Reference Coupling: An Exploration of Inter-project Technical Dependencies and their Characteristics within Large Software Ecosystems

Kelly Blincoe[a,*], Francis Harrison[b], Navpreet Kaur[b], Daniela Damian[b]

[a] *University of Auckland, New Zealand*
[b] *University of Victoria, BC, Canada*

Abstract

Context: Software projects often depend on other projects or are developed in tandem with other projects. Within such software ecosystems, knowledge of cross-project technical dependencies is important for 1) practitioners understanding of the impact of their code change and coordination needs within the ecosystem and 2) researchers in exploring properties of software ecosystems based on these technical dependencies. However, identifying technical dependencies at the ecosystem level can be challenging.

Objective: In this paper, we describe Reference Coupling, a new method that uses solely the information in developers online interactions to detect technical dependencies between projects. The method establishes dependencies through user-specified cross-references between projects. We then use the output of this method to explore the properties of large software ecosystems.

Method: We validate our method on two datasets — one from open-source projects hosted on GitHub and one commercial dataset of IBM projects. We manually analyze the identified dependencies, categorize them, and compare them to dependencies specified by the development team. We examine the types of projects involved in the identified ecosystems, the structure of the iden-

*Corresponding author
Email addresses: kblincoe@acm.org (Kelly Blincoe), francish@uvic.ca (Francis Harrison), kaur.navpreet472@gmail.com (Navpreet Kaur), danielad@uvic.ca (Daniela Damian)

Preprint submitted to Elsevier *February 2, 2019*

tified ecosystems, and how the ecosystems structure compares with the social behaviour of project contributors and owners.

Results: We find that our Reference Coupling method often identifies technical dependencies between projects that are untracked by developers. We describe empirical insights about the characteristics of large software ecosystems. We find that most ecosystems are centered around one project and are interconnected with other ecosystems. By exploring the socio-technical alignment within the GitHub ecosystems, we also found that the project owners social behaviour aligns well with the technical dependencies within the ecosystem, but the project contributors social behaviour does not align with these dependencies.

Conclusions: We conclude with a discussion on future research that is enabled by our Reference Coupling method.

1. Introduction

Software is not developed in isolation anymore. Whether open source or corporate-led, software development takes place within "a collection of software projects which are developed and which co-evolve together in the same environment", and which are referred to as software ecosystems [1]. Within such ecosystems, projects depend on one another [1], and yet awareness of such dependencies is not trivial. Identifying technical dependencies to external projects within the ecosystem is important for two main reasons: First, developers need to understand how their tasks and code changes impact other projects and who they need to coordinate their changes with at the ecosystem level [1, 2]. For open source ecosystems in particular, it is also important for attracting new contributors [1] since dependencies to other projects within an ecosystem are more likely to attract attention. Newcomers might decide to join the project based on a deeper knowledge of the structure of the system as well as of its external dependencies. Second, information about software ecosystems and technical dependencies within them enables further exploration and modeling of software ecosystems, an area currently understudied in the software engineering litera-

ture [3].

However, identifying technical dependencies between projects on a large scale has proven to be difficult [4]. Existing static dependency analysis approaches do not identify dependencies across projects. Methods for extracting external dependencies from a project's source code or configuration files have been proposed [1, 4, 5, 6, 7, 8, 9, 10, 11], but these approaches limit the types of dependencies detected to explicit relationships. Implicit relationships like dependencies on web services, operating systems, or hardware are not always visible in configuration files or source code. Further, methods that extract dependencies from source code require large amounts of memory and computation time, so they cannot be employed across a large set of projects [12]. Methods that are applied to configuration or build files are not memory or computation-intensive, but such files are not always available or accurate since not all projects use a package manager.

Without a way to easily establish a comprehensive set of dependencies between projects, software practitioners are unable to quickly identify the external dependencies of a software project or understand where their project sits within a software ecosystem.

In this paper, we propose a new method, Reference Coupling, to detect cross-project dependencies that leverages solely the information in the developers' social interactions. The social aspects of a project and its surrounding ecosystem significantly influence the way in which the software project will evolve over time [13], and they cannot be ignored in the development of models, guidelines and best practices for the analysis and maintenance of software ecosystems' health [14]. Reference Coupling mines the references to other projects that developers make in their online interaction (referred to as cross-references henceforth). We validated the method in identifying true cross-project technical dependencies by using it in two large datasets of GitHub and IBM projects and comparing its results with manually identified cross-references in each dataset. We found that the Reference Coupling method does identify technical dependencies between projects, and we describe several properties of the ecosystems

identified using this method. Reference coupling represents a significant novel complement to the other existing, but insufficient, code-based or configuration file-based methods to identify external dependencies within a project's ecosystem. Our method identifies ecosystems of projects within the same organization, and also outside of the organization for which there are technical dependencies, often hidden or even unknown to developers within a project.

Having identified external technical dependencies to projects in our datasets, we further used our Reference Coupling method with a popular community detection algorithm [15] to identify ecosystems across all GitHub-hosted projects and explore socio-technical aspects within these ecosystems. We found that the developers' socio-technical behavior within GitHub project ecosystems differs between the project owners and actual contributors. Our analysis illustrates the potential for further analysis of software ecosystems' health, something difficult to assess given the dynamic nature of relationships within the ecosystem, as well as the lack of a centralized management structure for overseeing the ecosystem's health and survival, most often typical of open source projects [14].

Our previous conference publication reported on some elements of this work [16]; however, this paper introduces numerous extensions to our work. Specifically this paper extends our previous publications by:

- describing how to utilize the Reference Coupling method on a wide variety of software tools (compared to just GitHub in the conference publication).
- validating the Reference Coupling approach on a new dataset from a proprietary ecosystem of IBM projects.
- providing a more detailed understanding of the types of dependencies that are captured by the Reference Coupling method through the analysis and categorization of dependencies from the IBM ecosystem and a more detailed analysis of the dependencies on GitHub.
- extending the discussion to further describe how the Reference Coupling method can be used to support software developers and software engineer-

ing researchers.

The paper has also been significantly restructured to better describe our research methods, development, validation and application of the Reference Coupling technique.

The rest of the paper is structured as follows: Section 2 provides an overview of related work in software ecosystems and dependency conceptualizations. The Reference Coupling method is described in Section 3. Our research methods are presented in Section 4. Our results are presented in Section 5. In Section 6, we summarize our findings and discuss open questions for future research. We provide a brief conclusion in Section 7.

2. Related Work

The term software ecosystem (SECO) has emerged as a paradigm to understand the dynamics and heterogeneity in collaborative software engineering. Unlike natural ecosystems, however, there is no common definition for SECO. Two different perspectives on SECOs have been identified in the literature, namely business-centric and platform-centric [3]. The business-centric definition refers to the holistic, business-oriented perspective of a SECO as a network of actors, organizations and companies [17, 18, 19]. The platform-specific perspective emphasizes the social and technical aspects of a set of software projects, technical platforms, and communities, in line with work of [20, 21]. In our work, we take a platform-specific perspective to leverage and study the socio-technical relationships within ecosystems. The ecosystems we consider are not limited to only those projects in the same organization, but also include projects outside of the organization for which there are technical dependencies.

The software ecosystems that received the most attention in previous literature include those around Eclipse (e.g. [6, 22]), Ruby on Rails (e.g. [23, 24]), Apache (e.g. [7, 9]), and GNOME (e.g. [13]). Notable research developments also exist in the area of frameworks for analysis of Open Source Software Ecosystems (OSSECOs) (e.g. [21]), OSSECO health measurement (e.g. [25]), and tools

for visualizing OSSECO projects (e.g. [26]).

Recent extensive literature surveys show a growing interest in studies of ecosystems, both in the domain of proprietary [27] as well as open source software development [3]. In the proprietary software space, the focus has been on the organizational and business aspects of the ecosystems, with a clear lack of deeper investigations of technical and collaborative aspects of work [27]. In the open source software ecosystems space, the pressing research challenges include the development of methods and tools for the ecosystem modelling and analysis, socio-technical theories to explain the interplay between the social and technical system within ecosystems, as well as the diagnostic and monitoring of ecosystem quality and health [3].

An important step in this direction lies with methods for the identification of a project's external technical dependencies within its ecosystem, in order to study its structure and socio-technical aspects. Analysis of a project's source code is a common technique to identify technical dependencies within a project (intra-project). However, these techniques do not scale up well to identify dependencies between projects (inter-project). Lungu et al. [5] describe several methods for extracting inter-project dependencies by considering external method and class calls in a project's source code. However, when investigating a large number of projects, obtaining the source code for every project is not always feasible. Collecting source code data across an entire versioning system would require multiple Terabytes of data and more than a year in processing time [12]. Ossher et al. [4] introduced a technique that analyzes import statements in Java source code to resolve inter-project dependencies. Businge and Serebrenik [6] employ a similar technique in their study of the Eclipse ecosystem. However, this technique still requires obtaining a large amount of source code and, therefore, requires a large amount of memory. These techniques, therefore, are limited in the number of projects that can be studied.

Previous studies have also proposed ways to identify technical dependencies without relying on analysis of source code. One method is to identify technical dependencies by examining declared dependencies from a project's configuration

files or its build files from a dependency management tool like Maven [1, 7, 8, 9, 10, 11]. However, not all projects declare dependencies in configuration files or employ a dependency manager, and, even for those that do, the data can be missing. Bavota et al. [9] found that this information was missing in 37% of releases in a study of the Apache project. Syeed et al. [23] extracted metadata on inter-project dependencies from the published specifications at rubygems.org in their study of the Ruby on Rails ecosystem. However, the specified dependencies may be out of date and the approach is specific to only projects that publish dependency specifications.

Considering the social aspects of software ecosystems are also important [3]. The work of Mens and his colleagues highlights the role that social aspects play in the future understanding and development of tools, prediction models, guidelines and best practices that allow ecosystem communities to improve upon their current practices [13]. Several other studies [28, 29] have explored ways to detect social connections in software ecosystems. These studies used community detection algorithms to detect communities across GitHub projects focusing on relationships between developers.

We use technical dependencies for community detection since the structure of an ecosystem is defined by its technical dependencies [5]. Thung et al. [30] constructed similar project-to-project networks for GitHub-hosted projects. In their networks, edges between projects represent a single developer contributing to both projects. Since developers can often work on multiple independent projects, sharing developers is not an indication of a technical dependency and their network is more of a social perspective.

In our approach, we propose a method for automatic identification of technical dependencies that does not rely on analyzing source code, but takes advantage of the cross-references that can be made in developers' social interactions. These cross-references are user-specified links between a pair of projects. They are made in comments on work items, pull requests, issues, and commits as developers coordinate and manage their work dependencies. With these identified technical dependencies we were then able to conduct a socio-technical analy-

sis of the behavior of different project members, namely project owners and contributors in the GitHub ecosystem.

3. Reference Coupling

To identify dependencies between projects, we relied on comments made by the developers within one project's tasks, issues, pull requests and commits that cross-reference another project. Modern collaborative software development tools make it easy for developers to create links between projects within their comments. We call this conceptualization of dependencies between projects *Reference Coupling*.

To develop the Reference Coupling method, we manually examined cross-references between projects on several software development tools to understand how they could be automatically extracted. We examined comments that cross-reference other projects on the following, popular open source software code hosting platforms, forges, and issue trackers: GitHub, GitLab, BitBucket, SourceForge, and Jira. We also examined cross-references in a proprietary software ecosystem by examining the comments in an IBM set of products that together form the Rational solution for Collaborative Lifecycle Management (CLM).

We found that all of the open source software code hosting platforms, forges, and issue trackers we examined (GitHub, GitLab, BitBucket, SourceForge, and Jira) employ Markdown languages, which allow plain text to be formatted in a lightweight way. When a Markdown language is employed and a user cross-references another project in a comment using the appropriate syntax, a link to the other project is automatically created, making it easier to navigate between the projects. Due to the adoption of these Markdown languages, cross-references to other projects are created in a standard format to enable these automatic links to be created. GitHub, Bitbucket, and GitLab all extend the CommonMark specification [31], which was introduced to standardize markdown implementations. Jira and SourceForge, use their own Markdown languages.

Table 1: Syntax of Cross-References

Tool	**Link To**	**Syntax**			**Example**
		Project Identifier	**Artifact Type**	**Artifact ID**	
GitHub, GitLab, Bitbucket, and Jira	Issues/Pull Requests	OWNER/PROJECT	#	NUMBER	rails/rails#123
	Commits	OWNER/PROJECT	@	SHA	twbs/bootstrap@6e2a82
SourceForge	Issues	PROJECT[/SUBPROJECT]	bugs	NUMBER	allura:bugs:#123
		PROJECT[/SUBPROJECT]	features	#NUMBER	allura:features:#123
	Commits	PROJECT[/SUBPROJECT]	code	SHA	allura:code:3b9d48
IBM CLM	Work Item	—	[work] item	NUMBER	work item 123
		—	task	NUMBER	task 456
		—	story	NUMBER	story 789
		—	defect	NUMBER	defect 123
		—	URL	NUMBER	https://jazz.net/jazz/resource/itemName/ com.ibm.team.workitem.WorkItem/123

*Text in [ ] is optional

The one proprietary software tool ecosystem we reviewed, the IBM CLM suite, did not employ a Markdown language. However, it still allowed explicit links to be created between projects using a graphical user interface. When entering text in a comment, users can click on a button that will allow them to insert a link to a work item. This will open a window which will allow the user to search for the work item(s) they wish to link to.

For all of these tools, whether they employ a Markdown language or not, the format of the cross-references in the comment follows the same high-level pattern with an optional project identifier followed by the artifact type and the artifact identifier. The project identifier is optional in the cases that all projects in the ecosystem share the same instance of the project management tool, resulting in unique artifact identifiers across all of the projects, as is the case in the IBM CLM ecosystem. Given the commonality of the syntax across all of these tools, such cross-references can be automatically extracted. Thus, our Reference Coupling method identifies dependencies by considering the following pattern in comments:

<project identifier (optional) ><artifact type ><artifact identifier >

The detailed syntax for the various tools and artifact types is shown in Table 1.

If this method were to be implemented in a tool, these cross-references could

be automatically stored in a separate database table when they are created, making the detection of dependencies nearly real-time. Post-hoc analysis, where cross-references have not been previously extracted and stored, could be done in O(n) time where n is the number of comments to be analyzed since one computation is required for each comment to examine whether there is a cross-reference to another project.

4. Research Methodology

4.1. Research Questions

We validated the Reference Coupling method on two datasets — one from open-source projects hosted on GitHub and the one commercial dataset of IBM projects. Our validation was guided by the following research questions:

RQ1a: Does the Reference Coupling method identify inter-project technical dependencies on GitHub issue, pull request, and commit comments? If so, what are the characteristics of these dependencies?

RQ1b: Does the Reference Coupling method identify inter-project technical dependencies on IBM work item comments? If so, what are the characteristics of these dependencies?

We then explored the characteristics of as well as socio-technical alignment within the identified GitHub ecosystems by asking: *RQ2: What ecosystems exist across GitHub-hosted projects and what is their structure?*, and *RQ3: Do the project owners' and contributors' social behaviours align with the technical dependencies?*

4.2. Research Setting and Data Collection

To answer our research questions, we conducted an analysis of the comments and cross-references in the GitHub and IBM CLM projects.

4.2.1. GitHub

We obtained data from the GHTorrent [32] project, which provides a mirror of the GitHub API data. GHTorrent obtains its data by monitoring and record-

ing GitHub events as they occur. We used the MySQL 2014-04-02 dataset to obtain information on the projects since this paper extends our previous conference publication which used this dataset [16]. This dataset contains data on 2,399,526 repositories, 3,426,046 users, and their events — including commits, issues, pull requests and comments. We define a project as a repository and all of its forks as recommended by Kalliamvakou et al. [33].

Since the MySQL database contains only the first 256 characters of comments, we obtained all comments from GHTorrent's main MongoDB server in May 2014. The MongoDB contains the full text of all comments. These comments were downloaded and stored in a PostgreSQL database for analysis. No pre-processing was needed.

Using our Reference Coupling method, we identified 89,784 comments in the GitHub data with a cross-reference to another project[1]. There are 29,018 repositories (18,533 unique projects when forks are considered) that make a cross-reference to another project. While this is only a small portion of the total number of repositories in our dataset, this is expected since Kalliamvakou et al. [33] have found that the majority of the projects on GitHub are personal and inactive.

4.2.2. IBM Collaborative Lifecycle Management (CLM)

We collected data from the products in the IBM Rational solution for Collaborative Lifecycle Management (CLM). CLM brings together requirements management, quality management, change and configuration management, project planning and tracking on common uniform platform. CLM consists of number of products including Rational Team Concert (RTC), Rational Quality Manager (RQM), Rational DOORs Next Generation (DNG), Rational Requirement Composer (RRC), Rational Software Architect (RSA), Rational Rhapsody and Rational insight. The IBM CLM ecosystem is broken into 16 distinct projects.

[1] These cross-references and the scripts used to identify them are available in a replication package at https://doi.org/10.5281/zenodo.2555526

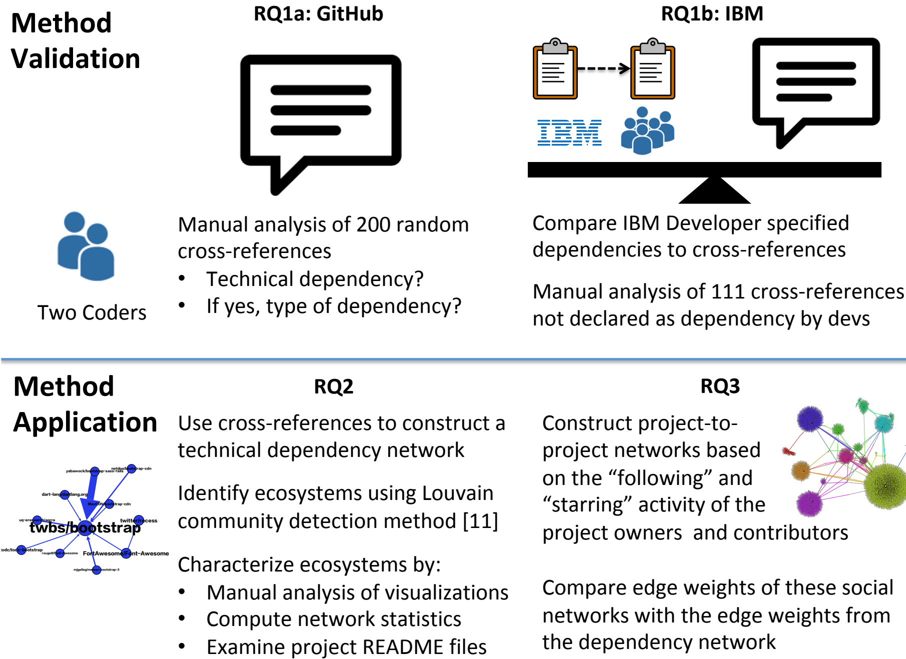

Figure 1: Summary of Research Methods

For each CLM project, we collected data on all work items created from 2005 to 2015. A work item is a task or an issue that must be attended to during development. The data was downloaded in XML format from the IBM REST API and converted into JSON and stored in PSQL tables. For each work item, we collected the metadata, history, and comments. No pre-processing was done on the data. The dataset consisted of 3,009 work items with a total of 17,708 comments. There were 635 cross-references to another project within those comments. All projects contain cross-references.

4.3. Research Methods

An overview of our research methods to answer each research question are shown in Figure 1 and described in detail in the following subsections.

4.3.1. Reference Coupling: Method Validation

To validate the Reference Coupling method, we used the method to identify cross-references between projects in both GitHub and the IBM CLM products. We automatically extracted these cross-references with pattern matching using Java Regular expressions [34]. Since we are interested only in relationships between projects, we filtered the cross-references to ignore references within the same project.

RQ1a: Does the Reference Coupling method identify inter-project technical dependencies on GitHub issue, pull request, and commit comments? If so, what are the characteristics of these dependencies?

To verify that the cross-references to other projects made in comments on GitHub identified through the Reference Coupling method are a valid conceptualization of dependencies, we examined 200 random comments which were classified as dependencies using the Reference Coupling method since they cross-referenced another project. Different types of comments may be made on different artifacts. To ensure our analysis included various types of cross-references that may occur, we ensured our randomly selected 200 comments were equally distributed across each of the following relationships: 1) commit comment cross-references another commit, 2) commit comment cross-references an issue or pull request, 3) an issue or pull request comment cross-references a commit, and 4) an issue or pull request comment cross-references another issue or pull request. Thus, there were 50 random comments selected from each of these types of cross-references.

This manual content analysis [35] was performed by two people familiar with software development practices. They classified a comment as a technical dependency if the comment described a work dependency, either direct or indirect, between the two projects. For each dependency, they also noted if the dependency was direct (between the two projects) or indirect (both projects depend on a third project).

The same two people further examined the comments that were classified as

technical dependencies to identify the types of dependencies that are identified using the Reference Coupling method. The dependencies were classified using common dependency types that can be declared in issue tracking systems:

- Duplicate: the issue/commits on the two projects are duplicates of each other. Within a project, duplicate issues would describe the same problem. Across projects, an example of a duplicate issue could be both projects have created issues to deal with a breaking API change from a shared dependency.

- Blocking: an issue/commit on one project is blocking work in the other project. For example, one project is waiting for the other project to release a promised API change before they can finalize a new feature that will be enabled by that API change.

- Resolving: an issue/commit on one project resolves an issue in the other project. For example, one project had security issues which it has inherited from a project it depends on. Once the other project fixes its security issues, the issue will be resolved in the dependent project as well.

- Affecting: an issue/commit on one project is impacted by an issue/commit on the other project. In other words, changes need to be made in the first project due to changes made in the other project. For example, a project deprecates an old API which would cause any projects using that old API to update to a more recent API.

When a dependency did not fit one of these categories, open coding was used to identify the type of dependency [36]. The cross-references that did not match one of the pre-defined dependency types were reviewed and conceptually similar comments were grouped into categories. This resulted in two new dependency categories being introduced, Leveraging and Updating, which are described in Section 5. Each coder independently did the manual analysis and coding. Then the two coders met to discuss their results and try to come to a consensus. The coders were able to come to a consensus for all items after this discussion,

Table 2: Inter-coder Reliability: Kohen's Kappa

	Initial Agreement	**Initial Cohen's Kappa**	**Final Agreement**	**Final Cohen's Kappa**
Existance of Dependency	99%	0.828	100%	1
Direct/Indirect	97%	0.807	100%	1
Affecting	88.5%	0.678	100%	1
Blocking	98.5%	0.911	100%	1
Duplicate	96.5%	0.895	100%	1
Leveraging	98.5%	0.660	100%	1
Resolving	92%	0.826	100%	1
Updating	98.5%	0.816	100%	1

resulting in 100% agreement. Table 2 shows the inter-coder reliability using Kohen's Alpha for each of the categories for the intial independent coding and the final agreed upon codes. Cohen's alpha was calculated using ReCal [37].

RQ1b: Does the Reference Coupling method identify inter-project technical dependencies on IBM work item comments? If so, what are the characteristics of these dependencies?

To validate the Reference Coupling method identifies true technical dependencies between the IBM projects, we are able to compare the dependencies identified using the Reference Coupling method to the developer declared dependencies. For each work item, the metadata contains information on dependencies between the work items captured by the developers along with the type of dependency. IBM developers can choose between 26 established dependency classifications such as Depends On, Blocks, Duplicate Of, and Resolves.[2] Of the 26 classification, only 14 are used by the IBM developers in our dataset. These 14 dependency classifications are described in Table 3. We grouped the classifications into types, since there are pairs of classifications which represent reciprocal relationships.

We analyze these developer specified dependencies to see how many were also identified by the Reference Coupling method. In addition, we also manually analysed the cases where the Reference Coupling method identified a dependency, but the IBM developers did not indicate the dependency within the work

[2] https://jazz.net/help-dev/clm/index.jsp?topic=%2Fcom.ibm.team.concert.sdk.doc%2Ftopics%2Fr_link_domains.html

Table 3: Types of Dependencies

Dependency Type	**Classification**	**Description**
Blocking	Blocks	The work item blocks work item X
	Depends on	The work item depends on work item X
Resolving	Resolves	The work item resolves work item X
	Resolved by	The work item is resolved by work item X
Duplicate	Duplicated by	The work item is duplicated by work item X
	Duplicate of	The work item is a duplicate of work item X
Affecting	Affected by Defect	A work item is affected by a defect
	Affects Plan Item	The work item impacts plan item X
Parent/Child	Parent	The work item is a parent of work item X
	Children	The work item is a child of work item X
Related	Related	The work item has a general relationship with work item X
	Related Change Request	The work item is related to a change request item
Planning	Contributes to	The work item contributes to work item X
	Tracks	The work item tracks work item X

item. We analysed these cases to determine if the Reference Coupling method identifies cases of true technical dependencies that had not been marked as such by the IBM developers. In the case that the manual analysis revealed a technical dependency, we also categorized the type of dependency using the same categories as used by the IBM developers to understand what types of dependencies Reference Coupling captures.

4.3.2. Reference Coupling: Method Application

RQ2: What ecosystems exist across GitHub-hosted projects and what is their structure?

To illustrate the applicability of our method, we constructed a network of the technical dependency relationships established through Reference Coupling as described in Section 3. The *Dependency Network* is defined as a directed graph $G_d =< V, E >$. The set of vertices, denoted by V, is all GitHub projects involved in at least one cross-reference. There are 18,533 projects in this set.

The set of edges, denoted by E, is a set of node pairs $E(V) = \{(x, y)|x, y \in V\}$. If the project represented by node x_i cross-referenced the project repre-

sented by node y_j, there is a directed edge from x_i to y_j. The weight of each edge is the count of cross-references for the pair of projects. We filtered the edges to only consider dependencies between nodes if the pair of projects have been cross-referenced two or more times to capture only the stronger dependencies.

It is important to note that this directed graph captures the direction of the cross-referencing comments and not the direction of the dependencies that those comments imply. A project could cross-reference another project because it is blocked by that project or because it is blocking that project. The nuances of the dependency direction are not captured by our method.

To identify ecosystems across projects hosted on GitHub, we used the popular Louvain community detection method [15] on the Dependency Network established through Reference Coupling. The Louvain method is a greedy optimization method that aims to partition a network into communities of densely connected nodes and optimize the modularity of the network. Modularity is defined as "the number of edges falling within [communities] minus the expected number in an equivalent network with edges placed at random [38]." High modularity scores indicate that there are dense connections within the communities but sparse connections across communities, showing that an optimal solution has been found. When high modularity scores are obtained, the communities have significant real-world meaning [15]. The Louvain method is comprised of two steps. It first optimizes modularity locally by looking for small communities. Then it aggregates the nodes in each small community and builds a new network with these aggregated nodes. It iterates on these two steps until the modularity is maximized. The Louvain method outperforms all other community detection methods in terms of both the modularity that is achieved and the computation time [15].

In our network, the identified communities represent sets of projects densely connected by technical dependencies. Since dependencies that exist between projects define the structure of an ecosystem [5], these communities represent software ecosystems.

To identify properties of the identified ecosystems, we:

- Analyzed visualizations of the Dependency Network. Visualizations of each ecosystem detected by the Louvain community detection method (as described above) were manually reviewed. We used the Gephi [39] graphing tool to create these visualizations. One of the authors inspected these visualizations of the network to visually identify patterns. The identified patterns were cross-checked by two of the other authors.
- Computed network statistics for each of the ecosystems, such as in-degree and out-degree of the nodes.
- Examined the types of projects involved in the ecosystems by reviewing the GitHub README files of the most well-connected project node in each of the ecosystems.

RQ3: Do the project owners' and contributors' social behaviours align with the technical dependencies?

To complement our investigation of technical dependencies and connectedness of projects in GitHub, we also sought to understand the social behaviour of project owners/contributors in relation to the ecosystems we identified. We studied two of GitHub's social relationships, following users and starring projects. On GitHub, users can *follow* other users to receive notification on their activity and *star* a repository to bookmark it or indicate interest in the project. To understand how the social behaviour of project owners/contributors relates to the identified ecosystems, we examine the alignment between social and technical connections between the projects.

To answer RQ3, we construct project-to-project networks based on the following and starring activity of the project owners and contributors.

Project Owners. We constructed two networks using the following and starring relationships by considering the actions of the project owners. The *Owner Stars Network*, $G_{os} =< V, E >$, and the *Owner Follows Network*, $G_{of} =< V, E >$, are both undirected graphs whose set of vertices is all GitHub projects involved in at least one cross-reference. For the Owner Follows Network, there is an edge from nodes x_i to y_j if the owner of project x_i follows the owner of

project y_j. There is an edge from x_i to y_j in the Owner Stars Network if an owner of any project in our dataset has starred both project x_i and project y_j.

Project Contributors. We constructed two additional networks using these following and starring relationships by considering the actions of the project contributors (users who have made commits on the project or are members of the project). The *Contributor Stars Network*, $G_{cs} =< V, E >$, and the *Contributor Follows Network*, $G_{cf} =< V, E >$, are also undirected graphs whose set of vertices is all GitHub projects involved in at least one cross-reference. The Contributor Follows Network has an edge from nodes x_i to y_j if a contributor of project x_i follows a contributor of project y_j. The Contributor Stars Network has an edge from x_i to y_j if a contributor to any project in our dataset has starred both project x_i and project y_j.

To compare the social connections with the technical dependencies, we compare the edge weights of these two networks with the edge weights of the Dependency Network constructed to answer RQ2. Pearson correlations were used since the data was normally distributed. The edge weights of these networks represent the following:

- Dependency Network G_d: Number of technical dependencies, measured through Reference Coupling, between the two project nodes.
- Owner Follows Network G_{of}: 0 if neither project owner follows the other, 1 if one project owner follows the other project owner, and 2 if both project owners follow each other.
- Owner Stars Network G_{os}: Number of project owners who have starred both projects.
- Contributor Follows Network G_{cf}: Number of contributors with following relationships for the pair of projects.
- Contributor Stars Network G_{cs}: Number of project contributors who have starred both projects.

5. Results

5.1. Method Validation on GitHub Data

RQ1a: Does the Reference Coupling method identify inter-project technical dependencies on GitHub issue, pull request, and commit comments? If so, what are the characteristics of these dependencies?

Of the 200 examined cross-references, we obtained 96.5% precision as 193 were found by the two manual coders to be true technical dependencies. We are unable to calculate recall of our method since we do not have a ground truth set of all dependencies.

Of these 193, 176 (91%) were direct dependencies between the two projects and 17 (9%) were indirect dependencies where the two projects both depend on a third project.

Dependency between the two projects. The most common type of dependency found was a direct technical dependency between the two projects. An example of a direct technical dependency is when an issue created in one project depends on a fix/update in another project. Another example is when a project needs to be updated based on changes made in another project.

Below we provide three examples of cross-reference comments that are indicative of direct technical dependencies between the two projects. Project names follow the pattern user/repository where user is the owner's GitHub login and repository is the name of the project repository.

Issue #449 on the sensu/sensu project describes an issue that is the result of the interaction between the sensu/sensu code and the ruby-amqp/amq-client library. The comment references a commit on the ruby-amqp/amq-client that fixes the issue.

> *"I verified that the problem is still the one referenced in ruby-amqp/amq-client#14. This fix is not merged with amq-client's '0.9.x-stable' branch. This is why I am still hitting it. The commit ruby-amqp/amq-client@60f1c59 is the fix but it resides only in the master branch."*

Issue #8 on the tsujigiri/axiom project notes that changes must be made to the code base to allow an upgrade to the latest release of the ninenines/cowboy project.

> *"Upgrade Cowboy: After Cowboy 0.6.1 Cowboy's http_req record was made opaque and can not be used directly anymore. I didn't really have the time yet to look into it, but it looks like we just need to remove all references to the record from the documentation and add directions on how to access cowboy_req:req() via the cowboy_req functions. See ninenines/cowboy#266 and ninenines/cowboy#267."*

Commit 81bbbec21c04b6392f6892f7735243387d295337 on the joyent/node project closes isaacs/node-graceful-fs issue #6, which describes a problem in the isaacs/node-graceful-fs code stemming from the use of joyent/node. GitHub allows automatic closure of issues through commit comments, even when the commit is in a different repository.[3]

> *"This fixes isaacs/node-graceful-fs#6."*

Both projects depend on a third project. We also identified some cases where the comments describe a dependency on a third project that is not cross-referenced. For example, everzet/capifony's pull request #376 cross-references composer/composer's issue #1453, but the problem stems from the use of the symfony/symfony project. After identifying the source of the problem, a new issue (#411) is created on the everzet/capifony project that identifies the changes that need to be made to the way the symfony environment is set so that the composer/composer code executes correctly.

> *"As described in #376 capifony should execute composer with the right symfony environment set. Currently, with `--no-scripts` option removed in #376, composer is always executing symfony scripts with default dev environment."*

[3] https://github.com/blog/1439-closing-issues-across-repositories

Table 4: Types of Dependencies Identified by Reference Coupling

Dependency Type	**Count (Ratio)**
Resolving	74 (38.3%)
Duplicate	43 (22.3%)
Affecting	43 (22.3%)
Blocking	18 (9.3%)
Updating	9 (4.7%)
Leveraging	6 (3.1%)

5.1.1. Dependency Categories

Most (178 of 193 or 92%) of the dependencies were able to be assigned to one of the four pre-existing dependency types (duplicate, blocking, resolving or affecting). The remaining cross-references were examined and open coding was used to generate additional dependency categories. Only two new categories were created by the two manual coders, both of which are specific to cross-project dependencies:

- Leveraging: the two projects depend on a third project and both experience the same issue due to this shared dependency. One of the projects leverages a solution to this problem that has been generated by the other project.
- Updating: one of the projects depends on the other project and is updating to a more recent version of the other project.

Table 4 shows the breakdown of how often each dependency type appeared in our data. Resolving is the most common dependency type.

Answer to RQ1a: The Reference Coupling method does identify inter-project technical dependencies on GitHub pull request, issue and commit comments; 96.5% of manually analyzed comments revealed technical dependencies between the cross-referenced projects. Of these, most are direct

Table 5: Types of Dependencies Not Identified by IBM Developers.

Dependency Type	**Count**
Duplicate	57
Related	42
Parent/Child	28
Affecting	10
Blocking	6
Planning	2
Resolving	1

dependencies between the two projects, but some are a shared dependency on a third project. We further classified these dependencies and found that the Reference Coupling method identifies a variety of dependency types. The most common type of dependency identified is Resolving: where a change in one project resolves an issue in the other project.

5.2. Validating Method on IBM Data

RQ1b: Does the Reference Coupling method identify inter-project technical dependencies on IBM work item comments? If so, what are the characteristics of these dependencies?

The Reference Coupling method identified dependencies between 2108 pairs of work items, 635 of which are inter-project dependencies. Of the inter-project dependencies, 146 (23%) were marked as technical dependencies by the IBM developers. Table 5 shows the types of dependencies that were both identified by Reference Coupling and the IBM developers. The most common types of dependencies that were identified by both the Reference Coupling method and the IBM developers are duplicates, related, and parent/child relationships.

However, there were 489 inter-project dependencies identified by the Reference Coupling method which were not marked as dependencies by the IBM

developers. To validate that our Reference Coupling method was, in fact, identifying dependencies, despite the fact that not all were marked as such within the repository, two coders familiar with software development practices reviewed a random sample of 111 of the 489 work items and manually assessed if the Reference Coupling method identified a valid technical dependency. The two coders achieved a 95% inter-coder reliability. They discussed any cases where their coding did not align and came to a concenus.

Of the 111 work items, all were found to have dependencies with other projects by the coders. The coders also categorized these 111 dependencies using the dependency types in Table 3. They also used an additional type (Unknown) to label dependencies that did not easily fit into one of the 26 pre-established dependency types.

Table 6 shows the types of dependencies that were evident among the pairs of work items. The most common dependency type that was identified by our Reference Coupling method but was not marked as a dependency by the IBM developers was 'Affected by Defect'. For example, this comment describes how a defect, which is associated to a different project, caused problems on the current project.

> *"The Validate is not defined error from comment 2 was fixed in defect 162650."*

Another common dependency type identified by Reference Coupling but not flagged as a dependency by the developers is 'Related'. For example,

> *"This is related to item 125838."*

The links between projects are not as clearly seen in the textual comments in IBM CLM, since the project associated with the work item is only available in the metadata on the work item. In these cases, it would be quite easy for developers to be unaware that a coordination need with another project exists since the fact that these work items are part of another project is not apparent.

Table 6: Types of Dependencies Not Identified by IBM Developers.

Dependency Type	**Count**
Affecting	73
Related	30
Planning	1
Blocking	1
Unknown	6

Answer to RQ1b: The Reference Coupling method does identify inter-project technical dependencies on IBM work item comments. It identifies a wide variety of dependency types. The most common type of dependency that is found with the Reference Coupling method but missed by the IBM developers is the 'Affecting' category.

5.3. Applying Reference Coupling to identify and examine GitHub ecosystems

5.3.1. Ecosystem Identification

RQ2: What ecosystems exist across GitHub-hosted projects and what is their structure?

Figure 3 shows the full Dependency Network, though for visibility we only display nodes with degree of 3 or greater. As visible on the graph, most of the nodes (10,484 of 18,533 projects or 57%) are part of the largest connected component (commonly referred to as the giant component [40]), which is the largest subgraph in which every node is connected to every other node by some path. The connected components isolated from the giant component are primarily comprised of *same owner* communities in which all nodes in the connected component are projects owned by the same GitHub user or organization. For example, the second largest connected component is comprised of 65 nodes, of which, all but two are owned by GitHub user deathcap.[4] Most of the nodes

[4]https://github.com/deathcap

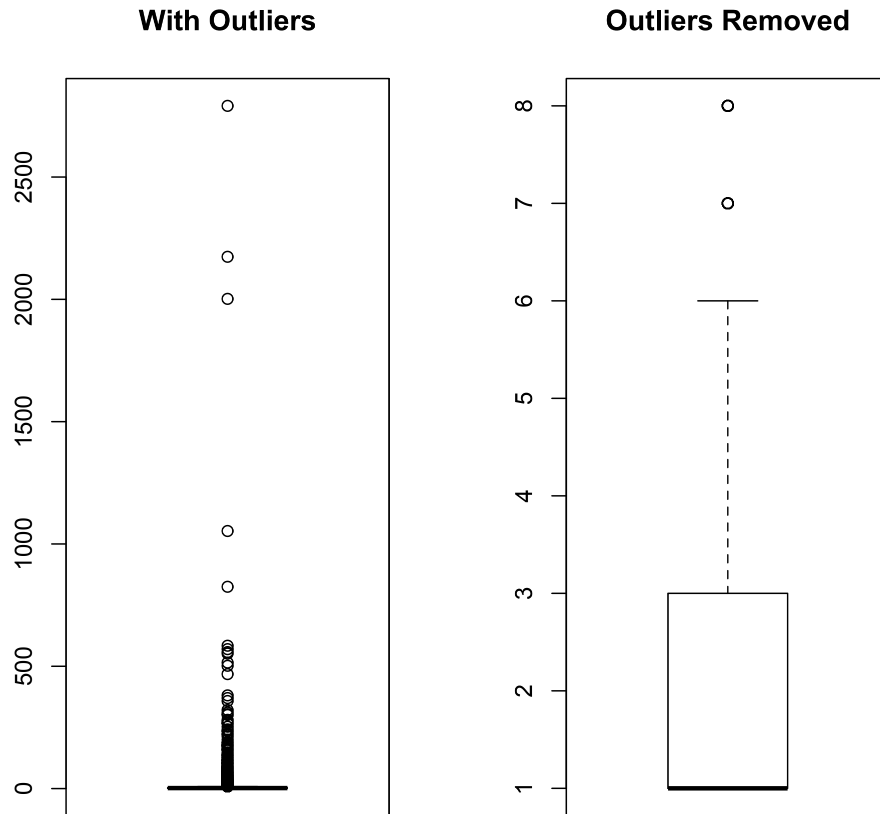

Figure 2: Number of unique cross-referenced projects for all projects that make at least one cross-reference to another project on GitHub.

isolated from the giant component are connected to only a small number of nodes. In fact, 75% of nodes not in the giant component are connected to only one other node.

Figure 2 shows boxplots for the number of unique cross-references for each project which cross-references at least one other project. These boxplots show that while some projects make cross-references to many other projects, most projects have only a small number of other projects which they cross-reference.

Since we are most interested in studying the popular GitHub ecosystems, we focus our analysis on the interconnected part of the network or the giant component. Figure 4 shows the giant component. The color of the nodes represent

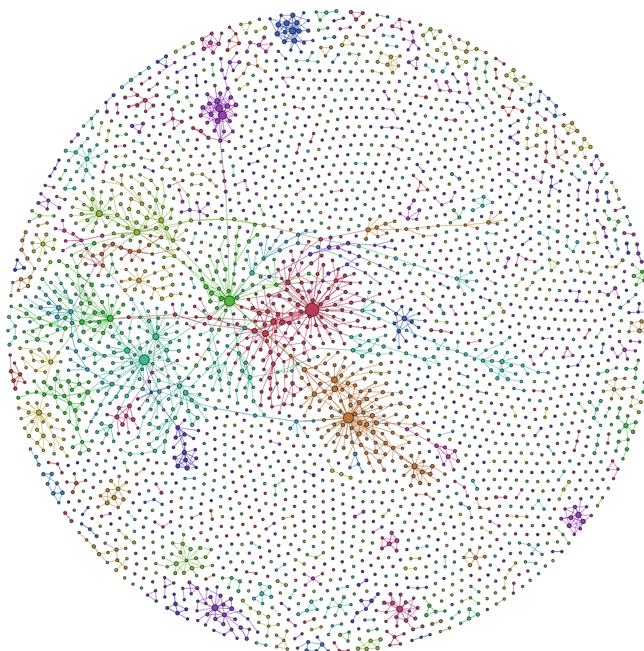

Figure 3: All GitHub projects with cross-references. The largest connected component (or giant component) is easily identified as the well-connected subgraph appearing in the center of the graph.

communities as detected by the Louvain method. We obtained a modularity score of 0.913 (out of a possible range of 0 to 1). This high modularity score indicates that the detected communities are much more tightly connected by technical dependencies than would appear in a random graph.

There were 43 ecosystems identified in this network. Nodes are sized according to their authority to display the nodes that are more prominent in each ecosystem. When a node has a high number of cross-reference relationships pointing to it, it has a high authority value [41]. Table 7 shows the most well-connected project node (highest Authority value) in each of the ecosystems.

Properties of GitHub Ecosystems

Ecosystems revolve around one central project. As depicted in Figure 4, each ecosystem appears to revolve around one main project. In Table 7, the most well-connected project node in each ecosystem is listed along with a description

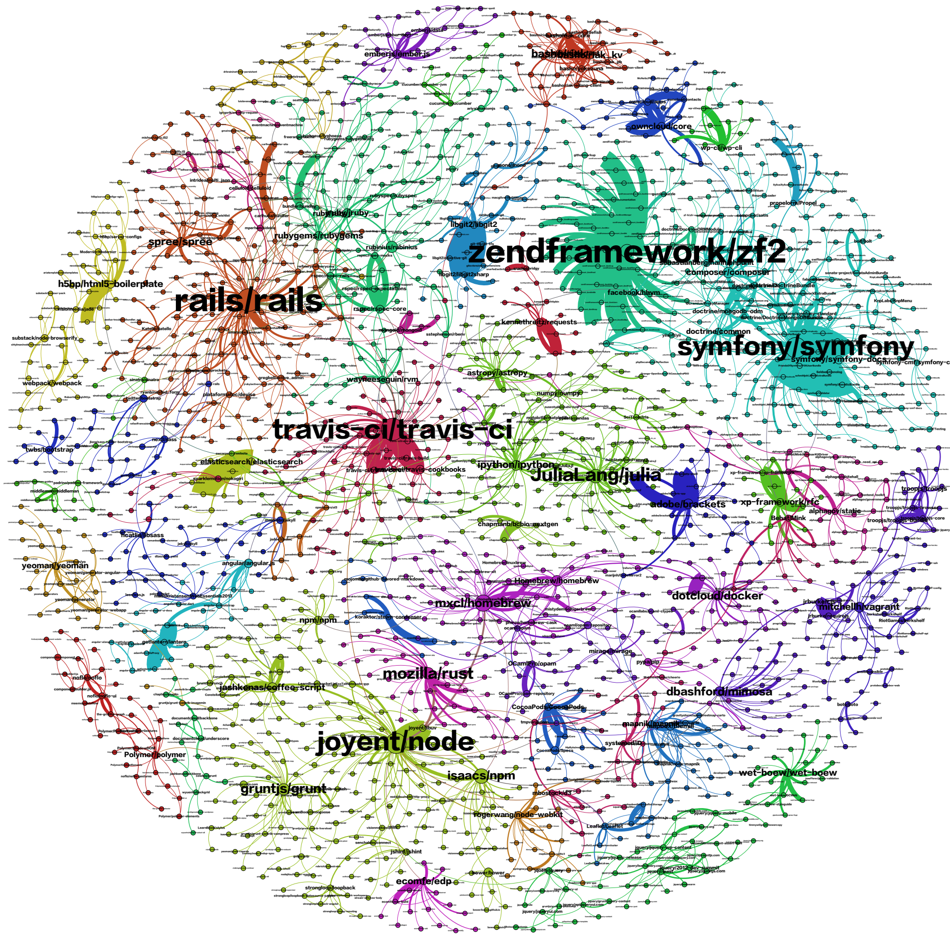

Figure 4: Ecosystems in the largest connected component of GitHub-hosted projects. Project names follow the pattern user/repository where user is the owner's GitHub login and repository is the name of the project repository.

of the project, the number of stars the project has, the size of the associated ecosystem, and the node's degree. Each of these projects has a higher in-degree than out-degree with the exception of the mxcl/homebrew project. On the other hand, low-degree project nodes are four times as likely to be dependent on another project than they are to have a project depend on them. This shows that ecosystems are being formed around a central project with the other projects in the ecosystem mostly depending on that central project. This results

Table 7: Ecosystems in GitHub. Details of the most well-connected node in each ecosystem.

Project	Description	Stars	Ecosystem Size	Degree (in,out)
joyent/node	Framework	39,373	10.08%	69 (53,16)
symfony/symfony	Framework	10,985	8.46%	93 (53,40)
rails/rails	Framework	29,744	7.92%	93 (65,28)
JuliaLang/julia	Programming Language	5,531	6.74%	51 (35,16)
rubygems/rubygems	Package Manager	1,304	6.04%	22 (14,8)
mxcl/homebrew	Package Manager	13,723	3.94%	48 (21,27)
zendframework/zf2	Framework	5,841	3.88%	72 (65,7)
travis-ci/travis-ci	Development Tool (Continuous Integration Platform)	3,693	3.50%	70 (54,16)
wet-boew/wet-boew	Framework	688	3.34%	19 (15,4)
twbs/bootstrap	Framework	41,828	3.29%	9 (9,0)
dbashford/mimosa	Development Tool (Browser development)	472	2.43%	25 (20,5)
h5bp/html5-boilerplate	Framework	31,926	2.37%	19 (15,4)
mitchellh/vagrant	Framework	9,274	2.10%	23 (15,8)
libgit2/libgit2	Library	5,161	2.05%	20 (11,9)
Behat/Mink	Development Tool (Testing)	673	1.99%	13 (9,4)
OCamlPro/opam	Package Manager	118	1.89%	9 (8,1)
basho/riak	Database	2,520	1.83%	27 (18,9)
Polymer/polymer	Library	8,787	1.83%	16 (11,5)
mapnik/mapnik	Development Tool (Toolkit for developing mapping applications)	1,003	1.78%	20 (12,8)
mozilla/rust	Programming language	5,604	1.78%	36 (29,7)
alphagov/static	Other (GOV.UK static files/resources)	67	1.73%	13 (10,3)
adobe/brackets	Development Tool (code editor)	23,921	1.46%	26 (16,10)
CocoaPods/CocoaPods	Development Tool (dependency manager)	5,711	1.46%	14 (9,5)
yeoman/yeoman	Development Tool (web development tools)	7,246	1.46%	18 (13,5)
angular/angular.js	Framework	42,950	1.40%	12 (8,4)
dotcloud/docker	Development Tool (application container engine)	14,270	1.35%	24 (19,5)
emberjs/ember.js	Framework	14,185	1.29%	20 (12,8)
owncloud/core	Other (personal cloud storage tool)	3,222	1.19%	26 (13,13)
typhoeus/typhoeus	Library	2,465	1.19%	6 (4,2)
facebook/hhvm	Other (Virtual machine)	11,506	1.08%	15 (10,5)
celluloid/celluloid	Framework	2,855	0.86%	9 (6,3)
xp-framework/rfc	Framework	0	0.86%	16 (14,2)
rogerwang/node-webkit	Framework	19,737	0.86%	16 (11,5)
ecomfe/edp	Development Tool (front-end development platform)	264	0.86%	18 (15,3)
kennethreitz/requests	Library	13,812	0.81%	13 (10,3)
documentcloud/underscore	Library	7,135	0.81%	6 (4,2)
middleman/middleman	Development Tool (website generator)	4,179	0.75%	8 (5,3)
elasticsearch/elasticsearch	Other (search and analytics tool)	10,700	0.70%	11 (11,0)
chapmanb/bcbio-nextgen	Other (RNA-seq analysis tool)	173	0.59%	10 (9,1)
wp-cli/wp-cli	Development Tool (command line interface for WordPress)	1,968	0.59%	13 (9,4)
cucumber/cucumber	Development Tool (Testing)	5,142	0.49%	7 (4,3)
jsdoc3/jsdoc	Development Tool (API documentation generator)	2,909	0.49%	6 (3,3)
propelorm/Propel	Development Tool (Object-Relational Mapping)	893	0.49%	7 (7,0)

in a star pattern. The twbs/bootstrap ego network (Figure 5) clearly depicts this pattern within the graph.

Predominant type of ecosystems is software development support. Interestingly, nearly all of the ecosystems are centered around projects whose purpose

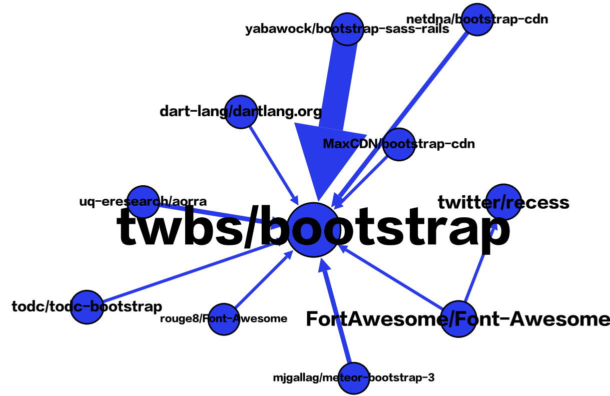

Figure 5: twbs/bootstrap Ego Network. Portraying a sample star pattern in the network.

Table 8: Ecosystem Types. Nearly all support software development.

Type	**Count**
Software Development Tool	14
Framework	13
Library	5
Package Manager	3
Programming Language	2
Database	1
Other	5

is to support software development, such as frameworks, libraries and programming languages. In fact, as shown in Table 8, of the 43 ecosystems, there are only 5 whose purpose is not to support software development. The 14 software development tools include a testing tool, a continuous integration platform, and an API documentation generator. The type of each ecosystem is also shown in Table 7.

Ecosystems are interconnected. The graph in Figure 3 shows two types of communities that occur in GitHub-hosted projects, those that are part of the largest connected component and those that are isolated from the largest con-

nected component. The majority of project nodes, 10,484 or 57%, are involved in the largest connected component, indicating that many ecosystems are connected to each other across the projects in our Dependency Network. The next biggest connected component in the graph is only 65 nodes indicating that the ecosystems that are isolated are small and have not attracted public attention.

Figure 4 displays the interconnected part of the network, and the connections between the ecosystems are apparent. As an example, Figure 6 shows the rubygems/rubygems ego network, clearly depicting its connection to the rails/rails project. This is not surprising, since the rubygems project is a package management framework for the Ruby programming language and rails/rails is a web application framework written in Ruby. There is a direct connection between the rubygems/rubygems and rails/rails nodes. In addition, there are projects, like carlhuda/bundler and airblade/paper_trail, which connect the two projects.

> **Answer to RQ2:** The Reference Coupling method can be used to identify and examine ecosystems. The predominant type of ecosystems on GitHub is software development support. Ecosystems tend to revolve around one central project and be interconnected to other ecosystems.

*5.3.2. Investigation of **Socio-Technical Alignment** within the Ecosystems*

RQ3: Do the project owners' and contributors' social behaviours align with the technical dependencies?

Project Owners: Table 9 shows strong, positive correlations between the technical dependencies and the social behaviour of the owners. Along with these strong correlations, Figure 7 shows a pronounced star pattern in the Owner Follows Network. This indicates that the project owners in an ecosystem tend to follow the owner of the central repository.

Project Contributors. As shown in Table 10, the social behaviour of project contributors does not align with the technical dependencies. This indicates

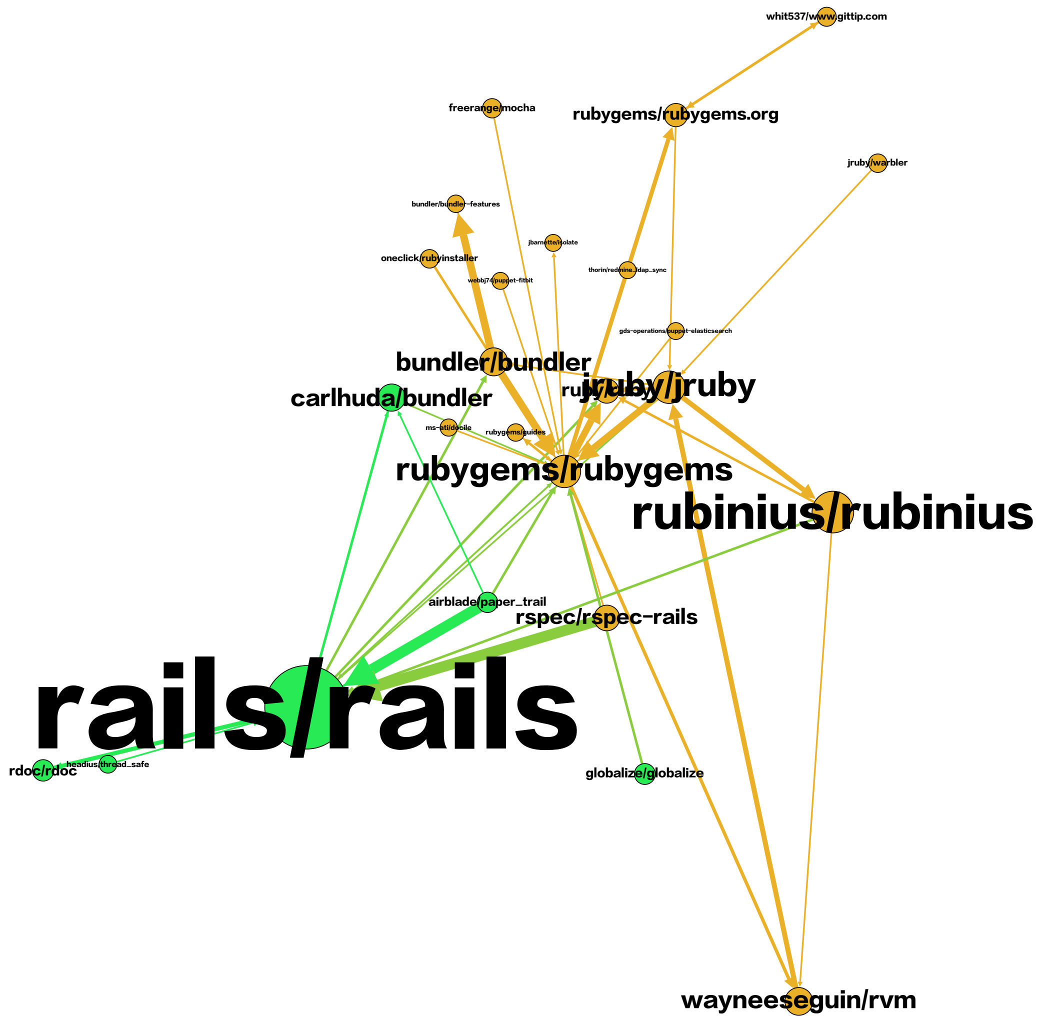

Figure 6: rubygems/rubygems Ego Network. Portraying connections between ecosystems.

Table 9: Project Owners: correlations between technical dependencies and social behaviour.

	Pearson Correlation	**p-value**
Technical Dependencies and Following	0.91	<0.001
Technical Dependencies and Stars	0.79	<0.001

that, while the project owners seem to follow the right people and are aware of the right projects based on the technical dependencies that exist in the ecosys-

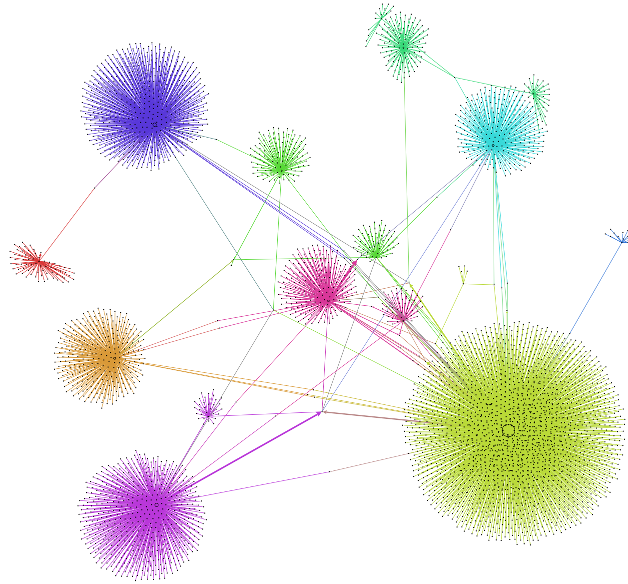

Figure 7: The Owner Follows Network, G_{of}.

Table 10: Project Contributors: correlations between technical dependencies and social behaviour.

	Pearson Correlation	**p-value**
Technical Dependencies and Following	0.0002	0.98
Technical Dependencies and Stars	0.001	0.88

tem, the social behaviour of project contributors is not aligned with project dependencies.

Figure 8 shows the Contributor Follows Network. As shown, the structure is quite different than the Dependency Network. Communities do not have one central project and the network is much more densely connected.

Answer to RQ3: The Reference Coupling method can be useful for other investigations of software ecosystems where identification of inter-project technical dependencies is needed. We investigated social behaviour in software ecosystems. We found that the project owners social behaviours

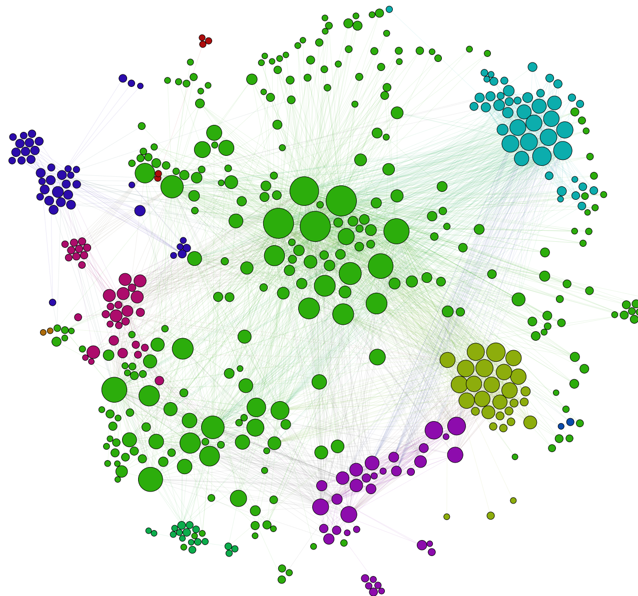

Figure 8: The Contributor Follows Network, G_{cf}.

do align with the technical dependencies, but the project contributors social behaviours *do not* align with the technical dependencies.

6. Discussion

The method we proposed in this paper, Reference Coupling, identifies cross-references to other projects within a project's ecosystem in the comments made by developers on project artifacts like issues, commits, pull requests or work items. We showed that the method outputs are a valid conceptualization of technical dependencies by analyzing the content of these cross-references and comparing the cross-references to dependency relationships identified by the development team.

Our method adds to the important, but scarce, research that leverages the social aspects of work within software ecosystems [13]. Reference Coupling detects technical dependencies that may not manifest themselves in source code by

identifying tasks, issues, pull requests, or commits that rely on another project, and, therefore, it can identify dependencies not identified by other methods. Our results show that the cross-references identify many different types of dependencies including duplicates, affecting, blocking, and related relationships.

Our method is analogous to the logical coupling method that detects dependencies within a project proposed by Gall et al. [42] except at the ecosystem level. Where logical coupling detects dependencies when artifacts have been worked on together, our method detects dependencies when issues, pull requests or commits have been worked in conjunction with another project (as evidenced through user-specified cross-references). Thus, the dependencies established through our method are those that are *logical*.

Limiting the detected dependencies to those that are logical is important when using those dependencies to identify ecosystems. Methods that detect technical dependencies between projects through analysis of code or configuration files may not be best suited for identifying software ecosystems. For example, when one project uses another project, it does not necessarily mean the two software projects are evolving together in the same environment, especially when the dependency is to an established, off-the-shelf software package. Thus, identifying all relationships that manifest in the source code or configuration files may result in dependencies that are not important for the identification of ecosystems.

For the GitHub projects, we used Reference Coupling to identify and explore ecosystems. To do this, we used a popular community detection algorithm [15] on the dependency network, which identifies clusters of nodes densely connected by technical dependencies. These detected communities represent software ecosystems. Through analysis of the resulting ecosystems found in GitHub-hosted projects, we showed that the ecosystems are centered mostly around projects that support software development through developing frameworks and toolkits. The predominant structure of the ecosystems is a star where one central project is the hub of the ecosystem.

Our method also allows for the identification of dependencies across ex-

tremely large sets of projects. For example, we ran our method on all public projects hosted on GitHub. Other methods that detect dependencies are limited to analyzing a given project or set of projects. Analyzing the dependencies of a popular project through its source code or configuration files to identify its ecosystem would not identify projects that rely on that project. We saw that most ecosystems across the GitHub-hosted projects are centered around one main project and many projects depend on that project without a reciprocal relationship. These relationships would be missed if only the dependencies of the main project were studied to identify its ecosystem. Since the ecosystems are not always well-defined, it would be impossible to know which other projects to consider for analysis. Thus, our method is better suited to identifying ecosystems since it is not limited in the number of projects it can analyze.

6.1. A Research Agenda

The ability to easily identify technical dependencies between a large set of projects opens the door for many interesting avenues of research.

Socio-technical analysis. Studies that have attempted to study how communication aligns with dependencies across projects have been limited to studying well-defined ecosystems where dependency information is publicized in some way. For example, dependencies can be made available through a project's configuration files, build files, or through publicly available dependency specifications [1, 7, 8, 9, 10, 23, 11]. Our method allows the identification of technical dependencies more broadly across projects and opens the door to continuing the study of socio-technical alignment across a larger set of projects and their stakeholders.

In this study, we found that when dependencies exist between a pair of projects, the project owners tend to be following the owner of the other project. Conway was the first to describe the possibility of an alignment between social connections and technical dependencies in software engineering projects, commonly referred to as Conway's law [43]. The transparent nature of GitHub could encourage technical connections between projects by providing an awareness of

activity across projects. An interesting future research question is understanding how and when these technical dependencies and social connections came to exist. Did the social connections exist first and result in a technical dependency or did the technical dependency exist first and result in a social connection? If the social connections existed first, what was the driver behind the creation of the technical dependency? Perhaps, the awareness of the other project, enabled through GitHub's notifications, was enough to spur a technical dependency indicating that GitHub's transparency is changing the landscape of OSS projects. These research questions could be investigated in future research.

While the project owners' social behaviours (following users and starring projects) aligned with the technical dependencies in our study, we did not witness such an alignment for all project contributors. The follower network of project contributors showed that there were no clear central projects and communities were densely connected. This is in contrast to the technical dependency network. These results align with recent research that found that the reasons behind following others extends beyond project coordination needs [44]. Future work should investigate the usefulness of following others for coordination purposes.

It is also worth studying in more detail the coordination needs of developers on OSS projects. Perhaps the mere existence of a technical dependency does not imply a coordination need, especially given the transparent environment of GitHub. Our previous work [45] begun this investigation, but coordination needs at the ecosystem level are also worthy of investigation.

Ecosystem emergence and evolution. The most prominent nodes in Figure 4 are not always the most popular projects on GitHub when considering the number of stars each project has. In fact, the two projects with the most stars, angular/angular.js and twbs/bootstrap, have significantly smaller ecosystem size and lower degree than other projects. Future work can investigate how and why ecosystems emerge and why some projects become popular without growing a large ecosystem. Such a study could include a temporal analysis of

the composition of the ecosystem and density of connections together with a temporal analysis of project history information such as number of contributors, forks, stars, etc. It would also be worth triangulating results with other information on important project events now commonly available through blogs and wikis. Such a study on the evolution of ecosystems can be a first step in understanding when and why projects accumulate an ecosystem.

Previous research has examined ecosystem growth [46], but this analysis was focused on the size of the code base (measured in lines of code) of all projects within an organization. We propose for future research to expand this view by considering the growth of an ecosystem also based on the number of projects that the ecosystem comprises of when considering technical dependencies.

Ecosystem size and strength of connections and project success. On many open source projects, volunteers are crucial to project success as they rely on volunteers to submit new features and fix bugs. As a project accumulates more projects in its ecosystem, it is also likely to increase its contributions as developers on dependent projects will be more likely to fix bugs that they encounter through their dependency. Future research could investigate this relationship to identify if the size of a project's ecosystem is a good predictor of various project health and success metrics like the number of contributions it receives or the number of forks it has.

Automatic detection of inter-project dependencies. Another avenue for future research is creating tools to make developers aware of their technical dependencies outside of their own project. It is important for developers to know who they need to coordinate with across the ecosystem and to understand how their tasks fit into the big picture. For example, when a developer on GitHub creates a cross-reference to an issue on another project, it could be useful for the developer to be made aware of other projects that have also cross-referenced that same issue. The issue may be causing problems in many projects and those project could minimize duplicated effort by being more aware of each other. Such a tool could increase awareness of coordination needs that extend outside

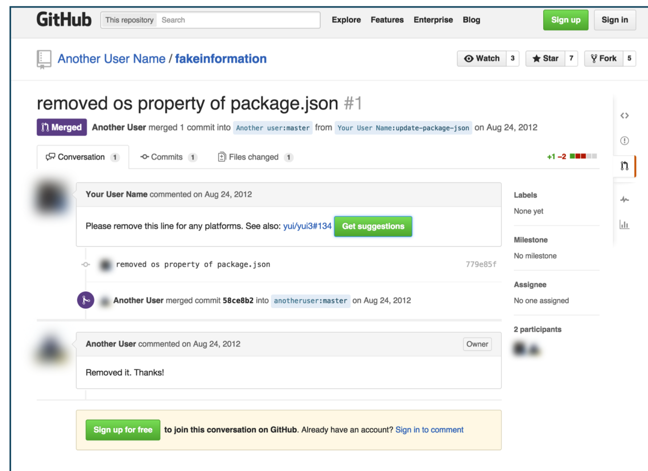

Figure 9: A proposed improvement to GitHub's Flavored Markdown, which would not only create a link to the pull request, issue or commit referenced in the comment, but would also allow users to click to see what other comments in other projects have also made the same reference.

project boundaries. For example, in Figure 9, we show a prototype we developed that improves GitHub's Flavored Markdown by allowing users to click on the "Get Suggestions" button to see a list of other comments from other projects that have made a reference to the same pull request, issue or commit. If there is a large number of other projects that share the dependency, natural language processing techniques could be used to summarize the information. This will allow developers to easily review the details from the other projects that share the same dependency. Similarly, other tools, including those used at IBM, could be modified to help developers become aware of inter-project dependencies that are automatically detected by our Reference Coupling method. Reference Coupling could be used to automatically create dependency relationships between issues or work items in a project's issue tracking system.

Automatic detection of inter-project dependencies can also be used by software engineering researchers in future unrelated studies. For example, a study of the effects of multi-tasking across multiple projects could include an analysis on the dependencies that exist between the projects to better understand the

reasons for multi-tasking or the amount of context switch that occurs. Many other software engineering research studies can benefit from an easy way to identify inter-project dependencies.

Automatic detection of ecosystems. With the ability to automatically detect individual technical dependencies between projects, it would also be useful to automatically detect and visualize ecosystems. Such a tool could help developers gain a better view of the ecosystem surrounding their project. It could also help researchers in future studies on software ecosystems.

6.2. Threats to Validity

One threat stems from our selection of the GHTorrent dataset to obtain the GitHub data. GHTorrent may not be a full copy of all GitHub data [32]. Nevertheless, it is a best-effort approach that has been widely accepted in the research community as evidenced by its inclusion as the dataset for the MSR 2013 Mining Challenge [47] and the many recent papers that utilize its data in their analysis.

The GHTorrent dataset used in our analysis is from 2014. The structure and dynamics of ecosystems could have changed since this dataset was captured. Future studies should reevaluate ecosystems using more recent datasets. Such a study can also consider ecosystem evolution since our results provide a snapshot that can be compared against. In this direction, Zhang et al. have used our Reference Coupling method to identify the GitHub projects in the Rails ecosystem [48]. They found that developers tend to make more cross-references to other projects over time.

Our manual exploration of cross reference comments illustrates a variety of types of technical dependencies found in GitHub cross-references, but these results can not be generalized. While we achieved saturation in our results, our results could be impacted by selection bias. To mitigate this, we ensured an equal number of comments for each source (commit, issue, pull request) were included in our sample. However, the types of dependencies identified seem reasonable for any software project. Future work can continue this investigation

by examining the content of cross-reference comments across a wide range of projects and code hosting environments.

Cross-references could appear more frequently when there are a higher the number of shared contributors between the two projects. We have not controlled for this since the presence of shared contributors does not diminish the existence of the dependencies. However, it could mean that dependencies on projects where there are no shared contributors are not easily found using this method. Future work should investigate this.

7. Conclusion

In this paper, we proposed a new method for detecting technical dependencies between projects, called Reference Coupling, which utilizes user-specified cross-references between projects. We validated this method on datasets from GitHub and IBM. We found that Reference Coupling identifies many dependencies which appear to be untracked by developers. The most common type of dependency that is untracked by developers but found with Reference Coupling is 'Affected by defect', which indicates that a task is impacted in some way by another defect. In these cases, especially, it could be useful for other developers to be aware of other projects that are impacted by the same defect. This could allow these projects to coordinate their efforts in creating workarounds or negotiating completion of the defect removal. The Reference Coupling method enables tools to be developed that can help developers become aware of these types of shared dependencies.

We also used our Reference Coupling method to identify ecosystems in GitHub-hosted projects by using an existing community detection algorithm to identify densely connected clusters of projects. Through an analysis of the identified ecosystems, we find that most ecosystems are centered around a single project. While small, unpopular ecosystems remain isolated, most ecosystems are interconnected. The isolated ecosystems tend to contain projects owned by the same GitHub user or organization. The popular ecosystems are mostly

centered around tools that support software development.

Our Reference Coupling method opens the door for future research in software ecosystems including studying the socio-technical relationships, evolution, health and success of ecosystems.

Acknowledgment

This work was partly funded by NSERC Canada. Thanks to Sunny Wang and Diksha Sharma for their assistance in the manual content analysis.

References

[1] M. F. Lungu, Reverse engineering software ecosystems, Ph.D. thesis, University of Lugano (2009).

[2] D. Cubranic, G. C. Murphy, J. Singer, K. S. Booth, Hipikat: A project memory for software development, Transactions on Software Engineering 31 (6) (2005) 446–465.

[3] O. Franco-Bedoya, D. Ameller, D. Costal, X. Franch, Open source software ecosystems: A systematic mapping, Information & Software Technology 91 (2017) 160–185. doi:10.1016/j.infsof.2017.07.007.
URL `https://doi.org/10.1016/j.infsof.2017.07.007`

[4] J. Ossher, S. Bajracharya, C. Lopes, Automated dependency resolution for open source software, in: Proceedings of 7th Working Conference on Mining Software Repositories, IEEE, 2010, pp. 130–140.

[5] M. Lungu, R. Robbes, M. Lanza, Recovering inter-project dependencies in software ecosystems, in: Proceedings of the International Conference on Automated Software Engineering, ACM, 2010, pp. 309–312.

[6] J. Businge, A. Serebrenik, M. van den Brand, Survival of Eclipse third-party plug-ins, in: Proceedings of 28th International Conference on Software Maintenance, IEEE, 2012, pp. 368–377.

[7] F. W. Santana, C. M. L. Werner, Towards the analysis of software projects dependencies: An exploratory visual study of software ecosystems., in: Proceedings of International Workshop on Software Ecosystems, Citeseer, 2013, pp. 7–18.

[8] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, D. M. German, Macro-level software evolution: A case study of a large software compilation, Empirical Software Engineering 14 (3) (2009) 262–285.

[9] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, S. Panichella, How the apache community upgrades dependencies: An evolutionary study, Empirical Software Engineering (2014) 1–43.

[10] D. M. German, J. M. Gonzalez-Barahona, G. Robles, A model to understand the building and running inter-dependencies of software, in: Proceedings of 14th Working Conference on Reverse Engineering, IEEE, 2007, pp. 140–149.

[11] S. Raemaekers, A. van Deursen, J. Visser, Measuring software library stability through historical version analysis, in: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, 2012, pp. 378–387.

[12] A. Mockus, Amassing and indexing a large sample of version control systems: Towards the census of public source code history, in: Proceedings of 6th Working Conference on Mining Software Repositories, IEEE, 2009, pp. 11–20.

[13] T. Mens, M. Goeminne, Analysing the evolution of social aspects of open source software ecosystems, in: S. Jansen, J. Bosch, P. R. J. Campbell, F. Ahmed (Eds.), Proceedings of the Third International Workshop on Software Ecosystems, Brussels, Belgium, June 7th, 2011, Vol. 746 of CEUR Workshop Proceedings, CEUR-WS.org, 2011, pp. 1–14.
URL http://ceur-ws.org/Vol-746/IWSECO2011-1-InvitedPaper-MensGoeminne.pdf

[14] T. Mens, B. Adams, J. Marsan, Towards an interdisciplinary, socio-technical analysis of software ecosystems health, in: Proceedings of the 16th edition of the BElgian-NEtherlands software eVOLution symposium, Antwerp, Belgium, December 4-5, 2017., 2017, pp. 7–9.
URL `http://ceur-ws.org/Vol-2047/BENEVOL_2017_paper_2.pdf`

[15] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, Journal of Statistical Mechanics: Theory and Experiment 2008 (10) (2008) P10008.

[16] K. Blincoe, F. Harrison, D. Damian, Ecosystems in GitHub and a method for ecosystem identification using reference coupling, in: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE Press, 2015, pp. 202–211.

[17] J. Bosch, From software product lines to software ecosystems, in: Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings, 2009, pp. 111–119. doi:10.1145/1753235.1753251.
URL `http://doi.acm.org/10.1145/1753235.1753251`

[18] S. Jansen, A. Finkelstein, S. Brinkkemper, A sense of community: A research agenda for software ecosystems, in: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume, 2009, pp. 187–190. doi:10.1109/ICSE-COMPANION.2009.5070978.
URL `https://doi.org/10.1109/ICSE-COMPANION.2009.5070978`

[19] K. Manikas, Supporting the evolution of research in software ecosystems: Reviewing the empirical literature, in: Software Business - 7th International Conference, ICSOB 2016, Ljubljana, Slovenia, June 13-14, 2016, Proceedings, 2016, pp. 63–78. doi:10.1007/978-3-319-40515-5_5.

[20] M. Lungu, Towards reverse engineering software ecosystems, in: 24th IEEE International Conference on Software Maintenance (ICSM 2008),

September 28 - October 4, 2008, Beijing, China, 2008, pp. 428–431. doi:10.1109/ICSM.2008.4658096.
URL `https://doi.org/10.1109/ICSM.2008.4658096`

[21] M. Goeminne, T. Mens, A framework for analysing and visualising open source software ecosystems, in: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), Antwerp, Belgium, September 20-21, 2010., 2010, pp. 42–47. doi:10.1145/1862372.1862384.

[22] D. Dhungana, I. Groher, E. Schludermann, S. Biffl, Software ecosystems vs. natural ecosystems: Learning from the ingenious mind of nature, in: Proceedings of the 4th European Conference on Software Architecture: Companion Volume, ACM, 2010, pp. 96–102.

[23] M. Syeed, K. M. Hansen, I. Hammouda, K. Manikas, Socio-technical congruence in the Ruby ecosystem, in: Proceedings of The International Symposium on Open Collaboration, ACM, 2014, p. 2.

[24] J. Kabbedijk, S. Jansen, Steering insight: An exploration of the Ruby software ecosystem, in: Software Business, Springer, 2011, pp. 44–55.

[25] S. Jansen, Measuring the health of open source software ecosystems: Beyond the scope of project health, Information & Software Technology 56 (11) (2014) 1508–1519. doi:10.1016/j.infsof.2014.04.006.

[26] M. Lungu, J. Malnati, M. Lanza, Visualizing Gnome with the small project observatory, in: M. W. Godfrey, J. Whitehead (Eds.), Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings, IEEE Computer Society, 2009, pp. 103–106. doi:10.1109/MSR.2009.5069487.
URL `https://doi.org/10.1109/MSR.2009.5069487`

[27] K. Manikas, K. M. Hansen, Software ecosystems–a systematic literature review, Journal of Systems and Software 86 (5) (2013) 1294–1306.

[28] S. Syed, S. Jansen, On clusters in open source ecosystems, in: Proceedings of International Workshop on Software Ecosystems, Citeseer, 2013, pp. 19–32.

[29] Y. Yu, G. Yin, H. Wang, T. Wang, Exploring the patterns of social behavior in GitHub, in: Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies, ACM, 2014, pp. 31–36.

[30] F. Thung, T. F. Bissyandé, D. Lo, L. Jiang, Network structure of social coding in GitHub, in: Proceedings of 17th European Conference on Software Maintenance and Reengineering, IEEE, 2013, pp. 323–326.

[31] J. MacFarlane, Commonmark spec. 2017, URL http://spec. commonmark. org/0.25.

[32] G. Gousios, D. Spinellis, GHTorrent: GitHub's data from a firehose, in: Proceedings of the 9th Working Conference on Mining Software Repositories, IEEE, 2012, pp. 12–21.

[33] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian, The promises and perils of mining GitHub, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 92–101.

[34] Oracle, Java platform standard edition 7 api specification: Package java.util.regex, `https://docs.oracle.com/javase/7/docs/api/java/util/regex/package-summary.html`, accessed 29 November 2016.

[35] K. A. Neuendorf, The Content Analysis Guidebook, Sage, 2016.

[36] J. Corbin, A. Strauss, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, Sage, 2008.

[37] D. Freelon, Recal oir: Ordinal, interval, and ratio intercoder reliability as a web service., International Journal of Internet Science 8 (1).

[38] M. E. Newman, Modularity and community structure in networks, Proceedings of the National Academy of Sciences 103 (23) (2006) 8577–8582.

[39] M. Bastian, S. Heymann, M. Jacomy, et al., Gephi: an open source software for exploring and manipulating networks., Proceedings of International AAAI Conference on Web and Social Media 8 (2009) 361–362.

[40] M. Molloy, B. Reed, Critical subgraphs of a random graph, The Electronic Journal of Combinatorics 6 (R35) (1999) 2.

[41] J. M. Kleinberg, Authoritative sources in a hyperlinked environment, Journal of the ACM 46 (5) (1999) 604632.

[42] H. Gall, K. Hajek, M. Jazayeri, Detection of logical coupling based on product release history, in: Proceedings of International Conference on Software Maintenance, IEEE, 1998, pp. 190–198.

[43] M. E. Conway, How do committees invent, Datamation 14 (4) (1968) 28–31.

[44] K. Blincoe, D. Damian, Implicit coordination supported by GitHub: A case study of the rails oss project, in: Open Source Systems: Adoption and Impact, Springer, 2015, pp. 35–44.

[45] K. Blincoe, G. Valetto, D. Damian, Do all task dependencies require coordination? the role of task properties in identifying critical coordination needs in software projects, in: Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, 2013, pp. 213–223.

[46] J. W. Paulson, G. Succi, A. Eberlein, An empirical study of open-source and closed-source software products, IEEE Transactions on Software Engineering 30 (4) (2004) 246–256. doi:10.1109/TSE.2004.1274044.

[47] G. Gousios, The GHTorent dataset and tool suite, in: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, 2013, pp. 233–236.

[48] Y. Zhang, Y. Yu, H. Wang, B. Vasilescu, V. Filkov, Within-ecosystem issue linking: a large-scale study of rails, in: Proceedings of the 7th International Workshop on Software Mining, ACM, 2018, pp. 12–19.