University of Passau

Department of Informatics and Mathematics

Master's Thesis

Analyzing the Evolution of Open Source Projects in the Face of Social and Organizational Change

Author:

Alexander Ebel

March 18, 2019

Advisors:

Prof. Dr.-Ing. Sven Apel

Chair of Software Engineering I

Claus Hunsen & Thomas Bock

Chair of Software Engineering I

Ebel, Alexander:
Analyzing the Evolution of Open Source Projects in the Face of Social and Organizational Change
Master's Thesis, University of Passau, 2019.

Abstract

Open source projects play an important role in modern software culture as they attract many software developers and consumers. Since many developers are contributing to the same repository and there often is no structured hierarchy, this construct seems to be very fragile. Thus, we are interested in the aftereffects of social and organizational change to an open source project covered by the media. Core developers leaving the project or the decision to switch to an open governance model are some example events. Also, we show the impact of major project releases on the repository. We use count-based information of the repositories such as developer count and commit count as well as network metrics to identify changes. We use plots which we create with the programming language R as well as tables to identify trends. In this thesis, we investigate the following repositories, which are hosted on the collaboration platform GitHub: Owncloud, Node.js, Qt, and Keras. We differ events by social or organizational cause. Additionally we separate the events by assuming positive or negative consequences. Depending on the event type, we can recognize changes through the commit data of each repository at the respected time period. In this thesis, we show that certain event groups do not necessarily imply the same consequences for count-based and network-based repository data.

Contents

List of Figures

List of Tables

1 Introduction

Open source projects occupy an important place in modern software development, as they are accessible for everyone and therefore often have a high participation number on collaborative platforms such as GitHub. Major motives why developers voluntarily work on open source projects are their need for the software, the enjoyment of the work itself and the enhanced reputation that may flow from making high-quality contributions [LVH04]. Node.js as an popular example for an open source project has currently over 2,000 contributors (as of February 2019) [nod].

In this thesis, we examine the resonance of open source software projects to special events happening within the particular community or leadership.

An example for a special event is the resignation of the Owncloud-Company founder Frank Karlitschek and other developers in April 2016. The founder quoted social and moral reasons for his decision. Five weeks after the event, Frank Karlitschek created a new fork called Nextcloud. As we expect both events to have an impact on the ownCloud project, we are going to analyze them.

In this work we investigate the effects of such events on the developer network and on count-based commit data. We expect those events to have consequences on the network structure or count-based data like lines of code or commits. In addition to special events, we also track main releases of projects. Node.js's major releases, for example, are cut from the master branch every six months. We also expect to see changes in terms of developer activity in those periods. We analyze the data from the Git repositories of our casestudies before and after a specific event happens. By doing so, we can compare both results and point out differences like the change of developer roles, for example. We use different network metrics to support our hypotheses.

In detail, in this thesis, we analyze four different well-known open source projects: ownCloud, Node.js, Qt, and Keras. We organize the events by choosing if the reason for the event has a social or organizational background. Also, we differ between positive and negative influence of the events. We found out that positive or negative social events can have a huge impact on the underlying data and network,

whereas organizational events tend to have a smaller impact but are often caused by previous downtrends.

In Chapter 2, we inform about previous work and the framework we use for creating our analyses. In the chapter thereafter, we step-by-step describe our approach to identify changes on the existing data. For analyzing the data, we use the programming language R, which is suited for statistical analysis and plotting graphs. In Chapter 4, we introduce our casestudies and the events we are interested in. We also take major releases into account. After that, we construct hypotheses for the events containing effects on count-based and network-based information in Chapter 5. In Chapter 6, we present and discuss the results which we get from our analyses for each project and check if our hypotheses hold. Finally, we have a conclusion for the complete thesis in Chapter 7.

2 Background

The emergence of distributed version control systems has led to the development of a new paradigm for distributed software development. Instead of pushing changes to a central repository, developers pull them from other repositories and merge them locally [GPD14]. GitHub is the pioneer in the field of distributed version control systems. As social applications on the web are getting more encouragement, it is only natural that collaboration platforms like GitHub follow suit by keeping the repositories of open source projects as transparent as possible. Previous work has shown that such transparency might lead to more commitment, higher quality, community significance, and personal relevance [DSTH12]. For our thesis, we can use the public available data for repository analysis.

Open source projects are not formally organized and do not have a pre-assigned command and control structure. Developers work on various areas by writing documentation, submitting bug reports, refactoring source code for different code areas of the code base, etc [BPD+08]. In comparison to commercial projects, open source projects do not have a fix organizational structure. This does not necessarily mean that commercial projects are more successful then open source projects. Henderson and Clark even point out that fixed organizational structures actually might hinder innovation [HC90]. Bird *et al.* [BPD+08] showed that even if open source projects do not have an organizational structure at first, subcommunities arise dealing with specific part of development. If we want to get a deeper understanding of how open source projects are organized, we also have to take a look at the onion model [NYN+02]. The onion model comprises several roles which appear in open source software projects. These roles include passive users of the software, testers, project leaders, core developers, contributing developers, etc [AVH10]. The model states that there is a clear and intentional expression of the substantial difference in scale between the group sizes fulfilling each role [JAHM17]. Several empirical studies substantiate this model and show that the number of code contributions per developer is described by heavy-tailed distributions, which means that a very small group of developers is responsible for performing the majority of work [MFH02, CWLH06, JAHM17]. Because only a very small group of developers takes care of most of the organizing work and additionally developers for open

source software projects have a high turnover rate, it is important to know how many developers can leave the project until the project starts to crumble. For this purpose, the truck factor helps to measure how prepared a project is to deal with developer turnover [APHV16]. The truck factor is defined as *"the number of people on your team that have to be hit by a truck (or quit) before the project is in serious trouble"* [WK03]. Avelino *et al.* [WK03] calculate the truck factor of a repository by following five steps. First, the latest point in the commit history is going to be checked out. Then, developers with multiple accounts have to be identified. In step 3, the history of each source-code file is being tracked. The next step defines the authors with their authored files. Finally, the truck factor can be calculated. The truck factor is a good example of what can be calculated by using commit data. We also analyze the change in developer count of projects, affected by social and organizational events. To be able to create such analyses, the open source software projects have to be mined and the results have to be visualized [JSS11]. For extracting the data from a GitHub repository, the tools Codeface[1] (by Siemens) and codeface-extraction[2], which are publicly available on GitHub, were used.

Now we will talk about methods for analyzing development data from a git repository. We have access to count-based data, like the commits of developers at a given time. With this kind of information, we can for example visualize the amount of active developers during a timespan. In open source software projects, we often encounter a huge amount of active developers. We can differ developers into core and peripheral. Core developers are characterized by prolonged, consistent, and intensive participation in the project. They often have extensive knowledge of the system architecture and have strong influence on decision making [JAM17]. On the other hand, peripheral developers are characterized by irregular, and often short-lived, participation in the project [JAM17]. As an example, we can use the commit count of developers to classify developers into core or peripheral at a certain time window. The amount of written code lines per developer can also be used to decide the developer role.

As coordination among developers is important in open source software projects, we are going to build networks which capture the cooperation between developers. Therefore we connect developers who worked on the same file. Figure 2.1 shows a small example network where the nodes represent developers and the edges between developers mean that they work on the same file. In this example, developer *Dev 1*, *Dev 2* and *Dev 3* worked on *File 1*. Additionally, developer *Dev 2* worked with *Dev 3* on *File 2* and with developer *Dev 4* on *File 3*. Those networks are used to understand the social behavior of developers and therefore represent socio-technical networks [JAM17]. If we want to evaluate developer networks, we can use network-based metrics to get information about the network in form of numeric values. In this thesis, we use network-based operationalizations for example the clustering coefficient and node degree of single developers. We talk more about those metrics in Chapter 3. Next to count-based operationalizations to determine the developer role, we can also use operationalizations on developer networks to identify the developer role. Another aspect is the stability of developer networks.

[1] https://github.com/siemens/codeface
[2] https://github.com/se-passau/codeface-extraction

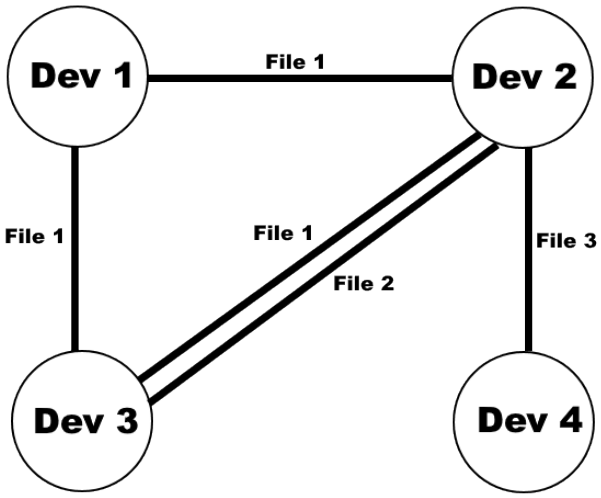

Figure 2.1: Example network with four developers working on three different files.

Developers in open source projects tend to have a high turnover rate. This might have an influence on the project. To identify the stability of a network, we can use a network-metric to calculate the scale-freeness of a network [JAM17]. If a network is in a scale-free state, it typically is robust against turnovers. We will talk more in detail about the scale-freeness of a network in Chapter 3. Joblin et al. dealt with developer networks and count-based information from GitHub repositories [JAM17]. The authors did an empirical study of 18 large open source projects to find out the evolutionary principles of developer coordination. They used network metrics like clustering coefficient and node degree for creating hierarchy diagrams and for classifying developer roles. They, for example, compared hierarchy diagrams of a project from an early state to a later state of the project. At the same time, count-based metrics were used to keep track of the developer count. The results contain the information that developers form a hierarchical structure at first and tend to become hybrid where only core developers are hierarchically arranged. The paper also concluded that projects exceeding 50 developers form scale-free networks in which the developer coordination is managed by a small number of developers as well as developers accumulate coordination with more developers over time [JAM17].

Joblin et al. covered the topic of identifying developer roles within open source projects by count-based and network-based approaches [JAHM17]. To support their results, the authors surveyed 166 developers to match the developers' perceptions with the results from the used count- and network-based metrics. Their paper describes the use of commit count, lines of code count, and mail count as count-based operationalizations to identify developer roles. In terms of network-based operationalizations, the authors used different approaches: Degree centrality was used to identify core-developers by counting the connections to other developers. The paper concluded that peripheral developers have only a limited number of interactions with the community. Another used metric was the eigenvector centrality to represent the importance of a developer by either connecting to many developers or by connecting to developers who are in a central position [JAHM17, BE05]. The authors also took the network hierarchy of the investigated projects into account to identify core and peripheral developers. In a study of 10 substantial open source projects, they found

out that commonly used count-based operationalizations for extracting developer roles are outperformed by network-based operationalizations [JAHM17].

We use a network library (CODEFACE-EXTRACTION-R) which contains functions for creating analyzable count-based information as well as networks by connecting developers working together. CODEFACE-EXTRACTION-R also contains functions to identify core and peripheral developers by different sources. The CODEFACE-EXTRACTION-R project on GITHUB was developed by the Chair of Software Engineering I at the University of Passau. In this thesis, we also use count-based and network-based operationalizations to identify core and peripheral developers at our dates of interest. We compare the results before and after an event to identify trends caused by this event.

3 Methodology

In this section, we are going to introduce our workflow. We split the casestudies into suitable time windows to gain expressive results. By matching the mined *git* data with our created time windows, we can build developer networks and retrieve count-based information for those periods. Additionally, we will describe the different type of data structures which we examine.

3.1 Workflow

In this thesis, we use commit data extracted from GitHub, for each of our casestudies. The data was extracted by the tool codeface-extraction which we referred to in the previous chapter. We now use the framework codeface-extraction-r to create expressive results from the commit data. Figure 3.1 shows the basic workflow of our analysis script.

We want to analyze the evolution of open source projects, affected by social and organizational events. We use the commit data of the git repository for every casestudy to gain count-based and network-based information. In step *a)* we create time windows for each casestudy. That means we split the whole project into uniform time windows from the start of the project until the end. For this we have decided to use three-month time periods, because previous work [JAM17] has shown that those time windows provide the best results in terms of plausibility. Figure 3.2 illustrates the lifetime of a project and how we split it into three-month time windows. We use this kind of splitting to initially get an overview for a complete casestudy. In step *b)* we now consider all the events and releases involved in our analyzes. For each event/release we create an individual splitting. Therefore, we also use three-month time windows with an overlap of 45 days. We start to cut the time windows six months before the event until six months after the event. By doing so, we get seven time windows where the first three and the last three windows do not contain the event at all. The fourth time window is where the event itself lies in the middle of the window. Figure 3.3 shows the splitting for an event/release in detail.

Once all time windows for a casestudy are constructed, we split the data by the time windows for our count-based information and additionally construct the developer

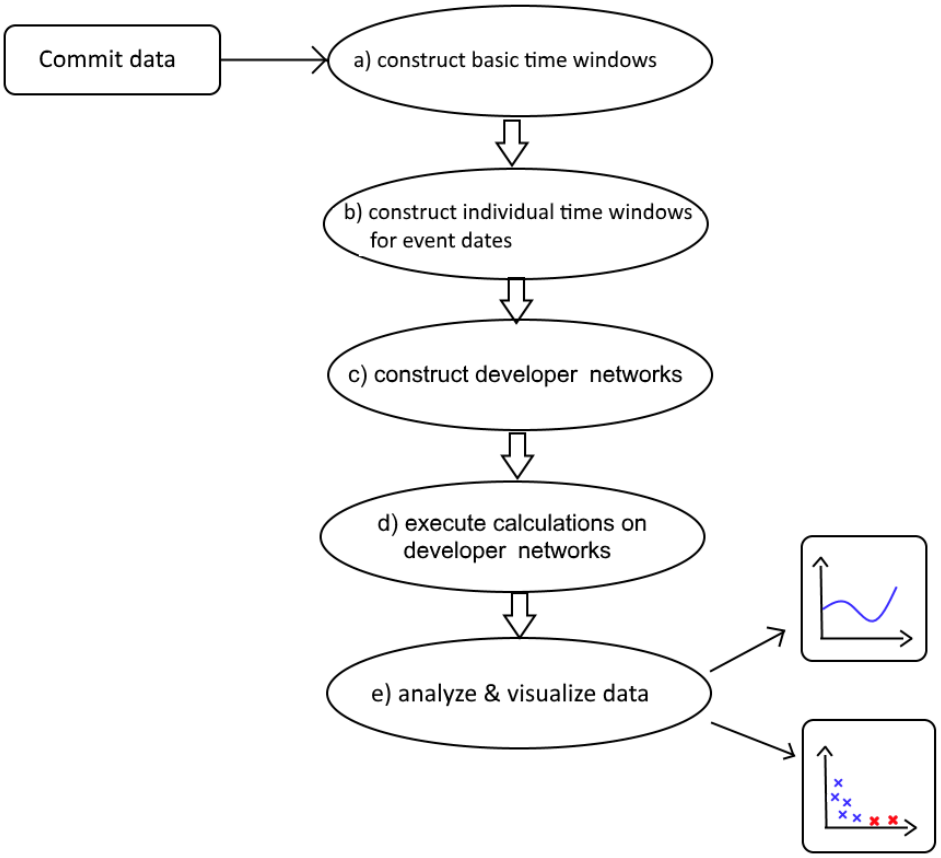

Figure 3.1: Basic analysis workflow. Step a) covers the process of generating time windows for a complete project. With step b), we create individual time windows for each event. Afterwards, we build developer networks for the constructed time windows in step c). Step d) represents count-based and network-based calculations. Step e) covers the analyzing and visualizing process.

networks based on the time windows for the network-based information. Step *c)* covers this process. As the network generating step takes much computing time, we only run that ones and store the results into a file. For further processing, we load the created file with all its information about the network.

Step *d)* is our actual analyze script for creating plots and calculating statistical information. We use the generated data and networks which are split by the time frames for creating count-based and network-based information. In the next sections, we will talk in detail about which data we are taking into account and how we are going to visualize it. The visualization and analysis are covered by the final step *e)*.

3.2 Count-Based Data

When we talk about count-based data we are primarily interested in the commit count and lines of code during a three month time window. We are also interested in the developer count which we split into core developer and peripheral developers. We also keep track of developers switching their role from one time window to the next time window, which we will cover in the role stability section (see Section 3.2.3).

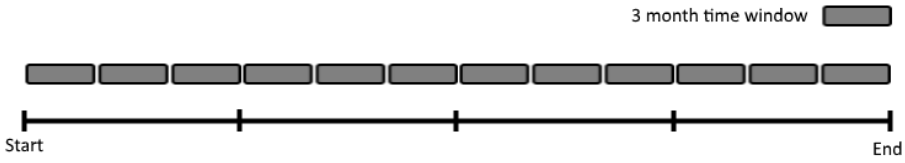

Figure 3.2: Splitting the complete project from start to end into three-month time windows.

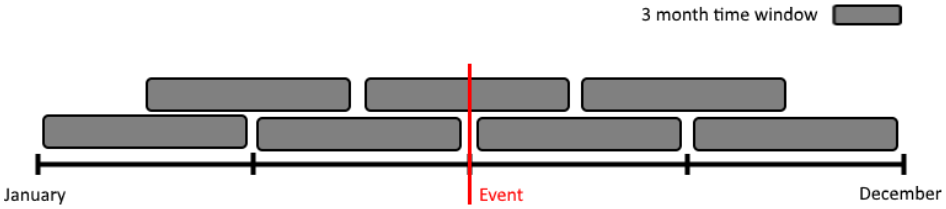

Figure 3.3: Splitting into three-month time windows with 45 days overlap for an individual event/release. The start is six months before the event and the end is six months after. In summary, there are seven time windows for an event.

3.2.1 Commits and Lines of Code

For each three-month time window, we are interested in the lines of code and commit count during that period. But we also have to take into account that the amount of written lines of code is not very expressive for a project. For example, computer generated code which was part of a commit would also count as lines of code, as well as deleted lines. Therefore, we have to be cautious with that information. We assume that the commit count on the other hand might be more expressive. Even though there might be regular commits caused by bots for cleaning up the project for example. For visualising that information, we apply the measured numbers for commit count and lines of code on the y-axis and provide the time line on the x-axis. Additionally, we will insert vertical lines at our chosen event- and release-dates of the respective repository. We do that for each individual event/release as well as for the whole project. Figure 3.4 shows an example plot for an event in the Node.js repository containing the amount of commits six months before and six months after the event. We create the same plot for lines of code where the amount of written code lines are represented by the y-axis. Additionally we generate the same plots for the whole project with the difference that we do not have overlapping time windows.

3.2.2 Developer Count and Role

An open source software project consists of peripheral and core developers, where core developers are consistent contributors and are essential for guidance and distributing tasks, whereas peripheral developers are persons who do not have much impact on the project but still have occasional commits. By investigating the amount of commits or written lines, we can use count-based metrics in order to point out core and peripheral developers. Communication between developers is not taken into account here. The network-based approach which considers connections between developers might be suited better for identifying core and peripheral developers. For

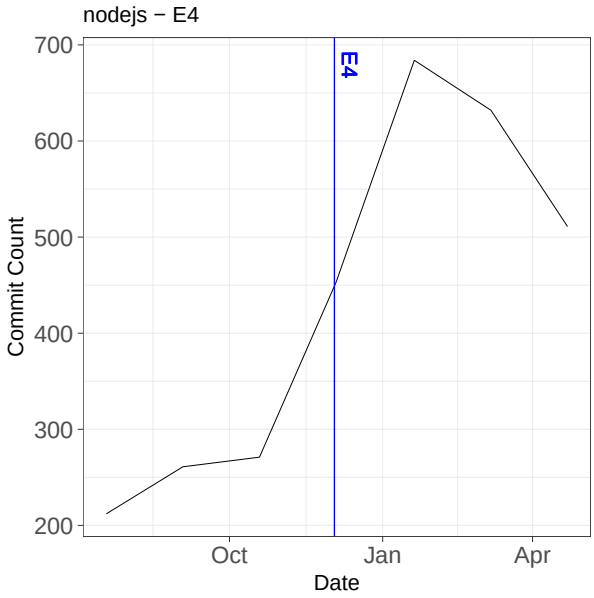

Figure 3.4: Example plot for visualizing the commit count for an event E4. The y-axis stores the amount of commits and the x-axis represents the timeline. Each point represents an overlapping three-month time window and not a concrete date.

the visualization, we choose the sum of core and peripheral developers to be on the y-axis, while we apply the time line on the x-axis. We will choose a three month time window to determine the developer role. We also will insert vertical lines at our chosen event- and release-dates of the respective repository. Figure 3.5 shows an example for a Node.js event, where the developer count is visualized. We not only create those plots for the events, but also for the complete project lifetime with the difference that we do not have overlapping time windows.

3.2.3 Role Stability

We want to visualize the role change of developers from one time window to the next time window. Because we will hypothesize predictions about change in developer roles at an event, we create expressive plots which show the role transition of core developers between two time frames. Those plots should visualize the amount of leaving and joining core-developers. Information like core developers moving to an inactive state or continuing as peripheral developers is depicted. We use the commit count to identify core and peripheral developers, because it is more suited than lines of code. Figure 3.6 shows an example role transition plot for an event.

3.3 Network-Based Data

Besides count-based data we also inspect the interaction between developers. Our created networks are undirected graphs, where the nodes represent the developers and the edges symbolize developers working on the same file. With that structure, we can calculate network metrics like average node degree or global clustering coefficient. We also can calculate the node degree and clustering coefficient for an individual developer. The node degree is the number of connections a node has to

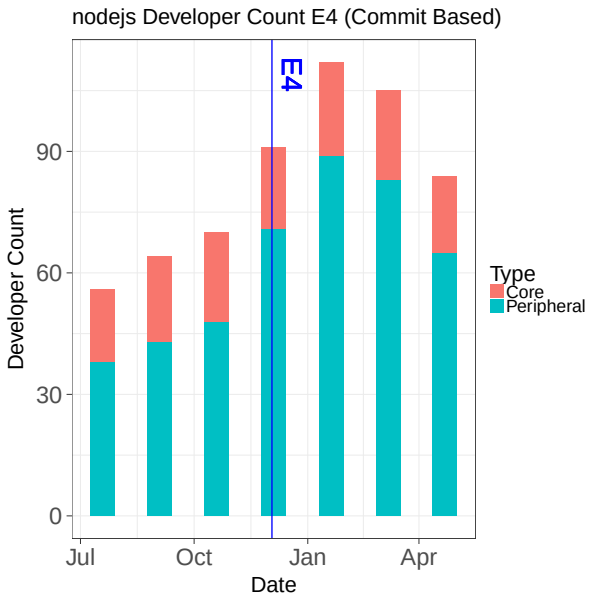

Figure 3.5: The bar plot shows the developer count for an event E4. The y-axis stores the count of developers and the x-axis represents the timeline. Each bar is split into core and peripheral, where core developers are marked red and peripheral are marked blue. The commit count was used to classify the developer roles. Each bar represents an overlapping three-month time window.

other nodes. The clustering coefficient describes the connectivity between neighbouring nodes of a node. Furthermore, we can determine if the network satisfies the conditions for scale-freeness. The state of scale-freeness is reached if there are hub nodes with an extraordinary large number of connections [DM13]. Those networks tend to be very robust and scaleable. An open source software project with a developer count greater than 50 usually is scale free [JAM17].

3.3.1 Developer Hierarchy

The local clustering coefficient and node degree of every developer is needed to build hierarchy networks. The relationship between node degree and clustering coefficient in a hierarchical network is described by $C(k) \propto k^{-1}$, where $C(k)$ is the clustering coefficient and k is the degree for a node [RB03]. In a hierarchical network, nodes at the top of the hierarchy have a high degree and a low clustering coefficient. Nodes at the bottom of the hierarchy have a low degree and a high clustering coefficient. The clustering coefficient describes the connectivity between neighboring nodes of a node. Figure 3.7 shows the clustering coefficient for the green node. As we can see, the clustering coefficient always is between zero and one. Zero means that there is no connectivity among the neighbors of the green node at all and one means that every neighbor of the node is connected to each other as well [BLM+06].

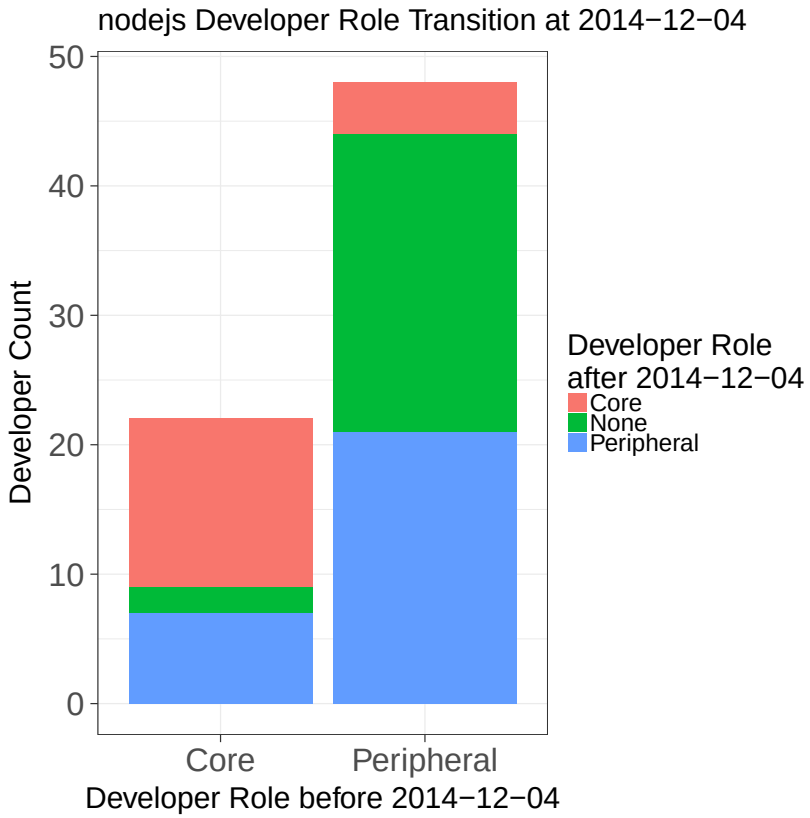

Figure 3.6: The bar plot visualizing the developer activity for two consecutive time windows. The first bar represents the core developers from the first time window. The colors express their movement towards a different developer type in the next time window, where red means that they are still identified as core developers. Blue means that they switched their role to peripheral and green means that they are not present at all in the new time window. The second bar represents the activity of peripheral developers from the first time window.

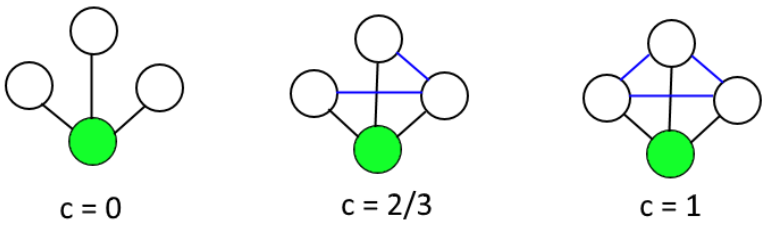

Figure 3.7: Example for the clustering coefficient for the green node.

The number of connections a developer has to other developers is called node degree. We can calculate the local clustering coefficient and node degree for each developer. To identify core and peripheral developers using the developer network structure, we need the node degree and clustering coefficient for each developer. Using this method to determine core and peripheral developers is more reliable than just using count-based data. This was proofed by Joblin et al. [JAHM17].

For visualizing this information, we choose to apply the logarithmic node degree on the x-axis and the logarithmic clustering coefficient on the y-axis. The result plot where each point represents a single developer can be used to visualize core and peripheral developers. Core developers are positioned at the bottom-right at the coordinate system. Developers with high degree tend to have edges that span across cohesive groups, thereby lowering their clustering coefficient. In Chapter 2, we mentioned that open source projects do not have an organizational structure at first and subcommunities arise dealing with specific part of development [BPD+08, JMA+15]. As core developers coordinate the effort of these subcommunities, they tend to have higher node degrees and smaller clustering coefficients. Peripherals are placed at the top left having higher clustering coefficients and smaller node degrees [JAHM17]. To classify core and peripheral developers we calculate a hierarchy value $\frac{deg}{cc}$ for each developer, where *deg* is the node degree and *cc* is the clustering coefficient. We order the developers by their hierarchy value and compute a threshold at the 80% percentile. Developers that have a value above the threshold are considered core, developers below are considered peripheral [JAHM17, CWLH06, MFH02]. The challenge here is to visualize the difference between two consecutive time windows. Additionally, we need to adjust the size of the x- and y-axis to be the same for both plots. Figure 3.8 shows an example how we visualize hierarchy for a release date of the KERAS project.

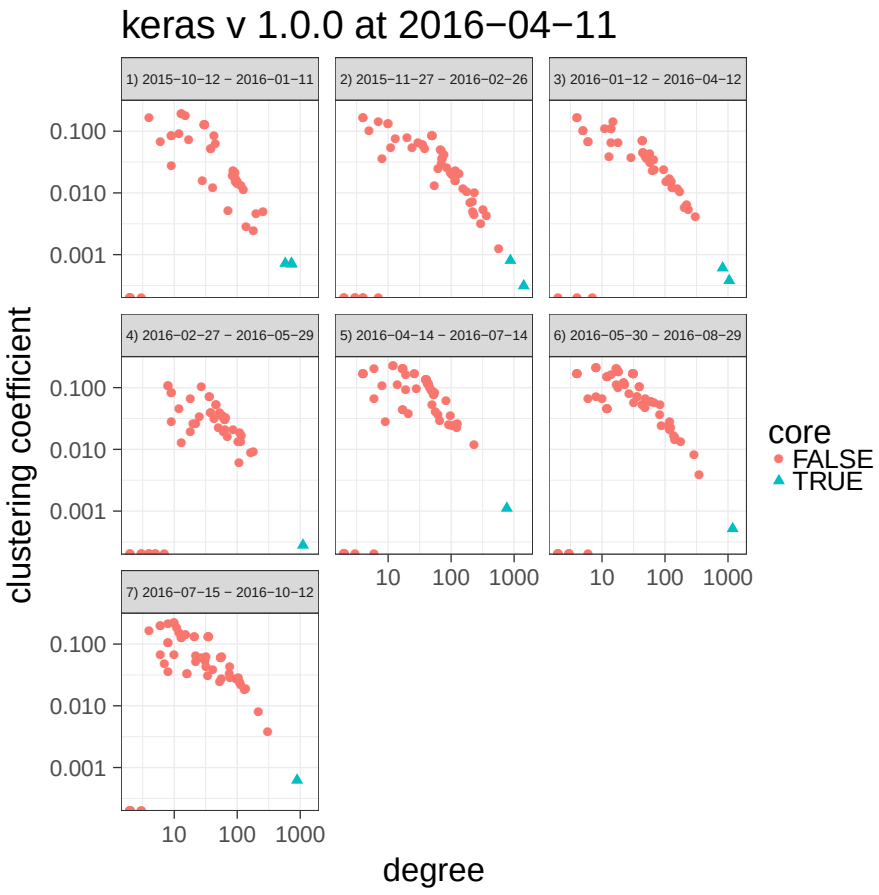

Figure 3.8: Example for seven consecutive hierarchy diagrams of a KERAS release. Each plot represents an overlapping three-month time window.

3.3.2 Scale-Freeness

Large networks sometimes show different characteristics for the degree of a node. If the node degree distribution follows a power law, those networks are called scale free [BA99, BEJ18]. Those networks tend to be very robust and scaleable. More than 50 developers being in a network often implies scale-freeness [JAM17]. CODEFACE-EXTRACTION-R provides us with the function to determine whether a network is in a scale-free state or not. For visualizing this kind of information, we use *True* and *False* on the y-axis symbolizing whether the network is in a scale-free state or not. A time line is provided on the x-axis. Figure 3.9 shows an example plot for a release of the KERAS project, displaying changes to the scale-freeness state of the networks.

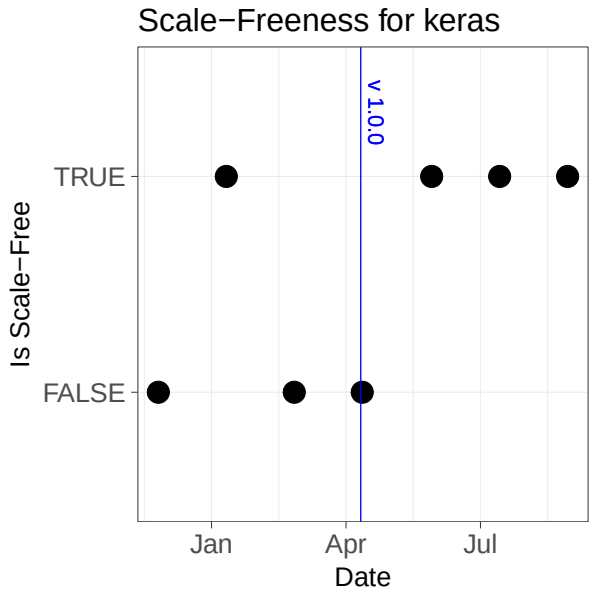

Figure 3.9: Scale-freeness example for a KERAS release. Each point represents an overlapping three-month time window and not a concrete date.

4 Case Studies

In this section, we present four different software projects which use Git as version control system. First, we provide basic information for each project, like the current contributor count, fork count and historical background. If available, we examine release periods to learn at which dates we should investigate the data later on. After that, we choose some major events which we expect to have consequences for the project. We group all events into four different event-groups: negative social events, positive social events, negative organizational events, and positive organizational events. In Section 4.5, we explain why those groups exist and why we grouped certain events into its respective group. We will then hypothesize on those consequences in Chapter 5 and check those in Chapter 6.

4.1 ownCloud

ownCloud is a client-server software for creating and using file hosting services founded by Frank Karlitschek. Development on GitHub started in January 2010. The corresponding company ownCloud Inc. was founded in 2011. The main programming language for ownCloud is PHP. The repository[1] currently has 455 contributors and was forked over 1800 times (as of September 2018).

4.1.1 Release Dates

As ownCloud has not a fixed schedule for their main releases, we pick them by choosing releases which have a new main version number. In addition, we pick release version 7.0.15 and 9.1 because they overlap with our event dates. In Table 4.1, we show all releases, starting with 1.0 in June 2010 and ending with 10.0.0 in April 2017.

[1] https://github.com/owncloud/core

version	release date
1.0	2010-06-24
2.0	2011-10-11
3.0	2012-01
4.0	2012-05-22
5.0	2013-03-14
6.0	2013-12-11
7.0	2014-07-23
7.0.15	2016-05-13
8.0	2015-02-09
9.0	2016-03-08
9.1	2016-07-21
10.0.0	2017-04-27

Table 4.1: ownCloud release dates from June 2010 to April 2017.

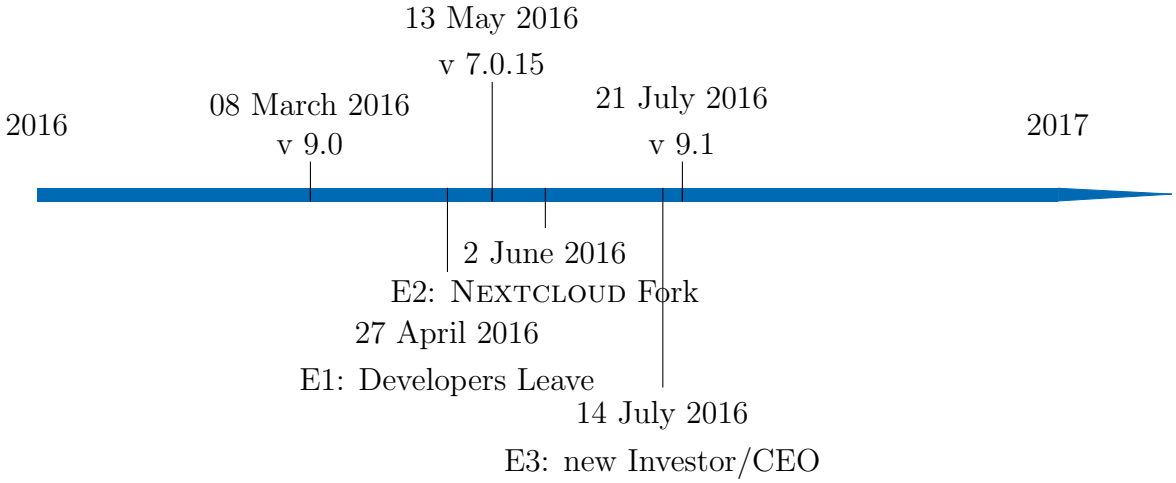

Figure 4.1: ownCloud event time line from 2016 to 2017.

4.1.2 Resignation of the Founder & Core-Developers – E1

In April 2016, founder Frank Karlitschek and other core developers leave the company. Karlitschek served as the chief technology officer of ownCloud Inc. and stated moral reasons for his resignation. The following quote is taken out from his blog:

> *With ownCloud 9.0 we released a huge milestone and an amazing product developed in close collaboration between the ownCloud company and the ownCloud community.* [Kar16a]

> *I thought a lot about this situation. Without sharing too much, there are some moral questions popping up for me. Who owns the community? Who owns ownCloud itself? And what matters more, short term money or long term responsibility and growth? Is ownCloud just another company or do we also have to answer to the hundreds of volunteers who contribute and make it what it is today?* [Kar16a]

Karlitschek writes about doubts and reasons why he is leaving the company. As we see here, he was an active part of the development until release 9.0. His resignation was also covered by the media [Gr6].

Management turnover is found to be negatively correlated with its consequences for the company. An improving performance after turnover can generally not be shown. CEO turnover often leads to diffusion of authority as well as change, regardless of direction [FK90, Mil93]. As Frank Karlitschek was the CTO of OWNCLOUD Inc., he was part of management and thus we evaluate this event because we expect to see changes. As this event was caused by moral conflicts, we associate this to a negative social event.

4.1.3 Start of Nextcloud which is an ownCloud Fork – E2

On June 02, 2016, Karlitschek created a fork of OWNCLOUD, which goes by the name NEXTCLOUD. In addition he wrote on his blog:

> *So today we are forking ownCloud into a new open source project called Nextcloud and we are also founding a new company called Nextcloud GmbH to offer Nextcloud software and services for companies and bigger organizations.* [Kar16b]

This fork is a consequence from Frank Karlitschek leaving the company five weeks before. Therefore, we assume this to be correlated to E1 and we identify the event as a negative social event as well.

4.1.4 New CEO and Finance Investments for ownCloud – E3

In July 2016, the OWNCLOUD company gets millions of dollars of finance investments and a new CEO [Bö16]. Rapid change of the OWNCLOUD management covered by *E1* probably led to this consequence several months after. We previously noted that a change in CEO often leads to negative consequences in terms of management. But as this event also comes with new investment money supporting the project, we classify this event to a positive organizational event.

4.2 Node.js

NODE.JS is a cross-platform JAVASCRIPT run-time environment that executes JAVASCRIPT. The repository[2] has over 2000 contributors and was forked over 12000 times. NODE.JS is often being used for back-end development of websites as it is suited to realize web servers. JOYENT is the development company which acted as sponsor at the very beginning of NODE.JS.

[2] https://github.com/nodejs/node

4.2.1 Release Dates

Major releases of Node.js are cut from the master branch every six months. Table 4.2 lists all releases which are of interest for us from February 2015 to October 2017.

version	release date
0.12.x	2015-02-06
4.x	2015-09-08
5.x	2015-10-29
6.x	2016-04-26
7.x	2016-10-25
8.x	2017-05-30
9.x	2017-10-01

Table 4.2: Node.js release dates from February 2015 to October 2017.

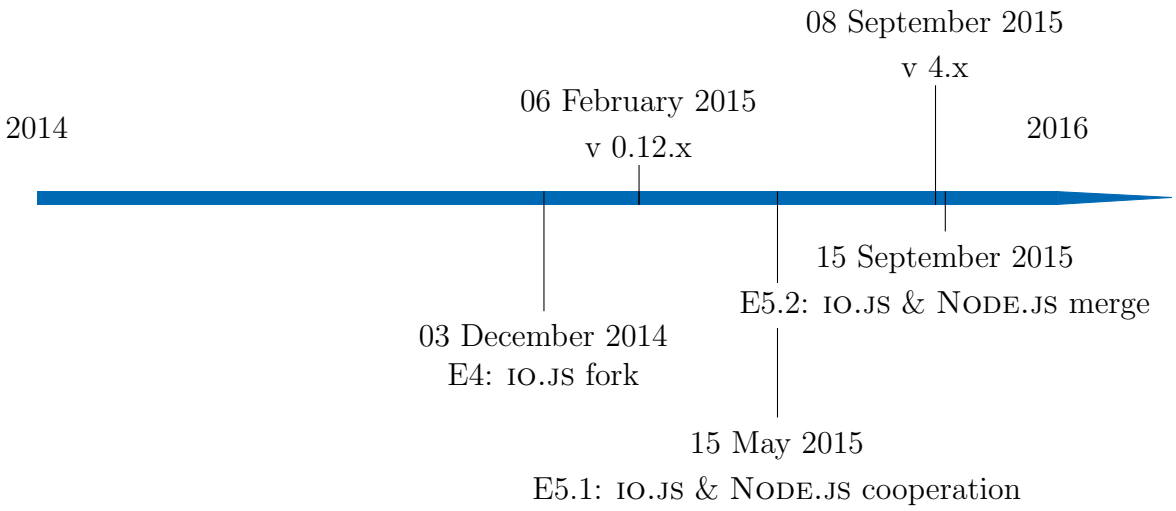

Figure 4.2: Node.js event time line from 2014 to 2016.

4.2.2 io.js Fork Caused by Dissatisfaction due to the Open Development Regulations – E4

In January 2014, many core developers were not satisfied with Joyent's efforts for an open development of the Javascript framework. This led to the fork io.js. The new fork uses the open governance model. The details for this model were introduced at the start of the fork [Amo15]. Those notes contained information about the tasks and authority of the technical committee. Additional information like technical committee meetings and technical committee membership are covered. Also the state of developers becoming contributors are written down in detail [Amo15]. Because

this event was triggered due to social conflict of some contributors we associate this event to be a negative social event.

4.2.3 Cooperation between io.js and Node.js – E5

On May 2015, both projects announced their cooperation to merge both projects into the new Node.js-Foundation. Until everything was settled, there were many discussions with contributors, developers and the leadership of both communities. Finally, the io.js community voted for joining the Node.js community once again. Until the merge is complete, io.js announces to continue their scheduled releases. We split the event into *E5.1* and *E5.2* where *E5.1* indicates the start of the cooperation and *E5.2* represents the end of the io.js and Node.js merge. The merge took place continuously from 15 Mai 2015 to 15 September 2015 [Dol15]. Because this event is correlated to *E4*, as a result after the social conflicts were solved, we associate this event to be a positive social event.

4.2.4 Ayo Fork Caused by Behavioral Code Violations – E6

In August 2017, there were again problems between the leadership and the community, because a member of the technical steering committee and the core technical committee violated behavioral codes and was not banned from the project. That did not only lead to disagreement within the leadership, but also to a new fork named Ayo. The new fork was mainly created out of protest against violations to the code of conduct. In addition to the new fork, three persons of the technical steering committee left Node.js. When we look at the progress of the Ayo repository, we can see that development stopped till mid of December 2017 [MS17, Baa17]. We associate this event to be a social negative event.

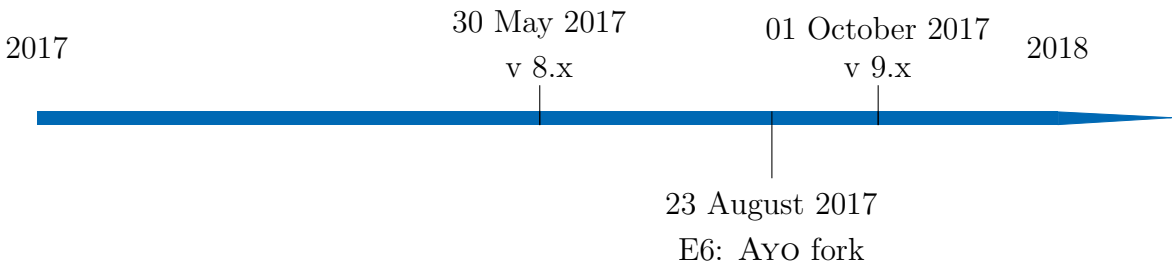

Figure 4.3: Node.js event time line from 2017 to 2018.

4.3 Qt

Qt is a cross-platform software framework with ready-made UI elements, C++ libraries, and a complete integrated development environment. Qt was developed with C++. Nokia acquired Qt in June 2008 and sold the commercial licensing part of Qt to Digia in March 2011. In August 2012, Digia fully acquired Qt from

Nokia, shortly before release Qt 5.0 [War12]. The repository[3] now has over 600 contributors and was forked over 300 times. The Qt repository is being regularly mirrored between qt-project.org and GitHub.

4.3.1 Release Dates

For Qt, we use two different repositories. For every release and event which happened before release Qt 5.0, we use the Qt4 repository[4]. We use the Qt5 repository[3] for every event and release past Qt 5.0. As Qt had many different releases during its lifetime, we only choose those which are near our dates of interest. Table 4.3 lists all those release dates.

version	release date
4.8	2011-12-15
5.0	2012-12-19
5.1	2013-07-03
5.2	2013-12-12
5.3	2014-05-20
5.4	2014-12-10

Table 4.3: Qt release dates from December 2011 to July 2015.

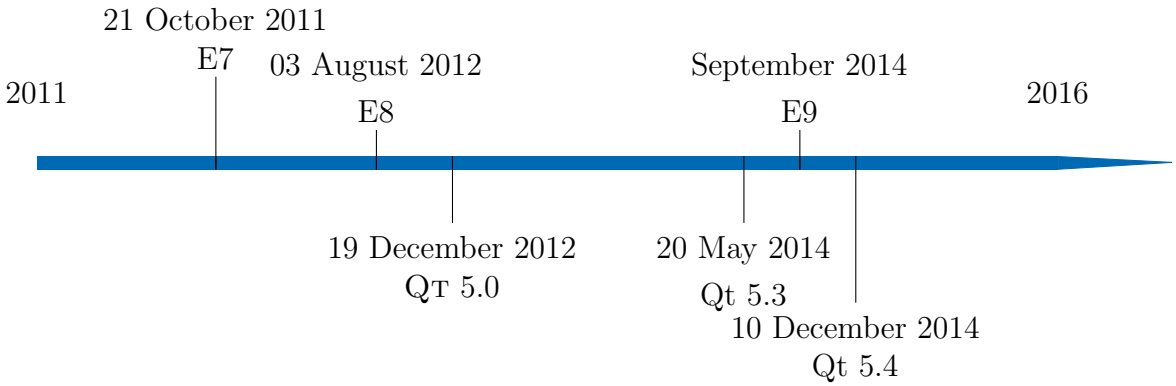

Figure 4.4: Qt event time line from 2011 to 2016.

4.3.2 Switch to an Open Governance Model – E7

On October 2011, the Qt development was under the open governance development model at the Qt project website. Previously, the qt code base was under open source. That means that all the source code is available to all developers in the Qt ecosystem, but the code development processes was primarily done by the paid

[3]https://github.com/qt/qtbase
[4]https://code.qt.io/cgit/qt/qt.git/

Qt development team. Whereas in the open governance model, the development model was changed in order to accept contributions from any member of the Qt ecosystem [qt:15, Kno11]. The QT development for release 5.0 was done under the open governance model. QT 5.0 was released about a year after the introduction of this new model. When this new model was introduced the development switched to a different repository. For this event, we only take the QT5 repository into account when mining and evaluating the data. The QT5 repository allows any contributor to take part in the development. We assume this event to be a positive organizational event.

4.3.3 Closing of the Qt-Development Department in Brisbane – E8

Nokia closes the Qt-development department in Brisbane till the end of August 2012 which consists of 60 employees. Three employees should handle open tasks until the end of December 2012 [Nan12]. This led to the question if those employees will continue their work on Qt in the community. During that time in September 2012 QT 5.0 was released. We associate this event to be a negative organizational event. We use both repositories to analyze this event.

4.3.4 The Qt-Company Takes Over the Qt-Business and Copyrights – E9

In September 2014, Digia transfers the QT-business and copyrights to their subsidiary company named "QT-Company" [Duc14]. As everything is now being handled by a single institution, we expect management to become more transparent and controlled. Thus we associate this event to be a positive organizational event. We use the QT5 repository to analyze this event.

4.4 Keras

KERAS is a neural-network library written in PYTHON and especially suited as a deep learning library. KERAS currently has over 700 contributors and was forked over 14,000 times. KERAS is capable of running on top of machine learning libraries such as TENSORFLOW, CNTK and THEANO[5].

4.4.1 Release Dates

For KERAS, we only pick release dates which are close to our picked event-dates. We are interested in the release dates covered by table Table 4.4 from December 2015 till November 2017.

version	release date
v 0.3.0	2015-12-01
v 1.0.0	2016-04-11
v 2.0.8	2017-08-25
v 2.0.9	2017-11-1

Table 4.4: KERAS release dates from December 2015 to November 2017.

[5]https://github.com/keras-team/keras

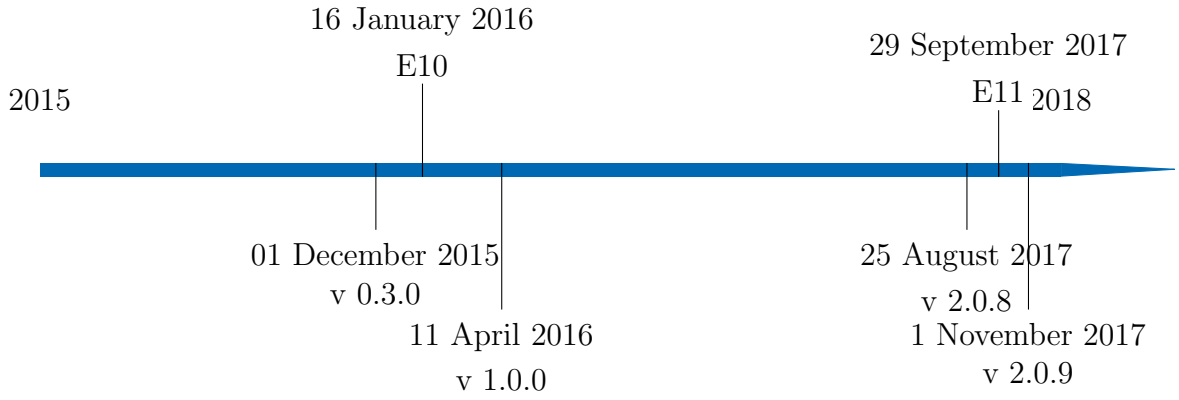

Figure 4.5: KERAS event time line from 2015 to 2018.

4.4.2 Cooperation Between Google and Tensorflow Announced – E10

The founder of Keras, François Chollet, who is a GOOGLE developer, announced the cooperation between TENSORFLOW and GOOGLE in the following reddit post:

> *Keras is gaining official Google support, and is moving into contrib, then core TF. If you want a high-level object-oriented TF API to use for the long term, Keras is the way to go [16].*

We associate this event to be a positive social event. The reason for the decision that this event represents a social event instead of an organizational one is that the opinion of contributors towards the cooperation might change. Developers might consider to stay or to leave the project.

4.4.3 Stop of Theano Support – E11

Announcement of KERAS not supporting THEANO after THEANO release 1.0 [Lam17]. This decision seems to be an aftereffect of E10, because TENSORFLOW which belongs to GOOGLE went into cooperation with KERAS about two years before. We assume that event to be a negative organizational event.

4.5 Event-Groups

We group multiple events which we consider alike. We expect those groups to have similar consequences for the data, which we examine.

Group one represents events, which were caused by negative social reasons. Core-developers were unsatisfied with organizational decisions and quit their contribution. Additionally all those events led to new forks. For Group 1 we pick events E1, E2, E4, and E6.

Group 2 represent events which also were caused by social reasons, but we expect them to have positive impact on the project. For group two we pick events E5 and E10. Both events lead to a cooperation of two different communities. In E5, we know definitely that changes are going to happen to the main project because a continuous merge of a fork was announced. In E10, a cooperation was announced but without a fixed schedule.

Groups 3 and 4 represent events, which have an underlying organizational change. We assume those changes to have a positive effect on the repositories. For group 3 we pick E8, E11: Both events are about to shut down: In E8, a development department shuts down and E11 was the announcement of stopping support for a third-party software.

Whereas Group 3 is about negative side effects of organizational change, Group 4 covers positive consequences for the projects. For Group 4, we pick E3, E7 and E9. We expect E3 to lead to positive change, caused by a new CEO and investment money. In E7 the Qt project switches to an open-governance model which is appreciated by open source developers. For E9, we cannot predict if the organizational change will have a positive effect on the repository. But transferring the Qt-Business into a representative company handling everything from there should be beneficial.

Project	Event	Description	Date
Group 1 - Social Events (negative) – (G1)			
ownCloud	E1	Resignation of the Founder & Core-Developers	2016-04-27
ownCloud	E2	Start of Nextcloud which is an ownCloud Fork	2016-06-02
Node.js	E4	io.js Fork Caused by Dissatisfaction due to the Open Development Regulations	2014-12-03
Node.js	E6	Ayo Fork Caused by Behavioral Code Violations	2017-08-23
Group 2 - Social Events (positive) – (G2)			
Node.js	E5	Cooperation between io.js and Node.js	2015-05-15
Keras	E10	Cooperation Between Google and Tensorflow Announced	2016-01-16
Group 3 - Organizational (negative) – (G3)			
Qt	E8	Closing of the Qt-Development Department in Brisbane	2012-08-03
Keras	E11	Stop of Theano Support	2017-09-29
Group 4 - Organizational (positive) – (G4)			
ownCloud	E3	New CEO and Finance Investments for ownCloud	2016-07-14
Qt	E7	Switch to an Open Governance Model	2011-10-21
Qt	E9	The Qt-Company Takes Over the Qt-Business and Copyrights	2014-09

Table 4.5: All events grouped by event-group.

5 Hypotheses

In the previous section, we listed some major events to our projects of interest. We also grouped them by social and organizational events. We now will hypothesize the effects on our data during those time periods for each group we created in the previous chapter.

5.1 Hypotheses for Group 1 (G1)

The effects of negative social events are covered by *Hypothesis 1.* We expect decreasing commit and changed line of code numbers because we assume many developers to leave and stop continuing their development. As we think that the developer count will decrease, we also suspect the node degree to drop for core developers. On the other hand, the clustering coefficient should increase because there will be less neighbouring nodes. We only hypothesize about changes to core developers because peripheral developers are not very expressive for the projects. As we think that the projects can loose stability, it might effect the scale-freeness in a negative way.

Hypothesis 1.1. *The lines of code and the commit count after the event will decrease.*

Hypothesis 1.2. *In general, the projects will loose contributors. More than the usual amount of developers which were previously identified as core developers will not be identified as core in the future.*

Hypothesis 1.3. *If the repository was in a scale-free state before, it might loose it.*

Hypothesis 1.4. *The clustering coefficient will get higher and the node degree will become smaller for core developers in the network hierarchy.*

5.2 Hypotheses for Group 2 (G2)

The effects of positive social events are covered by *Hypothesis 2.* As we expect many developers to join the project, it is only natural to think about increasing

commit and changed line of code numbers. With an increasing developer count, the node degree should increase. At the same time, the clustering coefficient should decrease due to an increase in neighbouring nodes. As the projects gain stability and developers they might hit the state of scale-freeness.

Hypothesis 2.1. *The lines of code and the commit count after the event will increase.*

Hypothesis 2.2. *In general, the projects will get more contributors. There will be more and new identified core developers.*

Hypothesis 2.3. *If the developer network was not in a scale-free state, then it is going to reach it.*

Hypothesis 2.4. *The clustering coefficient will slowly decrease, whereas the node degree will increase for core developers.*

5.3 Hypotheses for Group 3 (G3)

The effects of negative organizational events are covered by *Hypothesis 3*. We expect similar effects to the count-based and network-based information as we did for Group 1.

Hypothesis 3.1. *The lines of code and the commit count after the event will decrease.*

Hypothesis 3.2. *The amount of contributing developers will decrease.*

Hypothesis 3.3. *If the project was in a scale-free state, it will loose it and slowly recover into a scale-free state.*

Hypothesis 3.4. *The clustering coefficient for core developers will get higher, whereas the node degree will get lower.*

5.4 Hypotheses for Group 4 (G4)

The effects of positive organizational events are covered by *Hypothesis 4*. We expect similar effects to the count-based and network-based information as we did for Group 2.

Hypothesis 4.1. *The lines of code and the commit count after the event will increase.*

Hypothesis 4.2. *The amount of contributing developers will increase.*

Hypothesis 4.3. *If the project was not in a scale-free state, it will gradually achieve it.*

Hypothesis 4.4. *The clustering coefficient for core developers will decrease, whereas the node degree will increase.*

In the next chapter, we are going through the results for the events of each case study and check the hypotheses.

6 Results

In this chapter, we go step by step through each hypothesis and present and discuss the results of our analysis. We compare the results for each event in the specific event group. For each hypothesis, we conclude whether it holds or not.

6.1 Evaluation for Negative Social Events (G1)

Hypothesis 1 contains statements about the development of ownCloud and Node.js, which are affected by negative social events.

6.1.1 Hypothesis 1.1: Lines of Code and Commit Count

Hypothesis 1.1. *The lines of code and the commit count after the event will decrease.*

To prove this hypothesis, we use plots which contain the number of written code lines and commits during a three-month time window. When looking at the plots for *E1*, the resignation of the founder and core developers, in Figure 6.1 the number of commits drops drastically. The same is noticeable when looking at *E2*, the start of Nextcloud, in Figure 6.2. But at the same time the sum of written lines of code had a huge peak. Because that is an unexpected unproportional behaviour, we looked closer at the commits done during this period. We found out that two developers were involved in moving files from one place to another which caused them to have over 550,000 changed lines of code. The extraction tool accumulates the deleted lines and added lines even though the file was only moved from one place to another. That way we can conclude that the graphics for the amount of lines of code are not expressive. The events *E1* and *E2* are close to each other and thus both Figure 6.1 and Figure 6.2 contain the misleading number of written code lines.

When looking at events *E4* and *E6* in Node.js, we have a completely different behavior than expected. Instead of decreasing numbers in commits and lines of code, we have increasing numbers (see Figure 6.3 and Figure 6.4). For *E6*, the

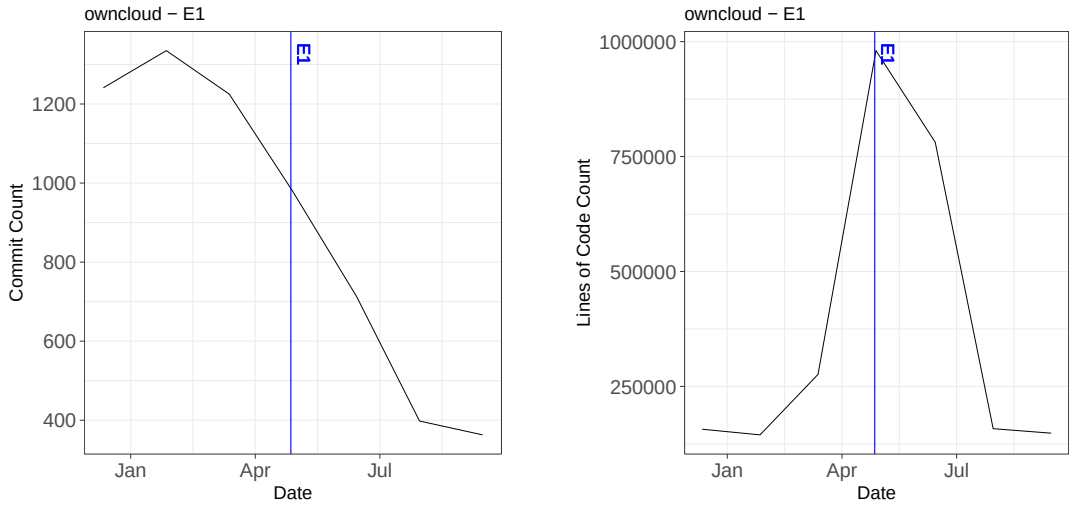

Figure 6.1: Commit count and changed lines of code count for ownCloud E1.

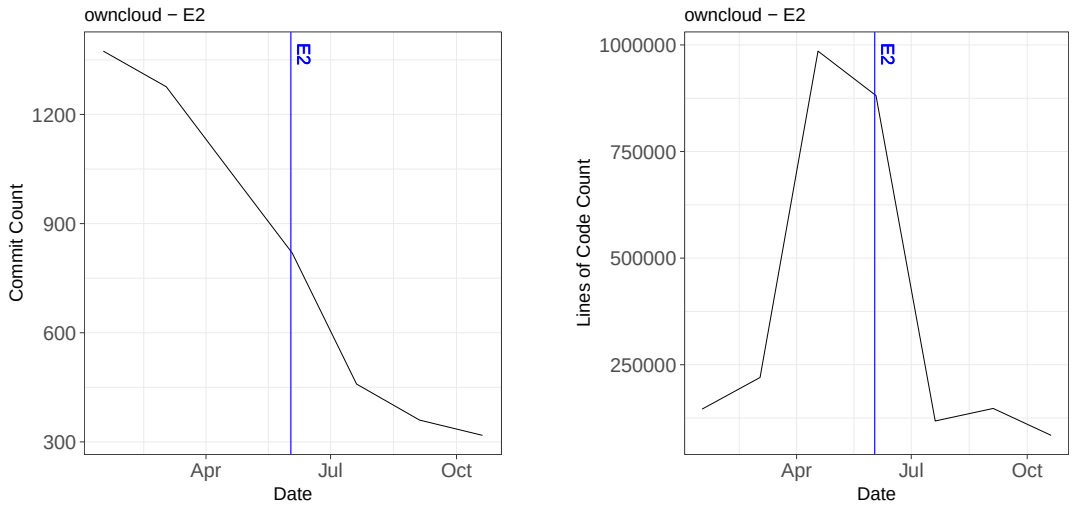

Figure 6.2: Commit count and changed lines of code count for ownCloud E2.

start of Ayo fork, we assume that the results might be plausible because the fork was only a small one. When looking into the fork on GitHub we can see that the project has only a few contributions and the lifetime of the project was very short with approximately six weeks. That is why we conclude that this fork does not have an impact on commit count and lines of code of Node.js. Whereas *E4*, the start of io.js fork, which was actually a bigger fork at that time, also does not show the expected behavior. To understand this behavior for *E4*, we had to look back in the repository. In the next chapter, we noticed that the developer count for Node.js stagnated for about two years before the event and the Node.js community got shaken awake by the event. We will discuss this result in Section 6.1.5. As we have contrary results for *Hypothesis 1.1* for some events in Group 1, we have to reject this hypothesis.

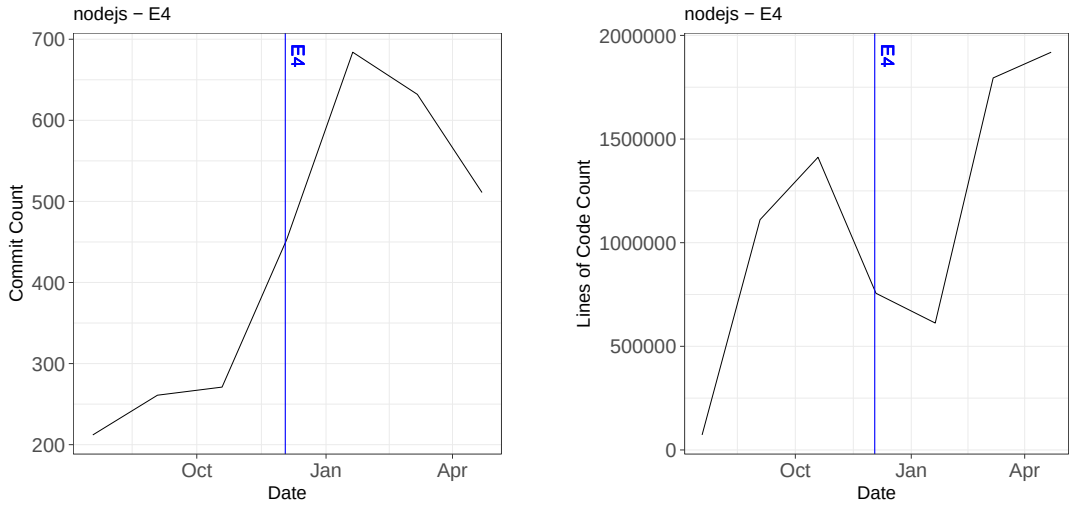

Figure 6.3: Commit count and changed lines of code count for Node.js E4.

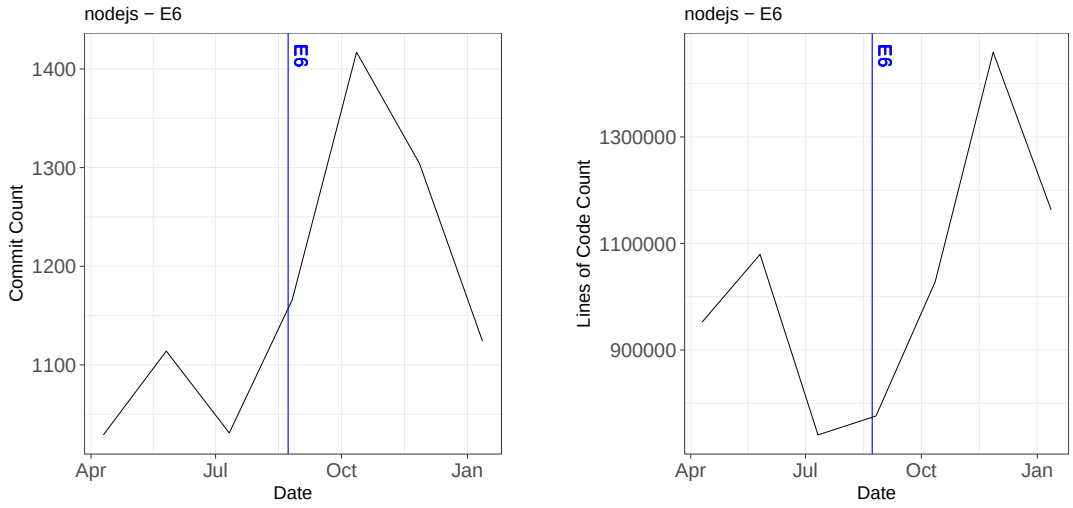

Figure 6.4: Commit count and changed lines of code count for Node.js E6.

6.1.2 Hypothesis 1.2: Developer Count and Type

Hypothesis 1.2. *In general, the projects will loose contributors. More than the usual amount of developers which were previously identified as core developers will not be identified as core in the future.*

When looking at Figure 6.5 and Figure 6.6, for *E1* and *E2*, we can see that there is a drop in developer count after the event occurred. Before the event happened, we have a developer count of over 60 and this number continuously sinks under 40 developers over the course of 9 months. So, we can register a loss of about 30% of active developers. This result matches with the result from *Hypothesis 1.1* for *E1* and *E2* where we already noticed a reduced commit count.

If we look for the developer numbers in Node.js (*E4* and *E6*) in Figure 6.7 and Figure 6.8, we have the opposite result than expected. As we got growing commit numbers for those events in the previous section, it is natural to see a growth in developer count as well. The extraordinary growth of developer count at *E6* does

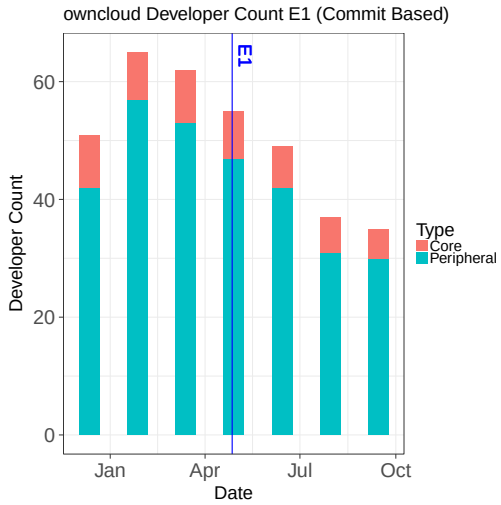

Figure 6.5: Developer count for ownCloud E1.

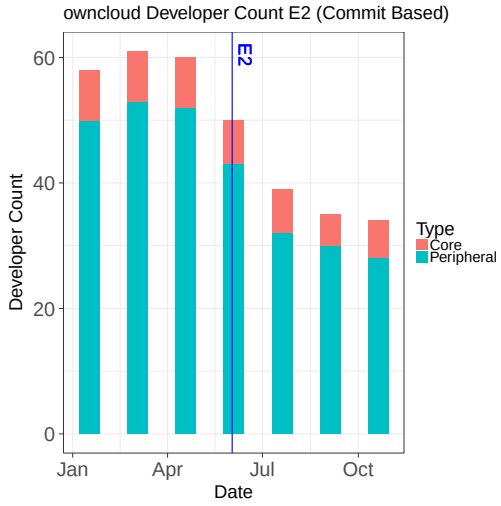

Figure 6.6: Developer count for ownCloud E2.

not look natural. The explanation for that might be visible in Figure 6.13. This plot contains vertical lines for events which we observe at the specific date as well as the release *9.x* for ownCloud. The reason for the extreme growth of developer numbers at *E6* might therefore be caused by release *9.x* which is almost at the same date.

Furthermore, the hypothesis states that more core developers switch to an inactive/peripheral state compared to the overall project lifetime. Table 6.1 contains the average percentages for the ownCloud and Node.js repository for core developers becoming inactive or peripheral. Additionally, the table shows the percentages for the events we are interested in this hypothesis. The values for the events are created by examining the data six months before and after the event, whereas the project values are calculated throughout multiple years. For an event with seven three-month time windows where each time window has an overlap of 45 days, we have six consecutive time windows. For all consecutive time windows, we calculate the role transition percentages. Afterwards, we calculate the average over all

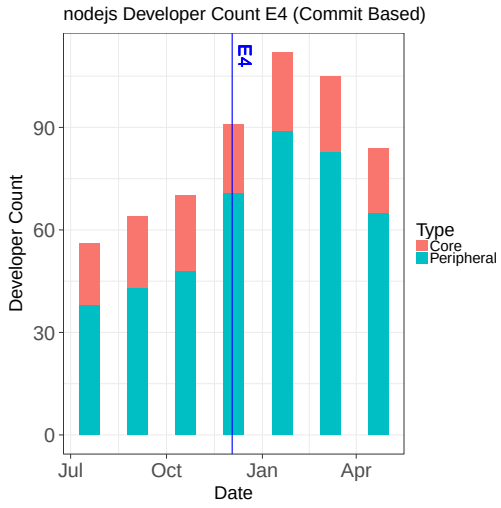

Figure 6.7: Developer count for Node.js E4.

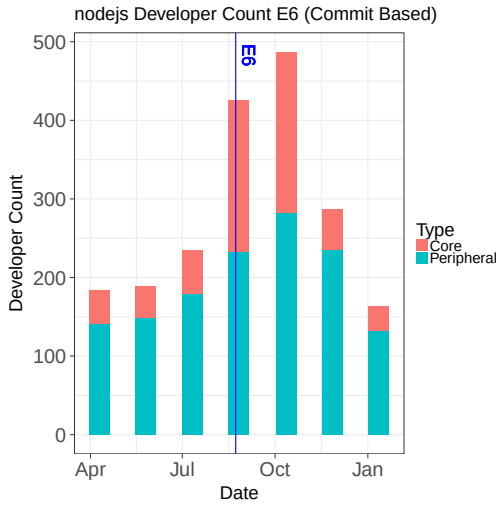

Figure 6.8: Developer count for Node.js E6.

consecutive time windows. To be able to compare the role transition percentages of an event with the percentages of the complete project, we have to do the same calculations for consecutive time windows for the complete lifetime of a project.

The values for *E1* and *E2* are exactly as we expected, where the percentage (5.2% and 4.8%) of core developers becoming inactive is higher then for the complete project (4.0%). For *E1*, it is even a 30% increase compared to the ownCloud project. Surprisingly, the percentages for core developers becoming peripheral at the events (12.4% and 8.9%) are smaller then for the average project (12.8%).

When looking at *E4* and *E6* for Node.js, we also have higher percentages of core developers becoming inactive, than for the project itself. *E6* is standing out with 16.7% compared to the overall project with 6.8%. The reason for that might be the Node.js release *9.x* which we discussed before. Figure 6.13 additionally covers the developer roles in red (core) and blue (peripheral). When looking at the bars for *E6* and its neighbours, we can see a huge change in core developers joining and leaving right after the release *9.x*. When we look closer at the developer activity

in Figure 6.9 for the time period of event *E6* and two followup time windows, we can detect a similar behaviour. The first plot at 2017-08-26 shows that almost every core developer stays in core, which is the average percentage when looking at Table 6.1. The amount of peripheral developers becoming core developers at that time is higher than the average for the ownCloud project. Therefore, we can see a higher amount of core developers (200) in plot two at 2017-10-12 and three at 2017-11-27. At 2017-10-12, we can see a role transition of more than 50 core developers to peripheral/inactive. The number is even higher at 2017-11-27 where more than 150 core developers become peripheral/inactive. This can only be explained by taking release *9.x* of ownCloud into account. *9.x* is almost at the same date as *E6* and this event does not seem to have a big impact on the repository as we showed in the previous sections. As we have contrary results for the trend of the developer count for the events of Group 1, we have to reject *Hypothesis 1.2*.

Project/Event	∅ Core to Inactive	∅ Core to Peripheral
ownCloud	4.0%	12.8%
ownCloud *E1*	5.2%	12.4%
ownCloud *E2*	4.8%	8.9%
Node.js	6.8%	15.4%
Node.js *E4*	7.3%	17.1%
Node.js *E6*	16.7%	17.1%

Table 6.1: Developer role change for ownCloud and Node.js and events *E1*,*E2*,*E4* and *E6*.

6.1.3 Hypothesis 1.3: Scale-Freeness

Hypothesis 1.3. *If the repository was in a scale-free state before, it might loose it.*

Figure 6.10 shows the scale-freeness of the constructed developer networks, for ownCloud and Node.js, at the appropriate dates. ownCloud became scale free shortly after 2014. It started to loose its scale-freeness some months before *E1* and *E2* were announced. When only looking at the plot, it seems that the events for ownCloud were looming after the project started to loose its scale-freeness again. ownCloud never reached continuous scale-freeness after the events. We know that scale-freeness also depends on the developer count [JAM17]. When looking at Figure 6.13 for ownCloud, we can see that the developer count never recovered after the events and therefore the developer network probably never reached scale-freeness again.

Node.js, however, became scale-free immediately after event *E4* occurred. With event *E4*, we already have unexpected results for commit count, lines of code and developer count. Therefore, it is not surprising that scale-freeness also behaves in a different way than expected. As the developer count and developer contributions started to increase after *E4*, it makes sense that the project gets into a scale-free state as well. Regarding *E6*, Node.js never looses its scale-freeness. When we have checked the trend of the developer count in the previous section, we already noticed that *E6* with its ayo fork does not have a huge impact on the project.

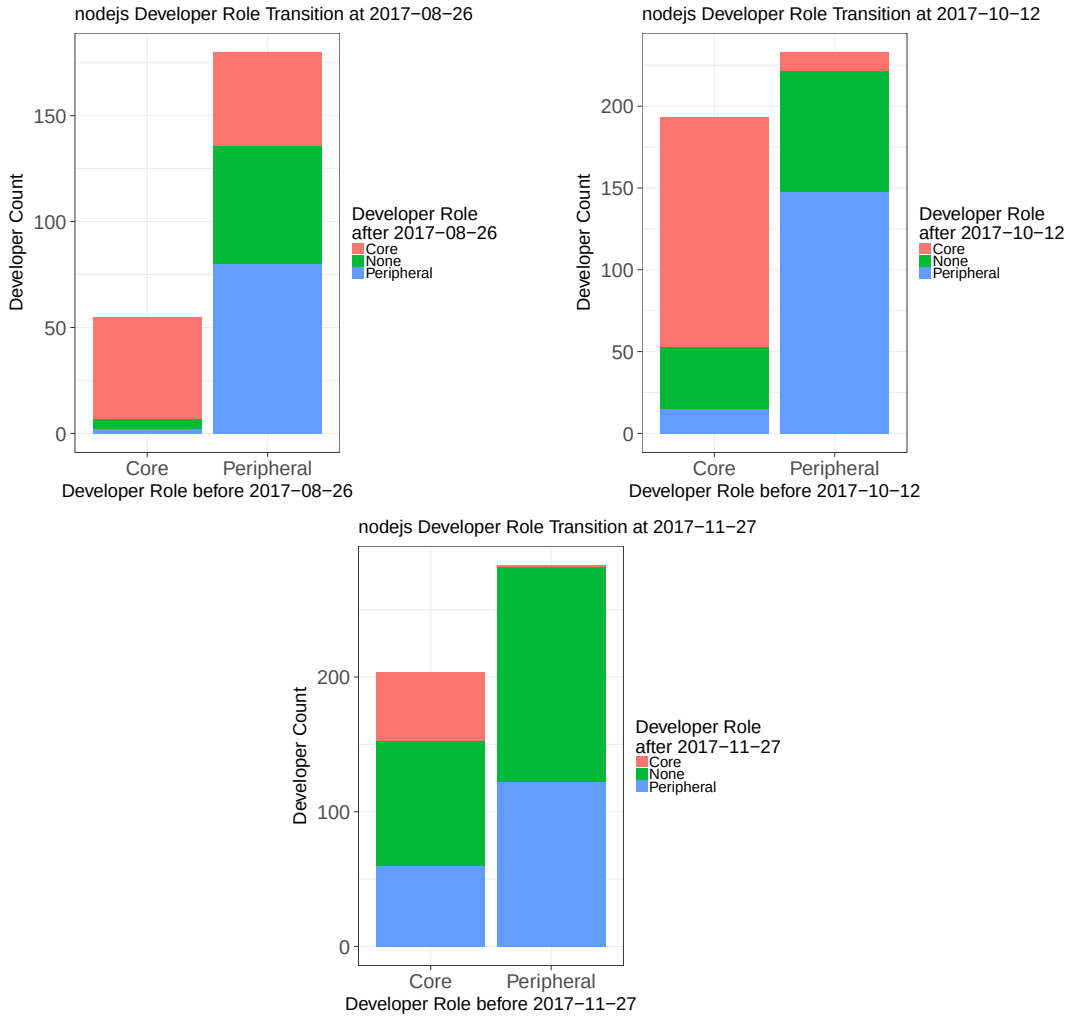

Figure 6.9: Developer role changes for E6. The first plot at 2017-08-26 contains event E6. A high amount of core developers (200) can be identified at 2017-10-12 and 2017-11-27. At 2017-10-12 more than 50 core developers transition to peripheral/inactive. More than 150 core developers become peripheral/inactive at 2017-11-27.

As a consequence, the scale-freeness state of a developer network depends highly on the trend of developers leaving or joining the project. Developers joining the project tend to lead the project into a scale-free state, as we can see clearly with event *E4*. For this hypothesis we can conclude that it is not enough to just use the type of event to predict the scale-freeness state of a project. Therefore *Hypothesis 1.3* does not hold.

6.1.4 Hypothesis 1.4: Developer Hierarchy

Hypothesis 1.4. *The clustering coefficient will get higher and the node degree will become smaller for core developers in the network hierarchy.*

To check this hypothesis, we will take a look at hierarchy diagrams with the logarithm of the clustering coefficient for each developer on the y-axis and the logarithm of node degree on the x-axis.

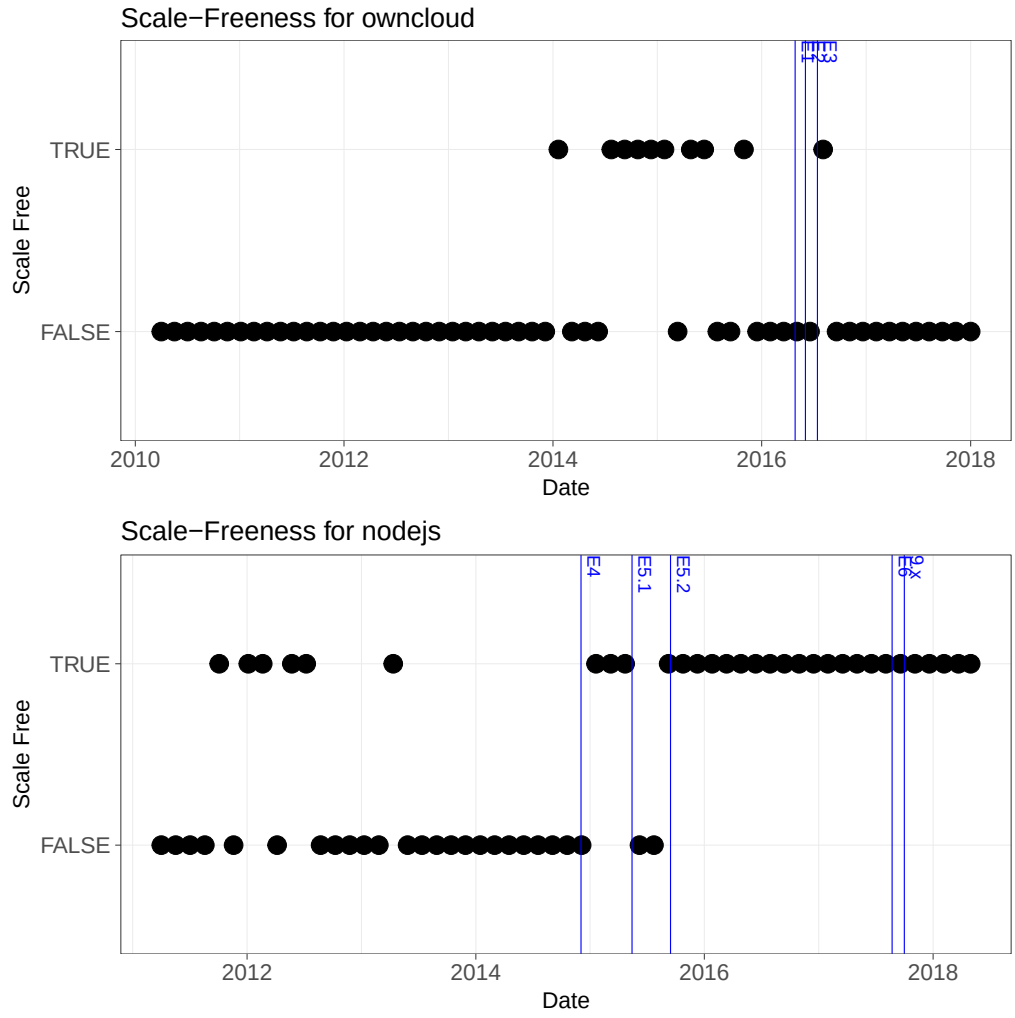

Figure 6.10: Scale-freeness for ownCloud and Node.js.

With Figure 6.11, we visualize the hierarchy of *E1* six months before and six months after the event date using overlapping three-month windows. When looking at the diagrams, we can detect a decrease of developers overall. In this hypothesis, we are interested in node degree and clustering coefficient of core developers. Those are marked as blue triangles at the bottom right in the diagrams. For *E1*, we can clearly see that the clustering coefficient for core developers becomes gradually higher, whereas the node degree gets smaller for core developers. *E2* for ownCloud follows the same pattern (see Figure A.1 in the appendix). The decreasing node degree and increasing clustering coefficient shows that there is a decrease in coordination done by core developers. These observations match our hypothesis.

In contrary, Figure 6.7 displays an increase in developer count at E4. Therefore, Figure 6.12 shows that the clustering coefficient for core developers continuously gets smaller and the node degree gets higher. Thus, the coordination work for core developers increases. The difference in clustering coefficient and node degree between core and peripheral developers is unexpected high for the second and third time window and gets smaller at the event date (plot four). The hierarchy plot for *E6* does not contain any conspicuousness and the cause for changes in the hierarchy

diagrams can not be distinguished between the event itself and the release *9.x* which are almost contemporaneous.

The evolution of node degree and clustering coefficient of core developers can not simply be decided by the type of the event. And therefore this hypothesis does not hold for every negative social event.

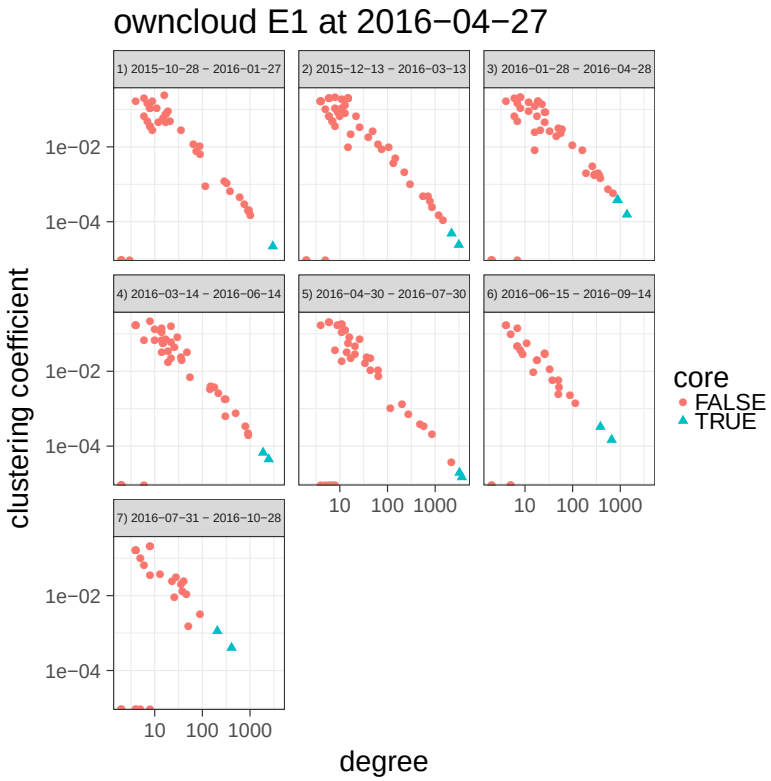

Figure 6.11: Network hierarchy for *E1*. The clustering coefficient for core developers becomes gradually higher and the node degree gets smaller.

6.1.5 Conclusion for Hypothesis 1

Hypothesis 1 covered the events which we categorized as negative social events. Therefore, we predicted negative consequences for our count-based measurements, as well as network stability in form of scale-freeness and network hierarchy. But we learned that those negative events do not necessarily imply negative consequences. In case of the events for ownCloud, our hypotheses met the expectations, whereas Node.js with event *E4* behaved completely in the opposite direction than expected. We examined the past years of the projects as well and found out that Node.js was standing still years before the actual event *E4* happened. The perceived observations lead us to the estimation that the Node.js community got shaken awake by the event and the project started to thrive. ownCloud, on the other hand, had growing developer numbers and even reached scale-freeness continuously before the actual events happened.

Therefore, we assume that, by taking the past of the project into consideration, we could have made better forecasts for the repository. Humans often do not act as

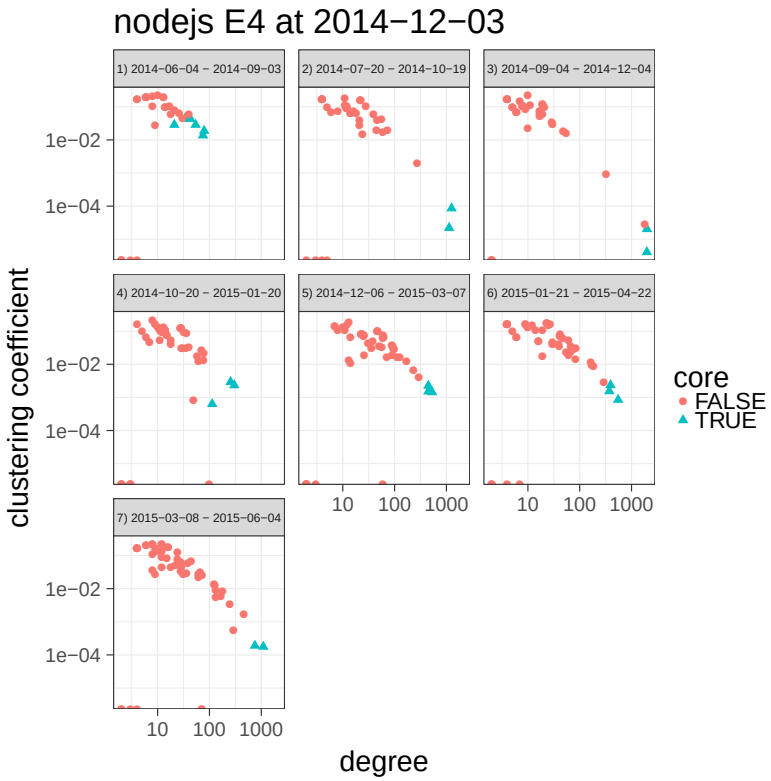

Figure 6.12: Network hierarchy for *E4*. The clustering coefficient for core developers continuously gets smaller and the node degree gets higher.

expected and therefore it might not even be enough to just take the past of the repository into account. *Hypothesis 1* does not hold for all social negative events.

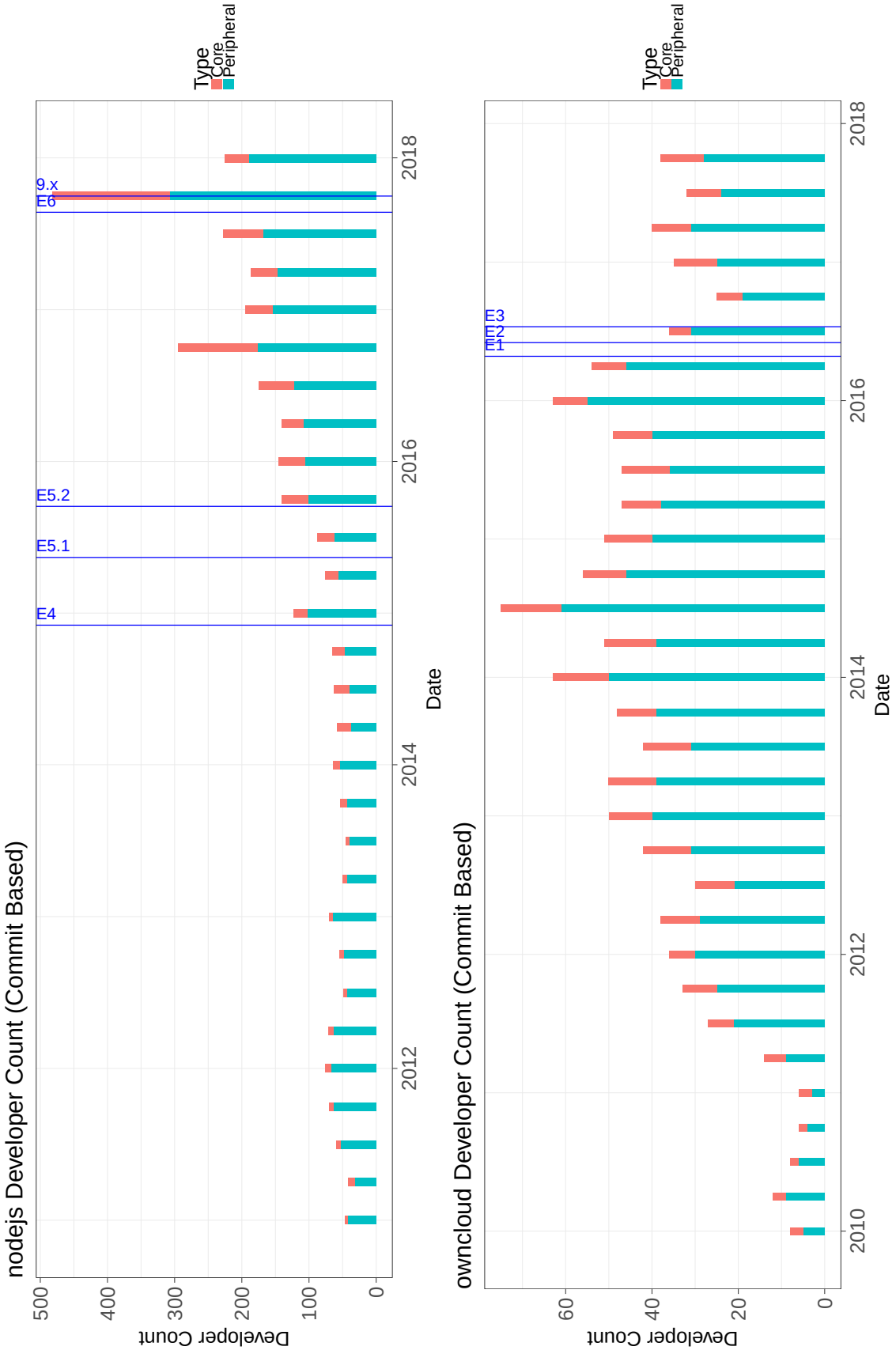

Figure 6.13: Developer count for NODE.JS and OWNCLOUD.

6.2 Evaluation for Positive Social Events (G2)

Hypothesis 2 contains statements about the development of the Keras and Node.js, affected by positive social events. The event *E5* is the cooperation between io.js and Node.js. We split the event into *E5.1* and *E5.2* where *E5.1* indicates the start of the merge and *E5.2* represents the end of the io.js and Node.js merge. The event *E10* for Keras is the announcement of the cooperation between Google and TensorFlow.

6.2.1 Hypothesis 2.1: Lines of Code and Commit Count

Hypothesis 2.1. *The lines of code and the commit count after the event will increase.*

Figure 6.14 and Figure 6.15 contain an overview over the commit count and lines of code count for *E5.1* and *E5.2*. Before the event *E5.1*, the commit count and changed lines of code count were at their peak and started to follow a downtrend. As expected, we can see a slow increase of commit count and changed lines of code shortly after event *E5.1*. Figure 6.15 shows a steady increase in commit count and changed lines of code count starting shortly before *E5.2*. The peak before the event *E5.1* can be explained by release *0.12.x* at 2015-02-06.

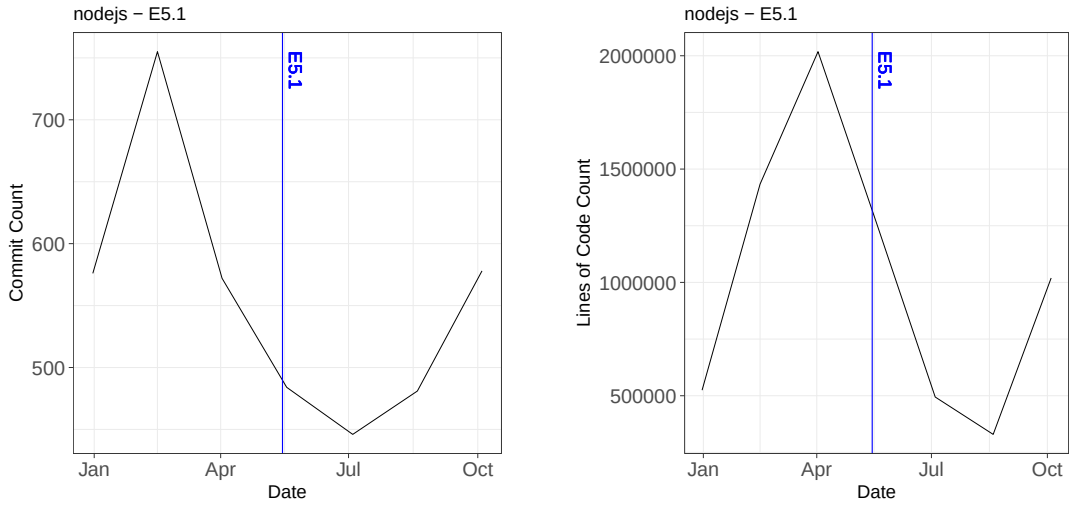

Figure 6.14: Commit count and changed lines of code count for Node.js E5.1.

Figure 6.16 displays the commit count and the changed lines of code for Keras six months before and six months after the event and Figure 6.17 contains the commit count over the whole lifetime of Keras. When looking at Figure 6.16, we can see a peak in commit count and lines of code before the actual event. The reason for that might be Keras release *v 0.3.0* which was at 2015-12-01 shortly before the event. The values slowly decrease after the event. When looking at the bigger picture in Figure 6.17, we can see that the commit count stabilizes after some months.

Concluding for *Hypothesis 2.1*, for event group *G2* we can say that both events had major releases before the actual event and therefore it is hard to distinguish the cause for the extraordinary change in commit count and lines of code. Therefore,

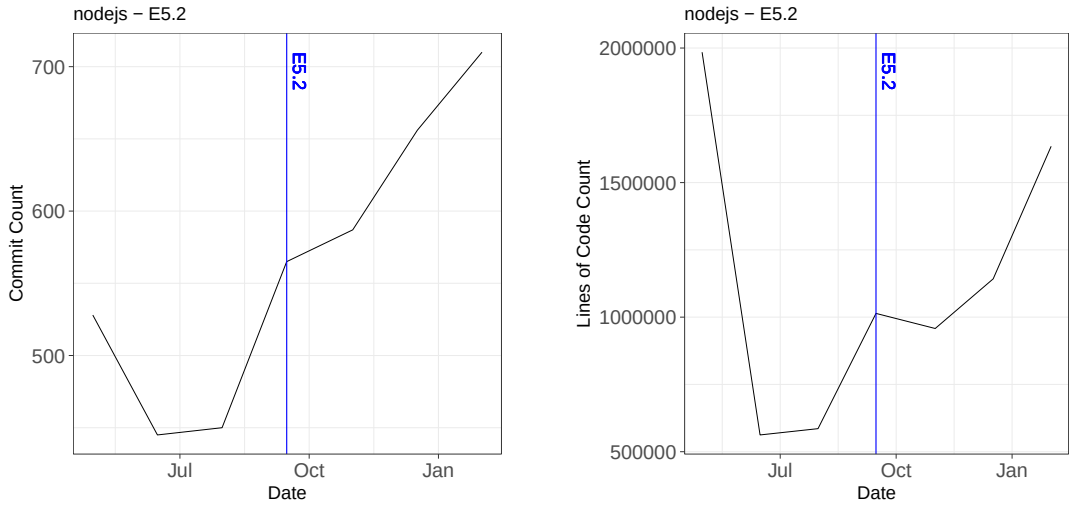

Figure 6.15: Commit count and changed lines of code count for Node.js E5.2.

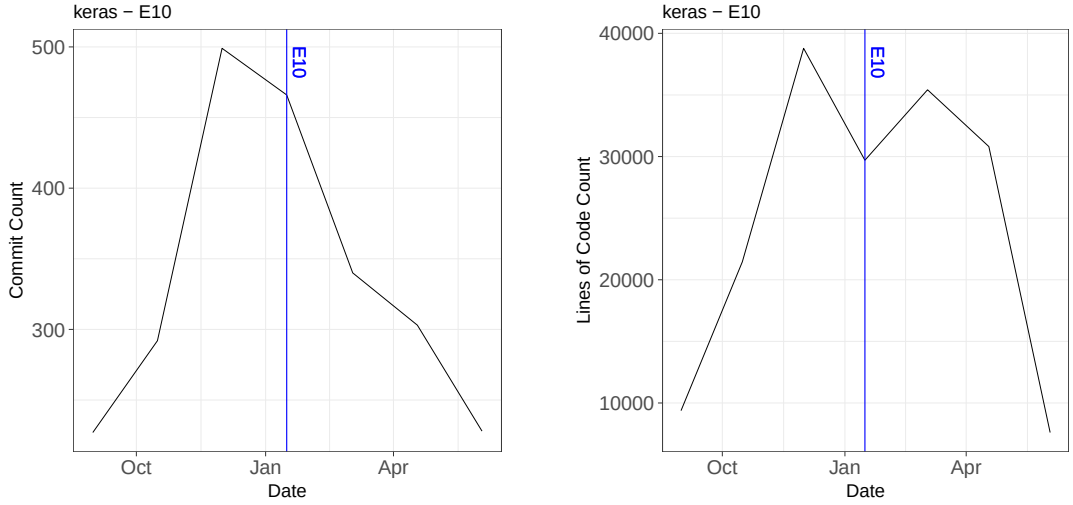

Figure 6.16: Commit count and changed lines of code count for Keras E10.

our perceived count-based data for those events are not very representing. Even though event *E5* shows a slight increase, it is not enough to generally claim that this hypotheses holds. Therefore, we reject *Hypothesis 2.1.*

6.2.2 Hypothesis 2.2: Developer Count and Type

Hypothesis 2.2. *In general, the projects will get more contributors. There will be more and new identified core developers.*

To evaluate this hypothesis we will look at the contributor count first. We check the developer count for our events of interest within a six month range before and after the event. In Figure 6.18, we see a slight increase of developers after event *E5.1.* The peak before the event can be explained with release *0.12.x.* Figure 6.19 shows a huge increase of developers after the merge with io.js is done (*E5.2*). This leads to the conclusion that developers who switched to work on the io.js fork are now coming back to the Node.js community.

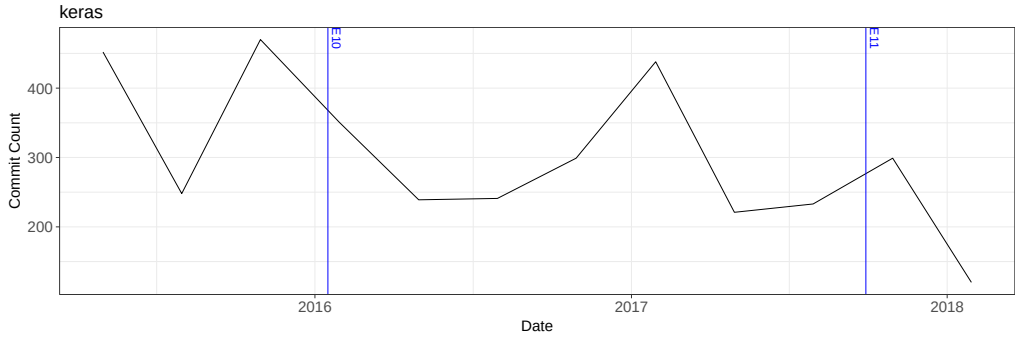

Figure 6.17: Commit count for KERAS.

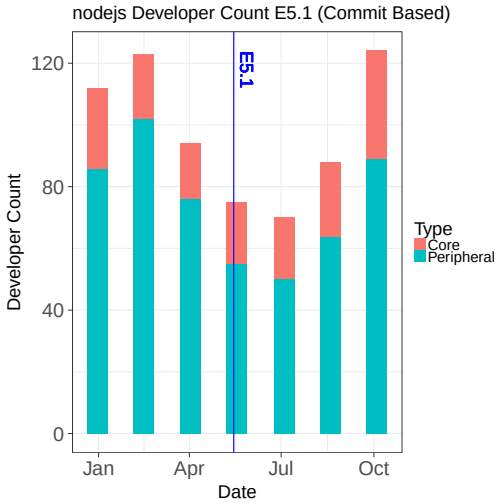

Figure 6.18: Developer count for NODE.JS *E5.1*.

KERAS only shows a small increase of developers after the event E10 occurred (see Figure 6.20). But when looking at the bigger picture in Figure 6.21 for the whole KERAS project, we can see a steadily increase in developer numbers over several months.

The second part of this hypothesis states that there will be more and new identified core developers. To show the role transition for current core developers to inactive or peripheral, we created Table 6.2. In contrast to *Hypothesis 1.2*, we have lower percentages for core developers becoming inactive during the events compared to the whole project. This supports our statement that there will be more identified core developers as the old ones tend to not get inactive. When looking at the bar charts (see Figure 6.18, Figure 6.19, and Figure 6.20) where core developers are marked red, we can definitely see an increase of core developers towards the end of the x-axis.

E5.2 leads to an immediate increase of developers, whereas *E10* only results into a slight increase over a long period of time. When looking objectively at *Hypothesis 2.2*, we can conclude that the hypotheses holds. But as both events had a major release before we cannot exclude after effects, caused by the releases, on our perceived data.

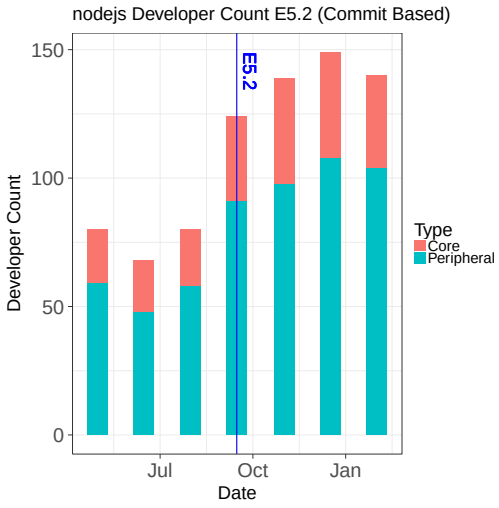

Figure 6.19: Developer count for NODE.JS *E5.2*.

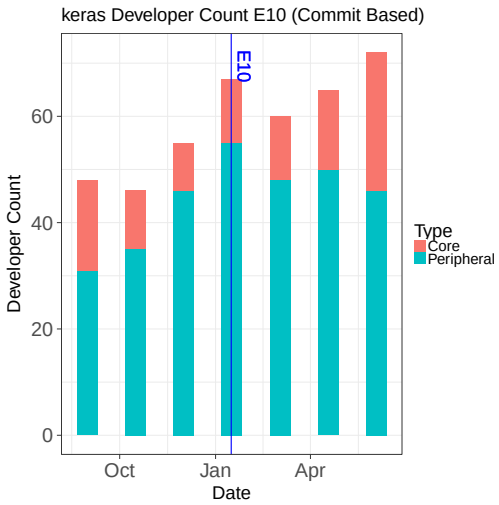

Figure 6.20: Developer count for KERAS *E10*.

6.2.3 Hypothesis 2.3: Scale-Freeness

Hypothesis 2.3. *If the developer network was not in a scale-free state, then it is going to reach it.*

To check this, we look at Figure 6.10 for NODE.JS and at Figure 6.22 for KERAS, which visualize the scale-freeness state of the developer networks for the whole lifetime of the projects. NODE.JS was in the scale-free state before the beginning of the event *E5* at *E5.1*. NODE.JS became scale-free after event *E4* (IO.JS fork). It lost the scale-freeness state about five months after *E5.1* happened. It seems that the project was in an uptrend after *E4*, but still was not well established. After the merge was done at *E5.2*, the developer network became scale-free again and never lost it ever since.

Figure 6.22 shows that the developer network for KERAS became scale-free for several months after the event *E10*. As there was a release for KERAS right before the event,

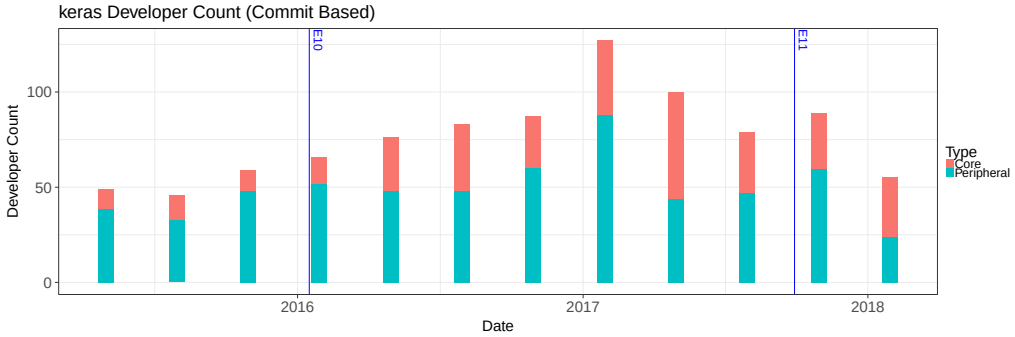

Figure 6.21: Developer count for KERAS.

Project/Event	∅ Core to Inactive	∅ Core to Peripheral
NODE.JS	6.8%	15.4%
NODE.JS *E5.1*	1.6%	16.4%
NODE.JS *E5.2*	4.8%	16.2%
KERAS	28.1%	14.1%
KERAS *E10*	21.8%	16.2%

Table 6.2: Developer role change for NODE.JS and KERAS and events *E5.1*,*E5.2*, and *E10*.

it is not possible to distinguish if the release or event *E10* is responsible for leading the network into a scale free-state.

Because both developer networks for NODE.JS and KERAS were immediately scale-free after the events, we claim that this hypothesis holds.

6.2.4 Hypothesis 2.4: Developer Hierarchy

Hypothesis 2.4. *The clustering coefficient will slowly decrease, whereas the node degree will increase for core developers.*

For evaluating this hypothesis, we look at the hierarchy plots for the two events as we did in Section 6.1.4. Since we split event *E5* into *E5.1* and *5.2*, we can distinguish between the beginning and the end of the event. Therefore, we see that in the first two hierarchy plots in Figure 6.23 before the actual event the clustering coefficient for core developers is about 0.001. Afterwards, the clustering coefficient for core developers starts to become smaller over time. There are also periods when the clustering coefficient returns to be over 0.001. But that overall only happens twice for *E5.1* and *E5.2* (Figure 6.24). A continual trend for the node degree of core developers is not visible. The value is at almost 1000 for the most time in *E5.1* and *E5.2* Without taking the hypothesis into account and only look at the stability of the network by applying a linear function through the dots in the diagram, we can clearly see that the points in the coordinate system approach a stable linear function especially when looking at the last plot in Figure 6.24.

For *E10*, Figure 6.25 shows a slight increase of the node degree for core developers over the inspected year. At the beginning, the value for node degree was at about

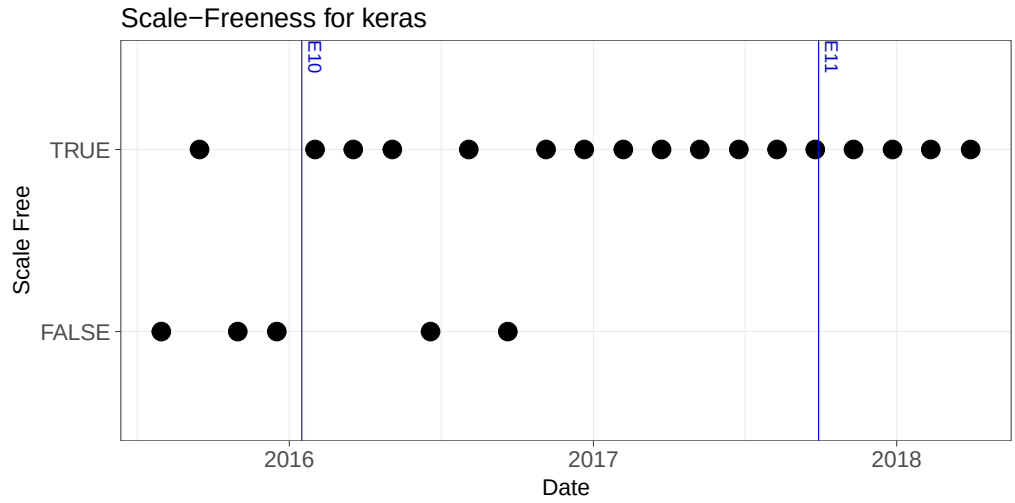

Figure 6.22: Scale-freeness for KERAS.

500 and grew to almost 1000 during the event. A slight downward trend for the clustering coefficient is visible. Thus, it looks like the coordination work of core developers increases.

As we showed in Section 6.2.2, both events had only small changes in developer numbers at the event dates. Therefore, the changes in the network hierarchy seem to be very small. Hence, the results for those events are not enough to claim that *Hypothesis 2.4* always holds.

6.2.5 Conclusion for Hypothesis 2

When evaluating the results for each hypothesis, we can see that the events clearly affect the scale-freeness and the developer count. Both events had a peak in developer numbers right before the event which probably were caused by previous releases. Those peaks went down at the event date and started to increase steadily for several months when we looked at commit-based classification for developers. Also the amount of identified core developers increased for both events. The developer networks for the projects always became scale-free after the events, even though NODE.JS lost it during the merge with IO.JS. Afterwards, it reached scale-freeness again. Thus *Hypothesis 2.2* and *Hypothesis 2.3* hold. However, the commit count and the changed lines of code are not very representative due to previous events like release dates in our case. Also, we cannot see a clear trend in the hierarchy diagrams for all events. Therefore, *Hypothesis 2.1* and *Hypothesis 2.4* do not hold. If we compare *Hypothesis 2* to *Hypothesis 1*, which covered negative social events, we can clearly see that the effects of positive social events on the network and count-based data tend to be positive.

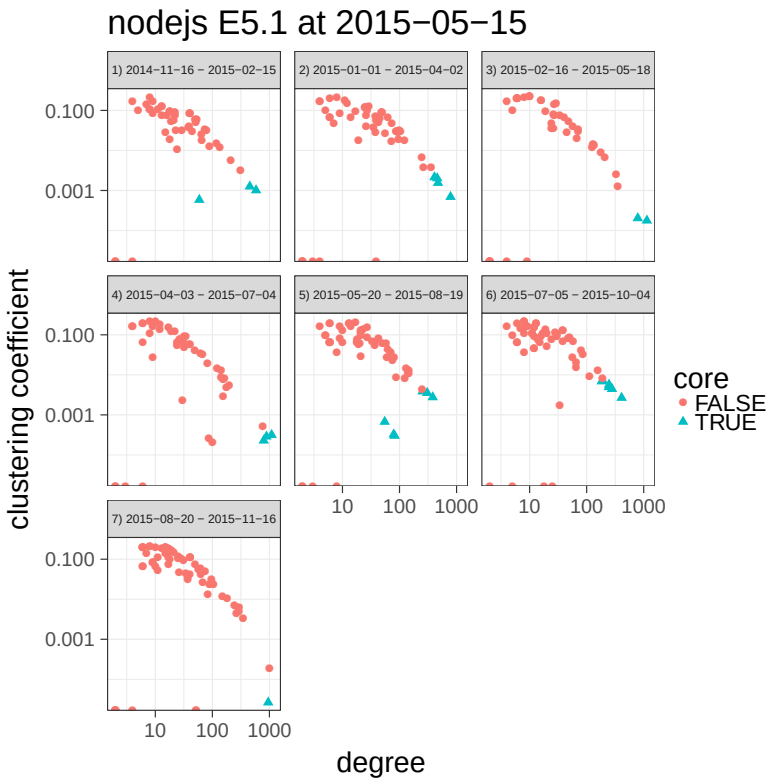

Figure 6.23: Network hierarchy for *E5.1*.

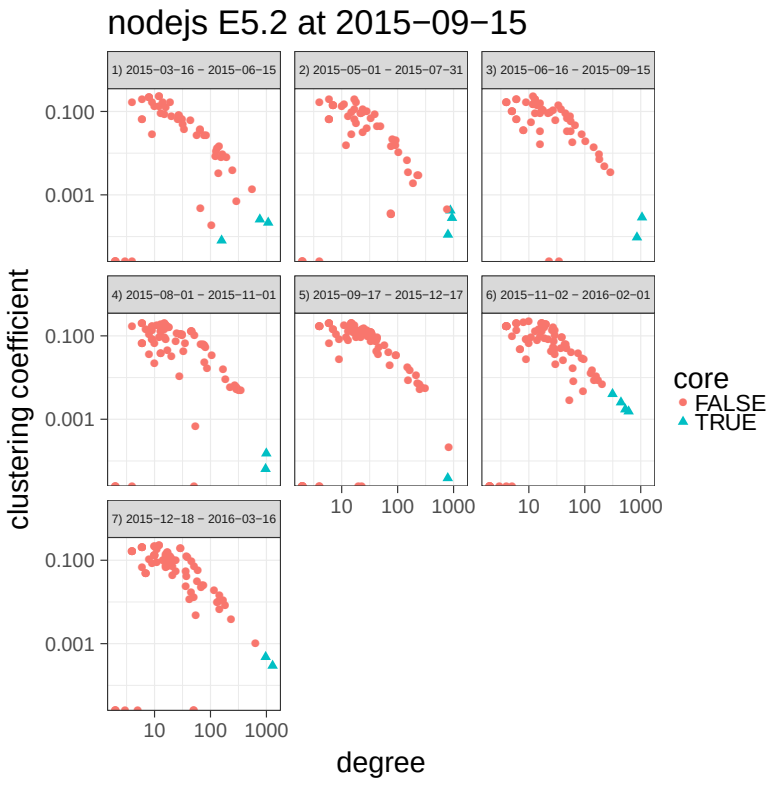

Figure 6.24: Network hierarchy for *E5.2*. A continual trend for the node degree of core developers is not visible.

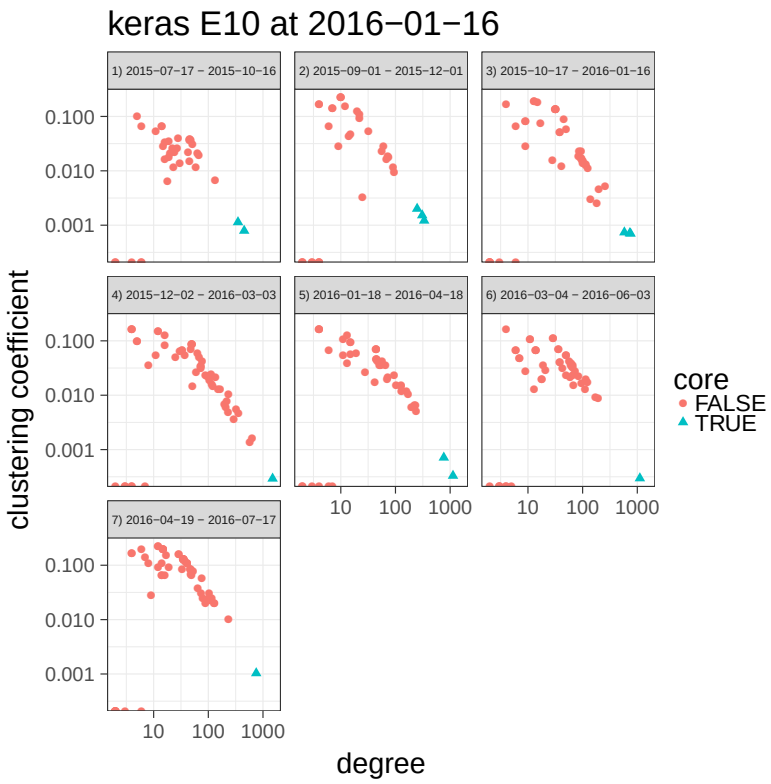

Figure 6.25: Network hierarchy for *E10*. The clustering coefficient for core developers becomes slightly smaller and the node degree gets continuously higher.

6.3 Evaluation for Negative Organizational Events (G3)

Hypothesis 3 contains statements about the development of Qt and Keras, affected by negative organizational events.

6.3.1 Hypothesis 3.1: Lines of Code and Commit Count

Hypothesis 3.1. *The lines of code and the commit count after the event will decrease.*

By examining the commit count and changed lines of code count for event *E8* in repository Qt4 in Figure 6.26, we encounter unusual numbers. The commit count six months before the event was 350 and it dropped to 200 shortly before the event. It started to grow to approximately 300 afterwards. The reason for that behavior might be release *5.0* of Qt which was at 2012-12-19, 4 months after the event. We assume that developers pushed more commits before the release. The extraordinary spike in changed lines of code after the event cannot be explained naturally. When looking at the new repository Qt5 for Qt, which was under open governance, in Figure 6.27, we can notice a downtrend in commit count and changed lines of code from 6 months before until 6 months after the event. If we compare the Qt4 and Qt5 repository, we notice that Qt5 has commit numbers between 1400 and 2400 for the inspected event period of *E8*, whereas Qt4 had its peak at 350 commits. Thus we assume the Qt5 repository to be more expressive for the consequences of *E8*.

Keras, on the other hand, also had releases shortly before and after the event *E11*. Release *2.0.8* was at 2017-08-25 and release *2.0.9* at 2017-11-01. Therefore, this might explain the up and down of the commit count and changed lines of code during the investigated period in Figure 6.28.

Hypothesis 3.1 only holds for *E8* and the new Qt5 repository. Thus the hypothesis does not hold overall.

6.3.2 Hypothesis 3.2: Developer Count

Hypothesis 3.2. *The amount of contributing developers will decrease.*

When looking at the Qt5 repository for Qt, we can recognize an upwards trend in the developer count several months before the event *E8*, visualized by Figure 6.31. By looking closer at the surrounding three-month windows for *E8* in Qt5, we can see a slight drop in the developer count (see Figure 6.30). This behavior would would match our hypothesis. In contrary to the Qt5 repository, the Qt4 repository follows a downtrend in developer count several months before the actual event *E8* (see Figure 6.31). We suspect that the decreasing developer numbers are part of the reason for *E8* (closing of development department) happening, but this is just speculation based on the perceived results. An immediate drop of the developer count after the event for Qt4 in Figure 6.29 is not recognizable.

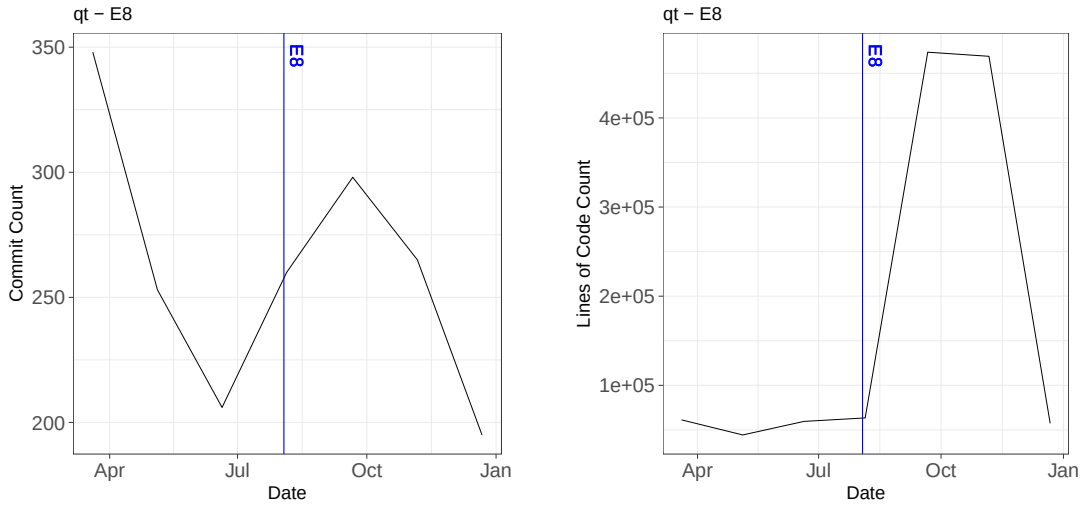

Figure 6.26: Commit count and changed lines of code count for Qt E8 (Qt4).

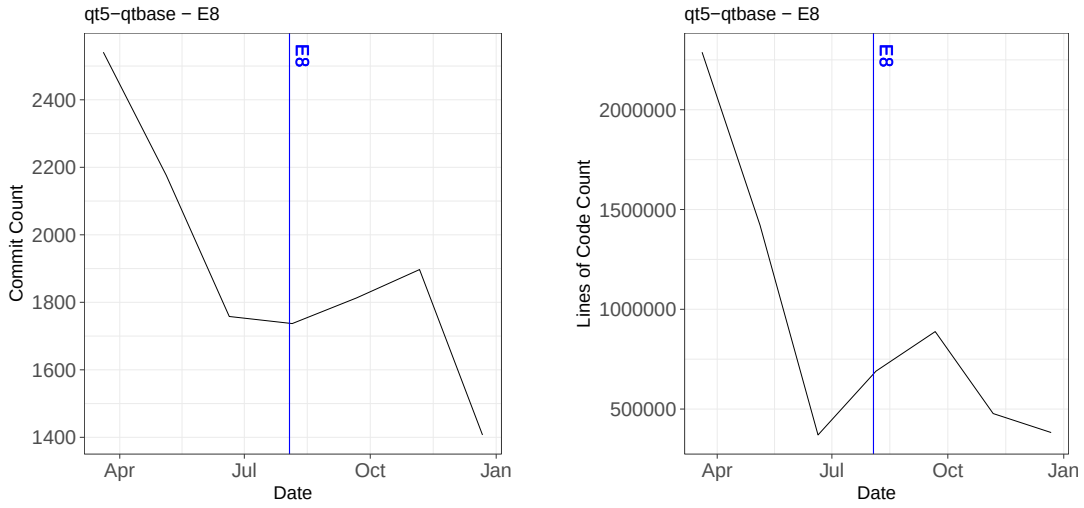

Figure 6.27: Commit count and changed lines of code count for Qt E8 (Qt5).

As well as the Qt4 repository of Qt (*E8*), Keras (*E11*) does not show immediately decreasing developer numbers in the three-month windows after the event (see Figure A.3 in the appendix). A reason for that might be a release shortly after the event, which we have already discussed in Section 6.3.1. However, we can see a decrease in developer numbers in the long term in Figure 6.21.

Hypothesis 3.2 only holds for *E8* and the Qt5 repository. E11 and *E8* (Qt4) show different results than expected. Thus we have to reject *Hypothesis 3.2*.

6.3.3 Hypothesis 3.3: Scale-Freeness

Hypothesis 3.3. *If the project was in a scale-free state, it will loose it and slowly recover into a scale-free state.*

To check the scale-freeness of Qt, we look at Figure 6.32, containing the information for network scale-freeness for both Qt repositories. The network was in a scale-free

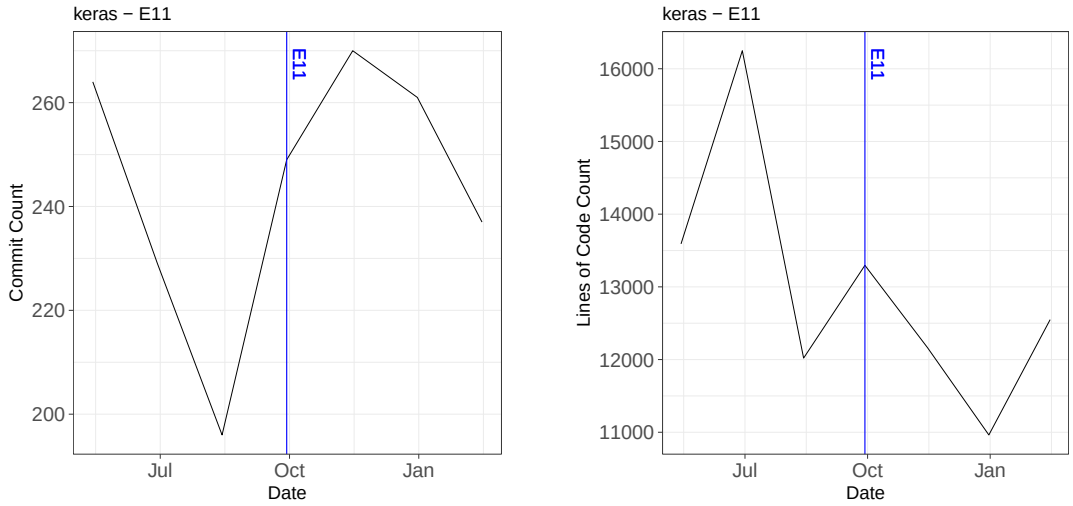

Figure 6.28: Commit count and changed lines of code count for Keras E11.

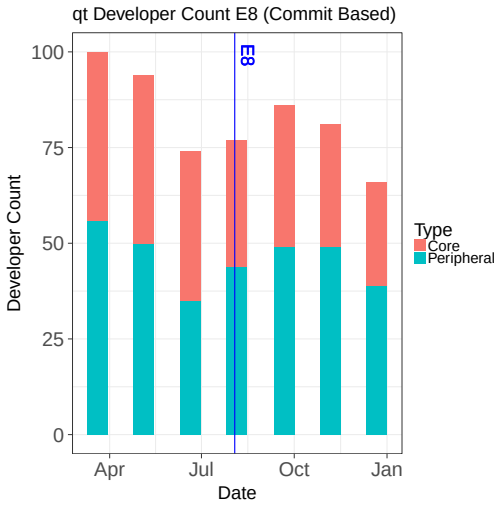

Figure 6.29: Developer count for Qt *E8* (Qt4).

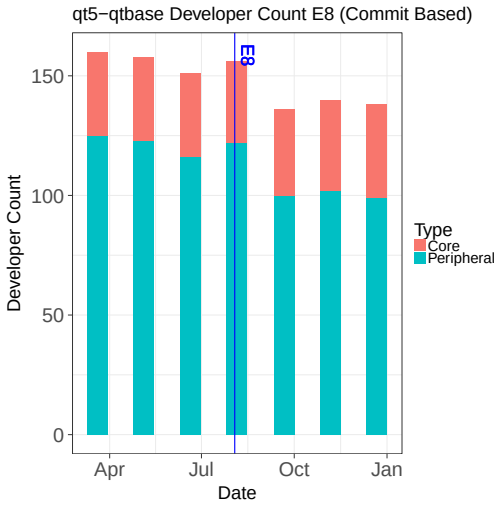

Figure 6.30: Developer count for Qt *E8* (Qt5).

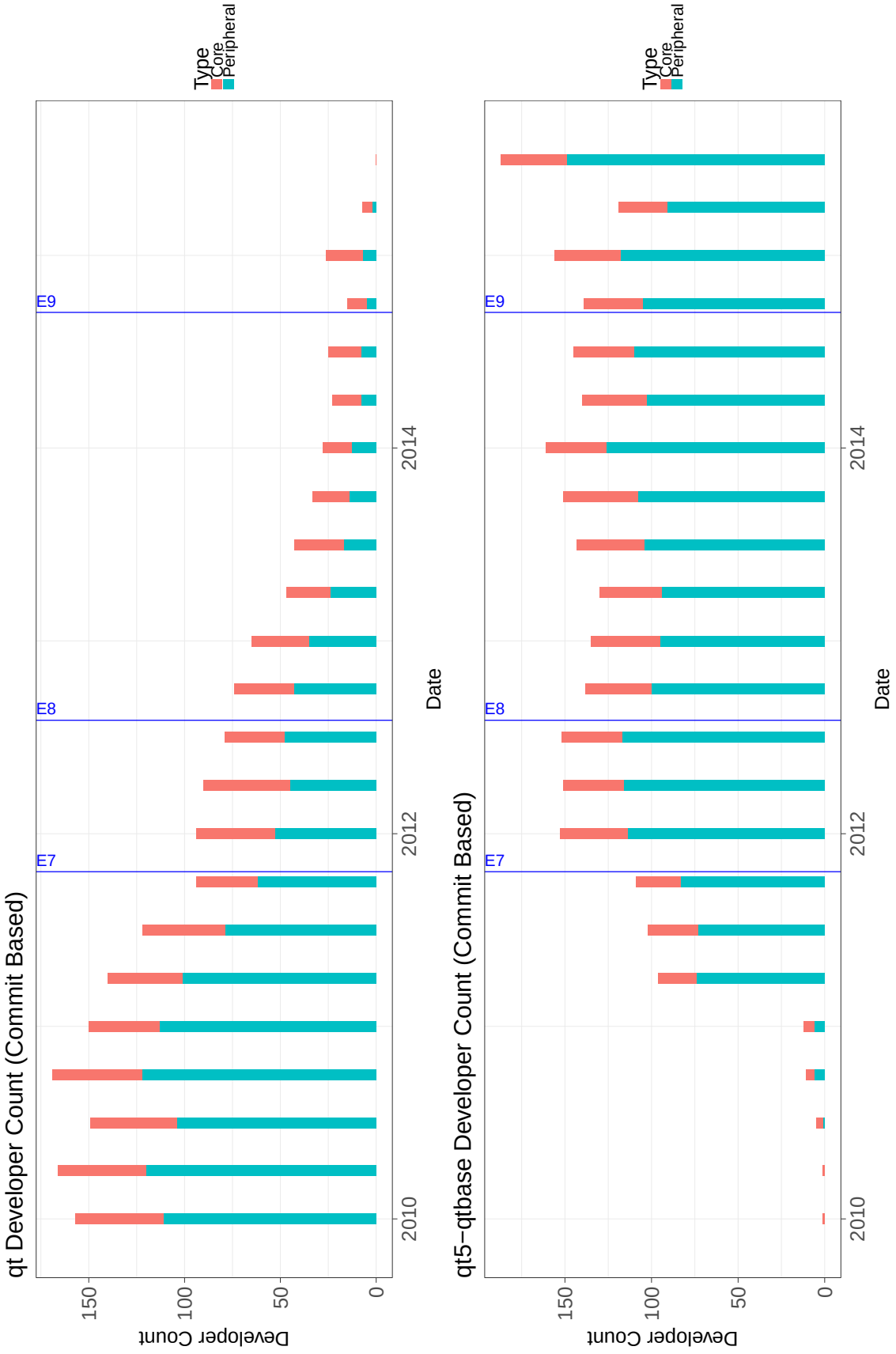

Figure 6.31: Developer count Qt.

state for a long time before the event and even several months after the event *E8*. The project lost its scale-freeness in 2013, when looking at the QT4 repository. We can explain that by looking into the plot which contains the developer count for the whole project in Figure 6.31. In 2013, the developer count of QT drops under 50 and the projects looses its scale-freeness. As we already have discussed in the previous section that we do not assume the event to be the reason for decreasing developer numbers, we also assume that the event is not the reason for QT to loose scale-freeness in 2013. In contrary, the new QT repository was in a scale-free state months before the event and never lost its scale-freeness ever since.

KERAS, on the other hand, achieved the scale-free state months before *E11* and never lost it since then (see Figure 6.22).

Summarizing the results, we reject *Hypothesis 3.3*.

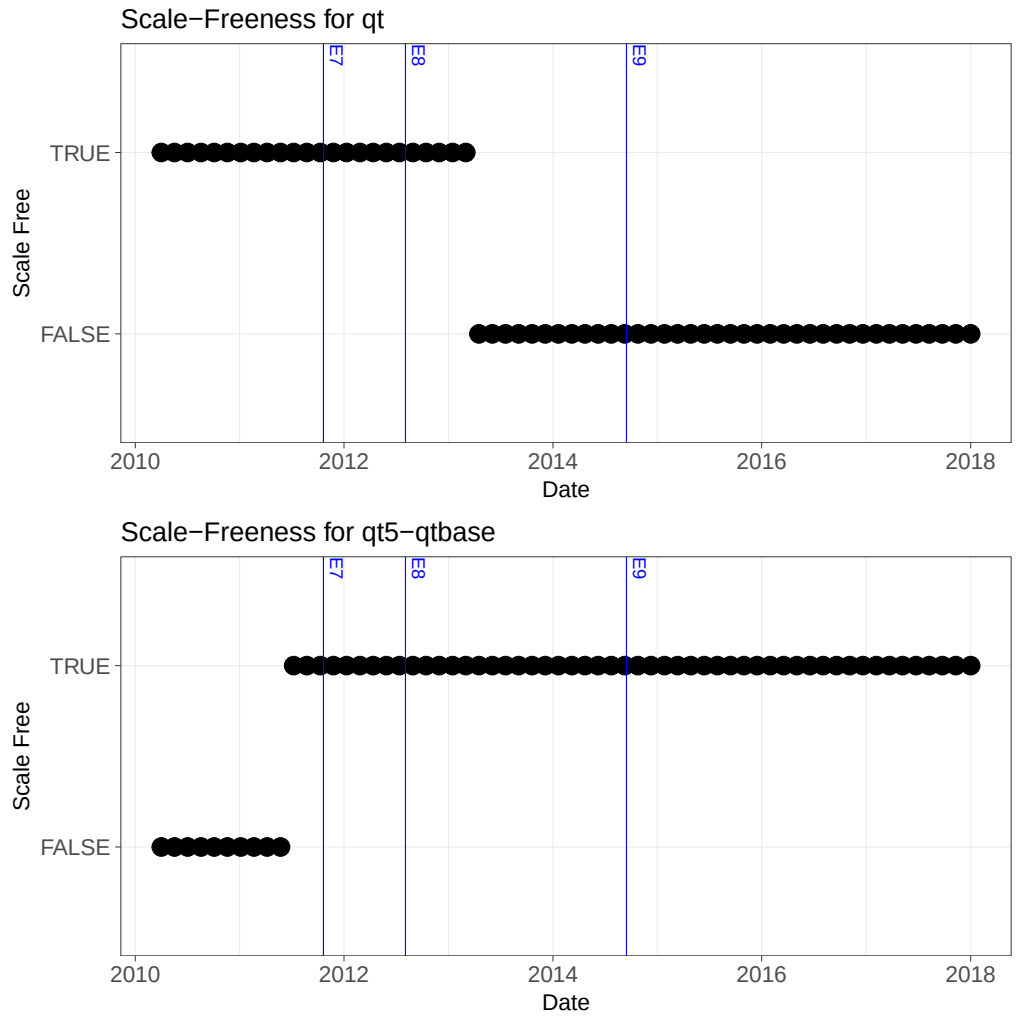

Figure 6.32: Scale-freeness for QT. The first plot shows the old QT repository. The second one represents the new QT repository.

6.3.4 Hypothesis 3.4: Developer Hierarchy

Hypothesis 3.4. *The clustering coefficient for core developers will get higher, whereas the node degree will get lower.*

Other than expected, the hierarchy networks for *E8* show decreasing clustering coefficient and increasing node degree values for core developers in the hierarchy plots in Figure 6.33. The hierarchy plots do not show developers whose clustering coefficient have a value of 0. We can clearly see an increase of peripheral developers showing up in the hierarchy plot for the fourth and fifth time window. That means that the overall clustering coefficient of developers exceeds 0 while the developer count only slightly increases when looking at the hierarchy diagrams for the old Qt repository (Figure 6.33). The last plot from the hierarchy diagrams of event *E8* shows a smaller amount of peripheral developers. The reason for that might be that, after closing the development department in Brisbane, the developers had time until end of December 2012 to finish their open tasks on Qt [Nan12]. This leads us to the assumption that after the department was closed and developers had to finish their work within the next months affected the amount of required files the developers had to edit. Thus, the values for clustering coefficient and node degree for core developers do not match our expectations for the old Qt repository. When looking at the hierarchy diagrams for the Qt5 repository, we can recognize a single core developer with higher node degree and lower clustering coefficient after the event. It seems that this one core developer, who shows up in the hierarchy network after the event, has to handle most of the coordination work. But in general, the node degree for developers decreases and the clustering coefficient increases.

Keras shows no striking changes to its hierarchy within six months before and six months after the event *E11* (Figure 6.34). A change in coordination effort of core developers is not visible.

This hypothesis does not hold when considering our results.

6.3.5 Conclusion for Hypothesis 3

To conclude, we can say that perceived changes to the count-based data and the network structure are not enough to claim that this hypothesis holds. Keras did not seem to be affected at all by event *E11*. Also Qt was on a downtrend before the event *E8* occurred and therefore we are not able to distinguish if the event was the reason. The new Qt repository matches our hypothesis more when only looking at developer numbers and hierarchy plots. But in general, it is not enough to prove that the hypothesis always holds. Thus, negative organizational change does not necessary come with changes to the count-based information and network structure, at least not in the short term we analyzed.

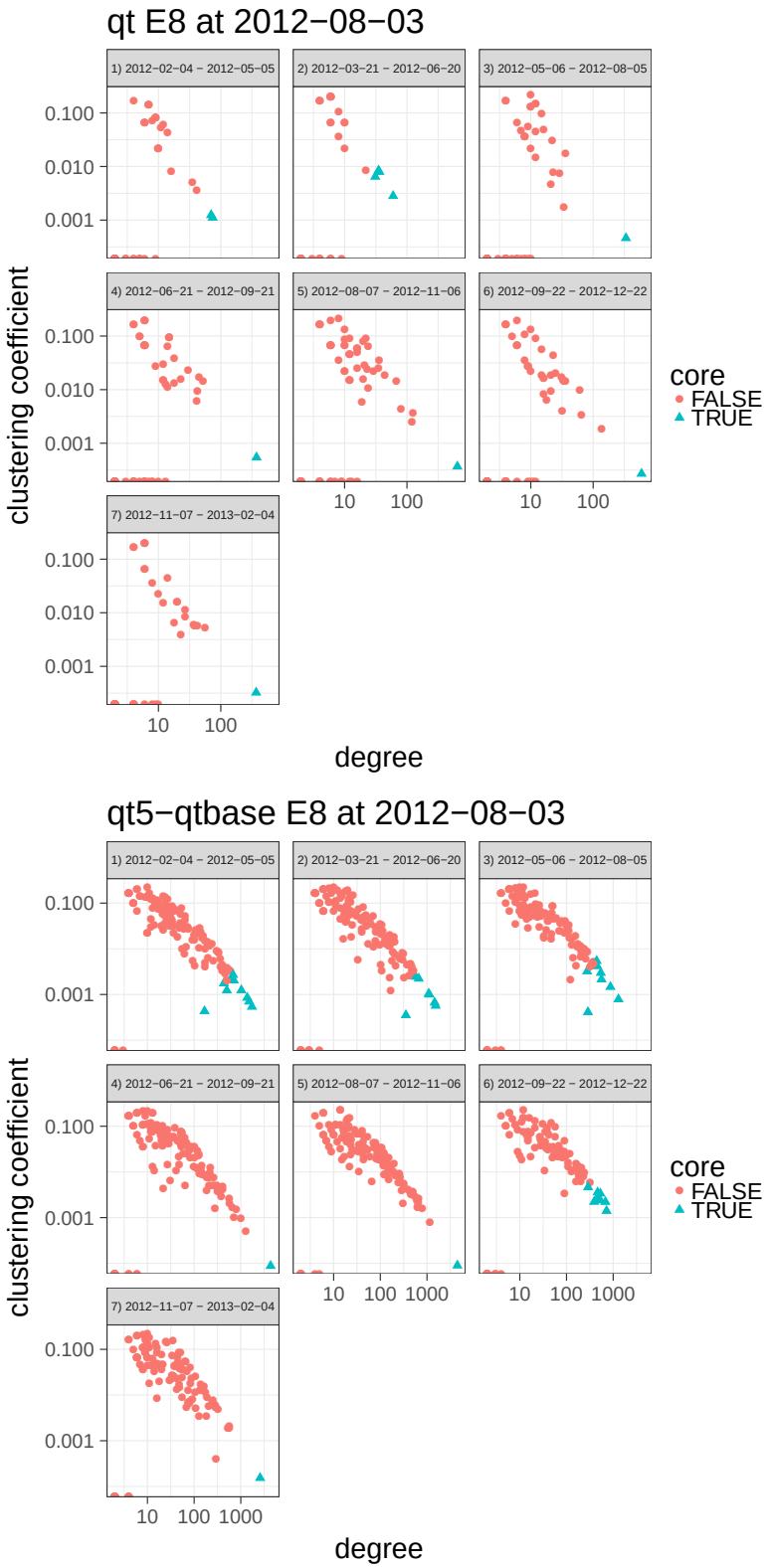

Figure 6.33: Network hierarchy for *E8*. The first plot shows the old Qt repository. The second one represents the new Qt repository. The hierarchy networks for *E8* (Qt4) show decreasing clustering coefficient and increasing node degree values for core developers. The hiararchy networks for Qt5 show decreasing clustering coefficients for core developers.

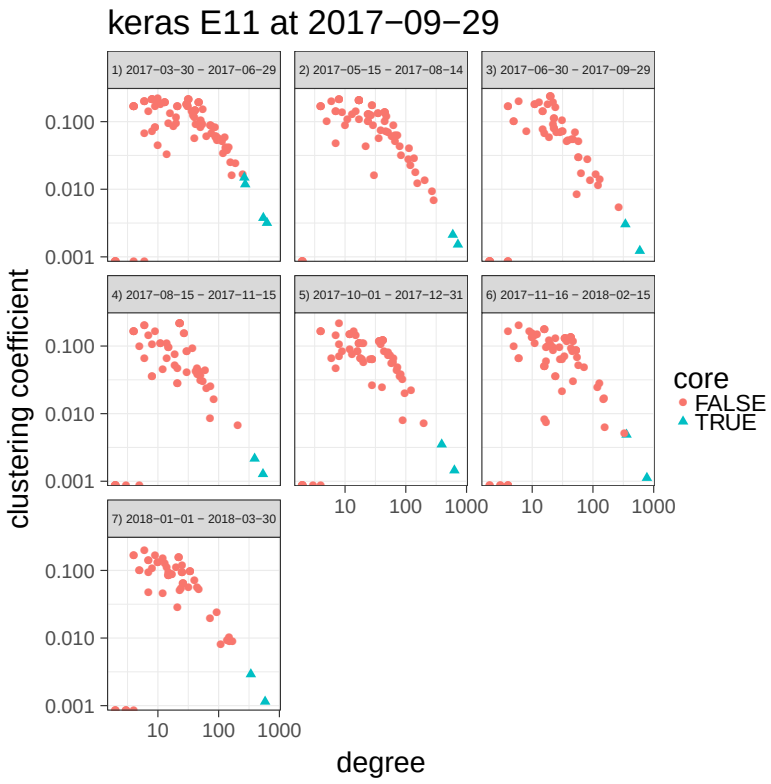

Figure 6.34: Network hierarchy for *E11*. No striking changes are recognizable.

6.4 Evaluation for Positive Organizational Events (G4)

Hypothesis 4 contains statements about the development of ownCloud and Qt, affected by positive organizational events.

6.4.1 Hypothesis 4.1: Lines of Code and Commit Count

Hypothesis 4.1. *The lines of code and the commit count after the event will increase.*

When we look at the commit count and changed lines of code for *E3* (new ownCloud CEO and finance investments) in Figure 6.35, we can see decreasing numbers after the event and even before the event. The peak in lines of code can be explained by previous commits which we discussed in Section 6.1.1 (moving of files). Against our expectations we got decreasing numbers. This can be explained by the previous events *E1* and *E2* (shortly before *E3*) which negatively affected the count based data. Thus, we assume that *E3* was a countermeasure against the downtrend. Even though it was not effective in the near future, we can see a slight uptrend when looking at the commit numbers for the whole ownCloud project in Figure A.4 some months after.

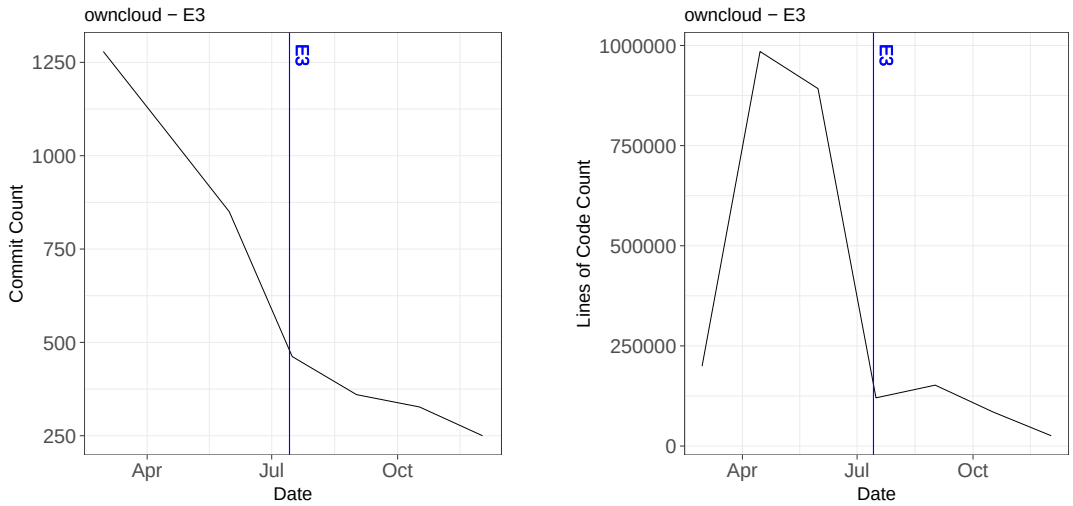

Figure 6.35: Commit count and changed lines of code count for ownCloud E3.

When investigating the Qt events, we only inspect the new Qt repository (Qt5) which officially started with event *E7* and the introduction of an open governance model. For Qt at events *E7* (Figure 6.36) and *E9* (Figure 6.37), we recognize a steady uptrend in commit numbers before and after the events. The lines of code for both events do not show a recognizable trend.

Conclusively, we can say that ownCloud with *E3* has decreasing commit numbers and changed lines of code before and after the event. Even though Qt matches our hypothesis in terms of commit count, it is not enough to claim that this hypothesis holds.

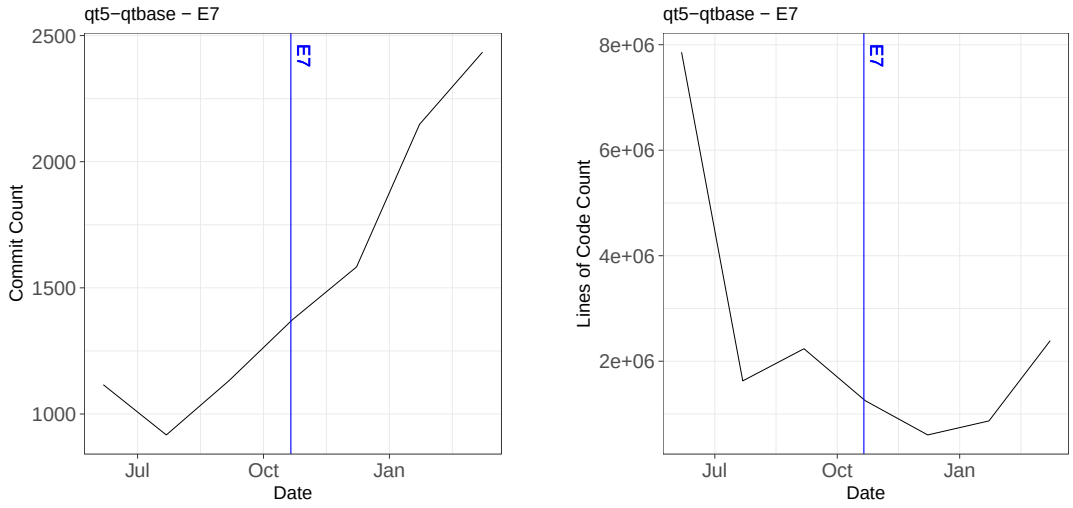

Figure 6.36: Commit count and changed lines of code count for Qt E7.

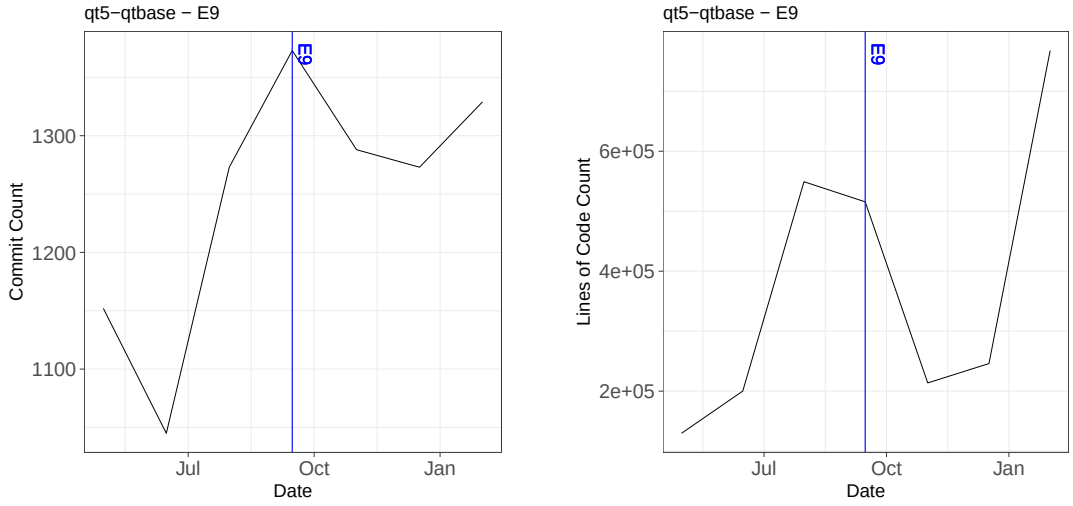

Figure 6.37: Commit count and changed lines of code count for Qt E9.

6.4.2 Hypothesis 4.2: Developer Count

Hypothesis 4.2. *The amount of contributing developers will increase.*

After evaluating Hypothesis 4.1, we already do not expect to see an immediate increase in developer numbers for ownCloud. When checking *E3* for ownCloud in Figure 6.38, we can see the drop from 60 active developers six months before the event to 20 developers six months after the event. As discussed in the previous section, we assume events *E1* and *E2* to be the cause for it. Looking at the bigger picture of ownCloud in Figure 6.13 shows that ownCloud regains developers after more then six months have passed since the event.

With Figure 6.39 and Figure 6.40, we visualize the developer count for *E7* and *E9* within a one-year range. *E7* led to continually increasing developer numbers, whereas the developer numbers six months before and six months after *E9* do not change a lot.

Overall, we have the same conclusion as in the previous section: The plot for ownCloud (*E3*) displays decreasing numbers in the developer count shortly after the event. An uptrend is noticeable when looking several months after the event (Figure 6.13). *Hypothesis 4.2* holds for *E7*, but not for *E9* when evaluating the Qt events of Group 4. Thus, we have to reject *Hypothesis 4.2*.

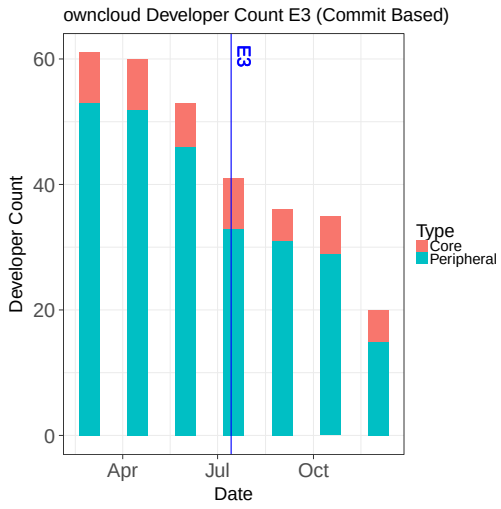

Figure 6.38: Developer count for ownCloud *E3*.

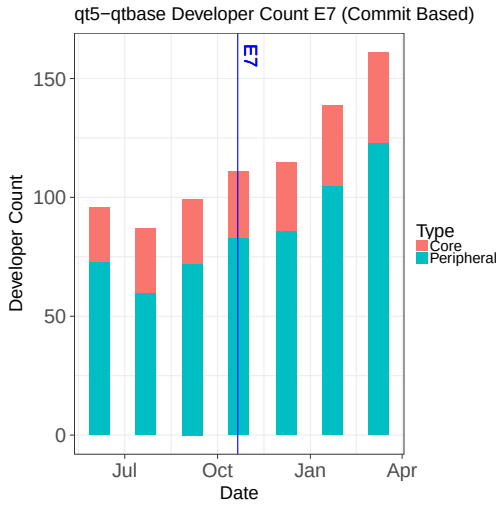

Figure 6.39: Developer count for Qt *E7*.

6.4.3 Hypothesis 4.3: Scale-Freeness

Hypothesis 4.3. *If the project was not in a scale-free state, it will gradually achieve it.*

Figure 6.10 and Figure 6.32 contain the scale free states for the ownCloud and Qt project. The network for Qt became scale free shortly before *E7* and never lost its scale-freeness ever since. ownCloud, on the other hand, briefly was scale-free after the event *E3* for one time window but never gained its scale-freeness back. Hence, the hypothesis does not hold.

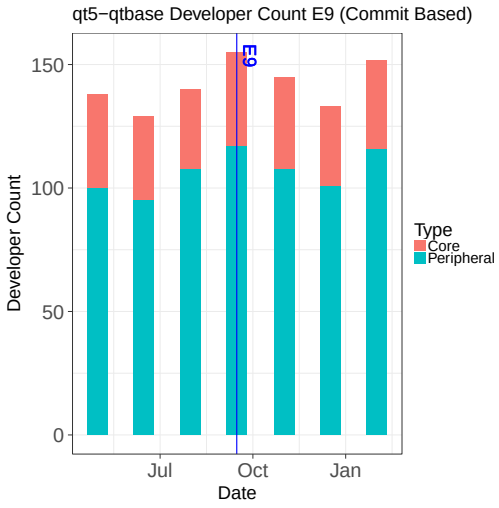

Figure 6.40: Developer count for Qt *E9*.

6.4.4 Hypothesis 4.4: Developer Hierarchy

Hypothesis 4.4. *The clustering coefficient for core developers will decrease, whereas the node degree will increase.*

As we already have seen different results in the previous hypotheses for ownCloud than expected, it is only natural that we will perceive different results in the hierarchy diagrams than claimed with *Hypothesis 4.4*. When looking at Figure 6.41, we can see that the clustering coefficient for core developers gradually increases and the node degree decreases as well. This effect can also be explained by *E1* and *E2*, which were the reason for the downtrend. The decreasing node degree and increasing clustering coefficient shows that there is a decrease in coordination done by core developers. In Figure 6.42 for Qt, we can see an increase in developer count towards several months after the event *E7*. A real trend for the clustering coefficient and node degree for core developers is not recognizable. Even though, there are two hierarchy diagrams showing an outlier core developer with increased node degree and decreased clustering coefficient. For *E9* we can see that the overall structure changes towards developers having a higher node degree and a smaller clustering coefficient. Thus, the coordination work increases overall.

The results for *E3* do not support the hypothesis and thus the hypothesis does not hold.

6.4.5 Conclusion for Hypothesis 4

Hypothesis 4 covered the events which we categorized as positive organizational events. Therefore we predicted positive consequences for our count based measurements, as well as network stability like we did for *Hypothesis 2*. Based on the developer count of ownCloud(Figure 6.13), we can see that *E3* was probably triggered by a previous downtrend. Especially the developer count was decreasing before the actual event date. We only can see a small uptrend in developer count months after the actual event. Especially the scale-freeness was never regained for ownCloud. But this is not enough to claim that this hypothesis holds, even though the results for Qt matches our hypotheses in most of the aspects.

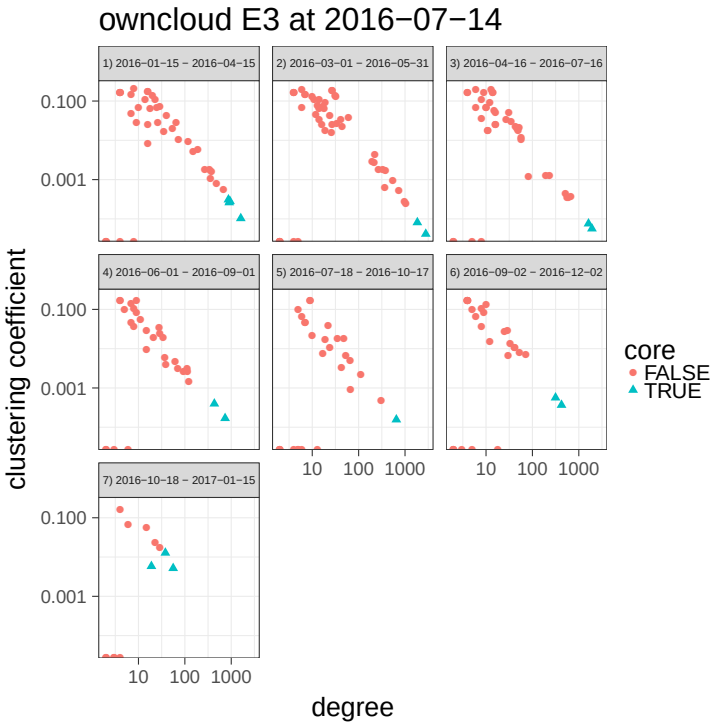

Figure 6.41: Network hierarchy for *E3*. The clustering coefficient for core developers gradually increases and the node degree decreases.

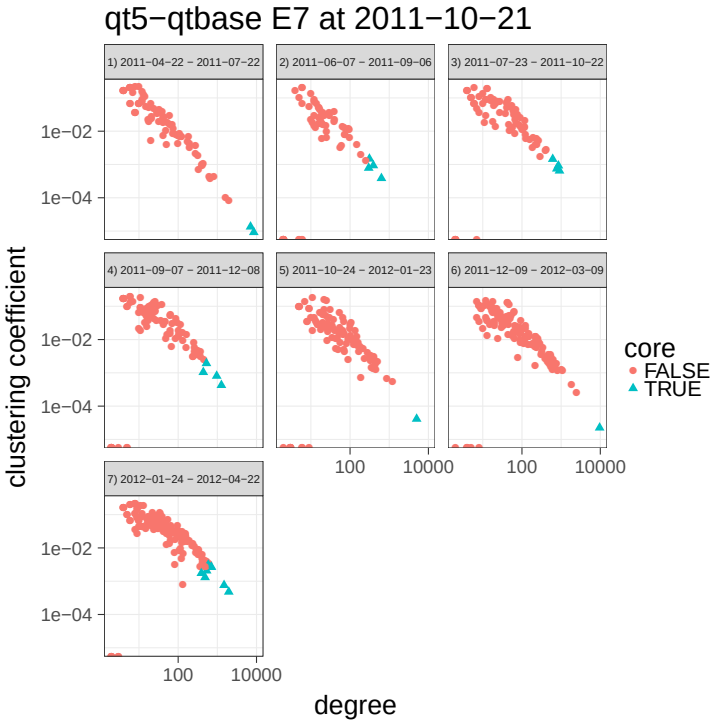

Figure 6.42: Network hierarchy for *E7*. A trend for the clustering coefficient and node degree for core developers is not recognizable.

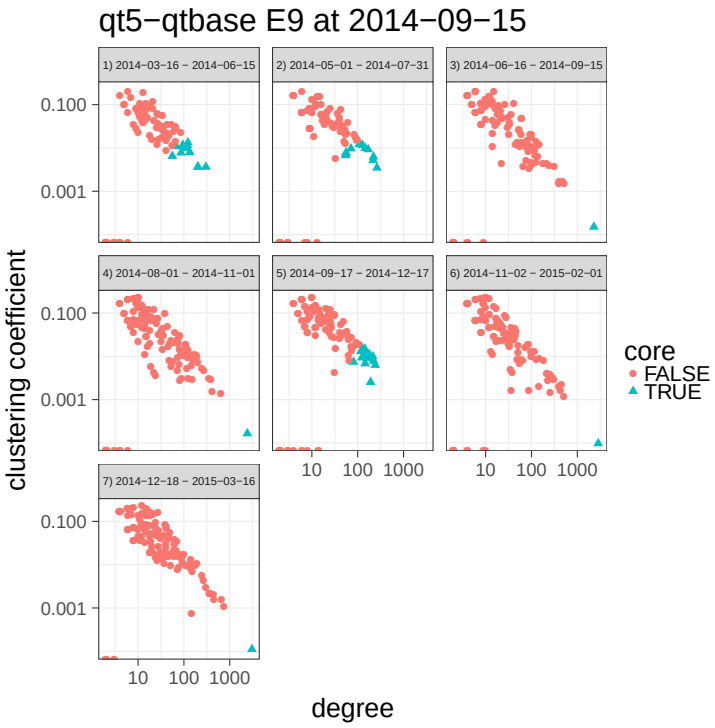

Figure 6.43: Network hierarchy for *E9*. The overall structure changes towards developers having a higher node degree and a smaller clustering coefficient.

6.5 Evaluation of Releases

In this section, we discuss the effects of releases on the repositories. When we evaluated the previous hypotheses, we often encountered different results than expected. Sometimes we suggested that releases which occurred before had an impact on the results. That is why we are going to examine major release dates for Owncloud, Node.js, Qt, and Keras. For Qt, we used the Qt5 repository which came with the switch to an open governance model, except for release 4.8 which was based on the old Qt4 repository. We focus on the trend of developer count six months before and six months after the release compared to the actual release date. Additionally, we look if the project is in a scale-free state six months before and after the release date. We mark projects as being scale-free if the projects are in a scale-free state more than 50 percent of the time in the three-month windows six months before and six months after the release.

Table 6.3 summarizes the information about the trend of developer count and scale-freeness for the release six months before and six months after the release date. If we take Node.js release 0.12.x at 2015-02-06 for example, we can see that the developer count increases by 37% towards the release date. We calculate this percentage by accumulating the developer count for three time windows before the release. Afterwards, we calculate the average developer count for the three time windows and compare the value with the developer count at the release time window. The developer count decrease by 33% towards six months after the release. We calculate this value similar as we did for the previous one, except we use the three time windows after the release date to calculate the average. The project was in a scale-free state most of the time six months before the release and lost its scale-freeness for over 50% of the time windows six months after the release.

Evaluating the table, we can see that there is almost always an increase in developer count before the release and a decrease afterwards for Owncloud. The same results are visible for Node.js except for release 8.x, which shows an increase in developer count after the release date. Also Node.js release 6.x shows a different trend, but the values are almost zero percent. Qt, on the other hand, has almost always decreasing developer numbers before a release and increasing numbers after a release. The developer numbers for Keras at the releases are quite close to each other. But release 2.0.8 shows a massive downtrend before the release date and release 1.0.0 shows an huge increase after the event.

When looking at the scale-freeness six months before and after the release dates we can see that the networks almost always stay in the same state. There are only 5 out of 29 releases which show a change of network state. Three of them show the change of a not scale-free state towards a scale-free state. Those three releases also have increasing developer numbers before and after the release.

To conclude, we can say that the developer count is almost always affected by release dates. Release 0.12.x for Node.js and 5.0 for Qt lead towards a none scale-free state. When looking at the developer count for those releases, we can see huge decreasing numbers. Overall, the scale-free state before and after a release stays the same for almost every release we checked.

Project	Release	Date	Dev Count Before	Dev Count After	Scale Before	Scale After
Owncloud	1.0	2010-06-24	17 %	-44 %	False	False
Owncloud	2.0	2011-10-11	39 %	-21 %	False	False
Owncloud	3.0	2012-01-01	-29 %	51 %	False	False
Owncloud	4.0	2012-05-22	16 %	-18 %	False	False
Owncloud	5.0	2013-03-14	26 %	-21 %	False	False
Owncloud	6.0	2013-12-11	17 %	0 %	False	False
Owncloud	7.0	2014-07-23	25 %	-25 %	False	False
Owncloud	7.0.15	2016-05-13	-12 %	-26 %	False	False
Owncloud	8.0	2015-02-09	-17 %	-5 %	False	False
Owncloud	9.0	2016-03-08	11 %	-20 %	False	False
Owncloud	9.1	2016-07-21	-43 %	-26 %	False	False
Owncloud	10.0.0	2017-04-27	20 %	-18 %	False	False
Node.js	0.12.x	2015-02-06	37 %	-33 %	True	False
Node.js	4.x	2015-09-08	33 %	29 %	False	True
Node.js	5.x	2015-10-29	34 %	9 %	True	True
Node.js	6.x	2016-04-26	-2 %	5 %	True	True
Node.js	7.x	2016-10-25	55 %	-33 %	True	True
Node.js	8.x	2017-05-30	-3 %	65 %	True	True
Node.js	9.x	2017-10-01	52 %	-35 %	True	True
Qt	4.8	2011-12-15	-23 %	1 %	True	True
Qt	5.0	2012-12-19	0 %	-11 %	True	True
Qt	5.1	2013-07-03	7 %	9 %	True	True
Qt	5.2	2013-12-12	-2 %	0 %	True	True
Qt	5.3	2014-05-20	-14 %	7 %	True	True
Qt	5.4	2014-12-10	-11 %	9 %	True	True
Keras	0.3.0	2015-12-01	16 %	11 %	False	True
Keras	1.0.0	2016-04-11	3%	31%	False	True
Keras	2.0.8	2017-08-25	-53%	14%	True	True
Keras	2.0.9	2017-11-01	-9%	-2%	True	True

Table 6.3: Releases for Owncloud, Node.js, Qt and Keras and their evolution six months before and after the release in terms of developer count and scale-freeness. "Dev Count Before" and "Dev Count After" represent the the difference between the developer count of the release date and the average developer count for three time windows before and three time windows after a release.

6.6 Threats to Validity

Internal Validity. We inspect repository data and developer networks over time. Choosing three-month time windows does not necessarily imply reliable results. However, we choose three-month time windows based on the experience of previous work [MW11, JAM17]. Depending on developer productivity and participation of each project, smaller or bigger time windows might be a better fit. Another threat is that we cannot determine the exact date for an event and their consequences. Consequences can show up days, months or even years before or after an event. When investigating an event, we choose to analyze the commit data six months before and six months after the event. A one year period for evaluating events does not necessarily deliver expressive results. Choosing smaller or larger time periods could lead to better results. However, by using overlapping time windows for the events, we tried to get additional information to support our statements at the event dates. Also developers having different GitHub accounts for their working space for example can manipulate our perceived data. Therefore core developers could be falsely classified as peripheral. Another threat might be bots doing cleaning commits once in a while, which also can distort our results. Further, only analyzing commit-based data without taking communication channels like mailing lists, chats, GitHub comment sections, etc. into consideration can lead to wrong results. The selection of events is also important. By choosing similar events or events without impactful consequences, we can get misguided by our perceived results. Therefore, we tried to select different event types and ordered them into different event groups to soften the consequences.

External Validity. We examined events for four different open source software projects which are all different in size and population. Only picking open source projects is an external threat to validity. By choosing different sized projects and projects with different publicity we try to soften the consequences. For example Keras is mainly known by people working in the field of machine learning, whereas ownCloud serves a broader audience. Thus, consequences can have different magnitudes. Also, the contributor number for three-month time windows of our evaluated events, range from less than 50 to over 200 active developers. Unexpected human behavior is also an external threat. We can not predict the human response to events and thus we cannot always see expected changes to the data. Another external threat to validity is to choose the correct active repository of a project for mining. As we have seen with Qt, two different repositories were overlapping until the development of one repository finally stopped.

7 Conclusion

With this thesis, we wanted to investigate consequences for open source software projects caused by special events within the respective community. Therefore, we chose four different open source projects and inspected different events occurring to them. We ended up with 11 different events and ordered them into social or organizational change. Additionally, we divided the events by their expected positive or negative consequences for the community. We created an analyzing script for building developer networks based on commit data, calculating count-based and network-based metrics, and visualizing the results in form of expressive plots.

After identifying the events, we constructed hypotheses for each event group. Those hypotheses contained assumptions towards the evolution of the repositories. The hypotheses covered changes to count-based information and developer networks.

We found out that some social and organizational events show changes to developer count, scale-freeness, node-degree and clustering coefficient of core developers. At the same time, we found contrary results than we expected from certain events to have. Thus, we assumed that we cannot predict if contributors react positive or negative towards certain events. Human behavior and their decision making seem to be a disturbing factor. Nevertheless, we found out that social events have a higher impact on the perceived data and network then organizational ones.

With this thesis, we examined GIT data within six months before and six months after an event. We primarily evaluated the results visually. A future approach could be to analyze a complete repository and statistically find abnormal data or network behavior at a specific time. Afterwards, the community and the media can be checked in order to find social or organizational events or even releases at that time window. Statistical analysis in combination with a visual approach might deliver clearer results. Also taking communication channels like mailing lists, chats and GITHUB comment sections into account, when building the developer networks, would lead to better results. Knowing the consequences of events beforehand, would allow the management of open source software projects to act appropriately and to minimize the negative consequences.

A Appendix

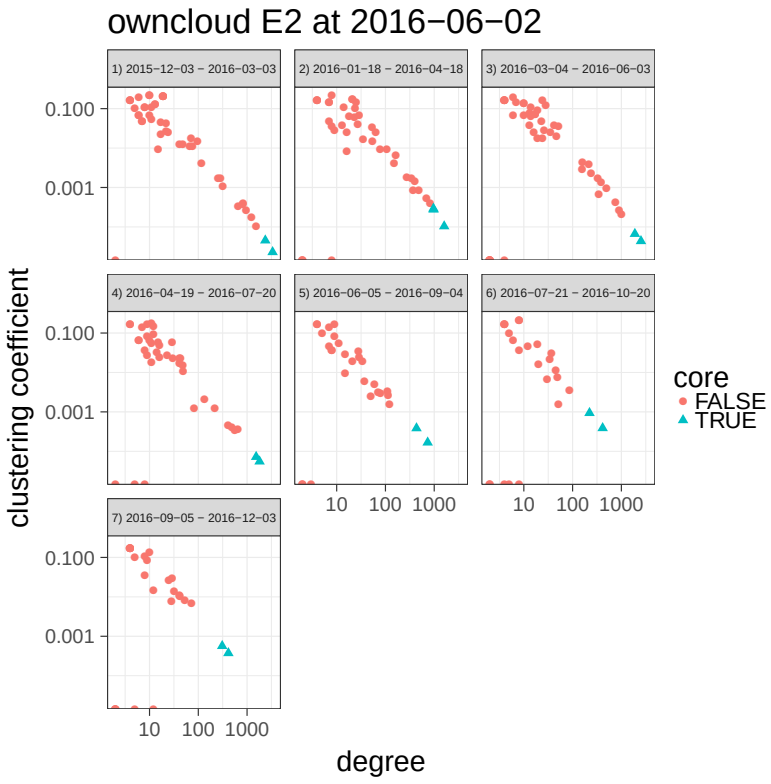

Figure A.1: Network Hierarchy for *E2*.

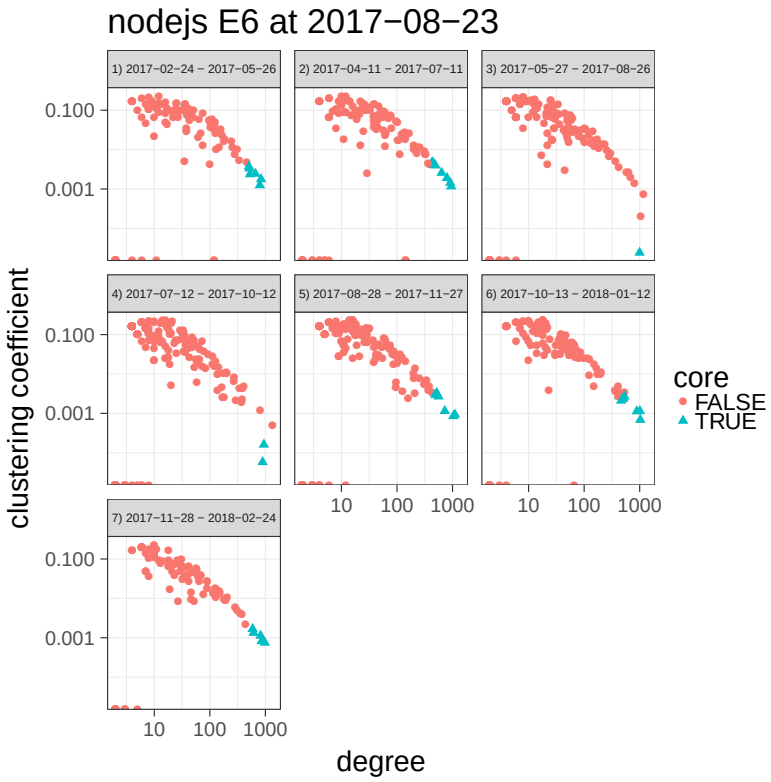

Figure A.2: Network Hierarchy for *E6*.

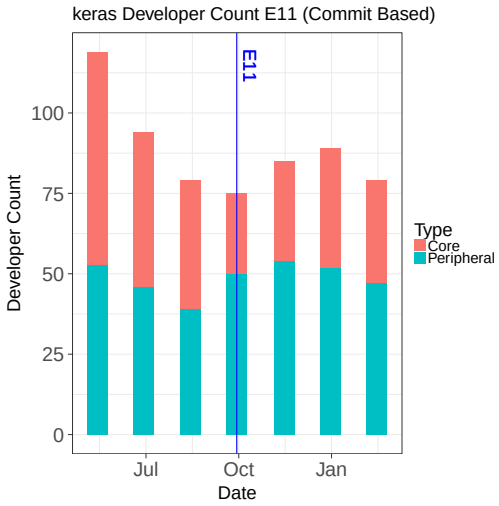

Figure A.3: Developer count for KERAS *E11*.

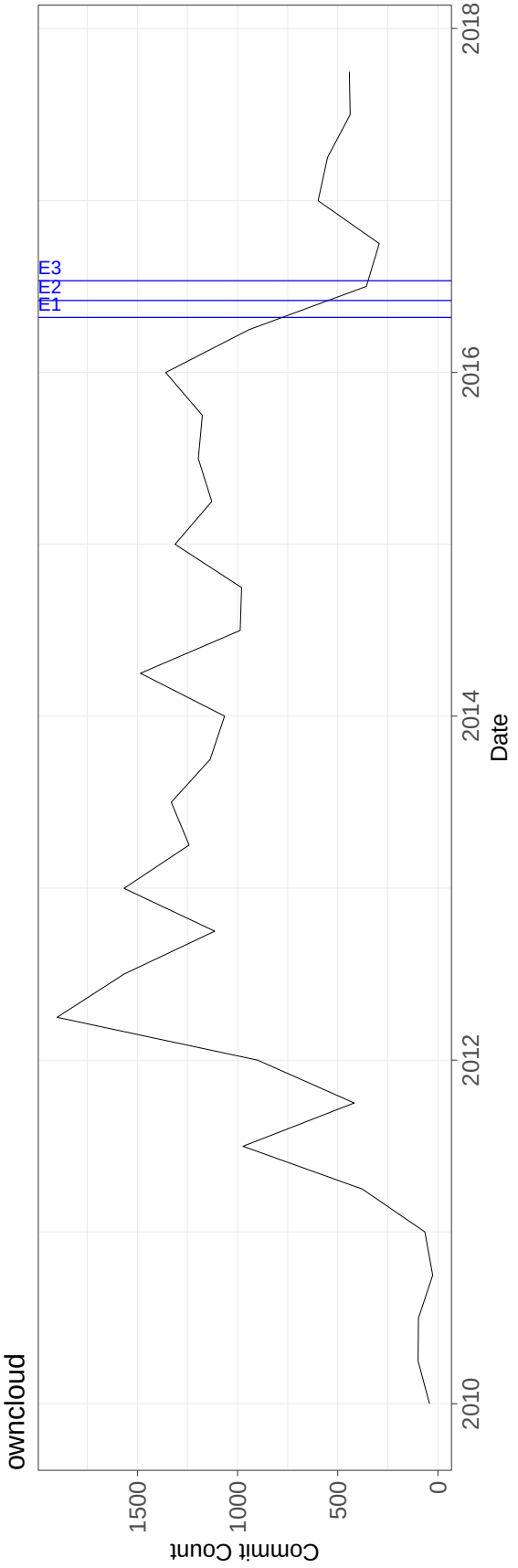

Figure A.4: Commit Count Owncloud

Bibliography

[Amo15] Amorelandra. io.js Project Governance. Website, February 2015. Available online at https://github.com/nodejs/node/blob/v1.x/GOVERNANCE.md; visited on February 4th, 2019. (cited on Page 18)

[APHV16] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. A novel approach for estimating truck factors. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016. (cited on Page 4)

[AVH10] Chintan Amrit and Jos Van Hillegersberg. Exploring the impact of soclotechnlcal core-periphery structures in open source software development. *journal of information technology*, 25(2):216–229, 2010. (cited on Page 3)

[BA99] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999. (cited on Page 14)

[Baa17] Hans-Joachim Baader. Ayo.js: Node.js abermals geforkt. Website, 2017. Available online at http://www.pro-linux.de/news/1/25085/ayojs-nodejs-abermals-geforkt.html; visited on January 5th, 2019. (cited on Page 19)

[BE05] Ulrik Brandes and Thomas Erlebach. Network analysis: methodological foundation. 2005. (cited on Page 5)

[BEJ18] Stephen P Borgatti, Martin G Everett, and Jeffrey C Johnson. *Analyzing social networks*. Sage, 2018. (cited on Page 14)

[BLM+06] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4-5):175–308, 2006. (cited on Page 11)

[BPD+08] 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*, pages 24–35. ACM, 2008. (cited on Page 3 and 13)

[Bö16] Thomas Börger. ownCloud sichert Finanzierung und erweitert Geschäftsführung. Website, July 2016. Available online at https://owncloud.com/de/owncloud-sichert-finanzierung-und-erweitert-geschaeftsfuehrung/; visited on February 4th, 2019. (cited on Page 17)

[CWLH06] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. Core and periphery in free/libre and open source software team communications. In *System Sciences, 2006. HICSS'06. Proceedings of the 39th Annual Hawaii International Conference on*, volume 6, pages 118a–118a. IEEE, 2006. (cited on Page 3 and 13)

[DM13] Sergei N Dorogovtsev and José FF Mendes. *Evolution of networks: From biological nets to the Internet and WWW*. OUP Oxford, 2013. (cited on Page 11)

[Dol15] Mike Dolan. Node.js and io.js leaders are building an open, neutral Node.js Foundation to support the future of the platform. Website, May 2015. Available online at https://nodejs.org/en/blog/community/node-leaders-building-open-neutral-foundation/; visited on February 4th, 2019. (cited on Page 19)

[DSTH12] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*, pages 1277–1286. ACM, 2012. (cited on Page 3)

[Duc14] Chris Duckett. Qt hot potato spun out from Digia into fourth home. Website, August 2014. Available online at https://www.zdnet.com/article/qt-hot-potato-spun-out-from-digia-into-fourth-home/; visited on March 4th, 2019. (cited on Page 21)

[FK90] Eugene PH Furtado and Vijay Karan. Causes, consequences, and shareholder wealth effects of management turnover: A review of the empirical evidence. *Financial management*, pages 60–75, 1990. (cited on Page 17)

[GPD14] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014. (cited on Page 3)

[Gr6] Sebastian Grüner. Owncloud-Gründer und -Technikchef verlässt das Unternehmen. Website, April 2016. Available online at https://www.golem.de/news/frank-karlitschek-owncloud-gruender-und-technikchef-verlaesst; visited on January 5th, 2019. (cited on Page 17)

[HC90] Rebecca M Henderson and Kim B Clark. Architectural innovation: The reconfiguration of existing product technologies and the failure of established firms. *Administrative science quarterly*, pages 9–30, 1990. (cited on Page 3)

[JAHM17] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. Classifying developers into core and peripheral: An empirical study on count and network metrics. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 164–174. IEEE, 2017. (cited on Page 3, 5, 6, 12, and 13)

[JAM17] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. Evolutionary trends of developer coordination: A network approach. *Empirical Software Engineering*, 22(4):2050–2094, 2017. (cited on Page 4, 5, 7, 11, 14, 32, and 62)

[JMA+15] 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 37th International Conference on Software Engineering-Volume 1*, pages 563–573. IEEE Press, 2015. (cited on Page 13)

[JSS11] 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*, pages 24–31. ACM, 2011. (cited on Page 4)

[Kar16a] Frank Karlitschek. big changes: I am leaving ownCloud, Inc. today. Website, April 2016. Available online at https://karlitschek.de/2016/04/big-changes-i-am-leaving-owncloud-inc-today/; visited on January 5th, 2019. (cited on Page 16 and 17)

[Kar16b] Frank Karlitschek. Nextcloud. Website, June 2016. Available online at https://karlitschek.de/2016/06/nextcloud/; visited on February 4th, 2019. (cited on Page 17)

[Kno11] Lars Knoll. The Qt Project is live! Website, October 2011. Available online at https://blog.qt.io/blog/2011/10/21/the-qt-project-is-live/; visited on January 5th, 2019. (cited on Page 21)

[Lam17] Pascal Lamblin. MILA and the future of Theano. Website, September 2017. Available online at https://groups.google.com/forum/#!topic/theano-users/7Poq8BZutbY; visited on January 5th, 2019. (cited on Page 22)

[LVH04] Karim R Lakhani and Eric Von Hippel. How open source software works:"free" user-to-user assistance. In *Produktentwicklung mit virtuellen Communities*, pages 303–339. Springer, 2004. (cited on Page 1)

[MFH02] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002. (cited on Page 3 and 13)

[Mil93] Danny Miller. Some organizational consequences of ceo succession. *Academy of Management Journal*, 36(3):644–659, 1993. (cited on Page 17)

[MS17] Rainald Menge-Sonnentag. Node.js: Ärger um Code of Conduct führt zu einem Fork. Website, 2017. Available online at https://www.heise.de/developer/meldung/Node-js-Aerger-um-Code-of-Conduct-fuehrt-zu-einem-Fork-3810543.html; visited on January 5th, 2019. (cited on Page 19)

[MW11] Andrew Meneely and Laurie Williams. Socio-technical developer networks: Should we trust our measurements? In *Proceedings of the 33rd International Conference on Software Engineering*, pages 281–290. ACM, 2011. (cited on Page 62)

[Nan12] Dan Nancarrow. Nokia closes Australian development office. Website, August 2012. Available online at https://www.smh.com.au/technology/nokia-closes-australian-development-office-20120803-23kc4.html; visited on January 5th, 2019. (cited on Page 21 and 51)

[nod] Node.js JavaScript runtime. Website. Available online at https://github.com/nodejs/node; visited on February 28th, 2018. (cited on Page 1)

[NYN+02] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the international workshop on Principles of software evolution*, pages 76–85. ACM, 2002. (cited on Page 3)

[qt:15] Qt Project Open Governance. Website, January 2015. Available online at https://wiki.qt.io/Qt_Project_Open_Governance; visited on January 5th, 2019. (cited on Page 21)

[RB03] Erzsébet Ravasz and Albert-László Barabási. Hierarchical organization in complex networks. *Physical review E*, 67(2):026112, 2003. (cited on Page 11)

[War12] Tom Warren. Digia acquires Qt from Nokia, plans to enable it on Android, iOS, and Windows 8. Website, August 2012. Available online at https://www.theverge.com/2012/8/9/3229785/digia-nokia-qt-acquisition; visited on March 4th, 2019. (cited on Page 20)

[WK03] Laurie Williams and Robert Kessler. Pair programming illumi—nated addison wesley, 2003. (cited on Page 4)

[16] Fran cois Chollet. Deep Learning for Coders. Website, January 2016. Available online at https://www.reddit.com/user/fchollet; visited on January 5th, 2019. (cited on Page 22)

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Name

Passau, den 18. März 2019