use and have proven to be a viable way to analyze the developer communities of OSS projects. In this thesis, we build file-based artifact networks from the commit data of six different OSS projects taken from GitHub. We split the commit data for all of our sample projects into observation windows of equal length using a sliding-window approach and analyze how artifact networks evolve over the course of a project. During our analysis, we characterize the artifact networks using several network analysis metrics. Our results indicate that artifact networks have many commonalities which are independent of the number of commits and number of developers in a project. Additionally, we use a clustering algorithm to detect clusters in artifact networks and show that they represent modules in the software project. Finally, we compare the clusters detected in artifact networks to clusters found in developer communication networks built from issue data of the same projects using set similarity metrics. We find that the similarity between the sets of files committed by clusters in the developer communication networks and the files in clusters in the artifact network is substantially higher for a single developer cluster, while there are multiple other clusters in the communication network that commit to only a few selected artifact clusters. Our results indicate that the most active developers often communicate in a single cluster, while less active developers tend to form their own clusters. Additionally, they indicate the existence of spontaneously arising, task-focused developer clusters in the development of an OSS project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

OSS projects make up an increasingly important part of today's world of software development and have also received an increasing amount of attention from various researchers in the software engineering domain. These projects are particularly interesting because of their developer communities, which include many voluntary contributors from all over the world. The resulting communities often have unique organizational structures that are different from the organizational structures found in classical software engineering which lead to OSS projects becoming a prevalent research topic in the software engineering domain.

Since many popular OSS projects involve hundreds of developers, manual analysis of their organizational structure can be a tedious task, which raises the need for automated approaches that work on a big scale. A popular approach is using commits or communication data of OSS projects to build networks and analyze these by applying tools and metrics from social network analysis [1, 28, 41]. Developer collaboration networks in which developers are connected to each other if they commit changes to the same files have been used to detect community structures in OSS projects and gain insight into the collaboration of community members [20, 21]. Developer communication networks that are based on either mail or issue data have also proven to be a reliable way to analyze the developer structures [4, 37].

Many previous studies have solely focussed on the organizational structure of OSS projects without taking their software structure into account. Networks that are built from artifacts, such as files or functions, that are committed together in a single commit provide a network approach to analyze which parts of a project are dependent on each other during its development. We can use developer communication networks and artifact networks to compare the organizational structure of a developer community to the dependencies of artifacts in the same project. Using this approach, we try to detect common patterns and antipatterns in the communication of developers of the same OSS project in relation to which parts of the project they are working on. This insight can be used in future work to give both developers and researchers a better understanding of how developers should coordinate based on which part of a project they work on.

1.1 GOAL OF THIS THESIS

In this thesis, we want to investigate the characteristics of file-based artifact networks that are built from commit data of six real-world OSS projects from GitHub and analyze the relationship of artifact networks to the respective developer communication networks built from metadata of the GitHub issues of these projects. To do this, we split the observation windows of each of the sample projects into six-month-long observation windows using a sliding-window approach and apply several metrics from network theory to the artifact networks to gain insight into their characteristics as well as how these characteristics evolve over time. After that, we apply a clustering algorithm on the resulting networks for each of the observation windows. We then use the relationship between developers and their committed files during an observation window to map the detected clusters in the artifact networks to the nodes of the developer communication networks and vice versa. Finally, we compare these clusters to each other using the previously mentioned mapping and various metrics from set theory to gain insight into how developers communicate with each other in relation to which parts of a project they are working on.

1.2 OVERVIEW

In **Chapter** 2, we provide background information on OSS development, GitHub, network theory, and further knowledge that is relevant to this thesis.

In **Chapter** 3, we present our research questions and introduce the methodology we use to build artifact and developer communication networks as well as our approach to characterizing artifact networks and comparing them to developer communication networks. Additionally, we give a short summary of how our methodology is implemented and which tools we use for our analysis.

In **Chapter** 4, we evaluate the results of our characterization of artifact networks and the comparison between clusters detected in artifact networks and developer communication networks and discuss their possible implications. We also discuss internal and external threats to the validity of our results.

In **Chapter** 5, we take a look at related studies to the topic of this thesis.

In **Chapter** 6, we summarize the content of this thesis, give a conclusion, and list some suggestions for future work.

2 BACKGROUND

This chapter contains background information about different topics that are needed for this thesis. We first give an overview of OSS development on GitHub. After that, we introduce some basics of network analysis, as well as our definitions of artifact networks and developer networks. Finally, we give an overview of different set similarity metrics that we use to compare clusters in different networks.

2.1 OSS DEVELOPMENT ON GITHUB

GitHub[1] is a code hosting platform for software development that provides remote repositories using git[2]. With over 96 million users and more than 41 million public repositories as of september 2022[3], it is by far the most popular platform of its kind. In addition to its basic repository hosting functionality, it provides features such as bug tracking, software feature requests, task management, wikis, and, most importantly for this thesis, issues.

The issue system is a functionality available to any project hosted on GitHub. Along with mailing lists, they are one of the most important means of communication between developers in modern OSS projects [6]. In public repositories, anyone can create a new issue or comment on an existing one. Issues usually contain bug reports, feature requests, or suggestions for changes or extensions to the existing software.

To contribute changes or extensions to the source code, developers can link pull requests to issues. Pull requests contain one or multiple commits that are not yet merged with the main repository of a project. Once a pull request gets accepted, these commits are integrated into the main repository. Only the maintainers of a project, a small group of core developers with additional permissions, are allowed to decide which pull requests are accepted into the repository. This leads to maintainers being among the most active developers when it comes to their participation in issues. However, Destefanis et al. [12] found that many less active developers and even regular users that do not contribute any code to the project are actively commenting on issues as well, oftentimes even more than the active developers of a project. Additionally, some popular projects use bots to manage their issues and pull requests, for example, to filter out duplicates, ensure that developers sign license agreements or automatically run tests on proposed code changes [39]. Since issues are a central point of communication for maintainers, regular developers, and even users that are not directly involved in the development of a project, they are a suitable source to mine communication data and use it to understand the social structure of OSS projects.

1 https://github.com/
2 https://git-scm.com/
3 https://github.com/search?q=is%3Apublic (accessed at 2022-09-27)

2.2 NETWORK ANALYSIS

Networks are theoretical constructs that are frequently used to study relationships between people or objects involved in some form of mutual activity. Building networks and analyzing them is a common practice in several fields of research, such as bioinformatics, text analysis, genealogical research, and many more [10]. Over the course of this thesis, networks are used to analyze interactions between developers of OSS projects as well as relationships between files in the version-control systems of these projects. When used to describe interactions between people, networks are called social networks. Networks based on technical interactions are referred to as technical networks. The data to build such networks is usually taken from social-media platforms. In this thesis, we use data from GitHub repositories to build networks representing interactions between developers. These networks are also called *developer networks*. They have seen widespread usage in the software engineering domain, especially as a convenient data structure to analyze OSS projects [17, 18, 25]. We take a more detailed look at developer networks in Section 2.2.1. Section 2.2.2 describes a different type of networks called *artifact networks*, which describe relationships between files. Furthermore, we present some of the basics of graph theory that we need for this thesis in Section 2.2.3.

Networks can be represented mathematically by using graphs. A graph is a tuple $G = (V, E)$ where V is a set of vertices and E is a set of edges. An edge $e \in E$ is used to link two vertices, so the set of edges has the form $E = \{(u, v) \mid u, v \in V\}$ [10]. Graphs can be directed, which means that edges have a start and a target, or undirected, which means that edges are bidirectional. Formally, in a directed graph $(u, v) \neq (v, u)$ with (u, v) being the edge from $u \in V$ to $v \in V$. In an undirected graph, an edge from u to v is the same as an edge from v to u, so $(u, v) = (v, u)$. We use weighted graphs, in which every edge has a weight in the form of a natural number assigned to them. A formal representation of edge weights is given by a function $\omega\colon E \to \mathbb{N}$ that assigns each edge $e \in E$ a weight $\omega(e)$. Unweighted graphs are graphs with a $\omega(e) = 1$ for all $e \in E$. We can also simplify graphs by unifying all existing edges that connect the same vertices in the same direction into a single edge. The weight of each simplified edge is equal to the sum of weights of the edges in the unsimplified graph it was built from. In the case of the unsimplified graph being unweighted, its simplified version still contains the same information since the weight of an edge in the simplified graph equals the number of edges between the same vertices in the unsimplified graph. [10, 36]

2.2.1 *Developer Networks*

In general, a developer network is a social network that connects developers of a project based on a certain kind of interaction. Joblin differentiates between developer collaboration networks and developer communication networks [8, 18]. Developer collaboration networks are built from commit data from a version-control system such as Git. In these networks, developers are connected by an edge if and only if they have committed changes to the same artifact. The kind of artifact chosen to build a developer collaboration network differs based on the goal of the analysis, common artifact types are files or functions. Using developer collaboration networks has been a common research practice on OSS projects [20, 25, 29].

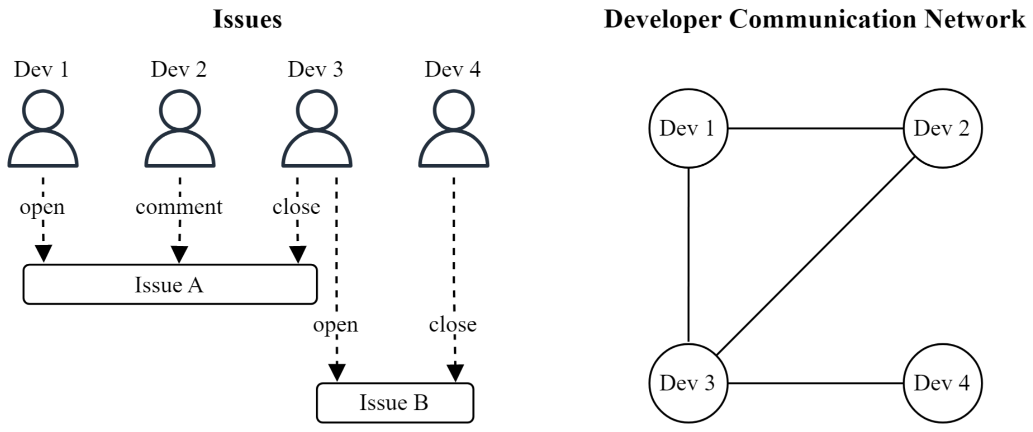


Figure 2.1: A developer communication network constructed from 2 issues with which a total of 4 different developers interacted. In the resulting network, developers are connected by an undirected edge if they have interacted with the same issue.

Developer communication networks are built from communication data rather than commit data. This data is usually taken from the main communication channel of a software project such as a mailing list or issues in the case of OSS projects. In developer communication networks, two developers are connected by an edge, if they have interacted in the communication channel, for example by one developer writing a mail to another developer or by two developers answering to the same issue. We only consider undirected developer communication networks, but there are also directed versions of developer communication networks, as for example proposed by Joblin [18]. Simplifying the complexity of human communication to a simple network has its weaknesses, as we can not reason about the content of the messages used to build the network. As a result of this weakness, it is unclear whether a message contains meaningful information (e.g., feedback or criticism) or is off-topic (e.g., private conversations between developers). However, several studies have shown that developer communication networks still convey valuable information about the communication between developers in a software project [4, 32, 40]. Since we want to compare the communication between developers to the structure of the project that they are working on, we use developer communication networks over the course of this thesis and do not use developer collaboration networks. An example for a developer communication network is shown in Figure 2.1.

2.2.2 *Artifact Networks*

Artifact networks are technical networks built from commit data that connect the artifacts of a software project based on a commonality between them. In the artifact networks used in this thesis, two files are connected by an undirected edge if they are committed together in a single commit. Depending on the number of commits in which two artifacts are committed together, they can also have multiple edges between them. However, we mostly consider simplified artifact networks in this thesis, which means that there is at most one edge between two artifacts, with a weight according to the number of times these files have been committed together assigned to it. The type of artifact chosen determines the granularity of

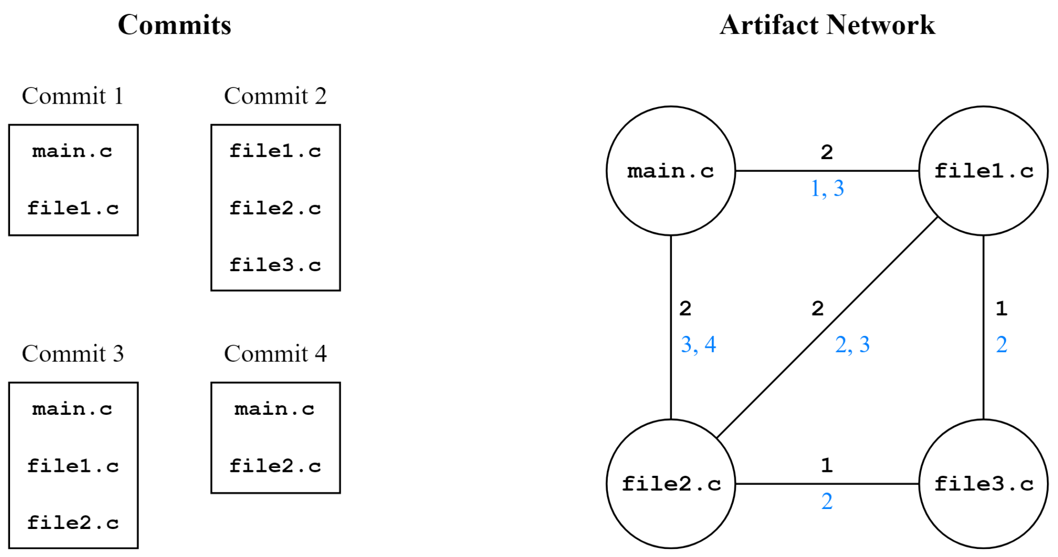


Figure 2.2: A simplified file-based artifact network constructed from 4 different commits which contain changes to the files main.c, file1.c, file2.c, and file3.c. Files are connected by an undirected edge if they have been committed together in a single commit. The edge weights represent the number of times edges have been committed together. The blue numbers correspond to the numbers of the commits that an edge was constructed from.

the information that we can extract from the network. In this thesis, we use file-based artifact networks which means that every vertex represents exactly one file. Choosing functions as artifacts would provide a more fine-grained approach, but in turn, we would drastically increase the number of vertices in many cases which can make the results more difficult to analyze. A less fine-grained choice could be, for example, the different software modules of a project. All these approaches have a common weakness: They couple lines of code based on their proximity to each other. Depending on the software project and programming language, there might be dependencies across the boundaries of artifacts that we miss when using artifact networks. However, we show that artifact networks provide a good heuristic to analyze real-world OSS projects and the dependencies among their artifacts. An example for a file-based artifact network is shown in Figure 2.2.

2.2.3 *Graph Theory*

The *density* $\rho(G)$ of an undirected graph $G = (V, E)$ is defined as the number of edges $|E|$ divided by the number of possible edges $\frac{|V|\cdot(|V|-1)}{2}$, so $\rho(G) = \frac{2\cdot|E|}{|V|\cdot(|V|-1)}$ [36]. By definition, $0 \leq \rho(G) \leq 1$ always holds. The density of a graph gives an indication of how strongly connected the vertices of the graph are to each other. A graph with a density close to 1 is called *dense*, while a graph with a density close to 0 is called *sparse*. However, there is no universal definition for a range of density values that is referred to as dense or a range of values that is referred to as sparse. [36]

In an undirected graph $G = (V, E)$, the *degree* $\delta(v)$ of a vertex $v \in V$ is the number of edges connected to it [36]. Loops, which are edges from a vertex to itself, are usually counted twice.

However, we do not use any graphs that contain loops in this thesis. Using the degree of a vertex, we can define the *average degree* of G as the average of all vertex degrees $\frac{1}{|V|}\sum_{v\in V}\delta(v)$. Similar to the density, the average degree of a graph can give us an indication of how strongly connected the vertices in the graph are. Unlike the density, the average degree of a graph is not bound between 0 and 1, but instead gives us an absolute value greater than 0. Consider two graphs $G = (\{A, B\}, \{(A, B)\})$ and $H = (\{A, B, C\}, \{(A, B), (A, C), (B, C)\})$. In this example, both graphs are fully connected. Both G and H have a density of 1, but G has an average degree of 1 while H has an average degree of 2. [36]

A *path* between two vertices v_0 and v_k in a graph $G = (V, E)$ is an alternating sequence $[v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k]$ of vertices and edges with $e_i = (v_{i-1}, v_i)$ and $v_i \neq v_j$ if $i \neq j$, i.e. all vertices in the sequence are distinct [10, 36]. The unweighted length of a path is defined as the number of edges in the path. Special kinds of paths appear in problems across many research fields where networks or graphs are used. For this thesis, we only use paths to calculate the *average path length avgp*, which is defined as the average unweighted length of the shortest path between all possible pairs of vertices. [10, 36]

A *clustering* $Cl(G) = \{Cl_1, \ldots, Cl_k\}$ of a graph $G = (V, E)$ is a partition of the vertex set V into non-empty subsets. A set Cl_i is called a *cluster*. In social networks, clusters are also referred to as communities. In principle, a graph can be partitioned however we like. However, we want to detect groups of vertices that are internally dense, meaning they contain a high number of internal edges, i.e., edges between vertices in the same cluster, while being only sparsely connected with other groups. While this way of phrasing it is more of an intuition than a formal definition of a good clustering, there exist many, often complex ways to determine the quality of a clustering. The measure of quality used to compare clusters often depends on the clustering algorithm used to detect the clusters, as many popular clustering algorithms try to maximize a certain quality metric for clusterings. [10]

Detecting clusters has been a prevalent research topic in network analysis for many years. Clusters have been used across many fields of research since they can be used to detect community structures in networks, which has proven to be especially helpful in the domain of social network analysis. Clusters can be used to detect target groups in users for marketing and recommendations, interactions between proteins in biological networks, analyzing collaborations in various domains, or, in our case, to detect communities in developer networks and find modules in artifact networks. [3, 21]

In this thesis, we use the *Louvain method for community detection* [7], which maximizes the *modularity* of a clustering and has the benefit of working on weighted graphs while also being relatively fast to compute. The modularity of a clustering $Cl = \{Cl_1, \ldots, Cl_k\}$ of a graph $G = (V, E)$ is defined as

$$Q = \frac{1}{2m} \sum_{v_i, v_j \in V} \left(\omega(v_i, v_j) - \frac{k_i k_j}{2m} \right) \delta(Cl(v_i), Cl(v_j))$$

$$m = \sum_{e \in E} \omega(e)$$

where k_i is the sum of weights of all edges attached to v_i, $\omega(v_i, v_j)$ is the edge weight of the edge from v_i to v_j, $Cl(v_i)$ is the cluster that contains v_i and δ is the Kronecker delta function with $\delta(x, y) = 1$, if $x = y$ and $\delta(x, y) = 0$, otherwise [30]. The modularity always has a value in the range $\left[-\frac{1}{2}, 1\right]$. Higher values indicate better clustering. A negative value indicates that there is no community structure present in a graph. It is important to note that, even though using the Louvain method to build a clustering that maximizes modularity, the Louvain method does not always yield optimal results. This, however, is the case for all clustering algorithms that have an acceptable running time, as they cannot brute-force every possible clustering and therefore have to rely on heuristics to maximize a clustering quality metric.

Before we apply a clustering algorithm to a network, we want to determine whether that network actually has a structure that allows us to detect clusters, as applying clustering algorithms to graphs that do not contain any community structure can yield misleading and unhelpful results. A network metric that gives a good indication of a graphs potential to contain community structures is the *clustering coefficient*, which is a rational number between 0 and 1. A high clustering coefficient indicates the existance of clusters that are strongly connected internally. There are multiple ways to calulate the clustering coefficient, many of which are equivalent to each other. We use the global clustering coefficient defined by $cc(G) = \frac{3|\triangle(G)|}{|\wedge(G)|}$ where $\triangle(G)$ is the set of triangles in a graph G and $\wedge(G)$ is the set of triples in G. A triangle is a subgraph containing three vertices that are connected with each other. Triples are subgraphs of three vertices that contains exactly two edges. [36]

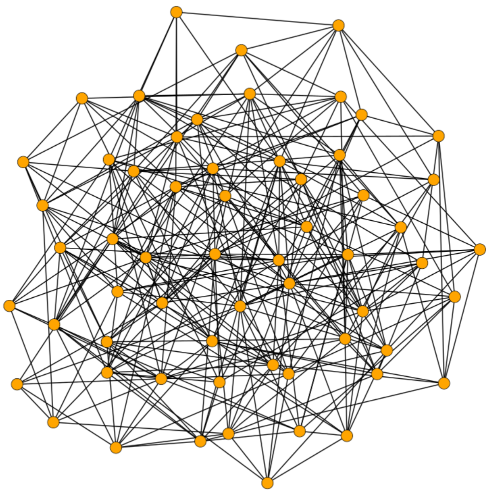
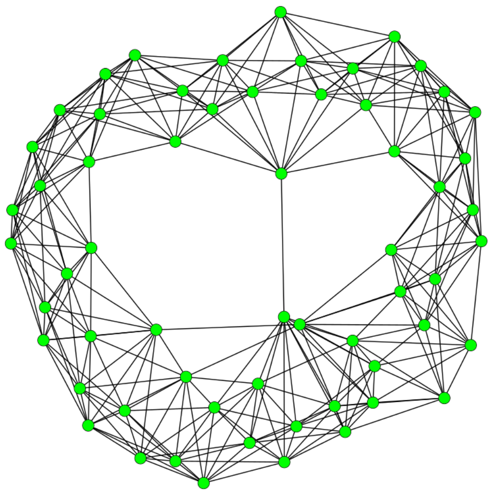



Figure 2.3: An Erdős-Rényi random network (left) and a small-world network (right). Both networks contain 60 vertices and 300 edges. In the small-world network, every node is strongly coupled with its neighbours. This structure leads to a high clustering coefficient of 0.63 while the network has a relatively low average path length of 2.87. The Erdős-Rényi random network, which does not fullfill the properties of a small-world network, has an even shorter average pathlength of 1.97 due to its completely random connections, but this randomness also leads to a much lower clustering coefficient of 0.16.

Using the clustering coefficient, we can check whether a network satisfies the properties of a *small-world network*. Small-world networks have a low average path length and a high clustering coefficient while still maintaining a relatively low number of total edges. Many real-world networks, especially social networks, have been shown to be small-world networks. Due to their high clustering coefficient along with their relatively low number of edges, small-world networks often contain a detectable community structure, which serves as a good starting point for a clustering algorithm. To check whether a given network G satisfies the small-world property, we first generate a random network H with an equivalent number of vertices and edges to G using the Erdős-Rényi method [14]. We then compute the small-worldness of our network G as follows:

$$smallworldness(G) = \frac{cc(G)/cc(H)}{avgp(G)/avgp(H)}$$

If the small-worldness of a graph is greater than or equal to 1, it satisfies the small-world property. An example for a random graph that was created using the Erdős-Rényi model next to a small-world graph is depicted in Figure 2.3. [36]

2.3 SET SIMILARITY MEASURES

In this thesis, we use several set similarity measures to compare sets of developers contained in the clusters of developer networks to sets of artifacts contained in the clusters of artifact networks. All the similarity measures presented in this section are meant to be applied to two finite sets.

The *Jaccard index* is one of the most well-known set similarity measures. It is defined as the size of the intersection of two sample sets A and B divided by the size of their union:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

By definition, $0 \leq J(A,B) \leq 1$ holds for any finite sets A and B. Additionally, we define $J(A,B) = 0$, if $A = \emptyset$ and $B = \emptyset$. The Jaccard index gives the similarity of two sets as a percentage. It has seen widespread use across many domains, especially in machine learning and computer vision. Due to it being developed by multiple researchers independently of each other, there are other names for the Jaccard index such as Jaccard similarity coefficient, Critical Success Index or Tanimoto index. [23]

The *overlap coefficient* is related to the Jaccard index, but it divides the size of the intersection by the size of the smaller of the two sets instead of the size of their union:

$$overlap(A,B) = \frac{|A \cap B|}{min(|A|,|B|)}$$

Additionally, we define $overlap(A,B) = 0$, if $A = \emptyset$ or $B = \emptyset$. By design, the results of the overlap coefficient always satisfy $0 \leq overlap(A,B) \leq 1$ for any finite sets A and B. As the name implies, the overlap coefficient measures the overlap between two sets. If either of the sets is a subset of the other, the overlap coefficient is equal to 1. This makes it easy to detect and visualize subsets in big datasets. The closer the result is to 1, the more elements of the smaller set are also contained in the bigger set. An overlap coefficient of 0.75, for example, would mean that 75% of the elements in the smaller set are also contained in the bigger set. [27]

Comparing results of using the overlap coefficient on sets from two different sources is difficult, as depending on the size of the sets we do not always divide by the size of the sets from the same source. To make the results of our analysis more comparable to each other, we introduce a modified version of the overlap coefficient called *completeness*, in which we predetermine one of the sizes of the input sets A or B as the divisor:

$$completeness_{B \subset A}(A, B) = \frac{|A \cap B|}{|A|} \qquad completeness_{A \subset B}(A, B) = \frac{|A \cap B|}{|B|}$$

Additionally, we define both versions of completeness to be 0 if either of the input sets is empty. This notion of completeness allows us to compare how many elements of B are contained in A ($completeness_{B \subset A}$) or vice versa ($completeness_{A \subset B}$). The advantage of using completeness over the regular overlap coefficient is that we can decide in advance which side of the overlap we are interested in. It is important to notice that completeness, unlike the overlap coefficient, is not symmetric. The results of this similarity measure can be interpreted analogously to the overlap coefficient.

3 METHODOLOGY

In this chapter, we present our approach for building and analyzing networks as well as its implementation. First, we provide our research questions. After that, we give an overview of the projects we analyze in this thesis. Then, we explain how we filter the data from GitHub and construct networks. Additionally, we describe our initial investigation on artifact networks, how we detect clusters in artifact networks and developer communication networks as well as how we compare the resulting clusters to each other. Finally, we provide an overview of the software we used to implement our methodology.

3.1 RESEARCH QUESTIONS

In this section, we present our two research questions for this study. The goal of this thesis is to compare artifact networks to developer communication networks using clusters detected in both network types. Since there has not been a lot of research done on artifact networks built from OSS projects to the best of our knowledge, we first want to gather information about the characteristics of artifact networks. This leads us to our first research question.

RQ1. *What are the characteristics of artifact networks and how do they evolve over time?*

The characteristics of artifact networks that we are interested in include various standard metrics from network analysis. By using observation periods of multiple years for all of the projects analyzed in this thesis and splitting them into several small observation windows as presented in Section 3.3.1, we can also analyze how these characteristics evolve over time. The characteristics of artifact networks as well as their evolution can then potentially be used for future research. In this thesis, we show that we can detect clusters with real-world implications in artifact networks. We use these clusters to answer our second research questions.

RQ2. *Which relationships between the communication of developers and the artifacts they commit can we detect by comparing clusters of artifact networks and developer communication networks?*

We analyze which relationships between clusters in artifact networks and developer communication networks can be detected. This also includes how these relationships evolve over the lifespan of a project. In Chapter 4, we discuss the implications the detected relationships have on the communication of developers of the different projects we analyze.

3.2 PROJECTS

We have chosen six different OSS projects that are all hosted on GitHub as a sample set for our analysis. All projects have been active for multiple years and multiple hundreds of developers have participated in the development of each of the projects at some point. The number of commits and issues in the projects we chose for our analysis differ greatly, with the number of commits ranging from 3492 up to 29150 and the number of issues ranging from 8762 up to 41259. The exact number of commits we extracted, the observation period, and the number of unique developers that committed changes to the code of the respective projects are listed in Table 3.1. The same statistics are listed for the extracted issues in Table 3.2. It is important to note that there are a lot of users participating in the issues of the projects that never made a commit to them, as can be seen from the substantially lower number of developers than the number of issue participants in each of the projects. Since most of the projects did not use issues from the beginning, the observation periods for the extracted issues are shorter than those for the extracted commits. However, we use the parts of the observation periods that do not feature any issues, as we can still build artifact networks from them and use them for our initial analysis of the characteristics of artifact networks.

Table 3.1: Number of commits, the number of commit authors, and the observation period the commits are extracted from for each project.

Project	# Commits	# Authors	Observation period
Deno	3492	350	2018-05-14 - 2020-12-22
OpenSSL	15438	418	1998-12-21 - 2020-02-17
Atom	16267	299	2011-08-19 - 2020-12-10
TypeScript	17973	470	2014-07-08 - 2020-12-22
Nextcloud	29150	673	2010-03-10 - 2020-09-22
Moby	14103	1160	2013-01-19 - 2020-12-22

Table 3.2: Number of issues, the number of authors participating in issues, and the observation period the issues are extracted from for each project.

Project	# Issues	# Authors	Observation period
Deno	8762	2952	2018-05-29 - 2020-12-22
OpenSSL	11196	3156	2013-09-05 - 2020-02-27
Atom	21182	20662	2012-01-21 - 2020-12-24
Nextcloud	22844	9369	2016-06-02 - 2020-10-03
TypeScript	41259	17716	2014-07-14 - 2020-12-23
Moby	41727	27795	2013-01-20 - 2020-12-22

DENO[1] is a runtime environment for JAVASCRIPT, TYPESCRIPT and WEBASSEMBLY. It is based on the V8 JAVASCRIPT engine, which is also used by GOOGLE CHROME, and RUST, a programming language heavily focussed on type safety and concurrent programming that has been getting increasing popularity in recent years. DENO has only been in development since 2018 and is therefore the most recently created project in our sample set. However, it has a very active developer community with 350 unique developers listed in our extracted commits.

OPENSSL[2] is a toolkit for general-purpose cryptography and secure communication. It has initially been developed as a toolkit for the SECURE SOCKETS LAYER (SSL) encryption. Nowadays, it is also used for TRANSPORT LAYER SECURITY (TSL). OPENSSL is based on the C programming language. It is the oldest project in our analysis, with an initial release in 1998.

ATOM[3] is a source-code editor based on ELECTRON[4], another popular OSS project that can be used to develop cross-platform desktop applications. It is owned by GITHUB and has been developed by its community. ATOM was first published in 2014, with the development running since 2011. As of 2022, GITHUB has officially ceased development of the project[5]. Even though our observation period for the ATOM project ends in December of 2020, we can already observe diminishing commit activity and fewer active developers at the end of the observation period, which can give us insight into how the artifact networks of projects with diminishing developer activity evolve.

TYPESCRIPT[6] is a scripting language that extends the JAVASCRIPT language. The goal of TYPESCRIPT was to fix various issues developers would face when building big applications in JAVASCRIPT. It was published by MICROSOFT in 2012 but has only been migrated to GITHUB in 2014. The repository was migrated by using a snapshot of the old repository which is why commits from before the migration are not included in the current repository.

NEXTCLOUD[7] is a software suite for creating and using file hosting services. It offers a server-side application running on LINUX as well as client-side applications on WINDOWS, ANDROID, MAXOS, LINUX and IOS. In our analysis, we only extracted commits and issues from the NEXTCLOUD SERVER repository[8], which is currently the most active of the over 260 different repositories connected to NEXTCLOUD. It has been forked from the OWNCLOUD[9] project by multiple former core developers in 2016, which lead to it having a very active developer community from the start.

MOBY[10] is a framework to assemble container systems. A container system encapsulates an application and the resources the application depends on, such as runtimes or system li-

1 https://deno.land/
2 https://www.openssl.org/
3 https://atom.io/
4 https://www.electronjs.org/
5 https://github.blog/2022-06-08-sunsetting-atom/
6 https://www.typescriptlang.org/
7 https://nextcloud.com/
8 https://github.com/nextcloud/server
9 https://owncloud.com/
10 https://mobyproject.org/

braries. This allows developers to port their applications to different computers without adapting the source code of the original software to different operating systems. It is based on Docker[11] which is a popular containerization software. Unlike Docker, which is suitable for enterprise usage, Moby is specifically tailored to developers who are interested in experimenting with containers or use them for debugging purposes and therefore allows a high amount of customization. In our sample set, Moby is the biggest project in terms of active developers, number of issues, and users commenting on issues, but it has a rather low number of commits. Just like for Atom, the developer activity in terms of commit count towards the end of our observation period for Moby slowly diminishes, but it never reaches numbers as low as Atom.

3.3 NETWORK CONSTRUCTION

In this section, we present how we partition and filter the commit and issue data and how we construct the artifact networks and developer communication networks from the filtered data.

3.3.1 *Data Partitioning*

Since the repositories we analyze have been active for multiple years, with some of them being created more than two decades ago, building a single artifact network from all commits or a developer communication network from all issues created over the lifespan of a project is likely to yield results that are hard to analyze. In most cases, changes or additions to a project only take a few days or weeks, so most files are only frequently added together for a short period of time. We provide evidence for this relationship between files through an analysis of artifacts with high degrees in Chapter 4. The developer community is also subject to many changes, as existing developers might change their role in the development of the project or leave it and new developers join the project [20]. These changes to the developer community are likely to make a developer communication network unrepresentative of real-world relationships if the issue data used to build the network is spread over a long observation window, as the network contains relationships that are outdated, with no way to distinguish them from more recent relationships.

To combat this issue, we split our data into six-months long observation windows using a sliding-window approach. In previous work, Joblin et al. [20] have shown that this approach to splitting commit data is suitable to analyze the evolution of developer collaboration networks. In this thesis, the n^{th} observation window is defined as a set C_n of commits and a set I_n of issues, such that $C_n = \{commit_t \mid t \in W_n\}$ and $I_n = \{issue_t \mid t \in W_n\}$ with $W_n = [t_0 + \frac{n}{2} \cdot (\Delta_{window} - 1), t_0 + (\frac{n}{2} + 1) \cdot (\Delta_{window} - 1)]$. $commit_t$ is a commit made at time t, $issue_t$ an event that happened to an issue at time t. An event can be, for example, the opening of a new issue, a new comment to an existing one, a developer subscribing to an issue, or a maintainer closing an issue. t_0 marks the time of the first entry in our dataset for the respective project. Joblin et al. used 90 day observation windows ($\Delta_{window} = 90$), but

11 https://www.docker.com/

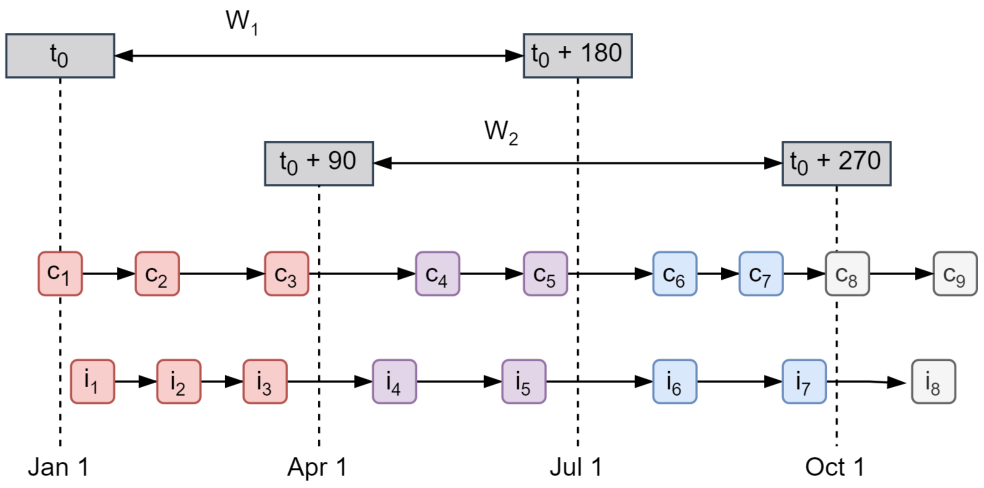


Figure 3.1: A dataset consisting of 9 commits $c_1, \dots, c_9$ and 8 issues $i_1, \dots, i_8$ is partitioned into two subsequent observation windows W_1 and W_2 with a length of 180 days. The start of W_1 (t_0) is based on the first entry into the dataset C_1. Red data points are part of C_1 (commits) or I_1 (issues), blue data points are in C_2 or I_2 and purple data points are contained in both C_1 and C_2 or I_1 and I_2. Grey data points do not fall within any of the two time windows displayed in this figure.

we use 180 day observation windows instead ($\Delta_{window} = 180$), as according to our investigation, shorter observation periods than 180 days lead to a very strong fluctuation in the characteristics of the resulting artifact networks. Figure 3.1 provides an example for our way of partitioning the commit and issue data.

After partitioning the data as described, we can build artifact networks from every set of commits C_n and developer communication networks from every set of issues I_n. Using the same observation windows for both sets ensures that comparing the artifact network built from C_k with the developer communication network built from I_n has real-world implications, as both these networks contain relationships from the same observation window W_k. By design, the first half of every observation window overlaps with the observation window before it and the second half with the observation window after it, which allows us to capture relationships at the cut between W_k and W_{k+2} using the networks obtained from W_{k+1}. These relationships would be lost if we do not use a sliding-window approach.

3.3.2 *Filtering the Input Data*

During our investigation, both artifact networks and developer communication networks have proven difficult to analyze when being built from the unfiltered commit data and issue data from GitHub. In this section, we point out the difficulties arising from using unfiltered data as well as filters we use to avoid them.

Type of change	# occurences in 30 commits with 21 - 30 files	# occurences in 60 commits with 31 or more files
Code style changes	8	12
New/updated dependency	11	9
Localization	0	9
New functionality	0	9
File renamed	1	6
Refactoring	3	5
Added/changed comments	3	4
New unit tests	1	3
Variable/function renamed	1	2
Bugfix	2	1

Table 3.3: Manual classification of changes done in 90 randomly selected commits.

The extracted commit data has already been prefiltered during the extraction process. Since in some cases, developers might have used multiple usernames or e-mail addresses, these have to be disambiguated. Additionally, commits made by bots are filtered out of the data, as they are not relevant to our analysis. Commits containing a large number of files also proved to be problematic for our approach, as each of the files is connected to each other file in the resulting artifact network. A commit with n files leads to a fully connected subgraph with n vertices and $\frac{n \cdot (n-1)}{2}$ edges. This almost quadratic relationship between the number of files committed and edges in the artifact network leads to larger commits having a much bigger influence on the resulting network. However, most of the commits that are interesting for our analysis include a relatively small number of files of less than 20 since we are mainly interested in commits that make changes to specific parts of the project to extend existing code or add new functionality. To prove our claim, we did a manual analysis of 5 randomly selected commits that contain between 21 and 30 files and 10 commits that contain 31 or more files for each of the 5 projects in our sample set. This leads us to a total of 90 randomly selected commits. We did a manual classification of the changes made by each of these commits. Only 9 of them contain the addition of new functionality and 3 of them contained a bugfix, which we consider to be the only relevant types of changes for our analysis that we found in the 90 commits. The exact results of this analysis are listed in Table 3.3. Additionally, less than 10% of the commits in our sample set change more than 20 files. Since commits with more than 20 files have a too big influence on the resulting artifact networks, rarely contain relevant changes to our analysis, and are relatively rare, we filter out all commits that make changes to more than 20 files.

Just like the commit data, the issue data has disambiguation applied to the names and e-mail addresses of the participating users, and bots are filtered out. When using the prefiltered data to build developer communication networks, these networks are usually too dense to analyze as issues contain a very large number of participants. As can be seen by comparing the number of authors in Table 3.1 to Table 3.2, many users comment on the issues of a project but never commit to it. Since we want to compare artifact networks and developer communication networks, users who do not commit anything are not relevant to our analysis. Therefore, we remove all users from the issue data of each observation window who did not commit anything before or during that observation window. In more formal terms, we remove users from all issues in I_k, if the user is not the author of any commit in $C_1, \dots, C_k$, for each k in $1, \dots, n$. This way of filtering out users also improves the results of our cluster detection on developer communication networks, as the resulting networks are less dense and contain less superfluous relationships between developers and other users.

3.3.3 *Network Configuration*

After we collect the data, partition it into 6 months long observation windows, and apply filtering to it, we can build networks from it. In this section, we describe how the networks are constructed for each of the two network types.

For an artifact network in the n^{th} observation window, the names of all files committed in the filtered commit data C_n are used as the set of vertices. The vertices are then connected by undirected edges if the files they represent have been committed together. After constructing the edges, the network gets simplified. Simplifying the network makes further computations during our analysis faster without changing the results, as the number of edges in the unsimplified network is encoded into the edge weights of the simplified network. Furthermore, we remove all isolated vertices, i.e. vertices that have no edges connected to them, since we want to avoid a large overhead of clusters that only consist of a single artifact.

The developer communication networks for the n^{th} observation period are built by using all developers that interacted with any issues in the filtered issue data I_n as the set of vertices. This set does not only include developers who commented on issues but also developers who interacted with the issues in any other form such as opening an issue, adding a tag to it, or closing it. Developers are then connected with undirected edges if they have been interacting within the same issue. Then, the developer communication networks are simplified to make further computations on these networks faster. Finally, we remove all isolated vertices since they would lead to one-man clusters, which do not give us any insight into the communication of developers.

3.4 ANALYSIS

Our analysis of networks is divided into three different steps. First, we analyze the artifact networks for each of our projects using different graph metrics to answer our first research question. Then, we use the Louvain clustering algorithm to detect clusters in both the artifact networks and the developer communication networks. In the final step, we use a comparison of these clusters to analyze relationships between the communication of developers and the part of the project they commit to answer our second research question.

3.4.1 *Initial Artifact Networks Analysis*

During the first step of our analysis, we only focus on artifact networks as there has not yet been a lot of research done on their characteristics, while developer communication networks have been analyzed in multiple other studies already [4, 32, 40]. We apply multiple graph metrics to each of the artifact networks for each project.

We are computing the number of vertices and edges, the density, the average degree, and the average path length of artifact networks. Using the number of vertices, we can infer how many different files are usually committed over the course of an observation window for a project. The number of edges, the density, and the average degree of the artifact networks give us insight into how strongly connected the networks are, i.e. how often different files are committed together. If artifact networks are relatively sparse despite being built from a high number of commits, we can conclude that the files that are committed together are usually committed together in certain groups instead of being randomly distributed, as a random distribution of jointly committed files would lead to a more dense graph. Since the observation periods for each of our projects is split into multiple observation windows from each of which we build a seperate artifact network, we can also investigate how these networks evolve over time. To do this, we first compute the average and the standard deviation for each of these metrics applied to the networks of each of the projects. The standard deviation can give us an understanding of how strongly the characteristics of artifact networks fluctuate over the course of our observation periods. In addition to the average and the standard deviation, we also plot the evolution of these metrics over time and do a manual analysis of the results.

Furthermore, we are interested in the clustering coefficient of artifact networks. Using it, we can check whether artifact networks fulfill the properties of small-world networks. If artifact networks are small-world networks, they should also contain detectable clusters which helps us during our analysis. If applicable, we also investigate how these characteristics of artifact networks relate to the number of commits, and the average number of files committed in the observation window the network is built from. Due to our large number of observation windows, we can also analyze how the clustering coefficient evolves over time as the development of a project progresses.

We then take a look at how often files are committed together, i.e., how high the average edge weight in the artifact networks is and whether we can detect files that depend on each other using the edge weight between them. Using edge weights, we could detect dependencies in the source code independent of the programming language used.

Most artifact networks contain vertices with high degrees that greatly exceed the average. These vertices are also called *hubs* [36]. Since there is no clear cutoff defined for how high of a degree a vertex has to have to be classified as a hub, we decided to analyze the ten artifacts with the highest degrees across all artifact networks. We discuss whether this set of files serves a central role in the source code of the analyzed projects and whether it stays the same over the course of the development of a project or if the central files change over time.

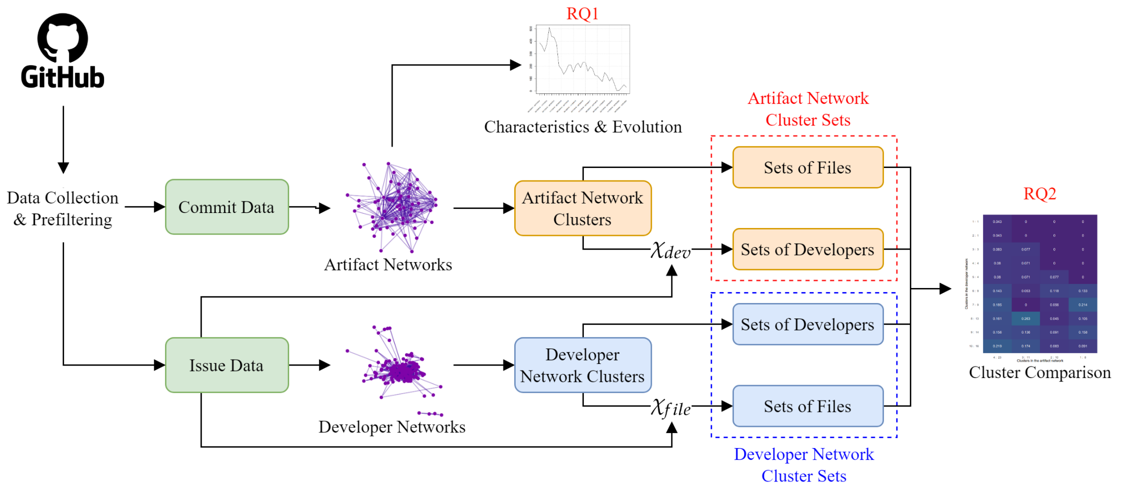


Figure 3.2: The workflow of our study approach.

3.4.2 *Cluster Detection*

During our initial investigation, we show that artifact networks have characteristics that allow us to detect clusters in them. In the second step of our analysis, we use the Louvain clustering algorithm to compute a clustering on all of our artifact networks. Then, we do an empirical analysis of these clusters and discuss whether they have any real-world implications on the projects the networks are constructed from. We use the same algorithm to compute clusterings on all constructed developer communication networks. Previous studies have shown that clusterings on different types of developer networks can be used to detect verified communities [1, 9, 21, 38], so we do not analyze whether the resulting clusters align with the real communities in the projects. However, we use the detected clusters in developer communication networks to compare them to the clusters found in artifact networks.

3.4.3 *Cluster Comparison*

In this section, we describe our method of comparing the clusters we computed in an artifact network to those we computed in the developer network for the same observation window. For simplicity, we describe every step of our analysis for exactly one artifact network and one developer communication network. In reality, we apply the steps described in this section to the pairs of artifact and developer communication networks for every observation window for each project in our dataset.

Since vertices in artifact networks and developer communication networks do not represent the same things, we need a way of mapping developers to files and vice versa before we can compare the clusters in the two different network types to each other. We denote the i^{th} cluster in the artifact network as A_i, where $i \in \{1, \ldots, k\}$ and k is the total number of clusters detected in the respective artifact network. Analogously, we denote a cluster in the developer communication network D_j, where $j \in \{1, \ldots, l\}$ and l is the total number of clusters detected in the respective artifact network. A cluster in an artifact network can be interpreted as a set of files, and a cluster in a developer communication network can be interpreted as a set of developers. Using the set of commits C that we used to build the artifact network, we can map a set of files F to the developers that committed them. This gives us the following conversion function:

$$\chi_{dev} : files \rightarrow developers$$
$$\chi_{dev}(F) = \{d \mid d \in developers, d \text{ has made changes to at least one} f \in F \text{ in } C\}$$

Conversely, we can use C to build a conversion function that maps a set of developers D to the files they committed:

$$\chi_{file} : developers \rightarrow files$$
$$\chi_{file}(D) = \{f \mid f \in files, f \text{ has been changed by } d \in D \text{ in } C\}$$

With the help of these conversion functions, we can build sets that we can compare to each other. We use three different kinds of comparisons to answer our second research question.

Developer Set Comparison. The most important step in our comparison consists of comparing the developers in the clusters of the developer network to the sets of developers we obtain by using the χ_{dev} on the clusters of the artifact network. We compute the Jaccard index $J(\chi_{dev}(A_i), D_j)$ and the completeness $completeness_{D_j \subset \chi_{dev}(A_i)}(\chi_{dev}(A_i), D_j)$ for every possible combination of $i \in \{1, \dots, k\}$ and $j \in \{1, \dots, l\}$. These two metrics give us a good indication of how strongly the group of developers that work on a cluster of files detected in the artifact network align with the clusters we detect in the developer communication network. We use completeness instead of the overlap coefficient as we are more interested in whether groups of developers that commit to the same cluster in the artifact network also form a cluster in the developer communication network rather than how often clusters of communicating developers work on the same cluster in the artifact network. Due to the nature of our comparison, actively committing developers will show up in multiple sets $\chi_{dev}(A_{i_1}), \dots, \chi_{dev}(A_{i_m})$, but will only show up in exactly one cluster D_j, as the clusters in the developer communication network are not overlapping and every developer is represented by exactly one vertex. However, the results we obtain from using only the completeness are fluctuating heavily for smaller sets $\chi_{dev}(A_i)$, which is why we additionally use the Jaccard index as well.

File Set Comparison. In the next step, we compare the files in the clusters of the artifact network to the files committed by developers in the clusters of the developer network. To do this, we apply the Jaccard index $J(A_i, \chi_{file}(D_j))$ and the overlap coefficient $overlap(A_i, \chi_{file}(D_j))$ to every possible combination of $i \in \{1, \dots, k\}$ and $j \in \{1, \dots, l\}$. Since, unlike in the case of comparing sets of developers, we are equally interested comparing the sets in both directions, we use the overlap coefficient, which is symmetric, instead of the unsymmetric completeness to compare sets of files. Using this comparison, we can analyze which parts of the project the files committed by a cluster of developers in the communication network correspond to. The results we obtain from this comparison mostly align with the comparison of sets of developers, but we still include it as the results may provide information that we might miss otherwise.

Committed Files Comparison. Finally, we compare the files committed by different clusters in the developer communication network to each other. For this part of the comparison, we only compute the overlap coefficient $overlap(\chi_{file}(D_i), \chi_{file}(D_j))$ for every possible pair of clusters in the developer network. The overlap coefficient gives us an indication of how strongly the parts of the project that different clusters in the developer communication network commit to overlap with each other. For two different clusters, this can lead to two particularly interesting cases:

- If the overlap coefficient is close to zero, this means that the developer clusters mostly work on different parts of the project.
- If the overlap coefficient is close to one, there are two possible implications. If one of the clusters committed changes to substantially more files, the other cluster works on only a subset of the parts of the project that the first one works on. If both clusters have edited a roughly similar number of files, they work on almost the same part of the project.

Active developers have to make changes to central parts of the project from time to time, independent of what other parts of a project they work on. This leads, at least, to values above zero for most of the compared developer clusters, but will in most cases not be enough overlap to lead to values close to one.

We do an empirical analysis of the results of these comparisons. The results of this analysis are presented in Section 4.2. Additionally, we discuss the possible implications of our results in Section 4.3.

3.5 IMPLEMENTATION

Our workflow to analyze networks based on OSS projects starts by first collecting the data from six different GitHub projects. After that, we process the data, filter it and use it to build networks. Finally, we run several analysis methods on the resulting networks. In this section, we give a summary of the software we use in our workflow.

3.5.1 *Data Extraction*

We extract the data from GitHub using CODEFACE-EXTRACTION[12], which is an extension to the Codeface[13] framework. The Codeface framework is a tool used to build a database consisting of metadata from a version control system such as GitHub. Codeface is able to process data from commits, and e-mails, but is unable to extract issue data. The issue data is extracted using GitHubWrapper[14], which uses GitHub's REST API. CODEFACE-EXTRACTION extracts the data from the database built by Codeface into a comma-separated list format, which makes processing it with our code easier. Additionally, it unifies the extracted commit and e-mail data with the issue data extracted by GitHubWrapper. The commit data is directly taken

12 https://github.com/se-sic/codeface-extraction
13 http://siemens.github.io/codeface/icse2017/#/home
14 GitHubWrapper is not yet officially released, but will be in the coming months.

from the official GitHub page of a project and includes all commits that were ever made to the extracted repository. The extracted data includes, for each commit, the timestamp of the commit, the author and their e-mail address, the hash, and the files and functions changed. The commit messages are also extracted, although we do not use them. The issue data is also directly taken from the official GitHub page of a project. It consists of all issues and pull requests including every event happening to issues such as comments, an issue being locked, or a pull request being accepted. The extracted issue data includes the number and titles of all issues as well as every event happening in every issue including the user that started the event, their e-mail address, the type of the event (e.g. a comment or opening an issue), its timestamp, and possibly some additional data depending on the type. Additionally, the data includes whether an issue is "open" or "closed".

The extracted data already already been prefiltered during the extraction process, which includes disambiguation of the author names and their e-mail addresses by using a heuristic by Oliva et al. [31] as well as filtering out bots. This prefiltering is not enough in our case, as the resulting networks still contain too many superfluous relationships, so we limit the sizes of commits and remove users that are not necessary for our analysis from the issue data (see section 3.3.2).

3.5.2 *Processing the Data*

To process and analyze the data we extract from Codeface, we use the R programming language[15], which is specifically designed for statistical computing. Additionally, we use the Coronet[16] library. Coronet provides functionality to build and analyze networks based on the data extracted from codeface. It is based on igraph[17], a popular network analysis library for R and several other programming languages. In our case, Coronet uses the commit data to build artifact networks and the issue data to build developer communication networks.

3.5.3 *Network Analysis*

The artifact networks we build with Coronet are first analyzed using the integrated graph metrics of igraph and Coronet. In this first analysis step, we take a look at the characteristics of artifact networks (see Section 3.4.1). After that, we use the Louvain clustering algorithm that is already implemented in igraph on both the artifact networks and the developer communication networks (see Section 3.4.2). The resulting clusters in the two different network types are then converted into lists of sets that can be compared to each other (see Section 3.4.3).

15 https://www.r-project.org/
16 https://github.com/se-sic/coronet/
17 https://igraph.org/

4 EVALUATION

In this chapter, we present the results of our analysis of artifact networks as well as the results of our comparison of clusters found in artifact networks and developer communication networks. Furthermore, we discuss the results and the answers they provide to our research questions. Then, we take a look at internal and external threats to the validity of our findings.

4.1 RESULTS: ARTIFACT NETWORK CHARACTERISTICS

In this section, we take a look at the characteristics of artifact networks that we constructed from the commit data of the projects presented in Section 3.2 and how these characteristics evolve over time.

We evaluate each metric used in our analysis separately. For each metric, we first present the general results across all projects, before pointing out projects which have differing results from the rest and giving possible explanations for these outliers. It is important to note that all the results in this section are computed on artifact networks from which isolated vertices were removed beforehand since we use the same networks to compute clusterings later, in which isolated vertices would lead to a large overhead of clusters that only consist of a single file. Since, depending on their quantity, isolated vertices can have a large impact on metrics such as clustering degree and density, not all of our results are generalizable to artifact networks that contain large numbers of isolated vertices. The evolution of most of the metrics that we evaluate in this section is depicted in Figure 4.1.

Table 4.1: Average and standard deviation of the numbers of vertices and edges across all artifact networks for each project. All values are rounded to two decimal places.

Project	# Vertices		# Edges	
	Avg.	SD	Avg.	SD
Deno	627.30	165.03	4867.50	1393.82
OpenSSL	233.42	147.17	1157.86	993.09
Atom	192.89	129.47	1089.43	1050.73
TypeScript	159.28	44.69	1278.24	544.75
Nextcloud	1112.17	475.64	7466.24	3381.94
Moby	767.00	405.06	5188.74	3595.62

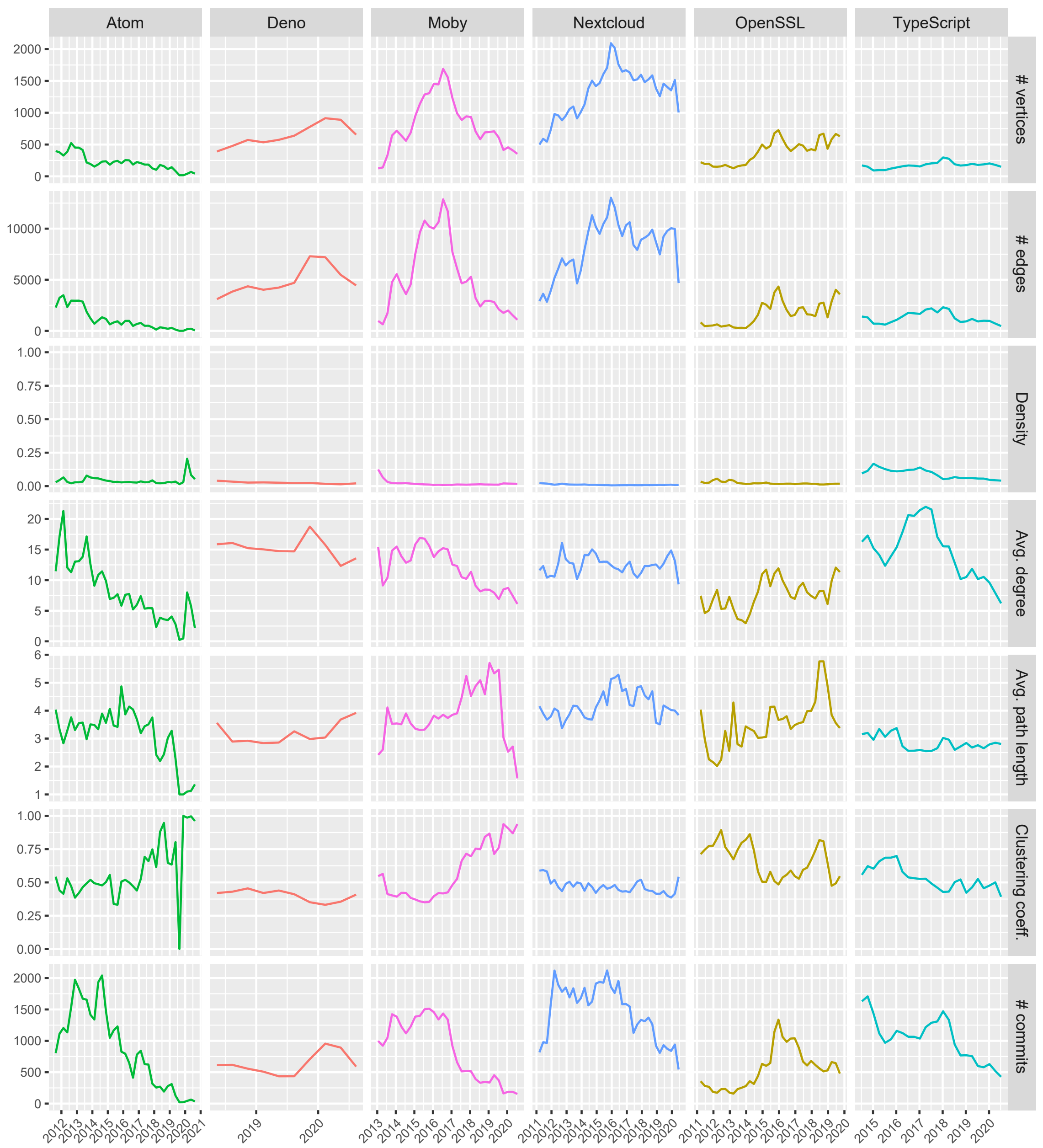


Figure 4.1: The figure shows the evolution of the number of vertices (i.e., the number of files), the number of edges (i.e., how often multiple files are committed together), the density, the average degree, the average path length, and the clustering coefficient across the artifact networks built from the projects in our sample set. Additionally, we provide the number of commits that the artifact network for the respective observation window is built from. The characteristics of the artifact networks built from observation windows earlier than 2011 from Nextcloud and OpenSSL are excluded to improve readability.

Table 4.2: Average values and standard deviation of different metrics across all artifact networks for each project. All values are rounded to two decimal places.

Project	Density		Avg. degree		Avg. path length		Clustering coeff.	
	Avg.	SD	Avg.	SD	Avg.	SD	Avg.	SD
Deno	0.03	0.01	15.55	1.65	3.20	0.39	0.40	0.04
OpenSSL	0.05	0.02	8.79	2.75	3.37	0.82	0.71	0.12
Atom	0.08	0.10	9.19	4.73	3.11	0.96	0.58	0.22
TypeScript	0.11	0.04	9.95	0.92	3.50	0.28	0.33	0.10
Nextcloud	0.02	0.04	12.97	2.16	4.05	0.66	0.49	0.09
Moby	0.02	0.03	12.60	3.21	3.83	0.97	0.58	0.20

The *number of vertices* in an artifact network represents the number of different files that were edited during the observation window that the network is built from. According to our findings, both the number of vertices and the *number of edges* depend heavily on the number of commits during that observation window. However, the number of vertices is capped by the number of files in the project, so there often are multiple commits involving the same files. As can be seen in Figure 4.1, the number of vertices and edges seem to not be correlated to the number of commits during the early stages of the development of a project, which is most likely due to many commits editing the same files at that point in time. However, in later observation windows, we can observe an almost linear relationship between the number of vertices and edges and the number of commits in an observation window. This is especially apparent in the second half of the observation periods of Deno, Moby, Nextcloud, and OpenSSL. Unfortunately, this relationship is often not consistent enough to use the number of edges and vertices as an indicator for developer activity, as can be seen in the first halves of the observation periods of the same projects. Because of the correlation between developer commit activity and the number of edges and vertices, we can expect a strong fluctuation of these values over time. The averages and standard deviations of these values for each of the projects in our sample set listed in Table 4.1 match our expectations, as especially the number of edges has a very high standard deviation due to the fluctuations in developer commit activity that can be expected in projects that have been running for multiple years.

Artifact networks are usually sparse, with most of the networks in our sample set having a *density* between 0.5% and 5%. Additionally, the density is mostly very stable across all artifact networks for a project, resulting in standard deviations of around one or two percent. The density of an artifact network does not heavily depend on the number of commits it is built from or the size of a project, as we can see from the densities in Figure 4.1 being quite stable despite the projects in our sample set featuring very different numbers of commits. However, if a project has a low development activity during a certain observation window, the resulting artifact network has a higher density, as both the number of edges and the number of vertices have a roughly linear relationship to the number of commits, but the density is defined as $\frac{2 \cdot |E|}{|V| \cdot (|V|-1)}$. Since the density has an inversely proportional relationship to the square of the number of vertices, this leads to the density being higher for observation windows with low commit count. Most artifact networks have an average density of around 3%, with the exceptions to this being the networks built from TypeScript and Atom. The high density of the artifact networks that are built from commit data taken from TypeScript is a result of filtering out unit tests. The source code of TypeScript contains a very large amount of unit tests. The *tests* folder in the root of the TypeScript repository contains more than 40000 files. Unit tests are often committed alone or together with a small number of other unit tests, which leads to them having a comparably low degree. Due to a large number of unit tests in the commits extracted from TypeScript, the resulting artifact networks were initially very sparse and hard to use for further steps of our analysis, so we filter out all unit tests for TypeScript. This leads to a higher density of the artifact networks built from TypeScript than those built from other projects which still contain unit tests. The high average density and high standard deviation of the networks built from Atom can be explained by its strongly diminishing developer activity at the end of our observation period, which also leads to substantially fewer commits per observation window and therefore to networks with few vertices and a high density. If we exclude the last two years of our observation period for Atom, i.e., we exclude the last seven observation windows, the densities of the artifact networks built from Atom only have a standard deviation of around 2%, which is a lot more similar to other projects. We detect this behavior across all of our results for Atom.

The *average degree* of artifact networks fluctuates depending on the number of commits, file structure, and number of files per commit of a project. While it usually stays between 5 and 20, the average degrees of the artifact networks we analyzed had nothing else in common, as even artifact networks for different observation windows of the same project can have very different average degrees. We were unable to find any commonalities in the evolutions of the average degree over time across different projects.

Unlike the average degree, the *average path length* is very stable across all artifact networks, averaging around 3 to 4. Our set of artifact networks contains no major exceptions when it comes to average path lengths, with the highest average path lengths we found being below 6 and the lowest ones being around 2. The average path lengths of similarly sized random networks that we built using the Erdős-Rényi method [14] yield average path lengths of around 2 to 4, depending on the number of vertices of the artifact network, so we conclude that the average path length of artifact networks is similar to the expected average path length for networks of that size.

The *clustering coefficient* is also stable across all artifact networks for most projects, with the exceptions being Atom and Moby. For both of these projects, there are some small networks with exceptionally high clustering coefficients towards the end of our observation period because of diminishing developer activity. However, the average clustering coefficient differs heavily between different projects. We attribute these differences to the different ways that projects are structured, which also leads to different structures in the artifact networks. The source code of TypeScript, for example, contains a large number of unit tests, which lead

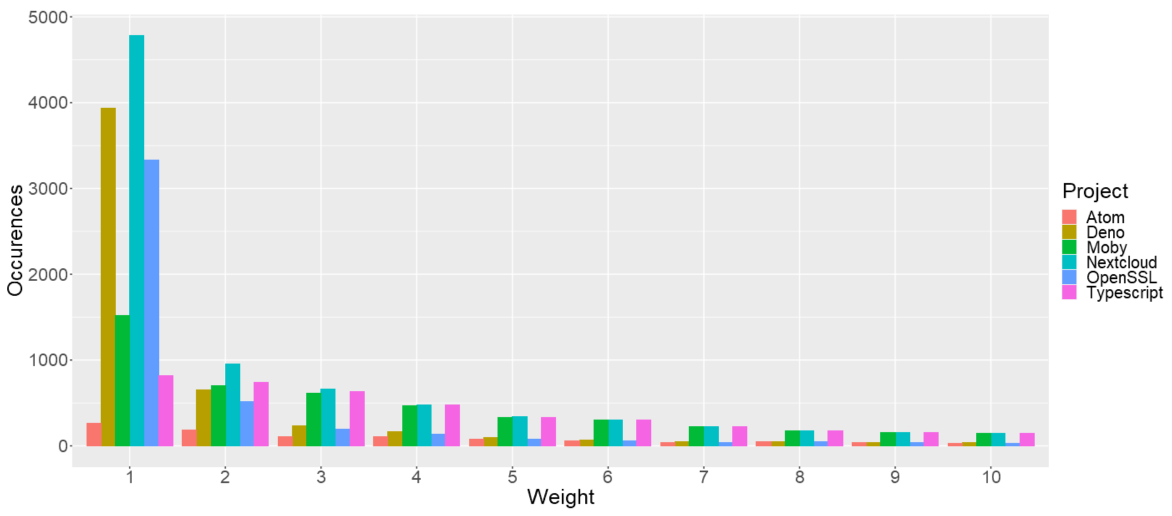


Figure 4.2: The figure shows the distribution of edge weights across all networks for each of the six projects in our sample set. Edge weights greater than ten are excluded from this figure, as they occur even less frequently than edge weights of ten.

to a significantly higher vertex count and more spread-out artifact networks than the other projects. Since unit tests in TypeScript are often committed together with only a very low number of other files, which are either the files whose functionality the tests are supposed to test or other unit tests, a large number of unit tests leads to less densely connected clusters and a lower clustering coefficient. OpenSSL, the project with the highest clustering coefficient, contains no dedicated files for unit tests in its main repository and is functionally divided into multiple modules of different functionality. Because of the structure of OpenSSL, which is comprised of several modules that are largely independent of each other (e.g., modules for different kinds of encryptions), commits frequently contain files that belong to the same module, which leads to very strongly connected clusters. In general, the artifact networks built from a project's commit data tend to form clusters. The quality of these clusters, i.e., their modularity, depends on the structure of the source code as well as on the number of auxiliary files such as unit tests or external dependencies that are committed in addition to the regular source code. We discuss the implications of these clusters at the end of this section.

Due to the low average path length and a comparably high clustering coefficient, we found that the artifact networks for all of the projects in our sample set are small-world networks. The only exceptions to this are two networks at the end of our observation period in Atom, which were both built from only two commits each. As such a low number of commits over the course of six months is very untypical for OSS projects, we conclude that artifact networks are small-world networks.

Before we take a look at the clusters we can detect in artifact networks using the Louvain clustering algorithm, we present two more interesting characteristics of artifact networks that we investigated during our initial investigation, but that we are not using for the analysis of our second research question. First, we evaluate the *edge weights* in artifact networks. The weight

Table 4.3: The 10 hubs with the highest degree in 3 artifact networks built from different observation windows of Deno. The set of the highest degree artifacts changes significantly over the course of just two years.

2018-05-13 - 2018-11-12		2019-05-13 - 2019-11-12		2020-05-13 - 2020-11-12	
Name	Deg	Name	Deg	Name	Deg
src/ops.rs	110	cli/lib.rs	125	cli/tests/integration_tests.rs	253
js/main.ts	98	cli/state.rs	122	cli/main.rs	119
js/unit_tests.ts	89	cli/ops.rs	120	cli/tsc.rs	108
src/handlers.rs	83	cli/worker.rs	108	cli/module_graph.rs	102
js/deno.ts	79	cli/ops/compiler.rs	75	cli/worker.rs	100
src/main.rs	77	js/compiler.ts	75	cli/rt/99_main.js	84
js/globals.ts	72	cli/deno_error.rs	70	cli/flags.rs	80
js/compiler.ts	70	js/dispatch.ts	69	cli/web_worker.rs	78
js/compiler_test.ts	64	cli/ops/workers.rs	67	cli/tsc/99_main_compiler.js	77
src/binding.cc	56	cli/flags.rs	64	cli/ops/worker_host.rs	70

of an edge between two artifacts represents the number of times these two artifacts are committed together during the observation window that the artifact network is built from. We expect that edge weights provide a way to analyze functional dependencies between files, as we expect files that are functionally dependent on each other to be committed together more frequently than those that do not contain any functional dependencies. In reality, we found that most of the edges in an artifact network have very low edge weights, which means that most files are only committed together one or two times during a six-month period. There are some exceptions to this as files that are central to a project are usually committed together very frequently. By looking at some samples of high edge weights, we could verify that high edge weights indeed seem to indicate functional dependencies between files, as many of the files that are connected by high edge weights are dependent on each other in obvious ways, e.g., subclasses or one file directly referencing code from the other file. However, the number of files that have very low edge weights in between them is substantially larger, with less than 10% of the edges in artifact networks having weights of 3 or higher. The distribution of edge weights for each of our sample projects is depicted in Figure 4.2.

We investigate the ten *hubs* with the highest degrees in different artifact networks. As can be expected, files that are hubs in an artifact network serve a central role in the development of a project during the observation window that the network was built from. We verified this by looking at different samples across all of our sample projects. Since we are not familiar with the source code of those projects, we cannot say with certainty that all hubs serve a central role in the source code, but we expect them to do so as we could verify the role of

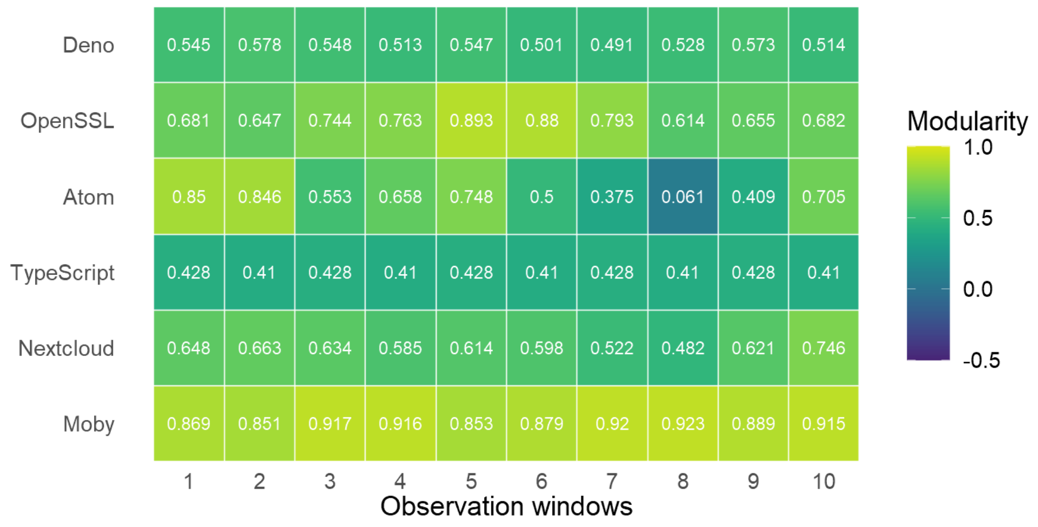


Figure 4.3: The modularity of the clusterings computed using the Louvain algorithm on the last 10 artifact networks for each project. The results for previous observation windows are similar to the displayed ones but are left out to increase readability. The low modularity in the observation windows 7, 8, and 9 of Atom is a result of low commit activity and therefore very small networks.

many hubs by looking at the names of the files, and the functionality present in those files. Surprisingly, the set of files that are hubs constantly changes, with files rarely being in the ten highest-degree hubs for longer than a year. An example of the evolution of hubs for the DENO project is listed in Table 4.3. Our findings lead us to the conclusion that hubs in artifact networks serve as a centrality measure to determine which files are important in the development of a project at a certain point in time. However, we only did a rough analysis of hubs and did not evaluate the accuracy of using hubs as a centrality metric, so the role of hubs in artifact networks is a topic for future research.

4.2 RESULTS: CLUSTERINGS

Before we present the results of our cluster comparison, we take a look at the clusters we detect in artifact networks and in developer communication networks using the Louvain clustering algorithm. After that, we present the general results of our cluster comparison across all projects. Finally, we take a look at the differences between the results for each of the projects in our sample set and the general results.

4.2.1 *Clusterings of Artifact Networks*

Our classification of artifact networks as small-world networks already gives us a good indication that artifact networks might contain detectable clusters. Additionally, we expect clusters of files that have a functional relation to each other to be detectable in artifact networks, as we expect files with such relations to be committed together more frequently than unre-

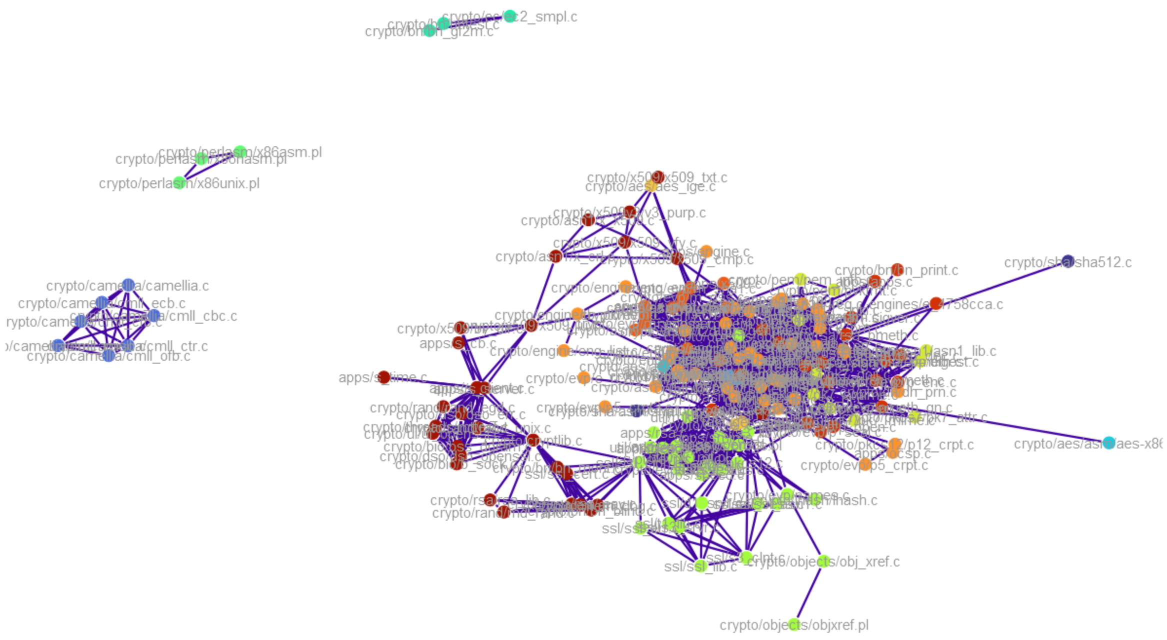


Figure 4.4: The figure shows a clustering we computed for an artifact network built from the commit data of OpenSSL (2006-03-22 - 2006-09-21). The vertices in different clusters have different colors. On the right, we can observe that multiple clusters are too densely connected to make out individual clusters from the plot. However, we can clearly see that the less densely connected parts of the network form clusters which have a high internal density and only a low number of edges to other clusters.

lated files. As an indicator for the accuracy of our computed clusterings in artifact networks, we compute the modularity (see Section 2.2.3) of each clustering. The resulting modularities, which are depicted in Figure 4.3, are typically above 50% (with the exception of TypeScript) and are even higher in many cases, which indicates that the detected clusters have significantly higher modularity than those we would detect in an arbitrary random graph, as a random graph would have a modularity score that is close to zero. We conclude that artifact networks built from commit data form clusters with strong internal connections. Since most artifact networks are quite large, they are relatively difficult to visualize. However, we provide a visualization for a comparably small artifact network from the OpenSSL project in Figure 4.4.

To evaluate the real-world implications of the detected clusters, we take a look at how the files that form a cluster are functionally related to each other. Since we are not familiar with the source codes of the projects in our sample set, we use examples that are simple enough to analyze. Due to the nature of OSS projects which should ideally be structured in a way such that new developers are quickly able to identify structures in the source code, we can often determine whether files in a cluster are functionally related by looking at filenames, function names or the name of the folders that they are in. While this way of determining whether clusters represent a group of functionally-related files or not has its flaws, for example, we might misclassify files that have ambiguous names or names that we as outsiders of a project do not understand, we deem our method good enough as we have a large sample set and can therefore generalize from the many examples that we can understand. We found that most clusters indeed consist of files that are very closely related to each other. Smaller clusters usually consist of files from the same module. In this thesis, *module* refers to a logical building block of a software project [16]. A module is often comprised of multiple files and can even contain multiple smaller (sub-)modules. Examples of modules can be utility files, input-output, server communication, and more. Unit tests also often form their own clusters, although they can also end up in clusters with the parts of the project that they are supposed to test as well. The central files in the development of a project usually form one or multiple larger clusters, depending on the project size. These large clusters often contain multiple modules at once, as there are too many edges between the most actively edited files of a project to divide them into multiple clusters using the Louvain algorithm. Because of the previously listed characteristics, we conclude that clusters in artifact networks provide a method to detect modules in OSS projects, which allows us to use these clusters to compare them to clusters in the developer communication networks in order to find out which parts of a project the different clusters of developers work on. However, this method of detecting modules in the source code has its flaws, which we discuss in Section 4.3.

4.2.2 *Clusterings of Developer Networks*

When applying the Louvain clustering algorithm on developer communication networks and doing an empirical study of the clusters, we found that most of the clusters found in issue-based developer communication networks fit at least one of the following three descriptions:

Type 1. The most active developers of a project communicate with more other developers than less active ones, which leads to these developers often forming dense clusters.

Type 2. Less active developers often form a cluster around one or multiple maintainers, as maintainers are the only group of developers that can accept or reject pull requests. In extreme cases, this can lead to clusters in which every developer that is not a maintainer has exactly one edge, which connects him to a maintainer. This case occurs if pull requests from less active developers only involve the developer that opened the commit and the maintainer that accepted or rejected it, with less active developers not commenting on any issues other than their own.

Type 3. Formerly active developers sometimes open issues or comment on them even if they do not commit changes to a project any longer. If the topic of such an issue starts a discussion among multiple developers that do not comment on any other issues, we obtain a small cluster in the developer communication network which only consists of the developers who commented on this exact issue.

There are also clusters that are a combination of these descriptions, for example, a dense cluster that mostly consists of very active developers might also contain some less active ones that only communicated with a single developer in the cluster. The listed characteristics are

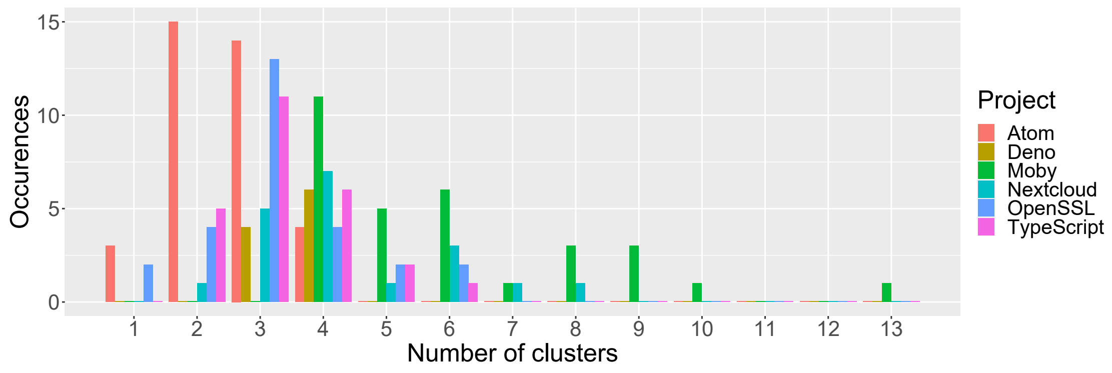


Figure 4.5: The amounts of clusters detected in developer communication networks across all observation windows.

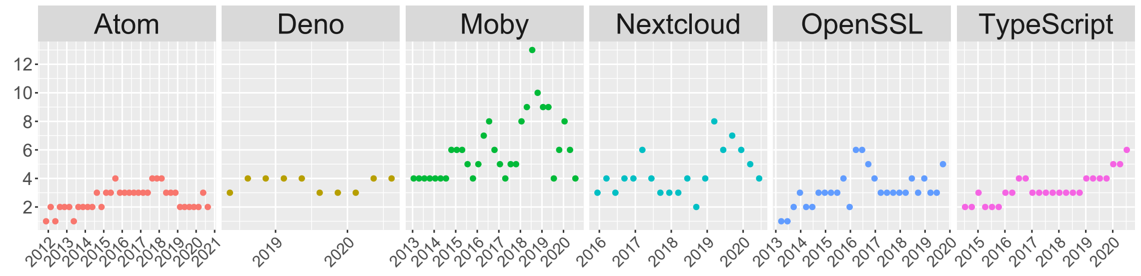


Figure 4.6: Evolution of the number of clusters detected in developer communication networks over the observation period for each project.

recurring patterns across most of the developer communication networks in our sample set and also play a significant role in the interpretation of our results, as they can be used to explain some of the patterns we detect during our cluster comparison. The number of clusters we detected across all developer communication networks of a project is shown in Figure 4.5. Additionally, the evolution of the number of clusters for each observation window over the course of the entire observation period for each project is shown in Figure 4.6.

In some cases, the number of detected clusters is only one or two. These cases often occur in the 9 to 18 months after the introduction of the issue system to a project, as there are not enough developers that use issues from the start to compute a clustering that is useful for our analysis. Additionally, we encountered these cases when a project was inactive over the course of an observation window, as, for example, during the last observation windows of Atom. We exclude all observation windows with developer communication networks in which only one or two clusters are detected from the comparison to clusters in artifact networks since most of these clusterings have very low modularity, which implies that they hardly convey any real-world meaning.

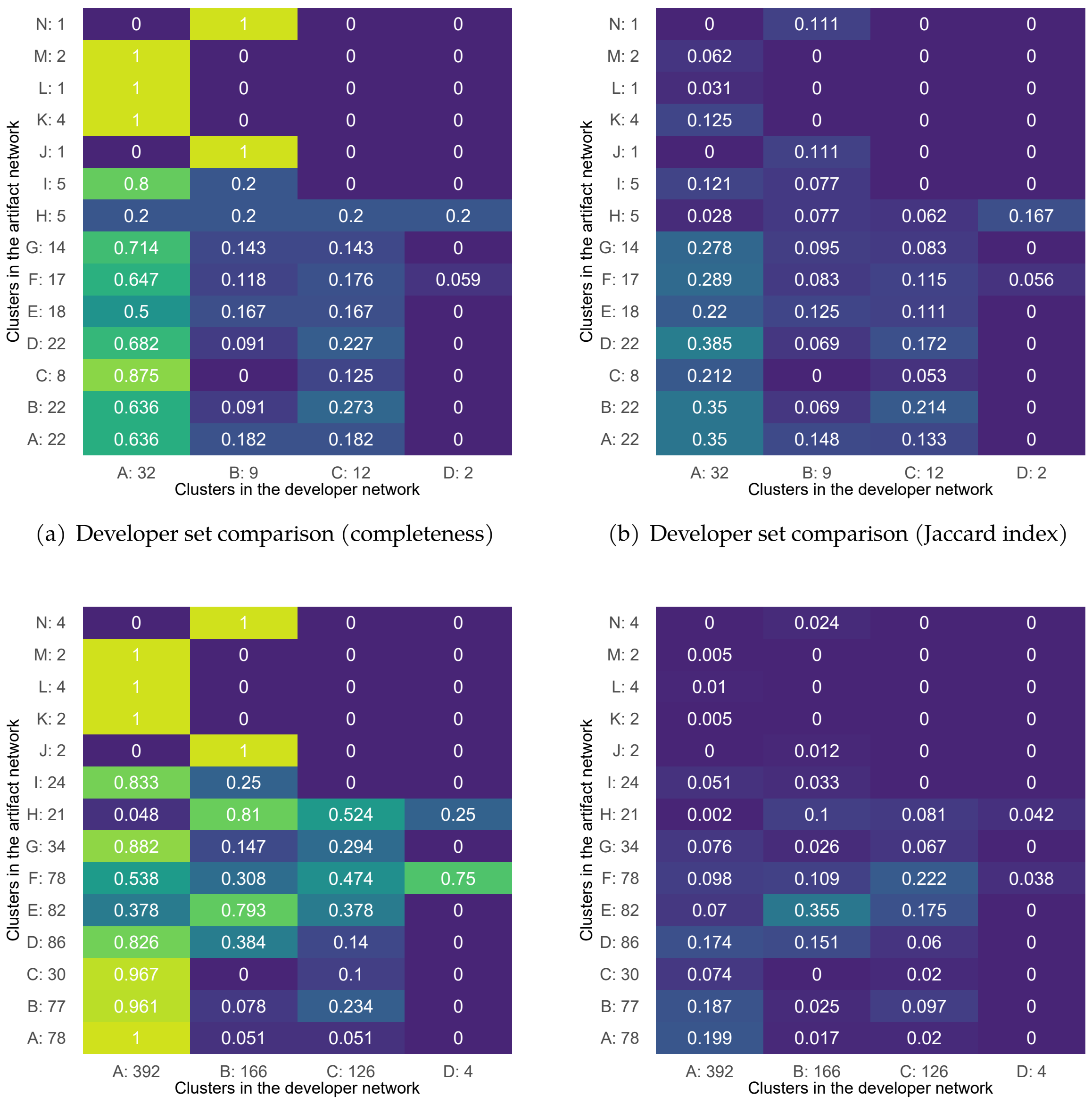
Most of the developer communication networks in our sample set are partitioned into three to five clusters of different sizes. Compared to the clusters in artifact networks, for which we sometimes detected more than 30 different clusters, the number of clusters in developer communication networks is very low. The low number of clusters is also a byproduct of the way the Louvain algorithm clusters vertices with degrees of one or two. While other clustering algorithms such as the Walktrap algorithm [34] often yield clusters that contain only one vertex with a degree of one or two, the Louvain algorithm tends to include these vertices into the clusters of the vertices that they are connected to. The vertices in clusters that fit our second type of clusters in developer communication networks would get split up into a large amount of very small clusters if we used a different algorithm to generate them. However, we prefer the way in which the Louvain algorithm clusters these cases, as small clusters are more likely to be overfitting to our sample data than larger clusters. Even though most developer networks do not contain more than five clusters, there are exceptions to this. These can occur in observation windows with larger numbers of issues and developers that interact with these issues, so we expect the number of clusters to be higher for more popular projects than the ones in our sample set.

4.2.3 *Cluster Comparison: General Results*

Next, we present the results of our cluster comparison. The results for the projects in our sample set have many commonalities, so we present the common results together. First, we extend our previous interpretation of artifact clusters from our analysis of artifact networks by looking at how many developers usually commit to which artifact cluster. Then, we provide results for the developer set comparison and the file set comparison for each project. Furthermore, we compare the sets of files comitted by different clusters in developer communication networks and analyze how strongly they overlap. Finally, we take a look at which exceptions to our general results we found in each project.

By looking at how much the developers of different clusters commit, we found that the developers included in a single cluster in a developer network are roughly grouped by their commit activity, even though the networks are built from issues. This indicates that developers tend to communicate with other developers that are equally as active as they are. However, there are some exceptions to this, e.g., a cluster that fits our Type 2 of clusters in developer networks will usually contain a lot of less active developers while the maintainer in the center of the cluster is a lot more active than the rest. Similarly, the most active developer clusters in terms of commits often contain a small number of less active developers.

Now, we take a look at the results of the developer set comparison and file set comparison, which we present together as they yielded similar results. We provide examples of the results of the developer set comparison and the file set comparison in the form of similarity matrices for one observation window per project. Unfortunately, we cannot include such an example for NEXTCLOUD, as the high number of clusters found in the artifact networks of NEXTCLOUD makes similarity matrices hard to read when printed on a regular-sized page. However, NEXTCLOUD has similar results to the rest of our sample projects, and the examples provided in this chapter should still contain enough information to generalize from them. The examples for the other projects are shown in Figure 4.7 (DENO), Figure 4.8 (OPENSSL), Figure 4.9 (ATOM), Figure 4.10 (TYPESCRIPT), and Figure 4.11 (MOBY). In the figures, each



(a) Developer set comparison (completeness)

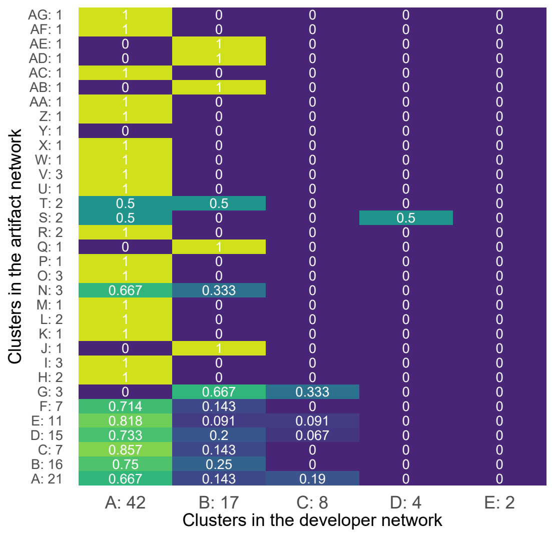
(b) Developer set comparison (Jaccard index)

(c) File set comparison (overlap coefficient)

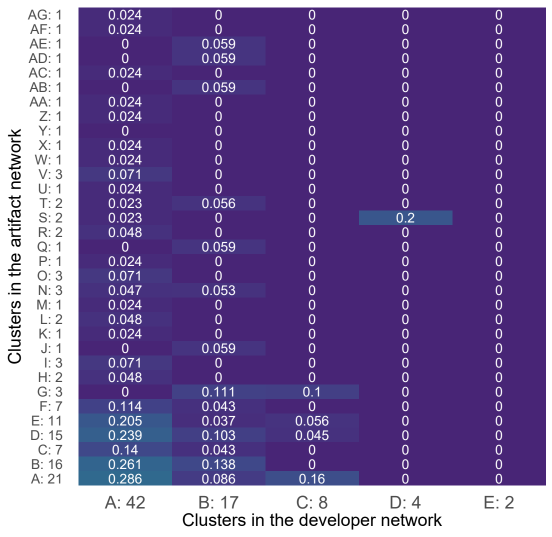
(d) File set comparison (Jaccard index)

Figure 4.7: Results of the cluster comparison for Deno for the observation window from February 2019 to August 2019.

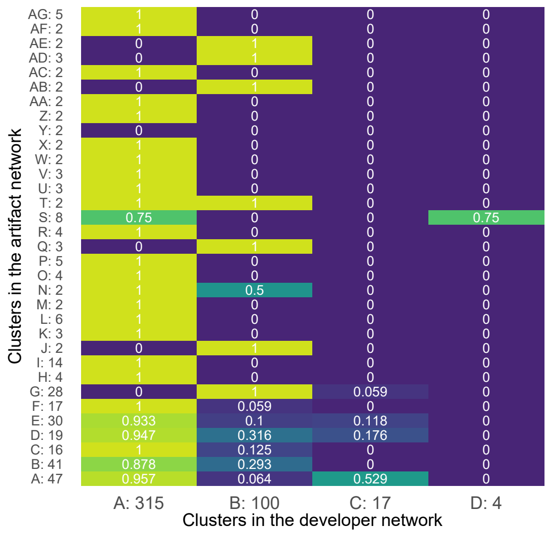
cluster is labeled with a letter and a number. The number represents the number of developers for the developer set comparison and the number of files for the file set comparison that is associated with the respective cluster. Each of the depicted observation windows yielded results that were comparable to most of the other observation periods of a project and are therefore representative of the results for the entire project that they are taken from. Further examples are provided in the Appendix A, where we provide results for one observation window in the early stages of development and one observation window near the end of our observation period for each project.



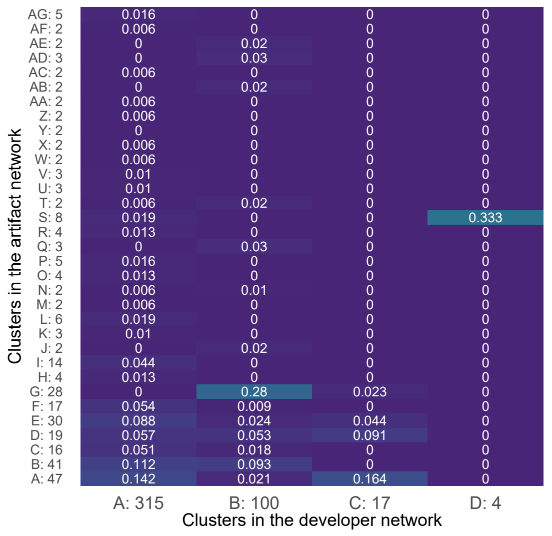
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)



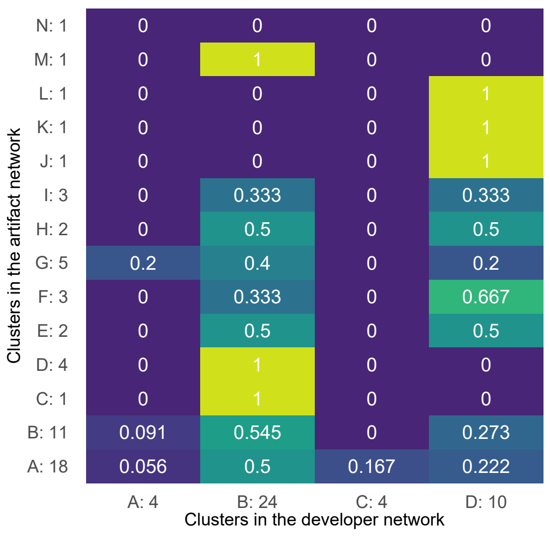
(c) File set comparison (overlap coefficient)



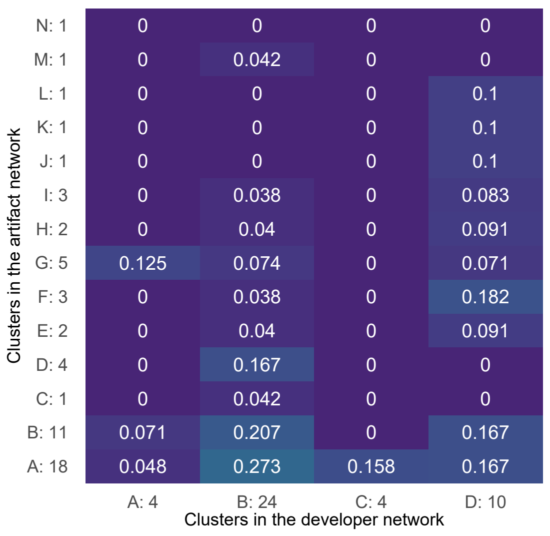
(d) File set comparison (Jaccard index)

Figure 4.8: Results of the cluster comparison for OpenSSL for the observation window from September 2016 to March 2017.

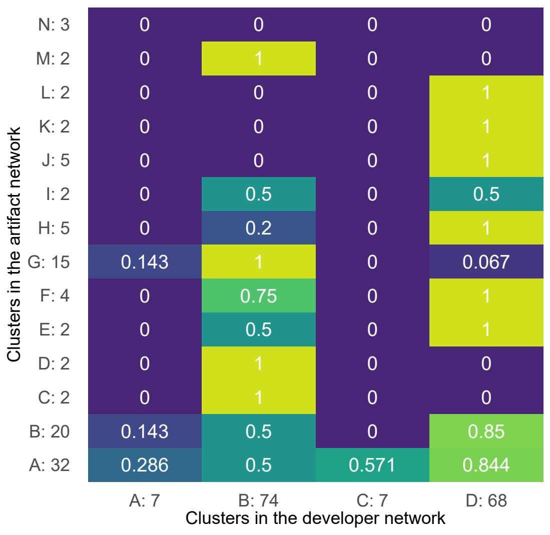
We describe our results of the comparison for project Deno, which are displayed in Figure 4.7 in detail, but we cannot do this for all of our results, as there are too many observation windows to analyze. However, the description of our comparison for Deno can be applied to the results of the other observation windows and projects as well. As we can see in both the completeness and the Jaccard index of the developer set comparison, cluster A in the developer network has the highest similarities to most of the clusters in the artifact network, with



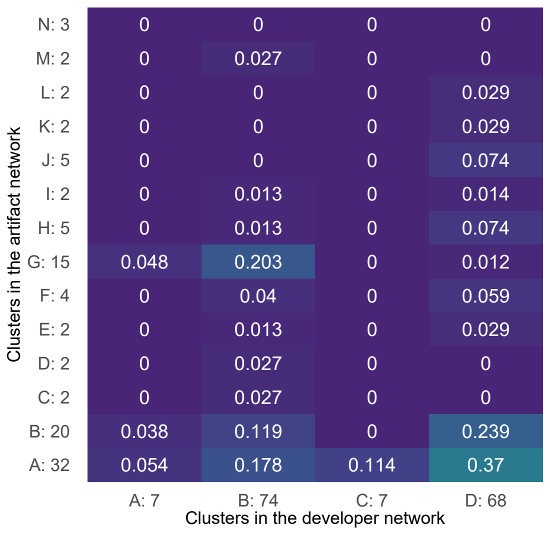
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)



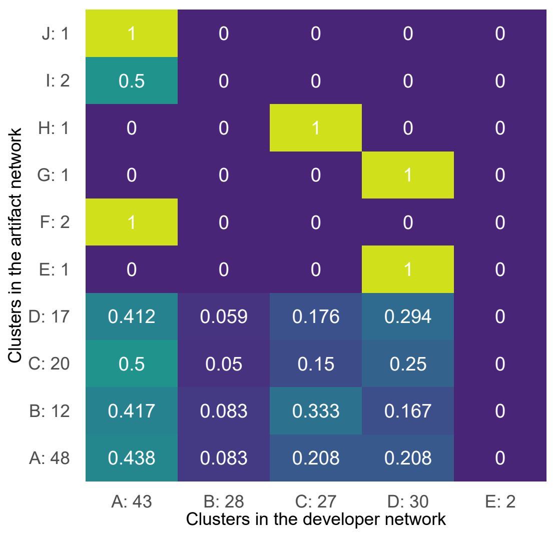
(c) File set comparison (overlap coefficient)



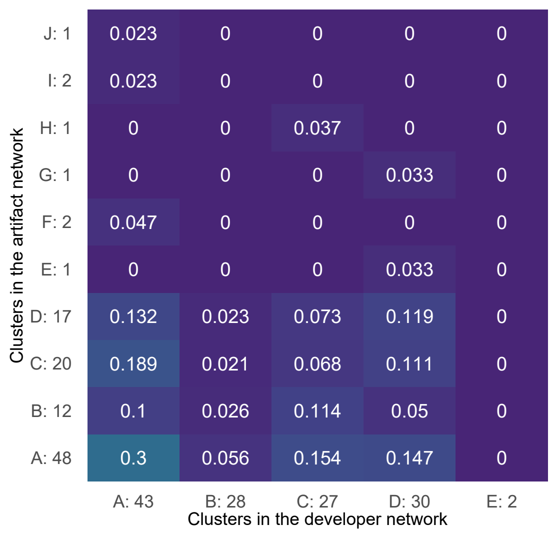
(d) File set comparison (Jaccard index)

Figure 4.9: Results of the cluster comparison for Atom for the observation window from February 2018 to August 2018, which is one of the last observation windows in which Atom still had an actively committing developer community.

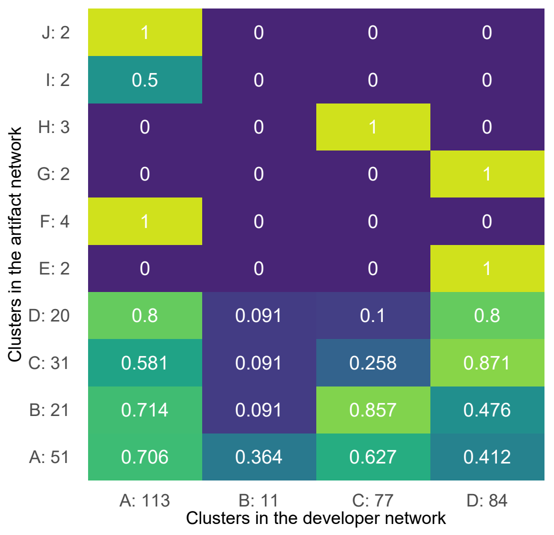
the completeness being between 0.5 and 1, except for the outliers H, J and N, and a Jaccard index of above 0.2 for artifact clusters A to G. The significantly lower values for the Jaccard index compared to the completeness are a result of the definition of these two metrics, as the Jaccard index divides the number of developers present in both clusters by the total number of developers in the union of both clusters, while the completeness only divides it by the number of developers that worked on an artifact cluster. Since the Jaccard index will always be relatively low, if the size of the clusters we compare is very different, we often rely on the



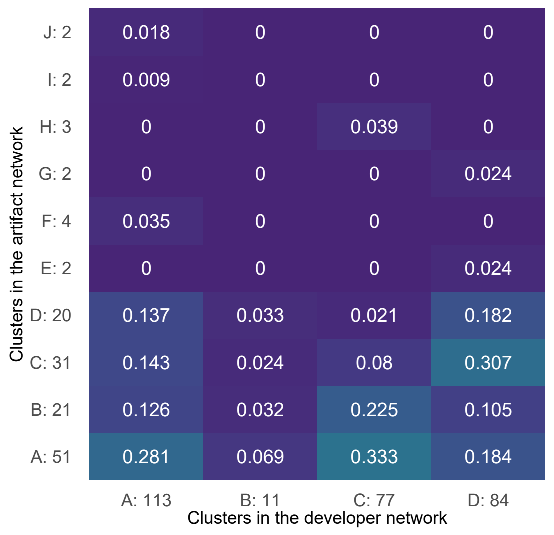
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)



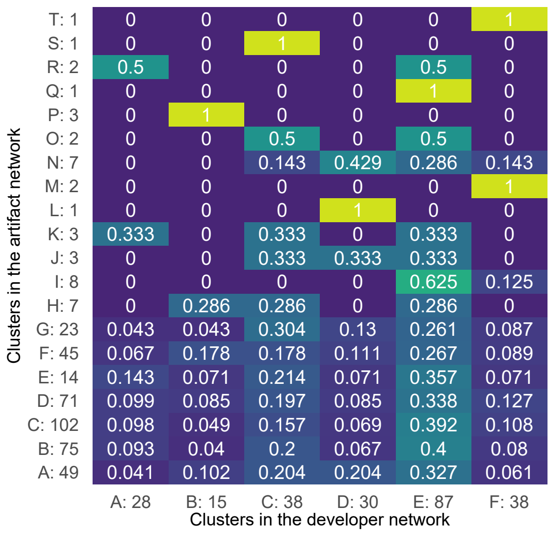
(c) File set comparison (overlap coefficient)



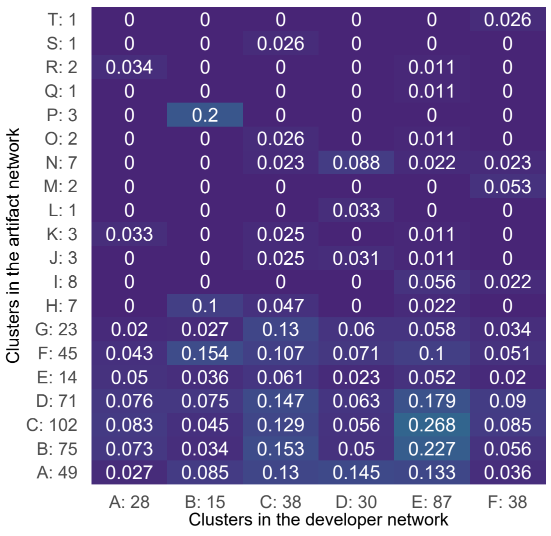
(d) File set comparison (Jaccard index)

Figure 4.10: Results of the cluster comparison for TypeScript for the observation window from April 2020 to October 2020. Unit tests were removed from the commit data before building the artifact networks for TypeScript.

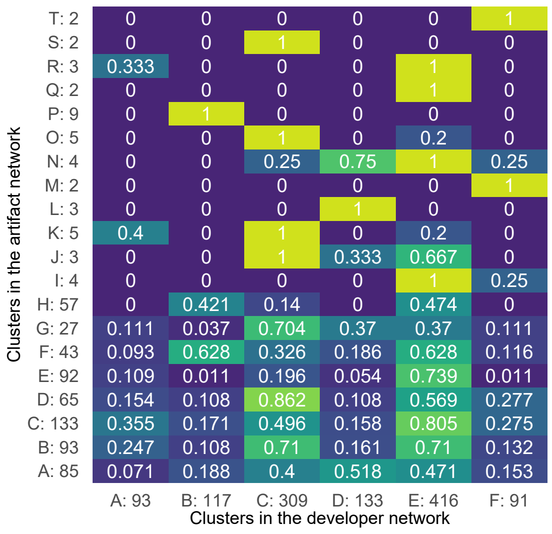
completeness during our analysis. The high completeness values for A indicate that a majority of the developers working on artifact clusters are grouped in developer cluster A. For example, the completeness of artifact cluster C and developer cluster A tells us that 87.5% (7 out of 8) of the developers working on C are in developer cluster A. Additionally, the overlap coefficient in the cluster file comparison shows that the set of files edited by developer cluster A has a high overlap with the sets of files in most artifact clusters except for E, H, J, and N. This lets us conclude that the developers in cluster A do not only make up a majority



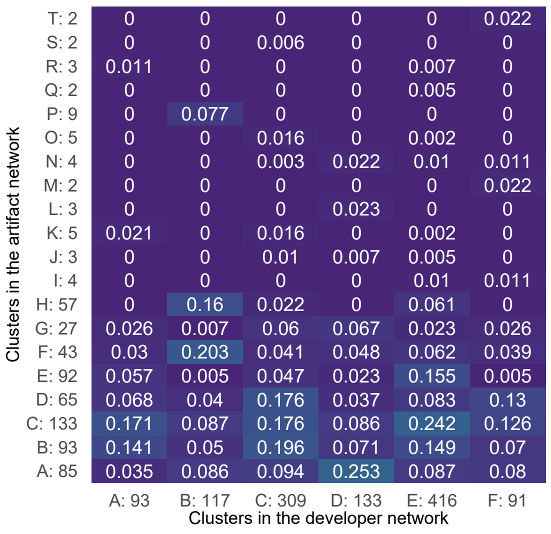
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)



(c) File set comparison (overlap coefficient)



(d) File set comparison (Jaccard index)

Figure 4.11: Results of the cluster comparison for Moby for the observation period from October 2014 to April 2015. Each cluster is labeled with a letter and a number.

of the developers working on many artifact clusters but that they are also responsible for the majority of changes made to these clusters. For the rest of our results, we refer to clusters like *A*, who are actively working on many parts of a project at the same time, as *central clusters*, as they serve a central role in the developer community of a project since they contribute a majority of the changes. We encounter central clusters in the results of all projects, e.g., *A* in Figure 4.8, *B* and *D* in Figure 4.9, *A* and *D* in Figure 4.10, and *E* in Figure 4.11.

It is immediately apparent that developer cluster *B* in Figure 4.7 has lower similarity values than *A* for most of the developer set comparison, except for clusters *J* and *N* who have been edited by only one developer each and therefore have a completeness of 1. However, the overlap coefficient of 0.793 to artifact cluster *E* and 0.81 to artifact cluster *H* in the file set comparison shows us that developers in cluster *B* have been major contributors to *H* and *E*. This is interesting, as developer cluster *B* contains only 3 of the 18 developers that have edited artifact cluster *E*, while *A* contains 9 of them. Still, the 3 developers in cluster *B* make up the majority of the changes to artifact cluster *E*, which shows us that in some cases, a few very active developers have a lot more impact on the development of a project than a large group of less active developers. Unfortunately, our analysis cannot always account for the impact of single developers, as the changes made by each developer in a cluster get attributed to the entire cluster. Still, in cases like the similarity between developer cluster *B* and artifact cluster *E*, we recognize that a small group of developers is responsible for the majority of changes made to that artifact cluster. When comparing the similarities computed for developer cluster *B* to those computed for *A*, we can see that, unlike *A*, the development activity of *B* is a lot less spread out across the project. Instead, small groups of developers in *B* focus on distinct tasks in the project. Since clusters like *B* are often, but not always, focused on a small number of specific parts of a project, we refer to them as *task-focused clusters* for the rest of our results. Task-focused clusters occur more frequently than central clusters. Similar developer clusters can be found in all projects, e.g., *C* in Figure 4.7, *B* and *C* in Figure 4.8, *D* in Figure 4.9, *C* in Figure 4.10 (although *C* could also be classified as a central cluster), and *A*, *B*, *C*, *D*, and *F* in Figure 4.11.

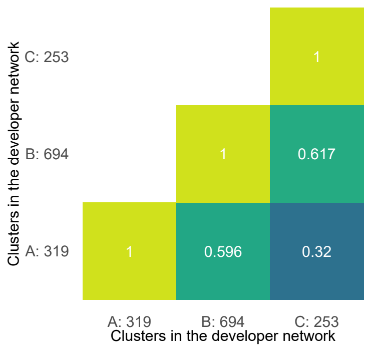
The final type of cluster to discuss in Figure 4.7 is developer cluster *D*. *D* only consists of two developers and has barely any similarity values that are not equal to zero in all of the comparisons. From the completeness in the developer set comparison, we can tell that one of the two developers in *D* has worked on artifact clusters *F* and *H*, although we cannot tell whether this is the same developer or if each of the developers in *D* has worked on one of the two artifact clusters. While we have relatively high similarity values between *F* and *D* for the overlap coefficient in the file set comparison, the Jaccard index in the file set comparison of only 0.038 between *F* and *D* and 0.042 between *F* and *H* shows us that the files changed by developer cluster *D* make up a very insignificant part of the total number of files changed in both clusters *F* and *H*. Because of their low impact on the development of a project, we call developer clusters like *D* *inactive clusters*. Similar clusters can be found in many observation windows across all projects, e.g., *D* and *E* in Figure 4.8, *A* and *C* in Figure 4.9, and *B* in Figure 4.10.

Next, we will provide some of the characteristics of the three types of developer clusters we described when analyzing Figure 4.7 from a more generalized perspective. In most of the developer communication networks, we can identify one or two central clusters (e.g., developer cluster *A* in Fig. 4.7), which have high similarity values to many artifact clusters in both the developer set comparison and the file set comparison. Central clusters usually include the most active developers of a project and often, but not always, have the highest developer count among all the clusters of its developer communication network. When comparing the files that are committed by central clusters in the developer network to the clusters in the artifact network, we often find that the developers in the central cluster commit changes to many different modules of a project instead of focussing on a small number of modules, which we would expect if the cluster was focussed on a major task such as developing new functionality. The central cluster usually consists of a set of very active developers who stay in the central cluster for multiple observation windows as well as several less active developers who change more frequently, so it is often a combination of the first two types of clusters in developer communication networks.

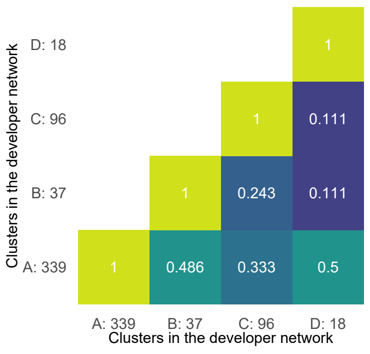
There are up to three task-focused clusters per network (e.g., developer clusters *B* and *C* in Fig. 4.7) which include developers who commit multiple changes to the project per observation window but do not work on as many different clusters in the corresponding artifact network as the developers in the most active cluster. The development activity of task-focused clusters seems to mostly be specialized on only a small number of tasks in the development of the project instead of editing many different parts like central clusters. We found that the Jaccard similarity and overlap that is computed based on the files in artifact clusters and the files edited by task-focused developer clusters change a lot over the course of multiple observation windows, which we attribute to the fact that task-focused clusters seem to form spontaneously around current tasks in the development of the project. The set of developers that are included in task-focused clusters also changes a lot over time which makes sense if these clusters do in fact form spontaneously around certain tasks. If there is a clear task that a task-focused cluster is focussed on, i.e., the commit activity of that community is concentrated on a few artifact clusters, task-focused clusters often fit the description of Type 1 clusters in developer communication networks. If the activity of a task-focused cluster is more spread out across many artifact clusters, it is more likely to fit Type 2, as the cluster is usually the result of a single maintainer accepting and rejecting independent pull requests of different developers.

There are often several inactive clusters (e.g., developer cluster *D* in Fig. 4.7), which usually range from two to ten developers. Inactive clusters consist of developers who previously contributed changes to the project but did not do so in the observation period that the developer communication network was built from or developers who do commit small changes that only affect one or two artifact clusters, e.g., bugfixes or small tweaks to existing source code. Inactive clusters often evolve around feature requests or off-topic discussions and are therefore representing Type 3 developer clusters. Additionally, we find that inactive clusters occur more frequently in projects with a larger developer community, as a larger community often coincides with a higher number of issues, which makes the existence of issues that are not connected to any central or task-focused cluster more likely.

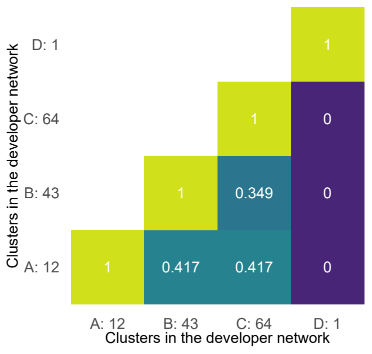
An example of the results of the committed files comparison, in which we compared the sets of files committed by different clusters in the developer communication networks to each other, is shown in Figure 4.12. For OPENSSL, *A* is a central cluster, *C* is a task-focused cluster, and *B* and *D* are inactive clusters. We can see that the changes made by inactive clusters have an overlap of about 50% to the central cluster, which means that, despite *A* committing significantly more files than *B* and *D*, the two inactive clusters still made changes to parts of the project that the central cluster did not change at all. Additionally, we can see that the



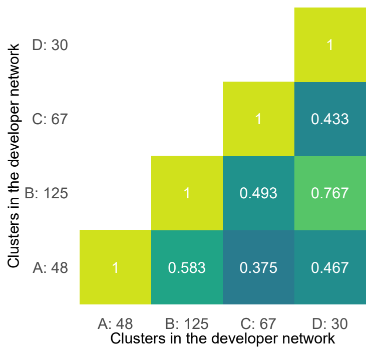
(a) DENO 02/2020 - 08/2020



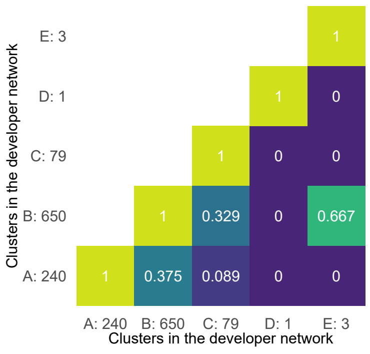
(b) OPENSSL 12/2018 - 06/2019



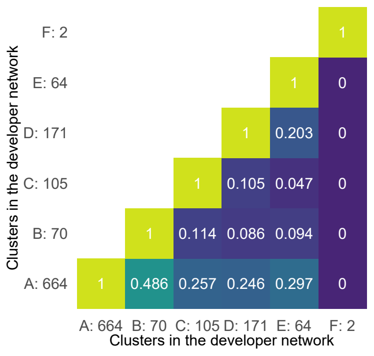
(c) ATOM 02/2018 - 08/2018



(d) TYPESCRIPT 07/2019 - 01/2020



(e) NEXTCLOUD 12/2019 - 06/2020



(f) MOBY 01/2018 - 07/2018

Figure 4.12: Results for the committed files comparison, for which we computed the overlap of the sets of files committed by different clusters in the developer network, for one observation window per project. The observation windows displayed in the figure are similar to the majority of the rest of the observation windows for each project. Each cluster is labeled with a letter and a number. The number represents the number of files edited by the respective cluster.

overlaps between *C* and other clusters are below 35%. As *C* is a task-focused cluster, the low overlaps shows us that the development activity *C* is focused on parts of the project which are only rarely edited by developers in other clusters.

In general, the overlaps between sets of files committed by different clusters in a developer communication network are largely consistent across multiple observation windows in the same project but can be very different across multiple projects. For example, many of the computed overlaps for TYPESCRIPT are very high, while MOBY often includes similarities that are close to zero. We also encountered comparably high overlaps for DENO, while NEXTCLOUD is an example of relatively low overlaps. Despite these fluctuations in the results, we could still make out patterns that hold for all of our projects. The overlap of the sets of files committed by central clusters is typically between 30% and 50% and largely consists of files that are frequently committed to. Additionally, we found that the sets of files committed by task-focused clusters overlap by between 10% and 40%, depending on the project and the current observation window. Since task-focused clusters usually do not consist of the same developers over multiple consecutive observation windows, the overlap of their committed files changes frequently as well. Surprisingly, the small set of files committed by developers in the inactive clusters in a developer communication network often does not overlap with the files committed by other clusters, despite larger clusters committing changes to a majority of the files of a project.

4.2.4 *Cluster Comparison: Project Results*

The previously presented results apply to the majority of the projects in our sample set. However, the results for each of the projects have some differences from our general results, which we present next.

OpenSSL & Nextcloud. Our results for both of these projects are very close to our general results. The only major difference is that some developer communication networks built from both OpenSSL and Nextcloud sometimes contain totally inactive clusters of previously active developers that commit no changes at all. These clusters seem to be the results of developers that only commented on a single issue and are usually very small in size, consisting of less than ten developers. In OpenSSL, we only encountered these clusters in two observation windows. In Nextcloud, 6 out of 19 observation windows contain totally inactive clusters. The only other project in which we encountered totally inactive clusters is Moby.

Deno. Out of all the projects in our sample set, Deno has the shortest observation period of only a little over two and a half years. Despite the project's short lifespan at the time of our analysis, we found that the developer community displays a very active usage of issues, which leads to developer networks of around 200 vertices in the last 3 observation windows while the developer networks built from different projects often contained around 100 vertices despite the projects being relatively similar in size. However, we found that the larger number of vertices did not influence the detected clusters, as they still resemble our general results. Usually, the developer networks built from Deno contain one central cluster, which is very actively committing changes, one or two task-focused clusters, and up to three inactive clusters which barely commit any changes. These inactive clusters exist in multiple different sizes and with varying activity, and they do not contain a consistent set of developers in consecutive observation windows, which leads us to the assumption that these communities form from discussions by developers who do not comment on any of the "main" issues of the project, i.e., the issues that are discussed by core developers. The inactive clusters usually make up about 20% of the total developer count in a developer communication network that is built from Deno.

Atom. Due to its strongly diminishing developer activity towards the end of our observation period, the developer networks built from the last observation windows of Atom have some unique properties. For the times in which Atom still had a more active developer community, the results are similar to those of other projects. As both the number of developers and the number of committed changes are significantly lower towards the end of the observation period, we detect fewer clusters in both of these networks. Surprisingly, we can still detect multiple clusters as unlike at the beginning of a project, the few developers that work on the project are actively using issues, so our dataset contains multiple hundreds of issue events for each observation window. However, we only detect between two and three clusters for each of the developer networks that have been built from data from the last two years of our observation period. These clusters consist of only three to ten developers. Additionally, some of the maintainers of the project commit directly to the source code without the use of pull requests or issues, presumably because they discuss changes in another channel of communication to that we have no access. We did not encounter this behavior in any other project that relied on pull requests.

TypeScript. When applying our approach to TypeScript, as it was described in Section 4.1, initially yielded barely any overlap between the files committed by developer clusters as well as barely any recognizable patterns when comparing artifact network clusters to the developer clusters, which made it very hard to interpret the results. This problem can be attributed to a very high number of unit tests, as previously mentioned in the results for our first research question. In the artifact networks used for our analysis, which have been built from commit data that has all unit tests removed, the detected clusters behave largely similar to the previously presented general results. However, TypeScript has an exceptionally high overlap between the files committed by different developer clusters, being above 65% in many cases. This is most likely a side product of filtering out the unit tests, as unit tests are typically only committed once and therefore produce no overlap between different developer clusters. Since other projects still contain the unit tests in their commit data, they typically have a lower overlap between the files committed by different developer clusters than TypeScript.

Moby. Compared to the other projects, which have between 299 and 673 different developers who committed changes to them during our observation periods, Moby has a very large developer community of 1160 different developers. This also shows in the developer communication networks built from the issue data, as Moby is the only project in which we consistently detect more than 4 clusters per observation window and reach as high as 13 detected clusters at the beginning of 2018. Surprisingly, most of these clusters consist of less than 10 developers and are inactive clusters. Just like the developer communication networks from other projects, the developer communication networks from Moby typically contain only one or two central clusters which actively commit changes to the project. Many of the artifact clusters are only changed by a single developer, but the developers making these small changes mostly end up in the central cluster. The sets of files that were changed by central clusters have an overlap of about 50% to 60%, which is similar to our results from other projects. Hence, we conclude that the higher developer account in Moby does not lead to a strong deviation from our general results and only increases the likelihood of the existence of small, inactive developer clusters.

4.3 DISCUSSION

After evaluating the results of our study, we discuss how they answer our research questions.

4.3.1 *Artifact Network Characteristics*

RQ1. *What are the characteristics of artifact networks and how do they evolve over time?*

We found out that artifact networks have a lot of common characteristics, with many of them even being consistent across multiple projects and many observation periods. The number of vertices and edges of artifact networks scales with the commit activity of the project that they are built from. As can be seen in Figure 4.1, there are strong fluctuations in the number of edges and vertices in many projects, which also shows us that the development activity in OSS projects, i.e., the number of commits, fluctuates over time.

Some other characteristics of artifact networks we analyzed are stable across all of the projects in our sample set, independent of the number of commits, developers, and files. Artifact networks are sparse and have a low average degree. Additionally, they have high clustering coefficients and comparably low average path lengths, which makes them small-world networks. If the artifact networks for a project turn out to have uncommon results for these characteristics, it is usually the result of a unique property of the project such as a large number of unit tests in TypeScript or the diminishing developer activity towards the end of our observation period for Atom.

Clusters, hubs, and edge weights, which behave differently depending on the project we analyze, can be used to gain insight into the structure of a software project. Our findings suggest that the clusters found in artifact networks represent modules in a software project. To be fair, these clusters can contain files of multiple modules or split a module into two clusters, depending on the clustering algorithm used to detect them and the number of times files of different modules are committed together. However, clusters in artifact networks provide a method to detect modules that is entirely independent of the programming language used, which can be helpful as many of the more accurate methods to analyze the structure of software projects are bound to specific programming languages. We also investigated vertex degrees in artifact networks and found out that artifact networks contain hubs that can be used to find out which files are most important to the development of a project at a certain point in time. Analyzing the edge weights of artifact networks has proven to be unhelpful, as different files in our sample projects are not committed frequently together. Most of the files that are connected by an edge are committed too rarely to use the edge weights in artifact networks as a way of detecting functional dependencies among files. The few artifacts that have high edge weights usually have dependencies that can easily be detected in other ways such as being subclasses of each other or the code in one file directly referencing code from another file. Since the number of dependencies we can detect in our sample projects using edge weights is low, we conclude that edge weights are not a suitable method to detect functional dependencies in files.

4.3.2 *Comparison of Artifact Networks and Developer Networks*

RQ2. *Which relationships between the communication of developers and the artifacts they commit can we detect by comparing clusters of artifact networks and developer communication networks?*

Our results indicate that clusters in developer communication networks are roughly grouped by their commit activity, as there are usually one or two very active clusters in terms of commits (central clusters), up to three less active ones (task-focused clusters), and multiple almost inactive clusters. The developers in these clusters do not seem to be stable subcommunities of the developer community, as the clusters in subsequent observation windows have varying activity, members, and member counts. A possible explanation for the fact that we detected exactly one central cluster for almost all of the developer communication networks could be that developers that contribute actively to a project are often involved in multiple different issues at once. This leads to them communicating with many different active developers in the same observation window. Hence, the most active developers often end up being grouped in the same central cluster of a developer communication network.

For the task-focused clusters, we found that they often commit to fewer different clusters in artifact networks. Additionally, the members of these task-focused clusters change even more frequently to the point where task-focused clusters hardly stay stable for longer than a single observation period, i.e., they usually do not persist for longer than six months. This part of our results aligns with the results of a study by Ashraf et al. [1], who found that the subcommunities found in developer communication networks are often spontaneously arising and focus on specific tasks that are important in the development of a software project at a certain point of time.

Inactive developer clusters mostly do minor changes, such as bugfixes, or do not commit anything at all. Inactive clusters often consist of developers who committed something in the past and do not actively participate in the development any longer. Yet, they still take part in discussions in issues from time to time. We assume that these former developers are still using the software that they previously worked on and therefore still discuss feature requests or bug reports that they are interested in. However, this is just a possible explanation and we did not an in-depth analysis of the developers in inactive clusters.

Unlike developer clusters, we cannot group artifact clusters into different types. However, we did find that larger clusters, which also often contain more important parts of the source code than smaller clusters (see Section 4.1), are usually changed by a large number of developers in each observation window. The smaller a cluster, the fewer developers committed changes to it. However, we assume that the number of developers working on an artifact cluster being higher for bigger clusters is not caused by the number of files in the cluster. Instead, we assume the connection to be the other way around: As files that are important at the current stage of development for a project are often edited together, they naturally form clusters. Since central files get more attention from the developer community than less important ones, there are more commits of these files which in turn increases the likelihood for each pair of central files to be connected by an edge. If there are many commits and therefore many edges between these files, they form a large cluster. So we conclude that files that are central in the development of a project are not forming big clusters solely because of their functional relationships, but because many developers make changes to them. This characteristic is also the most likely reason why even less active clusters in the developer communication networks frequently commit to these central clusters.

When taking a look at the similarity of the sets of files that different developer clusters commit to, we found that the central developer cluster usually has an overlap of around 50% with task-focused clusters. This is surprisingly low, as the most active cluster usually works on many different parts of the project at once and therefore covers a lot of files, so we would expect the overlap to the files committed by task-focused clusters to be closer to one. The overlap between the files edited by different task-focused clusters is usually around 10% to 40%. A possible explanation for these overlaps could be that task-focused clusters work on parts of the project that get less attention from the rest of the community, so the sets of files they work on only overlap partially with those that the rest of the community works on. Since changes to existing code often span across multiple files and the most important files of a project often have dependencies to many other files, even changes to files that otherwise do not get any attention from the rest of the community sometimes also involve the more important files. This would explain why there is an overlap between the smaller communities, despite them mostly working on different parts of the projects. The exact numbers for the overlap between communities are usually consistent for multiple observation windows of a project, despite the clusters often changing in size and members. The overlaps differ heavily between some projects, e.g., Deno often has overlap coefficients of around 70% between the sets of files committed by central clusters while the overlap coefficients for central clusters in OpenSSL stay around 40%. We attribute these changes to the differences in coordination among developers and file structures in different projects, but we could not determine a clear cause for them.

While our results had many commonalities across all of the projects in our sample set, we still found some unique properties in the results of Deno, Atom, TypeScript and Moby. We attribute these deviations from our general results to the differences in how these projects are developed, how developers communicate, and how active these communities contribute to the project. The initial difficulties we faced when analyzing TypeScript due to its large number of unit tests serve as a good example of a unique property of a project, as we did not face similar difficulties for any other project during our analysis. The results from Atom and Moby show us that the activity of the developer community has a major impact on the resulting developer communication networks. If applied to a larger number of projects, we expect our approach to yield largely similar results to the ones we already found, but there could be further exceptions to our general results that we did not encounter when analyzing the projects in our sample set.

4.4 THREATS TO VALIDITY

In this section, we discuss internal and external threats to the validity of our results.

4.4.1 *Internal Validity*

One of the main threats to the internal validity of our findings is the data we use to construct networks. Artifact networks are built from commit data which contains all changes that were made to the projects in our sample set during our observation periods. This, however, is not true for the developer communication networks, which are built from issue data. Many of the projects we analyzed introduced the use of issues a long time after already using *git* to commit their changes, so we are missing a lot of communication data during the early stages of our sample projects. Therefore, we only use observation windows for which we have issue communication data for our comparison of artifact networks and developer communication networks. In addition to issues, developers might use multiple channels of communication such as mailing lists or additional task management systems. If this is the case, we might miss some relationships between developers as we only use the issue data. However, all of the projects we analyzed use issues as their main channel of communication, so we capture most if not all of the important relationships among developers by only looking at issues.

Another threat is the validity of both the issue and the commit data. We used CODEFACE, CODEFACE-EXTRACTION, and GITHUBWRAPPER to extract the data from GITHUB, so we rely on the correctness of these three tools. CODEFACE disambiguates author names and e-mail addresses by using a heuristic by Oliva et al. [31]. In addition to that, all authors in the data were disambiguated manually, but we cannot guarantee that we did not miss any duplicates. Furthermore, some developers share e-mail addresses with other users. We did our best to detect these cases and attribute the commits made by shared e-mail addresses to all of the users using the respective address, as we cannot tell with certainty who of the shared owners is responsible for which commit or issue. If there are any cases we missed, we deem them insignificant enough to not be a major threat to the validity of our findings.

Using the Louvain algorithm to detect clusters in developer communication networks also poses a threat to validity. The Louvain algorithm yields nondeterministic results, which makes it difficult to reproduce the results of our analysis. However, we compared the modularities of several different clustering algorithms on both artifact networks and developer communication networks and found that the Louvain algorithm consistently yielded the best results for our analysis.
Furthermore, Joblin et al. [21] have shown that using a clustering algorithm that is able to detect overlapping clusters in developer collaboration networks yields significantly more accurate results than using a regular clustering algorithm. This could also apply to developer communication networks, so our decision to use the Louvain algorithm which only detects non-overlapping clusters might impact the quality of our results. However, Bird et al. [5] have shown that non-overlapping clustering algorithms yield accurate results when it comes to detecting communities in developer communication networks.

Moreover, our way of comparing clusters of files to clusters of developers is a very simplified view of the complex relationships between developers and the code they work on. Since our artifact networks are based on files, we value the contribution of a developer who just fixed a single typo equally as large as a developer who changed hundreds of lines of code in that file. Similarly, we group all developers who committed changes to a cluster of files into the same set, no matter how many of the files in that cluster they edited. Because of these two simplifications, the activity of developers has a much smaller impact on our results than it has in reality. Conversely, this problem also holds for our way of grouping files that were edited by a cluster in the developer network. Our approach does not include how many different developers in a cluster have worked on a certain file and how big the changes made to the file are. However, simplifications are a major part of our approach and finding a different approach without such simplifications is a nontrivial task.

4.4.2 *External Validity*

We expect our results to apply to most *GitHub* projects of similar sizes to the ones we used for our analysis that also actively use issues. However, even though we selected six projects of varying size, there are many projects with more commit activity or participating developers such as *Linux*[1] or *Visual Studio Code*[2], which might lead to different results. There is also an even larger number of projects that are substantially smaller and less active than the projects in our sample set. Since, to construct artifact networks and developer networks, we need a relatively high number of commits and issues, our results do not apply to such small projects.

Additionally, we only used projects that use the *git* version control system and *GitHub* issues. We do not expect other version control systems to have a major impact on the characteristics of artifact networks, as the overall usage of different version control systems is fairly similar. However, different channels of communication can change the way developers collaborate and might even impact the evolution of a project [13]. Therefore, our results do not apply to projects that are not hosted in the same way as our sample projects.

1 https://www.linux.org/
2 https://code.visualstudio.com/

5 RELATED WORK

There have been several studies that mined data from mailing lists, GitHub issues, or version control systems to analyze the developer interaction in open-source-software projects. Many of these studies used social network analysis to detect developer communities and analyze how these communities evolve over time. Similar to this thesis, some studies have also investigated which parts of a project these developer communities work on and how they coordinate based on the parts of a project they work on.

Joblin et al. [21] provide a fine-grained approach to detect communities in function-based developer collaboration networks. In their study, they used the OSLOM clustering algorithm to identify overlapping developer communities. They then verified their results in a survey involving 53 different developers of the projects they analyzed and showed that the communities that can be detected by using their approach have a real-world meaning. They also showed that communities detected in file-based developer collaboration networks tend to be a lot less accurate than the ones that can be detected in function-based developer collaboration networks. In another study, Joblin et al. [20] analyzed how developer collaboration networks built from OSS projects evolve over time. They have shown that developer collaboration networks tend to evolve into certain structures as the number of developers of a project grows. In a later study [19], they provided an approach to classify developers into core and peripheral based on developer collaboration networks. Their study showed that the network-based metric they proposed largely agree with previously established count-based operationalizations when it comes to classifying developers in OSS projects.

Avelino et al. [2] constructed developer collaboration networks from the commit history of the Linux kernel to analyze co-authorship in large OSS projects. They found that, in the Linux kernel project, experienced developers rarely collaborate with each other, but instead often collaborate with newer, less experienced developers.

Hong et al. [15] used bug reports to build developer networks. Bug reports have a very short-lived nature, so two developers interacting in the same bug report indicates a strong temporal relationship. They then used the Louvain clustering algorithm to identify communities and observed that the detected communities, active developers, and their relationships change continually. However, they also observed that the size of the detected communities and the overall characteristics of the developer networks stay relatively stable. Similarly, Canfora et al. [11] mined cross-system bug resolutions and built developer networks based on them. They showed that cross-system bug fixes are mainly done by developers with high commit count and active participation in the mailing lists of a project.

Panichella et al. [33] used developer communication networks built from both mailing lists and issue data to study how emerging teams evolve over time. They have shown that emerging teams tend to work on structurally related files and that these teams tend to regroup over time, i.e. that two or more developers will often end up on the same team even if the rest of the team changes over time. Bock et al. [9] proposed a community classification approach based on tensor decomposition that used developer networks. Their approach allows for a very detailed exploration of the group structure of developer communities in OSS projects. Additionally, they analyzed how the group structures in networks based on developer communication and developer collaboration align.

Lopez-Fernandez et al. [24, 26] did some of the earliest studies on developer networks constructed from version control data. Additionally, they analyzed module networks in which vertices represent a module and two modules are linked by an edge if a developer has contributed to both of them. In a way, module networks can be seen as a very coarse version of artifact networks. Lopez-Fernandez et al. showed that module networks are small-world networks. Our results extended these findings by showing that file-based artifact networks are also small-world networks.

In a more recent study, Ashraf et al. [1] investigated whether communities found in developer communication networks built from issue data overlap with groups of developers contributing code to the same subsystem. They found that communities found in developer communication networks are rather unstable over time and do typically not overlap with groups of developers that commit to the same subsystems. Conversely, they found that while most developers tend to work on the same subsystem for a longer period of time, they also communicate with varying developers that work on different subsystems. Similar to this thesis, their study compared communities found in developer communication networks to the developers working on certain parts of a project. However, we detect these groups by identifying clusters of files that are often committed together and deriving the groups of developers who work on these clusters, while Ashraf et al. group developers who actively contributed to the same part of the source code.

Bird et al. [5] also found that developer communication networks built from OSS project data usually contain a very volatile community structure since communities tend to form ad hoc according to current tasks in programming. Shihab et al. [35] also did not find any community structure other than short-term communities forming according to current tasks, so they concluded that there is little to no stable organizational structure in OSS projects. The more recent study by Ashraf et al. [1] supports their results.
To the best of our knowledge, no other studies analyzed artifact networks based on jointly committed files in version control systems of open-source software projects. Similarly, we have no knowledge of any previous studies which compared any kind of developer network to file-based artifact networks.

6 CONCLUDING REMARKS

In this final chapter, we summarize and conclude our study. After that, we provide some ideas on how our findings can be used for future work.

6.1 SUMMARY

The goal of this thesis was to analyze the characteristics of file-based artifact networks built from the commit data of OSS projects and to compare clusters detected in file-based artifact networks to those detected in developer communication networks. We extracted the issue and commit data of six different OSS projects on GitHub. The names and e-mail addresses of developers in the extracted data were disambiguated, bots were filtered out, and we removed commits with a high number of files from the extracted commit data. Additionally, we removed all interactions of users that did not previously commit any changes to a project from the project's issue data. Both the commit data and issue data were split up into overlapping six-month long observation windows using a sliding-window approach, which allowed us to analyze the evolution of the resulting networks. We then built undirected, file-based artifact networks from the commit data, in which two files gain an edge if they are committed together in the same observation window. We also built developer communication networks for each observation window, in which two developers are connected by an edge, if they interacted with each other in an issue, e.g., by commenting on it consecutively.

First, we analyzed the characteristics of file-based artifact networks. As expected, the number of vertices and edges of an artifact network scale with the number of commits that are used to build the artifact network and the number of unique files in these commits. We found that artifact networks have a low density, high clustering coefficients, and an average path length that is comparable to a random network of the same size. The high clustering coefficient in combination with our results for the average path length led us to the conclusion that artifact networks are typically small-world networks. Furthormore, we took a look at vertices with high degrees in artifact networks and found that they represent files that are important in the development of the software project at a certain point in time. When taking a look at edge weights in artifact networks, we found that most of the edge weights in artifact networks are too low to use them to detect functional dependencies between files.

Second, we computed clusterings on artifact networks and did an empirical analysis of the sets of files that end up in the same cluster. Our results suggest that clusters in file-based artifact networks represent modules in the software project. The accuracy of these clusters largely depends on the number of commits used to build the network, as a higher number of commits includes a higher number of dependencies between files. When looking at the clusters found in developer communication networks built from issue data, we unexpectedly found that there is a relatively low number of developer clusters per network. The developers in these clusters are roughly grouped by their committed activity. Additionally, we found that actively committing clusters in developer communication networks often form around a small group of maintainers.

Finally, we compared clusters in artifact networks to those found in developer communication networks by mapping sets of files to sets of developers and vice versa using commit data. When comparing developer clusters to the clusters in artifact networks they commit to, we grouped them into three different kinds of clusters. In most developer communication networks, we found at least one cluster that commits a lot more changes than the other clusters and also includes many maintainers of a project. Additionally, we found several smaller clusters that also actively commit changes but are a lot more focussed on a specific task in the development of the project than the biggest cluster. The least active clusters are often the result of discussions between former developers that are inactive during the observation period that the network is built from. The files changed by different developer clusters typically overlap by between 10% and 50%. The changes made by different developer clusters usually overlap on the central files of a project which have a lot of dependencies on other files, while less central files are rarely changed by developers in more than one cluster in the same observation period.

In conclusion, we found many commonalities in the characteristics of artifact networks built from different OSS projects. We also found a lot of commonalities across multiple projects in our cluster comparison, although there are multiple instances in which our results for a certain project had unique properties. For future work, utilizing and refining our approach and applying it to a larger sample set of projects could yield interesting insight into the commit-behavior of OSS communities.

6.2 FUTURE WORK

Our analysis of the characteristics of artifact networks can be used as a baseline for future studies on this type of network. Our characterization of artifact networks was based on simple graph metrics. Further studies could investigate the role of individual files and clusters in artifact networks by applying more sophisticated centrality measures and clustering algorithms to them. Additionally, characterizing function-based artifact networks in the same way that we characterized file-based artifact networks could provide further insights into how the choice of the artifact type influences the resulting networks.

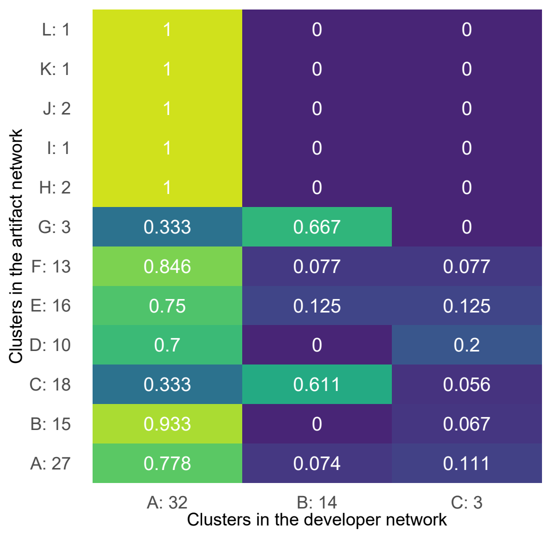
Furthermore, our sample set for this study was limited to six OSS projects from GitHub. To generalize our findings, projects of different sizes and with different work tracking systems would need to be analyzed. Ideally, our method of comparing artifact clusters to developer clusters should be improved in a way that allows for a more large scale evaluation, e.g. by automating parts of the evaluation, instead of a mostly empirical one.

Moreover, the comparison between clusters in artifact and developer communication networks can be improved by using a clustering algorithm that is able to detect overlapping clusters, such as the OSLOM [22] algorithm, on developer communication networks. Joblin et al. [21] have shown that using such an algorithm significantly increases the accuracy of the detected communities in developer collaboration networks, so the same might hold for developer communication networks.

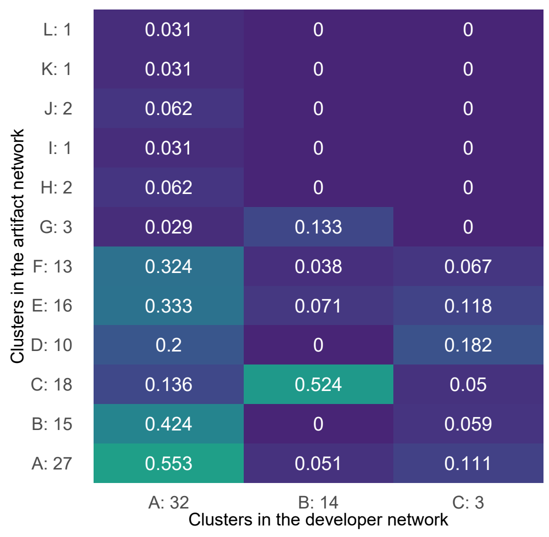
Our work showed that both the characteristics of artifact networks and the clusters in artifact and developer communication networks vary depending on the project that they are extracted from and also evolve over time. Further research is needed to contextualize these differences, i.e., find out which factors influence the characteristics of artifact networks and the clusters found in them. To do this, the characteristics and clusters can be compared to the evolution of software quality metrics of a project. Additionally, investigating how major events in the development of OSS projects (such as new commit policies or a change in the development goals) influence the resulting artifact network could help us understand how these events influence the development of OSS projects at the commit level.

A APPENDIX

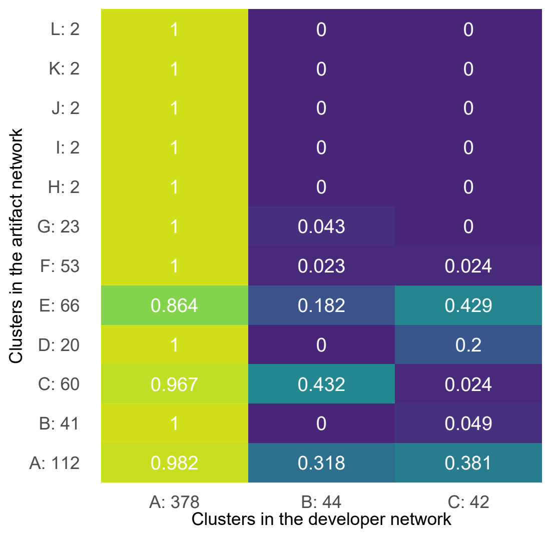
In the appendix, we provide further results of our cluster comparison. We include the results for one early observation window and the results for an observation window close to the end of our observation period for each of our sample projects. The results for Nextcloud are not included, as the artifact networks for Nextcloud contain too many clusters to print the resulting similarity matrices on a regular-sized page. Results for TypeScript were computed using commit data which had all unit tests removed from it. Just as in the similarity matrices in Chapter 4, each of the clusters in the similarity matrices is labeled with a letter and a number. The number represents the number of vertices, i.e., files or developers, in the respective cluster.



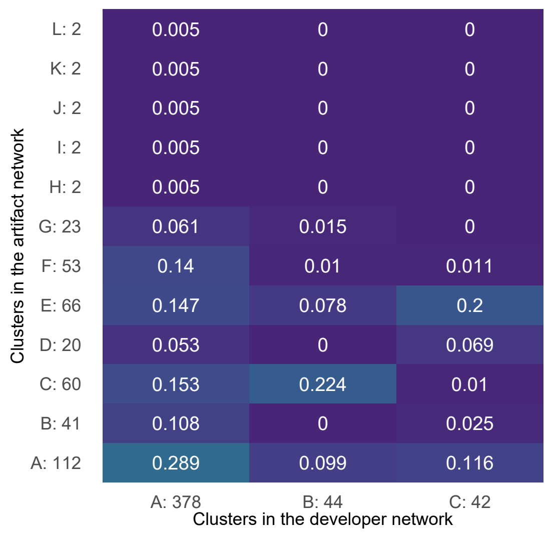
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)

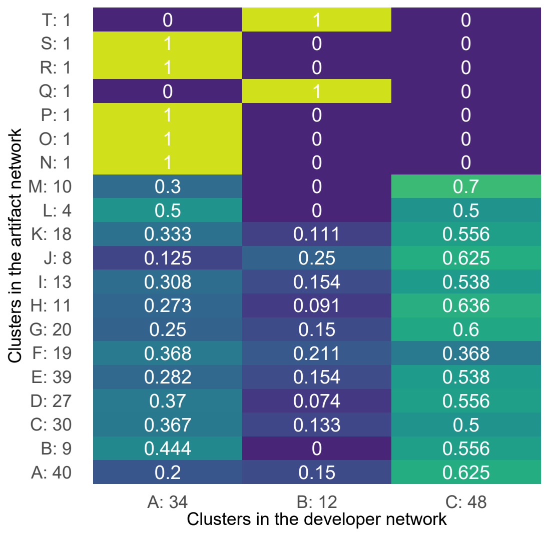


(c) File set comparison (overlap coefficient)

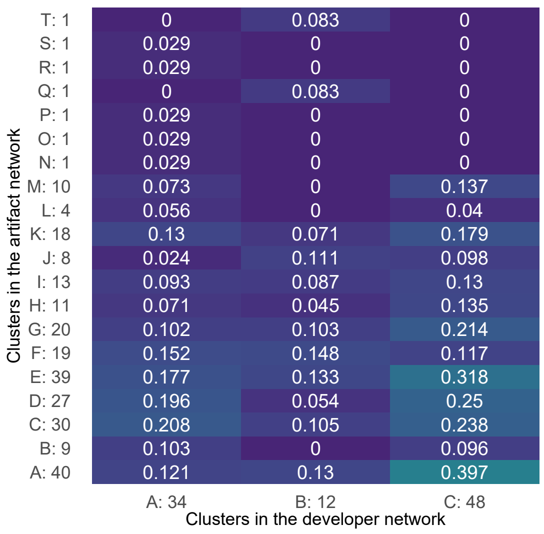


(d) File set comparison (Jaccard index)

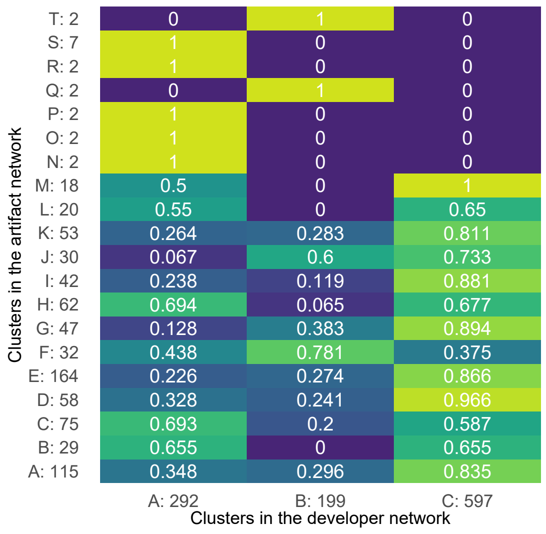
Figure A.1: Results of the cluster comparison for Deno for the observation window from May 2018 to November 2018, which is the first observation window for the project. Unlike the other projects in our sample set, Deno has been actively using issues from the start, so the results look very similar to our general results.



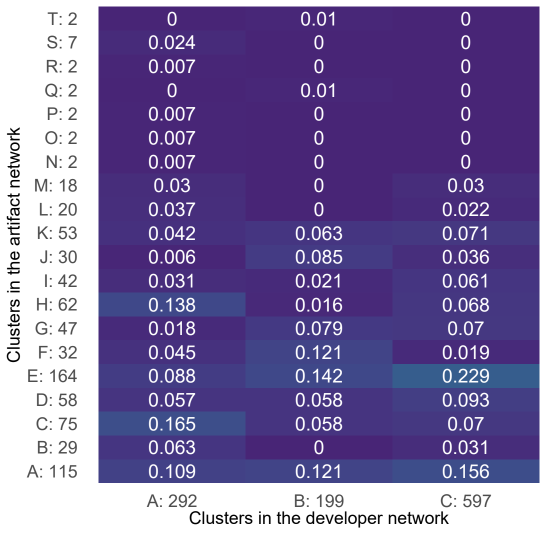
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)

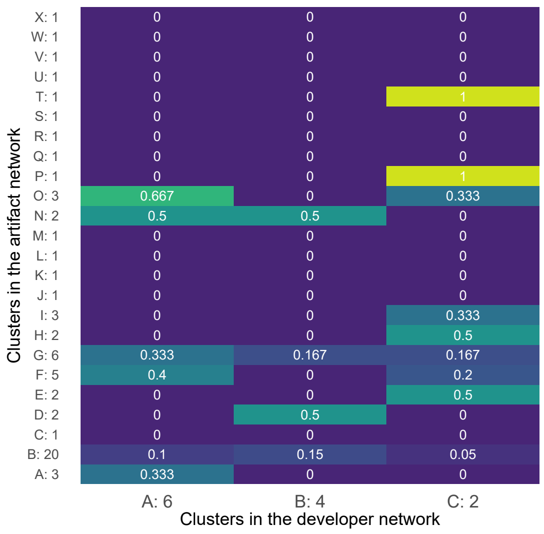


(c) File set comparison (overlap coefficient)

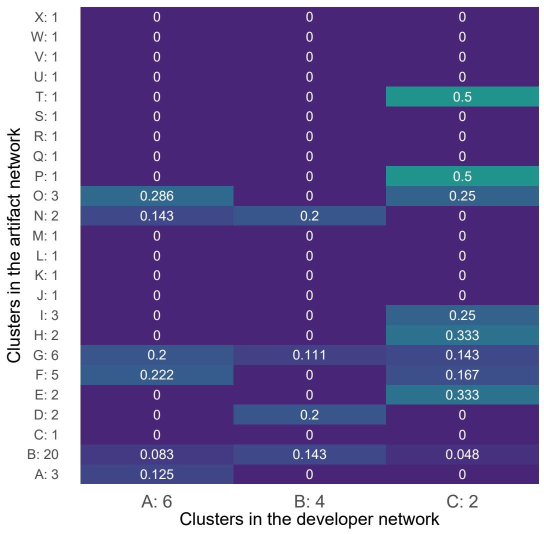


(d) File set comparison (Jaccard index)

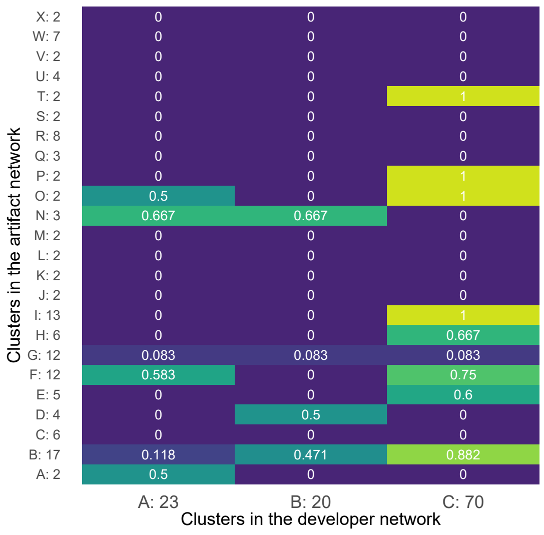
Figure A.2: Results of the cluster comparison for Deno for the observation window from November 2019 to May 2020, which is close to the end of our observation period.



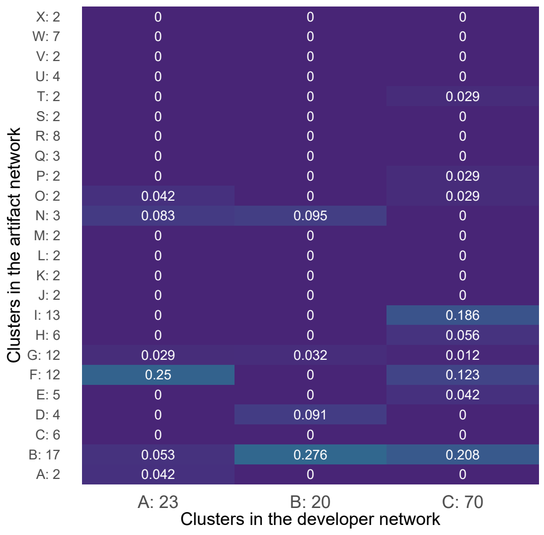
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)

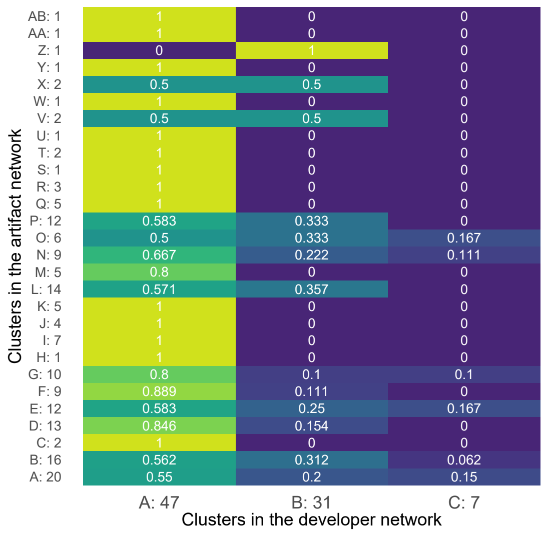


(c) File set comparison (overlap coefficient)

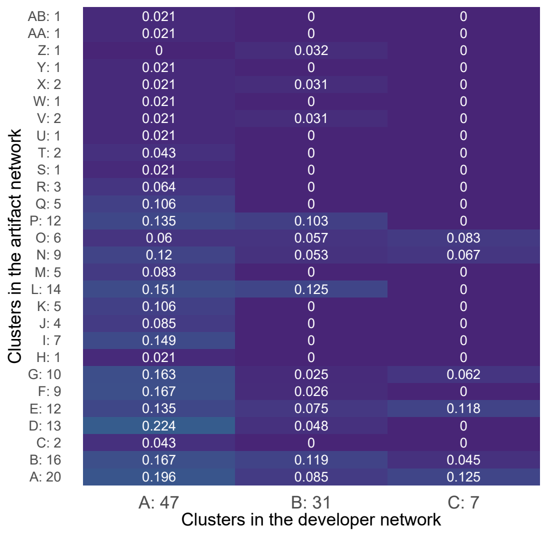


(d) File set comparison (Jaccard index)

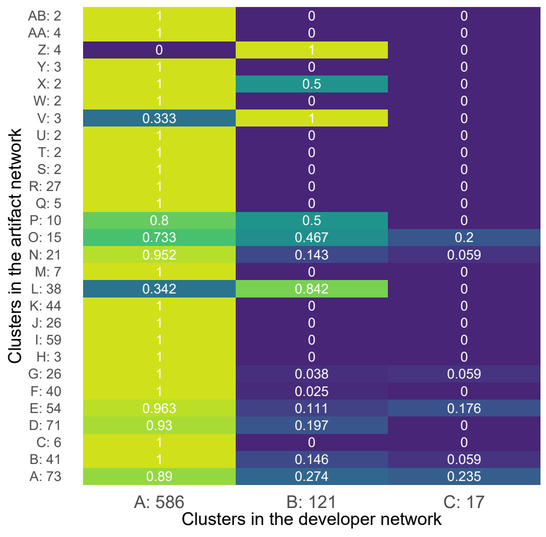
Figure A.3: Results of the cluster comparison for OpenSSL for the observation window from December 2013 to June 2014, which is the earliest observation window with more than one cluster in the developer network for OpenSSL.



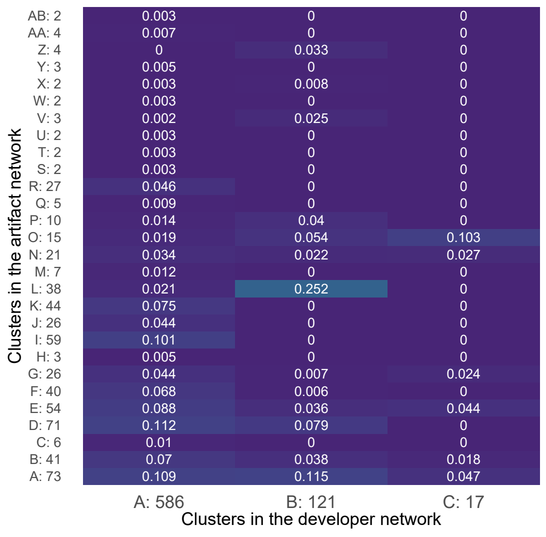
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)

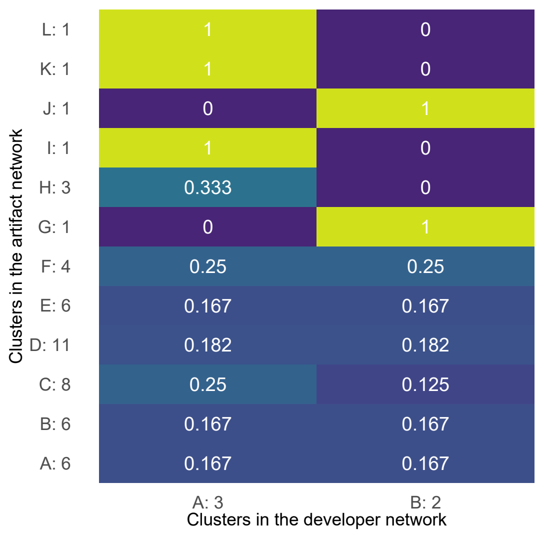


(c) File set comparison (overlap coefficient)

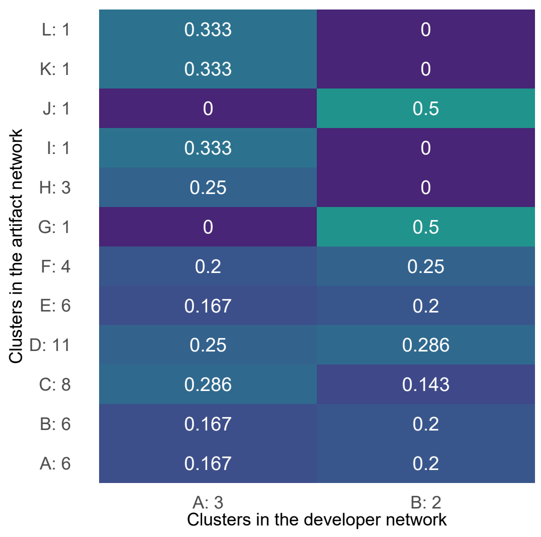


(d) File set comparison (Jaccard index)

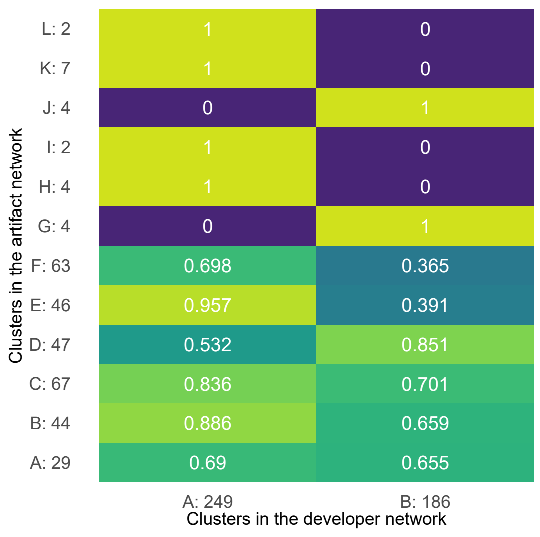
Figure A.4: Results of the cluster comparison for OpenSSL for the observation window from June 2019 to December 2019, which is one of the last observation windows for OpenSSL. Both the developer community and the number of files in the artifact network have significantly grown since the beginning of the project.



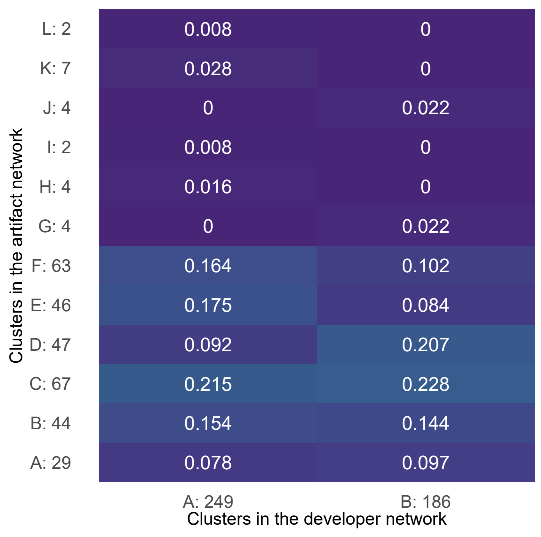
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)

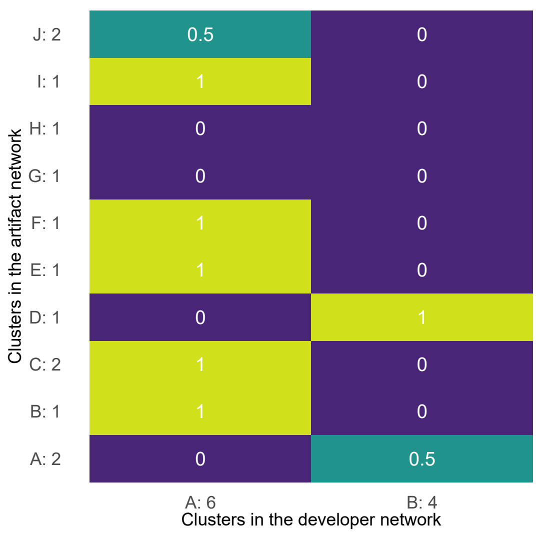


(c) File set comparison (overlap coefficient)

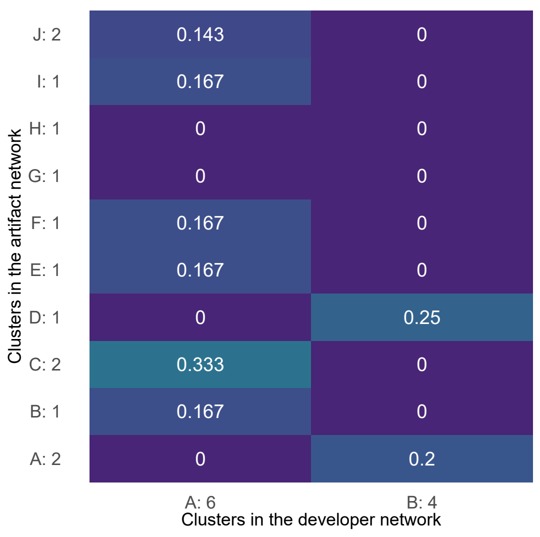


(d) File set comparison (Jaccard index)

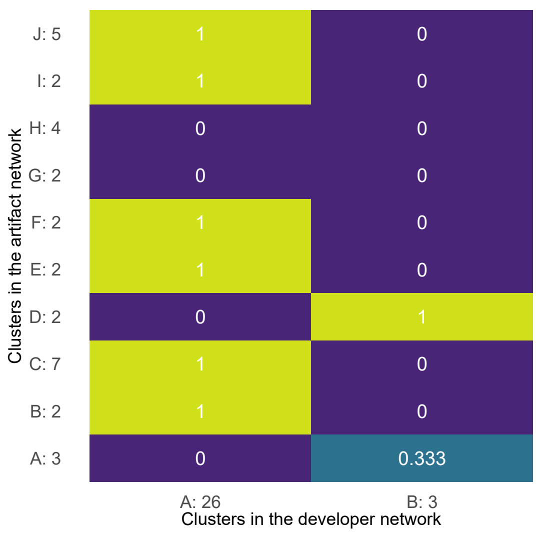
Figure A.5: Results of the cluster comparison for Atom for the observation window from February 2012 to August 2012. As this observation window is very early in the development of the project, there are only 5 developers using issues, while there are clusters in the artifact network that have been committed to by more than 5 developers.



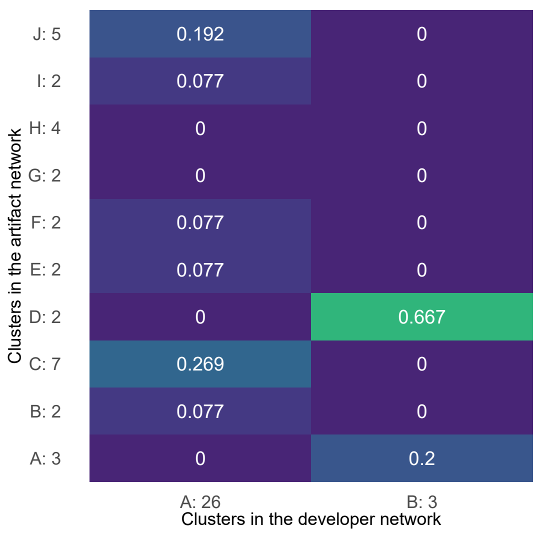
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)

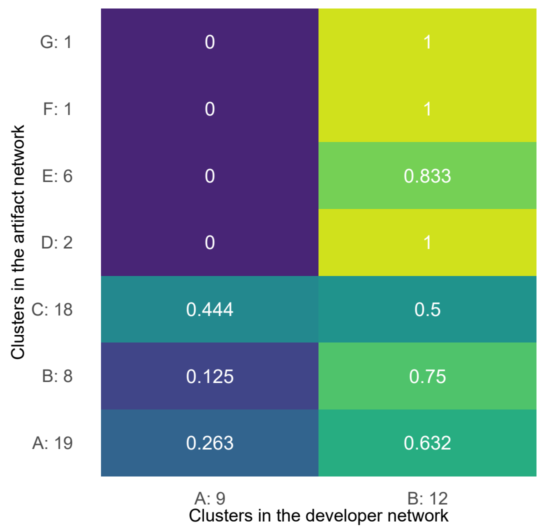


(c) File set comparison (overlap coefficient)

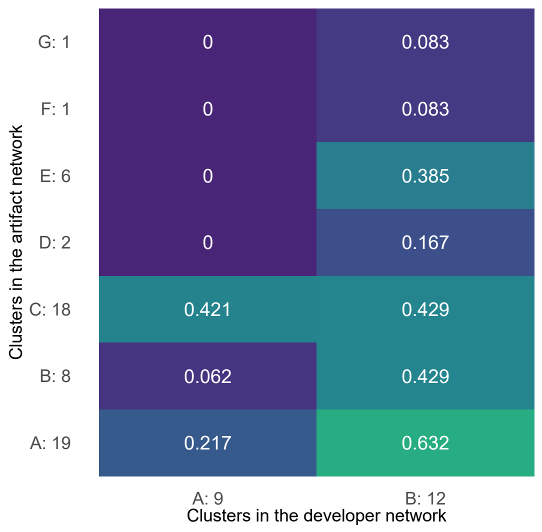


(d) File set comparison (Jaccard index)

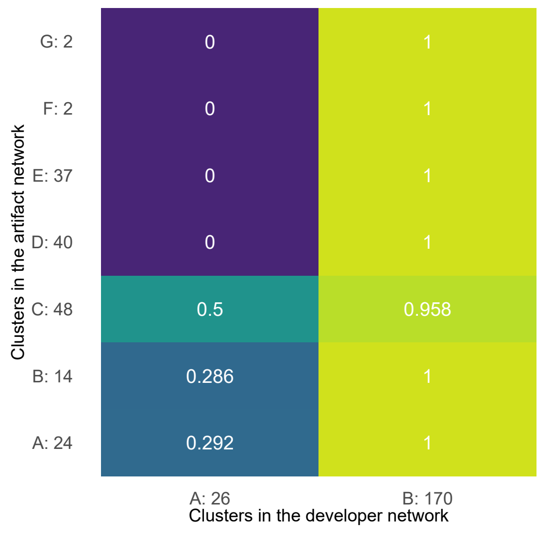
Figure A.6: Results of the cluster comparison for Atom for the observation window from August 2020 to August 2020. Since Atom has diminishing developer activity towards the end of our observation period, the depicted results are similar to those in the early stages of the development of a project.



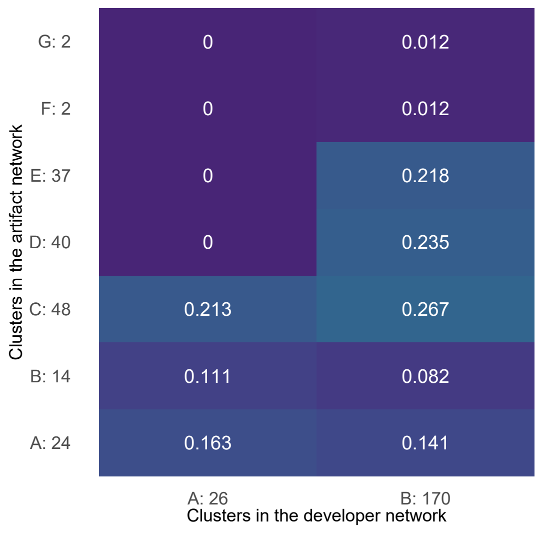
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)

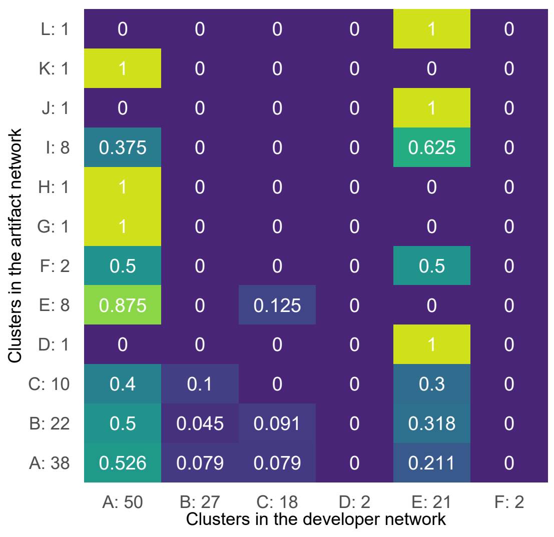


(c) File set comparison (overlap coefficient)

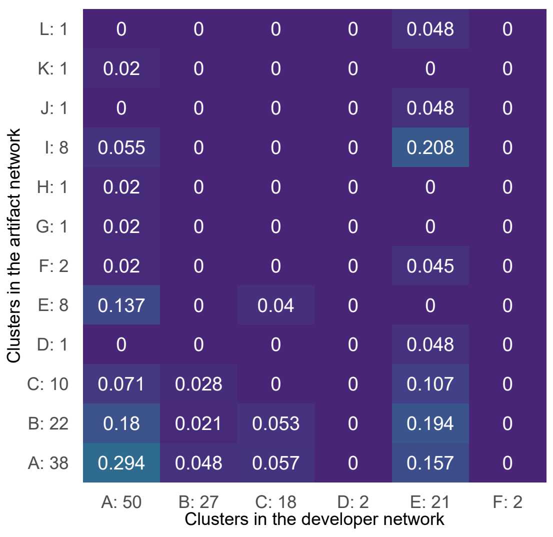


(d) File set comparison (Jaccard index)

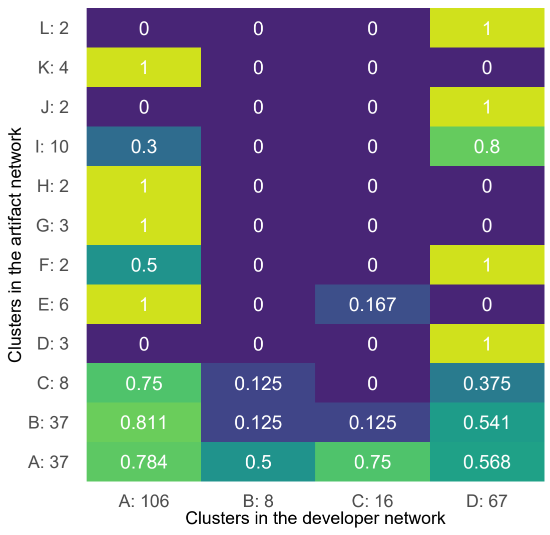
Figure A.7: Results of the cluster comparison for TypeScript for the observation window from July 2014 to January 2015, which is the first observation window in our project. Despite the developer communication network only containing two clusters, these clusters behave similarly to our general results, which is a characteristic that we did not encounter for developer communication networks with only two clusters that were built from other projects.



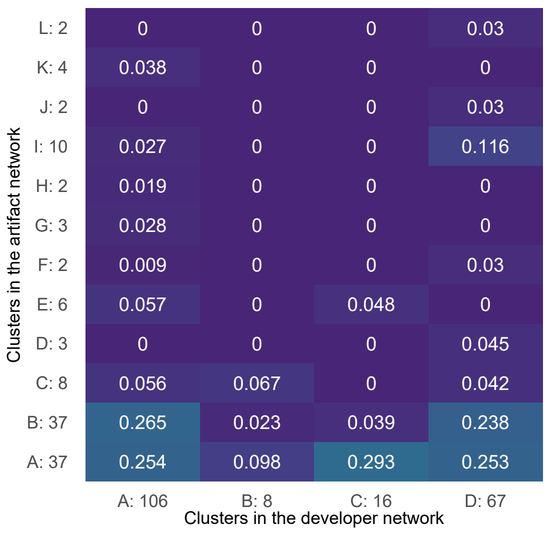
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)

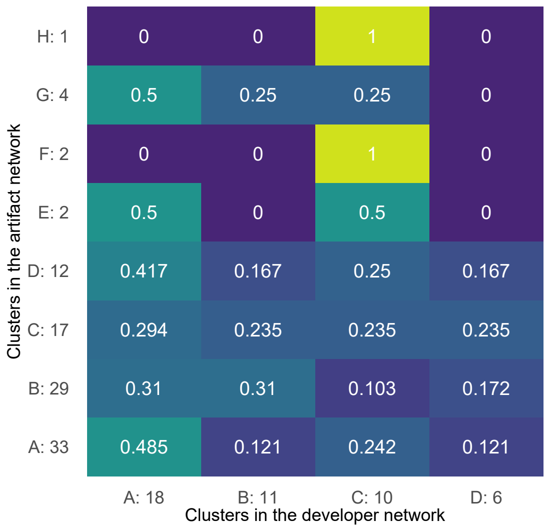


(c) File set comparison (overlap coefficient)

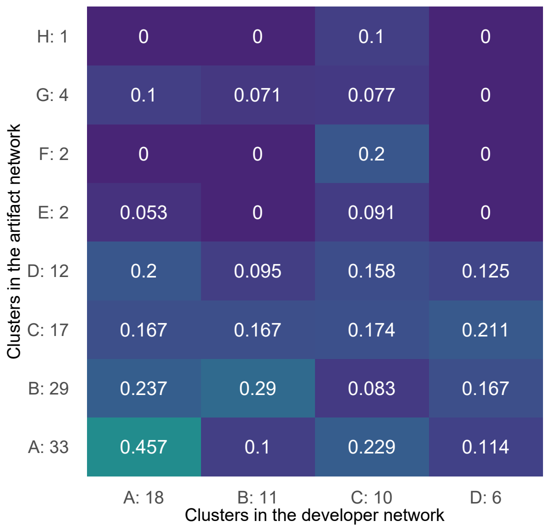


(d) File set comparison (Jaccard index)

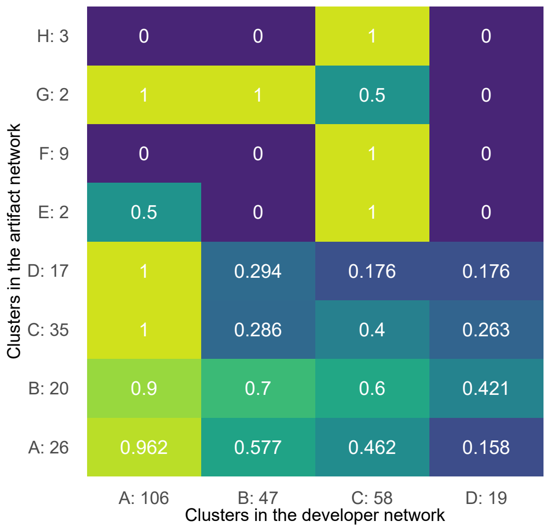
Figure A.8: Results of the cluster comparison for TypeScript for the observation window from July 2020 to December 2020, which is the last observation window for TypeScript.



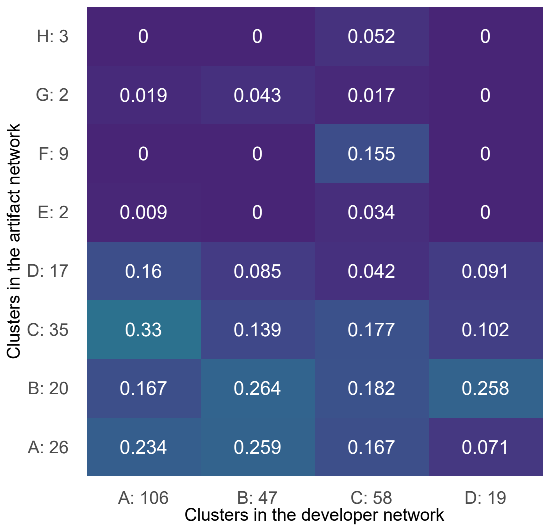
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)

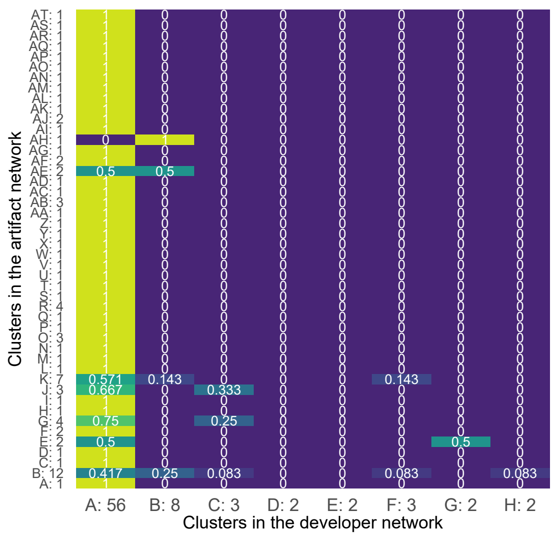


(c) File set comparison (overlap coefficient)

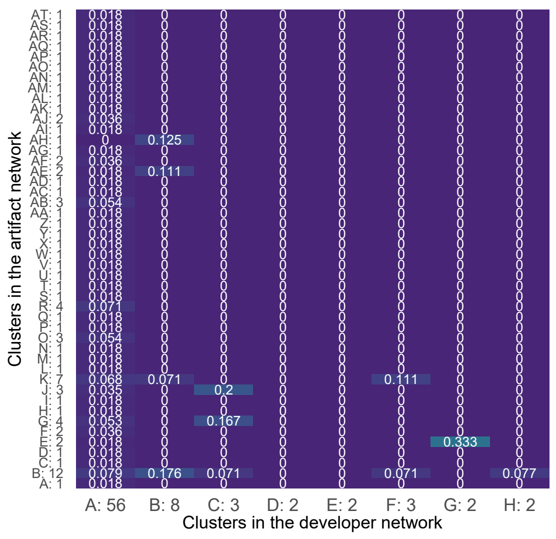


(d) File set comparison (Jaccard index)

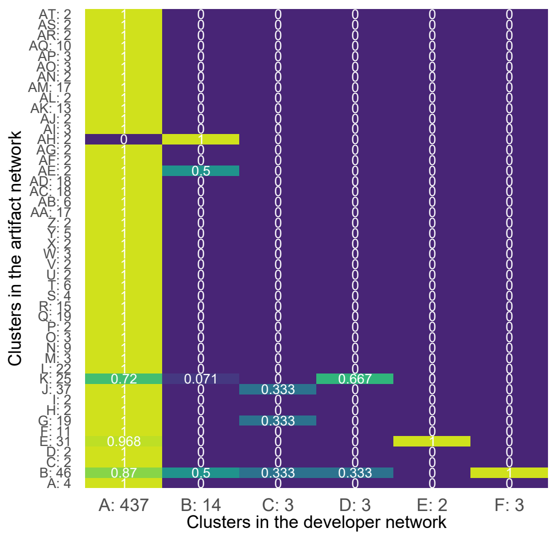
Figure A.9: Results of the cluster comparison for Moby for the observation window from January 2013 to July 2013. Despite this being the first observation window for Moby, a large part of the active developer community is already using issues.



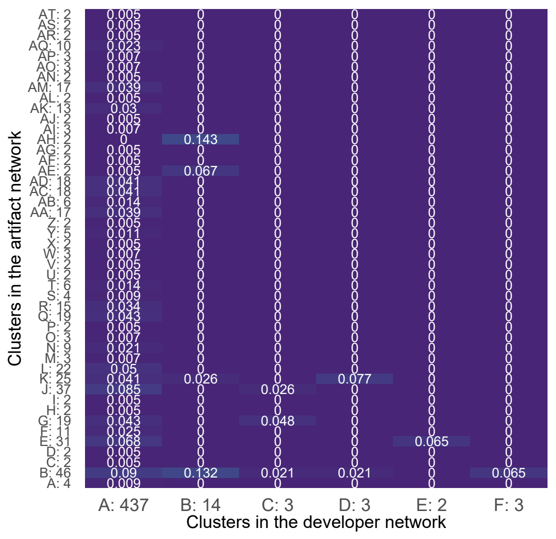
(a) Developer set comparison (completeness)



(b) Developer set comparison (Jaccard index)



(c) File set comparison (overlap coefficient)



(d) File set comparison (Jaccard index)

Figure A.10: Results of the cluster comparison for Moby for the observation window from January 2020 to July 2020, which is close to the end of our observation period. Most of the clusters in the artifact network are the result of changes made by a single developer. The results of Moby are unique compared to other projects since they feature a lot more small, inactive developer clusters.

BIBLIOGRAPHY

[1] Usman Ashraf, Christoph Mayr-Dorn, Atif Mashkoor, Alexander Egyed, and Sebastiano Panichella. “Do Communities in Developer Interaction Networks align with Subsystem Developer Teams? An Empirical Study of Open Source Systems.” In: *2021 IEEE/ACM Joint 15th International Conference on Software and System Processes (ICSSP) and 16th ACM/IEEE International Conference on Global Software Engineering (ICGSE)*. 2021, pp. 61–71.

[2] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. “Assessing Code Authorship: The Case of the Linux Kernel.” In: *Open Source Systems: Towards Robust Practices*. Springer International Publishing, 2017, pp. 151–163.

[3] Punam Bedi and Chhavi Sharma. “Community detection in social networks.” In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 6.3 (2016), pp. 115–135.

[4] Christian Bird, David Pattison, Raissa D’Souza, Vladimir Filkov, and Premkumar Devanbu. “Latent Social Structure in Open Source Projects.” In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, 2008, pp. 24–35.

[5] Christian Bird, David Pattison, Raissa D’Souza, Vladimir Filkov, and Premkumar Devanbu. “Latent Social Structure in Open Source Projects.” In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT ’08/FSE-16. Association for Computing Machinery, 2008, pp. 24–35.

[6] Tegawendé F. Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillère, Jacques Klein, and Yves Le Traon. “Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub.” In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 2013, pp. 188–197.

[7] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. “Fast unfolding of communities in large networks.” In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[8] Thomas Bock, Claus Hunsen, Mitchell Joblin, and Sven Apel. “Synchronous development in open-source projects: A higher-level perspective.” In: *Automated Software Engineering* 29.1 (2022), pp. 1–53.

[9] Thomas Bock, Angelika Schmid, and Sven Apel. “Measuring and Modeling Group Dynamics in Open-Source Software Development: A Tensor Decomposition Approach.” In: *ACM Transactions on Software Engineering and Methodology* 31 (2022), pp. 1–50.

[10] Ulrik Brandes and Thomas Erlebach. *Network analysis - Methodological foundations*. Springer Science & Business Media, 2005.

[11] Gerardo Canfora, Luigi Cerulo, Marta Cimitile, and Massimiliano Di Penta. “Social interactions around cross-system bug fixings: the case of FreeBSD and OpenBSD.” In: *Proceedings of the 8th working conference on mining software repositories*. 2011, pp. 143–152.

[12] Giuseppe Destefanis, Marco Ortu, David Bowes, Michele Marchesi, and Roberto Tonelli. “On Measuring Affects of Github Issues' Commenters.” In: *Proceedings of the 3rd International Workshop on Emotion Awareness in Software Engineering*. Association for Computing Machinery, 2018, pp. 14–19.

[13] Luiz Felipe Dias, Igor Steinmacher, Gustavo Pinto, Daniel Alencar Da Costa, and Marco Gerosa. “How Does the Shift to GitHub Impact Project Collaboration?” In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 473–477.

[14] P. Erdős and A. Rényi. “On Random Graphs. I.” In: *Publicationes Mathematicae* 6 (1959), pp. 290–297.

[15] Qiaona Hong, Sunghun Kim, S.C. Cheung, and Christian Bird. “Understanding a developer social network and its evolution.” In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 323–332.

[16] “ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary.” In: *ISO/IEC/IEEE 24765:2017(E)* (2017), pp. 1–541.

[17] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. “Mining and Visualizing Developer Networks from Version Control Systems.” In: *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*. Association for Computing Machinery, 2011, pp. 24–31.

[18] Mitchell Joblin. “Structural and Evolutionary Analysis of Developer Networks.” PhD thesis. Universität Passau, Germany, 2017.

[19] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. “Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics.” In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. IEEE Press, 2017, pp. 164–174.

[20] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. “Evolutionary Trends of Developer Coordination: A Network Approach.” In: *Empirical Software Engineering* 22 (2017), pp. 2050–2094.

[21] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. “From Developer Networks to Verified Communities: A Fine-Grained Approach.” In: *Proceedings of the International Conference on Software Engineering (ICSE)*. Vol. 1. 2015, pp. 563–573.

[22] Andrea Lancichinetti, Filippo Radicchi, José J Ramasco, and Santo Fortunato. “Finding statistically significant communities in networks.” In: *PloS one* 6.4 (2011).

[23] Michael Levandowsky and David Winter. “Distance between sets.” In: *Nature* 234.5323 (1971), pp. 34–35.

[24] Luis Lopez-Fernandez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. “Applying Social Network Analysis to the Information in CVS Repositories. Intl.” In: *Workshop on Mining Software Repositories (MSR)*. 2004, pp. 101–105.

[25] Luis López-Fernández, Gregorio Robles, Jesus Gonzalez-Barahona, and Israel Herraiz. "Applying Social Network Analysis Techniques to Community-Driven Libre Software Projects." In: *International Journal of Information Technology and Web Engineering (IJITWE)* 1 (2006), pp. 27–48.

[26] Luis López-Fernández, Gregorio Robles, Jesus Gonzalez-Barahona, and Israel Herraiz. "Applying Social Network Analysis Techniques to Community-Driven Libre Software Projects." In: *International Journal of Information Technology and Web Engineering (IJITWE)* 1 (2006), pp. 27–48.

[27] Vijaymeena M K and Kavitha K. "A Survey on Similarity Measures in Text Mining." In: *Machine Learning and Applications: An International Journal* 3 (2016), pp. 19–28.

[28] Gregory Madey, Vincent Freeh, and Renee Tynan. "The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory." In: *Proceedings of the Eighth Americas Conference on Information Systems*. 2002, pp. 1806–1813.

[29] Juan Martinez-Romo, Gregorio Robles, Jesus M. Gonzalez-Barahona, and Miguel Ortuño-Perez. "Using Social Network Analysis Techniques to Study Collaboration between a FLOSS Community and a Company." In: *Open Source Development, Communities and Quality*. Springer US, 2008, pp. 171–186.

[30] Mark EJ Newman. "Modularity and community structure in networks." In: *Proceedings of the national academy of sciences* 103.23 (2006), pp. 8577–8582.

[31] Gustavo A. Oliva, Francisco W. Santana, Kleverton C. M. de Oliveira, Cleidson R. B. de Souza, and Marco A. Gerosa. "Characterizing Key Developers: A Case Study with Apache Ant." In: *Collaboration and Technology*. Springer Berlin Heidelberg, 2012, pp. 97–112.

[32] Sebastiano Panichella, Gabriele Bavota, Massimiliano Di Penta, Gerardo Canfora, and Giuliano Antoniol. "How Developers' Collaborations Identified from Different Sources Tell Us about Code Changes." In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 251–260.

[33] Sebastiano Panichella, Gerardo Canfora, Massimiliano Di Penta, and Rocco Oliveto. "How the Evolution of Emerging Collaborations Relates to Code Changes: An Empirical Study." In: *Proceedings of the 22nd International Conference on Program Comprehension*. ICPC 2014. Association for Computing Machinery, 2014, pp. 177–188.

[34] Pascal Pons and Matthieu Latapy. "Computing communities in large networks using random walks." In: *International symposium on computer and information sciences*. Springer, 2005, pp. 284–293.

[35] Emad Shihab, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. "On the Central Role of Mailing Lists in Open Source Projects: An Exploratory Study." In: *New Frontiers in Artificial Intelligence*. Springer Berlin Heidelberg, 2010, pp. 91–103.

[36] Maarten van Steen. *Graph Theory and Complex Networks: An Introduction*. 2010.

[37] S.L. Toral, Rocio Martínez-Torres, and Federico Barrero. "Analysis of virtual communities supporting OSS projects using social network analysis." In: *Information and Software Technology* 52.3 (2010), pp. 296–303.

[38] Liang Wang, Ying Li, Jierui Zhang, and Xianping Tao. "Quantitative Analysis of Community Evolution in Developer Social Networks Around Open Source Software Projects." In: *arXiv preprint* (2022). Online first.

[39] Mairieli Wessel, Bruno Mendes de Souza, Igor Steinmacher, Igor S. Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A. Gerosa. "The Power of Bots: Characterizing and Understanding Bots in OSS Projects." In: *Proceedings of the ACM on Human-Computer Interaction* 2.CSCW (2018), pp. 1–19.

[40] Qi Xuan and Vladimir Filkov. "Building It Together: Synchronous Development in OSS." In: *Proceedings of the 36th International Conference on Software Engineering*. Association for Computing Machinery, 2014, pp. 222–233.

[41] Xiaolong Zheng, Daniel Zeng, Huiqian Li, and Feiyue Wang. "Analyzing open-source software systems as complex networks." In: *Physica A: Statistical Mechanics and its Applications* 387.24 (2008), pp. 6190–6200.