

# Software Developers Using Signals in Transparent Environments

Jason Tsay

CMU-ISR-17-102

April 2017

Institute for Software Research  
School of Computer Science  
Carnegie Mellon University  
Pittsburgh, PA 15213

**Thesis Committee:**

James D. Herbsleb, Co-chair

Laura Dabbish, Co-chair

Claire Le Goues

André van der Hoek, U.C. Irvine

*Submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy in Software Engineering.*

Copyright © 2017 Jason Tsay

This research was sponsored by the Center for the Future of Work at Carnegie Mellon University's Heinz College and by the National Science Foundation, under NSF awards IIS-1111750, ACI-1322278, IIS-1633083, and IIS-1546393. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Center for the Future of Work or the National Science Foundation

**Keywords:** Transparency, GitHub, transparent development environments, signaling theory, computer-support cooperative work, open source software, activity traces

## Abstract

One of the main challenges that modern software developers face is the coordination of dependent agents such as software projects and other developers. Transparent development environments that make low-level software development activities visible hold much promise for assisting developers in making coordination decisions. However, the wealth of information that transparent environments provide is potentially overwhelming when developers are wading through information from potentially millions of developers and millions of software repositories when making decisions around tasks that require coordination with projects or other developers. Overcoming the risk of overload and better assisting developers in these environments requires a principled understanding of what exactly developers need to know about dependencies to make their decisions.

My approach to a principled understanding of how developers use information in transparent environments is to model the process using signaling theory as a theoretical lens. Developers making key coordination decisions often must determine qualities about projects and other developers that are not directly observable. Developers infer these unobservable qualities through interpreting information in their environment as signals and use this judgment about the project or developer to inform their decision. In contrast to current software engineering literature which focuses on technical coordination between modules or within projects such as modularity or task assignment mechanisms, this work aims to understand how developers use signals to information coordination decisions with dependencies such as other projects or developers. Through this understanding of the signaling process, I can create improved signals that more accurately represent desired unobservable qualities.

My dissertation work examines the qualities and signals that developers use to inform specific coordination tasks through a series of three empirical studies. The specific key coordination tasks studied are evaluating code contributions, discussing problems around contributions, and evaluating projects. My results suggest that when project managers evaluate code contributions, they prefer social signals over technical signals. When project managers discuss contributions, I found that they attend to political signals regarding influence from stakeholders to prioritize which problems need solutions. I found that developers evaluating projects tend to use signals that are related to how the core team works and the potential utility a project provides. In a fourth study, using signaling theory and findings from the qualities and signals that developers use to evaluate projects, I create and evaluate an improved signal called “supportiveness” for community support in projects. I compare this signal against the current signal that developers use, stars count, and find evidence suggesting that my designed signal is more robust and is a stronger indicator of

support. The findings of these studies inform the design of tools and environments that assist developers in coordination tasks through suggestions of what signals to show and potentially improving existing signals. My thesis as a whole also suggests opportunities for exploring useful signals for other coordination tasks or even in different transparent environments.

# Acknowledgments

I first would like to thank my advisors Jim Herbsleb and Laura Dabbish for everything they have done for me during my time in the PhD program. I greatly value all of the many, many discussions we have had over the years as well as all of the support, advice, and feedback. I consider myself extremely fortunate to have had Jim and Laura as my advisors.

I also thank my committee members Claire Le Goues and André van der Hoek for their always insightful feedback and support. I've greatly enjoyed all of our discussions over the years.

As somebody I have worked closely with and also had many discussions with, I also would like to thank Colleen Stuart. Some of my fondest memories of working on research projects are the long discussions about GitHub between the four of us.

I also want to thank all of the PhD students and postdocs that have taken the time to help me, whether it's through writing groups or helping me practice for talks: Ben Towne, Ivan Ruchkin, Hanan Hibshi, Vishal Dwivedi, Arun Kalyanasundaram, Erik Trainer, Chris Bogart, Anna Filippova, Amber McConahy, and many others. Thank you all for reading my (sometimes very) rough drafts and/or sitting through talks (even multiple times!).

I would also like to thank and remember Chalalai "Jib" Chaihirunkarn. I wish that I could have helped her practice for a talk like she has done for me.

I want to thank all of the students, postdocs, and faculty in the Institute for Software Research and the Software Engineering PhD program. All of the people who have supported me through hallway conversations, SSSG questions and feedback, and so on are too numerous to name. In particular though, I would like to thank Josh Sunshine for his invaluable advice in progressing through the program and my officemates who I've had many long discussions with about life, job hunting, research: Wei Wei, Ashwini Rao, and Marat Valiev.

I also have had the privilege of working with people across various research groups

such as SCALE, Jim's research group, and Laura's CoEx Lab: Linda Argote, David Redmiles, Anita Sarma, Jonathan Kush, Jenn Marlow, Thomas LaToza, Mary Nguyen, Min Lee, Tatiana Vlahovic, Joseph Seering, Fannie Liu, and many, many others.

I would also like to thank all of the people I have met through organizing CMU eSports and CMUken. Managing the fighting game community in Pittsburgh has allowed me to make great friends and has given me many excuses to travel and make even more friends. I would like to thank of all my friends in Pittsburgh and the various cities, coasts, and countries for all of the good memories that we've shared.

Finally, I'd like to thank my family for always supporting me through everything. Coming from a family of engineers, I have my parents to thank for developing an interest in computers and software at an early age.

My time in the PhD program have been some of the best years of my life, thanks to all of the relationships and social links that I have been lucky enough to establish.

# Contents

<b>1</b>	<b>Developers Using Signals in Transparent Development Environments</b>	<b>1</b>
1.1	Transparency Enables Decentralized Coordination . . . . .	2
1.2	Challenges in Using and Understanding Transparency . . . . .	3
1.3	Approach: Signaling Theory to Model Information Usage in Transparent Environments . . . . .	5
1.4	Thesis: Software Developers Using Signals in Transparent Environments	6
1.5	Contributions . . . . .	6
1.6	Outline . . . . .	7
<b>2</b>	<b>Related Work</b>	<b>9</b>
2.1	Online Communities . . . . .	9
2.2	Open Source Software . . . . .	11
2.3	Transparent Development Environments . . . . .	13
2.4	Signaling Theory . . . . .	15
2.5	Summary . . . . .	17
<b>3</b>	<b>Research Context: GitHub</b>	<b>19</b>
3.1	Transparency Features in GitHub . . . . .	20
3.2	Contributions in GitHub: Pull Requests, Issues, Comments . . . . .	21
<b>4</b>	<b>Signals for Evaluating Contributions in GitHub</b>	<b>27</b>
4.1	Contributions in Transparent Development Environments . . . . .	30

4.1.1	Contributions in Open Source Software . . . . .	30
4.1.2	Contribution in Online Communities . . . . .	31
4.1.3	Evaluating Contributions in Transparent Development Environments	31
4.1.4	Hypotheses Development . . . . .	32
4.2	Methods . . . . .	35
4.2.1	Pull Request Selection . . . . .	35
4.2.2	Signal Measures . . . . .	36
4.2.3	Analysis . . . . .	39
4.3	Results . . . . .	40
4.3.1	Pull Request-Level Measures . . . . .	40
4.3.2	User-Level Measures . . . . .	42
4.3.3	Repository-Level Measures . . . . .	43
4.4	Discussion . . . . .	43
4.4.1	Technical Norms and Social Connection . . . . .	43
4.4.2	Decision-Making and Highly Discussed Contributions . . . . .	45
4.4.3	Audience Pressures . . . . .	47
4.4.4	Limitations . . . . .	49
4.5	Conclusion . . . . .	49
<b>5</b>	<b>Negotiating Contributions through Discussion in GitHub</b>	<b>51</b>
5.1	Contribution and Discussion in Online Work . . . . .	54
5.1.1	Discussions around Contributions in Online Communities . . . . .	54
5.1.2	Discussions around Contributions in Open Source Software . . . . .	55
5.1.3	Social Signals in Transparent Development Environments . . . . .	56
5.1.4	Development of Research Questions . . . . .	57
5.2	Method . . . . .	58
5.2.1	Data Collection . . . . .	58
5.2.2	Data Analysis . . . . .	60
5.3	Results . . . . .	60

5.3.1	Issues Raised Around Code Contributions . . . . .	60
5.3.2	Methods of Influencing the Decision Process for Code Contributions	64
5.3.3	Outcomes for Proposed Code Contributions . . . . .	67
5.3.4	Submitter’s Prior Experience . . . . .	69
5.4	Discussion . . . . .	70
5.4.1	Stakeholders Influencing the Outcome . . . . .	70
5.4.2	Power Relationships in Evaluating Contributions . . . . .	72
5.4.3	Developing Software Requirements through Discussion . . . . .	74
5.5	Conclusion . . . . .	75
<b>6</b>	<b>Signals for Evaluating Projects for Use or Contribution</b>	<b>77</b>
6.1	Open Source Software and Signaling . . . . .	80
6.1.1	Awareness and Open Source Software . . . . .	80
6.1.2	Signal Usage in Transparent Development Environments . . . . .	80
6.1.3	Signaling Theory as a Theoretical Lens . . . . .	81
6.1.4	Research Questions Development . . . . .	82
6.2	Qualitative Exploratory Interview Study . . . . .	83
6.2.1	Methods . . . . .	83
6.2.2	Results . . . . .	84
6.3	Quantitative Validation Analysis . . . . .	92
6.3.1	Methods . . . . .	93
6.3.2	Results . . . . .	98
6.4	Discussion . . . . .	100
6.4.1	Evaluating Projects and Signal Fit . . . . .	100
6.4.2	Implications for Transparent Development Environments . . . . .	103
6.4.3	Limitations . . . . .	104
6.5	Conclusion . . . . .	105
<b>7</b>	<b>Evaluating and Creating Signals for Community Support in Software Projects</b>	<b>107</b>

7.1	Community Support in Open Source . . . . .	110
7.1.1	Community Involvement in Open Source Software . . . . .	110
7.1.2	Community Involvement in Online Communities . . . . .	111
7.1.3	Developers Using Information in Transparent Environments . . . . .	112
7.2	Preliminary Interview Study . . . . .	112
7.2.1	Interview Methodology . . . . .	112
7.2.2	Community Support in Open Source Software Projects . . . . .	113
7.2.3	Research Question Development . . . . .	114
7.3	Community Support Modeling . . . . .	116
7.3.1	Dataset Collection . . . . .	116
7.3.2	Measure Development . . . . .	117
7.3.3	Analysis . . . . .	120
7.4	Results . . . . .	120
7.4.1	Model Fit . . . . .	120
7.4.2	Community Support Predictors . . . . .	121
7.4.3	Project State Measures . . . . .	122
7.5	Discussion . . . . .	123
7.5.1	Predicting Community Support . . . . .	123
7.5.2	Project State Affecting Community Support . . . . .	124
7.5.3	Implications for Software Engineering . . . . .	125
7.5.4	Limitations . . . . .	126
7.6	Conclusion . . . . .	127
<b>8</b>	<b>Future Work</b>	<b>129</b>
8.1	Methodology for Eliciting and Improving Signals . . . . .	129
8.2	Designing Improved Signals for Software Developers . . . . .	131
8.3	Designing Developer Tools and Transparent Development Environments . . . . .	133
8.4	Dynamic Signals for Tasks, Projects, and Users . . . . .	134

<b>9 Conclusions</b>	<b>137</b>
<b>A Pull Request Extended Discussion Sample</b>	<b>141</b>
<b>Glossary</b>	<b>143</b>
<b>Bibliography</b>	<b>145</b>



# List of Figures

2.1	Diagram of Signaling Process . . . . .	16
3.1	Example of GitHub Project Page with Watch, Star, and Fork Buttons Highlighted . . . . .	21
3.2	Example of GitHub User Profile Page . . . . .	22
3.3	Example of GitHub Activity Feed . . . . .	23
3.4	Example of GitHub Pull Request . . . . .	24
3.5	Example of GitHub Issue . . . . .	25
3.6	Example of Inline Code Comment in GitHub . . . . .	25
4.1	Interaction Plot of Test Inclusion and Contribution Discussion . . . . .	47
4.2	Interaction Plot of Social Distance and Contribution Discussion . . . . .	47
4.3	Interaction Plot of Prior Interaction and Contribution Discussion (prior interaction values are standardized) . . . . .	48



# List of Tables

1.1	Overview of Dissertation Studies . . . . .	7
4.1	Descriptives of Contribution Evaluation Signal Measures (Pre-transformation)	36
4.2	Multi-level Mixed Effects Logistic Model for Pull Request Acceptance . .	41
5.1	Description of Pull Request Sample . . . . .	59
5.2	Distributions of Pull Request Sample . . . . .	59
6.1	Interview Study Summary . . . . .	86
6.2	Unobservable Quality Types Summary . . . . .	86
6.3	Summary of Signal Types and Characteristics . . . . .	92
6.4	Descriptives of Project Evaluation Measures (Pre-transformation) . . . .	94
6.5	Quantitative Hypotheses for Project Evaluation Signals . . . . .	97
6.6	Usage Model Analysis Summary . . . . .	99
6.7	Contribution Model Analysis Summary . . . . .	101
7.1	Descriptives of Community Support Measures (Pre-transformation) (JavaScript Dataset) . . . . .	118
7.2	Comparison of Fit for Community Support Hierarchical Models (JavaScript Dataset)* . . . . .	121
7.3	Summary of Predictive Models for Community Support* . . . . .	123
8.1	Overview of Signaling Implications From Dissertation Studies . . . . .	129



# Chapter 1

## Developers Using Signals in Transparent Development Environments

One of the main challenges of modern software development is coordination. In contrast to traditional software development, the widespread availability of reusable open source libraries and frameworks greatly increases the efficiency of creating software systems and the quality of the resulting systems [Ajila and Wu, 2007]. Rather than implementing additional features in-house or through approved vendors, it is possible to simply establish a dependency with the appropriate external software project and integrate its functionality into a given software system. While this reuse is efficient, these external software projects are usually managed by their own independent developers. Establishing a dependency then involves giving up some control to this external project. For example, if a bug related to the dependency is found, it usually is not possible to directly fix the bug in the external project. The software developer in this case then needs to make a decision: create a (often brittle) workaround in their software system to account for this bug or coordinate with the external project and its developers to fix the bug [Shah, 2006]. For many developers, software development is not simply writing code but managing these dependencies. This management of dependencies between activities is the challenge of coordination [Malone and Crowston, 1994]. These coordination challenges extend beyond deciding to incorporate external software projects. Dependencies that developers must manage range from software projects to external code contributions to developers, designers, and other people. Developers must coordinate with these dependencies during various software development tasks, such as deciding whether to accept external code contributions, to recruit developers, how to handle bugs that are reported, and many others. While developers have long faced these coordination challenges, the nature of coordination evolves along with changes

to how software is developed.

The coordination challenges of software development are increasingly decentralized. For traditional software development, coordination decisions were often alleviated by central management. For the example of external software dependencies, in a centrally managed system, the decisions of which projects to use are prescribed top-down through road-mapping. As the pace of software development increases, these dependencies also develop and change independently, often outpacing what central management can control. Without central control, software developers and teams must increasingly make their coordination decisions independently. To make these decisions effectively, developers require information such as the potential dependencies available and the tradeoffs for interacting with certain dependencies. Returning to the example of a developer deciding whether to create a local workaround or coordinate with an external project to fix a bug, information about the external project and how its core team works is crucial towards informing this decision. For example, if the external project does not respond to outside code contributions, then any time spent attempting to coordinate with this project may be wasted. This information about dependencies is often difficult to directly observe, even for open source projects. Even if the codebase is freely available for an open source project, it may be difficult to observe useful qualities of the project such as their openness to outside contributions, ease-of-use, and maturity. However, the advent of transparency may enable new sources of information for developers making decentralized coordination decisions.

## 1.1 Transparency Enables Decentralized Coordination

With the advent of modern development environments, the resulting wealth of information about software projects and developers enables transparency of development work. Transparency is the “accurate observability of an organization’s low-level activities, routines, behaviors, output, and performance” [Bernstein, 2012]. In the case of transparent development environments, the “organization” may be a software project or a particular developer. Modern development environments make visible low-level development activities such as code commits and bug reports. This quality is present in open source software (OSS) and increasingly in engineering-focused companies such as Google where the development activity of all projects is visible to most developers. Modern transparent environments such as GitHub<sup>1</sup>, in contrast to traditional open source software projects, increase the scale of what activities are visible. Rather than only monitoring all the activity in a single project, transparent development environments make visible development ac-

<sup>1</sup><https://github.com/>

tivity of entire communities of software projects, the activities of the developers who work on them, the projects these developers work on, and so on [Dabbish et al., 2012]. For example, along with viewing all of the commits and commit authors for a project, transparent environments also make visible the other projects these developers have worked on.

Transparency enables developers to make independent, informed coordination decisions with dependencies through observing their activities. By freely observing how projects or developers work, software developers have the opportunity to enhance their own work by using this information to decide how they should coordinate with said projects or developers. For example, when a project manager receives a contribution from a newcomer, they choose whether to interact with the developer and their code contribution. In a transparent environment, project managers are able to view all of the prior projects that a newcomer might have participated in and evaluate them as signals of developer skill before deciding whether to accept a contribution or even potentially recruit the developer. Conversely, newcomers looking for open source projects to join may need to decide which projects are worth their time. In a transparent environment, these new developers might investigate what prior contribution attempts look like and how they have fared, as signals of openness and project norms.

## **1.2 Challenges in Using and Understanding Transparency**

The visibility of development activity that enables transparency also comes with the risk of overloading developers with information. The social media systems these transparent environments are based on are also associated with an overwhelming amount of information [Singer et al., 2014]. Many developers in these systems find it challenging to effectively consume the sheer amount of content, developing ad-hoc strategies to filter and skim. However, whereas missing content on Twitter often has little consequence, missing important work-related information in transparent environments may have negative impacts on productivity or project management. Especially with how visible work is, missing important cues in the environment such as submitted contributions or important discussions potentially creates negative impressions of the project or developer [Dabbish et al., 2012]. This is especially true at “web scale”, where developers are potentially wading through information from potentially millions of developers and millions of software repositories when making work decisions. Even if information is overwhelmingly available through these systems about specific projects or developers, they often do not directly answer questions developers have about projects or other developers. For example, a developer evaluating a project as a potential dependency may want to know how mature the

project is. Though a project's maturity is not directly visible, a developer might look at various signals the project broadcasts such as recent commit activity, number of versions, and the community size to infer the project's maturity indirectly. Due to the cost of navigating the firehose of development-related information in transparent environments and then additionally interpreting and making inferences, developers may be forced to develop ineffective ad-hoc techniques such as constantly checking feeds [Dabbish et al., 2012] or even refusing to participate in social aspects of development.

Overcoming risks such as information overload requires a principled understanding of how exactly information is used by developers in transparent environments. Though transparent environments generate an overwhelming amount of development-related information, there is an opportunity to target what is most informative for developers in making independent decisions. By exploring how developers use information in these environments, we further our understanding of what pieces of information are most informative for making coordination decisions. By doing so, I have the opportunity to design tools and practices and even new transparent environments that assist developers in making specific coordination decisions rather than simply making all work activity visible. Although we know much about the coordination challenges software developers face [Cataldo et al., 2006, Malone and Crowston, 1994], much of current software engineering literature focuses on either technical coordination between software modules or coordination within projects such as modularity or task assignment mechanisms [Crowston et al., 2008]. We know comparatively little about how developers coordinate with other projects or developers and even less about the decision process behind these coordination decisions. What are the coordination decisions that developers must make independently in these environments? How does the information that transparency provides about these potential dependencies assist in these coordination decisions? What do developers need to know in order to make these decisions? What are developers currently using in their environment to inform themselves? Most importantly, can we use theory to derive relationships between developers' coordination needs and the information that transparency makes visible? Rather than design for specific tasks or developers, certain types of information might be more generally useful for developers making coordination decisions. For example, signaling theory has a concept of "honest signals" [Connelly et al., 2010] that are resistant to deception which may be useful for developers wishing to obtain accurate information from their environment.

## 1.3 Approach: Signaling Theory to Model Information Usage in Transparent Environments

My approach to a principled understanding of how developers use information to coordinate in this new environment is to apply signaling theory as a theoretical lens and to use mixed empirical methods to explore and validate theories.

*Signals* are observable pieces of information that are used by a receiver to infer an unobservable quality of the signaler [Donath, 2005]. Signaling theory is used in economics to describe how parties behave in an environment with information asymmetry. Insiders (signalers) have information that they are privileged to (unobservable quality) and may emit signals indicating this information to outsiders (receivers). In Spence [1973]’s classic example, potential employees indicate their unobservable ability level to employers through the signal of educational credentials.

Signaling theory is a useful theoretical lens for coordination in transparent environments as it allows for describing the relationships between specific pieces of information (signals) about dependencies (signalers) and the unobservable qualities developers (receivers) infer about these dependencies to make decisions. “Dependencies” here refers to agents that developers may coordinate with, such as software projects and other developers. For example, a developer with commits to high-status projects may signal coding expertise [Dabbish et al., 2012] or a project with quick responses to pull requests may signal a culture open to outside contributions. By understanding what signals are used and the qualities developers wish to infer, there is the opportunity to assist developers in a principled manner by targeting specific qualities that inform coordination decisions. For example, if the quality of a project’s maturity is crucial towards informing the decision of whether to use this software project, then what signals most efficiently or accurately indicate a project’s maturity?

Signaling theory as a lens also provides constructs to describe aspects of signals such as cost, honesty, and fit [Connelly et al., 2010]. These constructs help explain why certain signals may be more useful than others. For example, certifications are *costly* signals that indicate the quality of a manufacturer’s process. The high cost of producing such a signal makes it useful because lower-quality manufacturers would need to implement many more costly changes compared to high-quality manufacturers in order to obtain the certification. Understanding both qualities that developers wish to know about dependencies and characteristics that make signals more or less useful allows for principled improvements to how developers use information in these environments.

## 1.4 Thesis: Software Developers Using Signals in Transparent Environments

### Thesis Statement:

*Signaling theory is a useful lens to understand how developers use information made visible in transparent development environments. Developers making key coordination decisions often must determine qualities about projects and other developers that are not directly observable. Developers infer these unobservable qualities through interpreting information in their environment as signals. Through this understanding of the signaling process, I can create improved signals that more accurately represent desired unobservable qualities.*

To support this claim, my dissertation work includes a series of empirical studies that aim to understand how developers in transparent development environments use signals. In these studies, I identify key decisions where developers use signals to inform their decision-making by inferring useful qualities about projects or other developers. These studies describe in detail this signal usage process of: receiving a piece of information, developing inferences from the information, using these inferences to estimate an unobservable quality about the project or developer that the developer actually wants to know, and using this judgment about the project or developer to inform their decision. The specific key decisions studied are evaluating code contributions [Tsay et al., 2014a], discussing problems around contributions [Tsay et al., 2014b], and evaluating projects. Using findings from the qualities and signals that developers use to evaluate projects, I create and evaluate an improved signal for community support in projects that is grounded in signaling theory. I compare this signal against the current signal that developers use, stars count, in a longitudinal study of community support in projects.

## 1.5 Contributions

- The application of signaling theory to software engineering and transparent development environments to model how developers use information in their environment to make coordination decisions.
  - Novel application of signaling theory to unintentional signals derived from development activity.

- Three studies of the signals developers use in transparent environments for three key tasks:
  - Evaluating code contributions (Chapter 4)
  - Discussing problems around contributions (Chapter 5)
  - Evaluating projects for usage and contribution (Chapter 6)
- A study of community support in projects (Chapter 7)
- Development and evaluation of an improved signal for community support based on signaling theory (Chapter 7)

## 1.6 Outline

Chapter 2 grounds my work by discussing prior literature in open source software, online communities, and transparent development environments. In this chapter, I also give a brief overview of signaling theory and its concepts and constructs. Chapter 3 describes the popular transparent environment of GitHub which is the research setting for the studies described in this thesis. Chapter 4 is a quantitative study that examines the social and technical signals that developers use when evaluating code contributions. Chapter 5 is a qualitative study that explores problem-solving discussions around contributions. Chapter 6 is a mixed-methods study of the signals and qualities developers use when evaluating projects when deciding to use or contribute to the project. Chapter 7 is a quantitative study of community support in projects. In this chapter, I use signaling theory to design an improved signal for support which I compare against signals developers currently use. Table 1.1 provides a summary of the studies. Finally, Chapter 8 discusses potential future work, and Chapter 9 concludes.

Table 1.1: Overview of Dissertation Studies

Coordination Task	Chapter	Study Type
Evaluating Contributions	4	Quantitative
Negotiating Contributions	5	Qualitative
Evaluating Projects	6	Mixed
Inferring Community Support	7	Mixed + Improved Signal



# Chapter 2

## Related Work

This dissertation is grounded in prior literature around how participants of online communities such as open source software projects coordinate. The coordination challenges that online communities and software projects face when managing members and contributions are well-studied and inform explorations of coordination challenges in transparent development environments such as the studies described in the following chapters.

Transparent development environments reveal important information about projects and users that developers can use to inform coordination decisions. Prior literature of how developers coordinate in these environments suggests that developers use transparency information to make useful inferences about potential dependencies. The studies described in the following chapters contribute directly to this literature by furthering our understanding of how developers coordinate in these environments for specific coordination tasks.

My approach to understanding how developers use information to coordinate in transparent environments makes use of signaling theory as a theoretical lens. Signaling theory and its constructs are useful in describing and reasoning about the relationship between specific pieces of information about projects or users and the inferences that developers may derive from such information.

### 2.1 Online Communities

To survive and thrive, online communities regularly perform key coordination tasks of managing community members and contributions. Successful online communities rely on members contributing their unique resources to the community, such as users uploading

videos on YouTube or posting pictures or comments on Reddit.

Kraut and Resnick [2012] analyzed the coordination challenges that online communities face regarding membership and contributions. They developed a number of design claims that assist online communities in addressing these challenges. Kraut and Resnick suggest that online communities addressing the coordination task of encouraging contributions have a number of methods to motivate members: matching members to contributions needed, making requests to members, using intrinsic and extrinsic motivators, and grouping members together. They review evidence showing that constant feedback to members, whether it be character levels in World of Warcraft or community comments in YouTube, motivates members to create more contributions. Similarly, combining contributions with social contact also encourages further contributions, for example, the GNOME software project encouraging socialization through forums and get-together conferences. They claim using the collective effort model [Karau and Williams, 1993] that commitment to an online community increases willingness to contribute. Kraut and Resnick also claim that encouraging commitment is a combination of affective commitment (attachment to the group or project), normative commitment (obligations to the community), and needs-based commitment. Lastly, Kraut and Resnick also claim that when dealing with newcomers, successful online communities must meet a number of challenges: attracting newcomers, selecting among the newcomers, retaining newcomers, socializing newcomers, and protecting existing members from potential problems newcomers may bring. When evaluating newcomers, communities will often screen potential members by using signals of whether or not a newcomer is a good fit. In order to gather information about these signals, diagnostic tasks are often used such as solving CAPTCHAs to screen automated attackers or acquiring experience points and weapons to signal character prowess in the online game World of Warcraft.

The online community of Wikipedia regularly faces the coordination challenges of managing contributions and dealing with newcomers. The two challenges are often related, as the contribution evaluation process can have an important impact on contributor motivation particularly for new members. In a study of the contributions of new editors on Wikipedia, Halfaker et al. [2011] found that reverts decreased motivation for newcomers. In particular, reverts from experienced editors were the most demotivating, suggesting that certain social interactions around contributions may have a particularly negative influence on the motivation of newcomers to contribute. Bryant et al. [2005] found that contribution acceptance is an important step in a new editor's socialization process. Newcomers learn the conventions and contribution rules of the Wikipedia community through observation (lurking) and direct mentoring from more experienced users. Related to this concept of mentoring is a community-wide norm of "don't bite the newcomers."

“Talk pages” are an important coordination mechanism to manage contributions for articles on Wikipedia. Viegas et al. [2007] found that the primary use for Talk pages were requests for coordination where editors discussed editing activities in advance. Kittur et al. [2007b] found that coordination work such as the discussions in these Talk pages were growing at a much faster rate than direct edits to articles. They find that these discussions also serve as a mechanism for building consensus and resolving conflicts. Towne et al. [2013] examined how users’ perceptions of article quality declined when coordination discussions were shown along with the article, especially if conflict was present in the Talk page. Arazy et al. [2013] found that when these conflicts were not sufficiently resolved in discussions, the disagreements would impede group performance through lower article quality.

Wikipedia also regularly faces political challenges, particularly between classes of users. Kittur et al. [2007a] suggested that there are two de facto classes of users on Wikipedia: 1) “elite” users such as administrators or high-edit users and 2) “common” low-edit users. Their study suggests that the influence of “elite” users has waned as much of the work on Wikipedia has shifted to “common” users. Forte and Bruckman [2008] described policy as the main governance mechanism on Wikipedia. Creating policy requires building consensus across groups of users while enforcing policy is handled by administrators. Though enforcement power is concentrated in the relatively small population of administrators, decisions by administrators are not enforced without widespread support from the larger community. Kriplean et al. [2007] found that during conflict, users will engage in political maneuvering (termed “power plays” in the work). These political strategies include arguing or redefining the scope of the article, referencing past policy or consensus in other articles, pointing to past work as an appeal to authority, and “elite” users threatening to leave articles.

## 2.2 Open Source Software

Open source software projects, similar to online communities, often regularly coordinate with developers through managing code contributions and newcomers. For this dissertation, I purposely disregard common coordination tasks for open source software projects that are wholly internal to the project, such as division of labor and task assignment [Crowston et al., 2008]. For the studies presented in the following chapters, I focus on coordination tasks that may be informed by transparency information. As much of the information that transparency makes visible is around external projects or developers, I accordingly focus on coordination tasks external to projects such as code contributions.

The distributed nature of open source software development [Mockus et al., 2002] encourages open source software developers to seek out information about their fellow developers in order to stay aware of their work activities. Gutwin et al. [2004] found that developers in open source software projects informed their coordination needs through seeking out information such as who is working on what part of the project. They found that work awareness information from simple text communication such as mailing lists and text chat was enough to satisfy most project coordination needs. Newcomer developers to an open source software project also needed to seek similar awareness information from text communication tools [Ducheneaut, 2005] in order to get “buy-in” from core project developers towards supporting their contributions. Rigby and Storey [2011] found in their study of peer review on open source software projects that project managers selecting which code contributions to review used similar work awareness information from the project’s mailing list. They also found in their study that developers also suffered from “too much awareness” and needed filtering techniques to manage the information overload. Guzzi et al. [2015] found that current awareness information in integrated development environments (IDE) is sufficient for developers to overcome coordination challenges such as simultaneous conflicting changes. However, breaking changes by developers on the same team were particularly difficult to deal with using existing IDEs.

Literature suggests that managing newcomers and their contributions in open source software projects is a complex social process. von Krogh et al. [2003] found in their study of the contribution process in the Freenet open source project that successful newcomers must follow “joining scripts” before submitting a contribution. These joining scripts involve participating in various aspects of the project such as lurking on the project’s mailing list, participating in technical discussions, and reporting bugs. They also found that the nature of discussions around contributions differed between developers that joined the project versus developers who did not. For example, the detail and specificity of feedback given was much more general for non-joiners. Ducheneaut [2005] found that developers also underwent a progressive socialization process before successfully contributing to the Python project. Core members on a project regularly evaluated contributions and contributors to ensure submitted code changes were technically sound. Successful socialization allowed potential submitters to learn project norms and to identify members of the core project team that participated in this evaluation process. In order to successfully start the contribution evaluation process, a submitting developer needed to “recruit” core members of the project as a network of “allies”, especially when proposing complicated or controversial changes. Regarding the transition from one-time contributor to more involved long-term contributors, Zhou and Mockus [2012] suggest that the difference is that long-term contributors are possess more “willingness” to help the project. They suggest that the nature of support actions these contributors participate in are a measure for willingness.

For example, the low cost of reporting an issue through a tool may be less involved than the higher cost of applying for an account in GNOME Bugzilla, creating a report, and filling in the bug reproduction template. Steinmacher et al. [2015] found that newcomers to open source software experienced social barriers when attempting to make their first contribution to project. These barriers include reception issues from late or non-existent responses, the social and technical capability of the newcomer, newcomer orientation in the form of mentorship, documentation problems, cultural differences such as rudeness, and technical hurdles.

As open source software often relies on the contributions of different software developers [Crowston et al., 2008], a key coordination task for members of software project teams is to evaluate and discuss contributions to ensure the integrity of the software project. A common method of evaluating code contributions is the peer review process. Rigby et al. [2008] found in their examination of different peer review processes in the Apache server open source project that early and frequent reviews of small contributions from the core team were effective in finding defects in contributions. In particular, the usage of the project mailing list allowed for self-selection of expert core members and a more open discussion between members. Ko and Chilana [2011] found that discussions around bug reports established scope, proposed ideas, identified design dimensions, defended claims with rationale, moderated the process, and finally made a decision. The most powerful factors in decision-making around a bug report were the participant’s authority (developers over users) and actions taken (writing a patch).

## **2.3 Transparent Development Environments**

Transparent development environments implement transparency features to provide new actionable information that enable developers to make highly informed coordination decisions. Transparency in this case refers to the “accurate observability of an organization’s low-level activities, routines, behaviors, output, and performance” [Bernstein, 2012]. In the case of transparent development environments, the “organization” may be a software project or a particular developer and the “activities” may include low-level development work such as code commits, bug reports, and discussions.

Developers inform coordination decisions by making inferences about other developers and projects using information made visible by transparency. As part of my early work on the transparent development environment of GitHub, Dabbish et al. [2012] found that developers used information such as the recency and volume of activity of a developer to infer their interest and level of commitment. They also found that developers inferred

the intention or “story” behind development activity by observing the sequence of actions over time. Regarding projects, developers inferred the relative importance of a project to the community by using signals for attention in the form of star and fork counts. They found that developers use these inferences to inform coordination activities such as managing projects, learning through observing, and managing their reputation in the GitHub community. For example, project managers inferred user needs by observing their activities in forks. Developers looking to learn how to improve their coding ability followed “coding rockstars” and observed their development work. Dabbish et al. also found that developers were very aware of an “audience” due to the transparency in the environment. This awareness of the audience influenced how developers worked such as making changes less frequently.

Literature on GitHub and other transparent environments has explored other coordination tasks and the inferences that developers make in order to inform these tasks. Marlow et al. [2013] found that GitHub developers used information in the environment in order to form impressions of users during three coordination scenarios: discovery of a developer through following, informing interactions through pull requests, and forming expectations about skills through pull requests with unknown developers. To form these impressions, developers used information from user profiles such as recent activity, project owned, languages used, and past comments to infer qualities such as coding ability and interaction style. Pham et al. [2013] found that when project managers are assessing how much testing a contribution requires, they used inferences for the type and target of the contribution and how much they trusted the submitter. Project managers tended to demand tests for contributions that introduced new features or targeted core functionality. Developers perceived to be more trusted received a less thorough assessment than unknown developers. Singer et al. [2013] in a study of developer profile aggregators found that developers and recruiters differ in the inferences they use to evaluate developers. Developers assess other developers often and infer qualities such as passion for technology, diversity, and standing in the community (called the “coder footprint” by Singer et al.). Recruiters on the other hand focused on filtering potential developers by inferring if they possessed relevant technical skills then whether they passed some baseline level of activity in open source software. They also inferred ability to learn quickly and passion for technology through looking at a developer’s programming language diversity.

The information that transparent environments make visible may also enable the creation of useful statistical models to understand different aspects of how developers coordinate. These models may also inform the creation of useful predictors for desired qualities in GitHub projects. Kikas et al. [2016] use a number of issue and project features to predict the probability of an issue closing at various points in its lifetime. Vasilescu et al. [2015]

use a longitudinal dataset to create gender and tenure diversity predictors for productivity. They find that both predictors are positive and significant. Casalnuovo et al. [2015] found that experience in a programming language and prior social links affect the productivity of newcomers to projects. Yu et al. [2015] found a number of social, technical, and process-related factors, in particular continuous integration-related, that affect pull request latency.

A challenge of working in transparent environments is that the information made visible is potentially overwhelming and noisy. Kalliamvakou et al. [2015] pointed out that GitHub, like many similar software hosting services, is mostly comprised of inactive projects with little activity. Very few projects use collaborative features such as pull requests [Kalliamvakou et al., 2015]. Singer et al. [2014] in their study of how open source developers use Twitter found that developers dealt with challenges in how to manage consuming large amounts of information through developing strategies such as filtering tweets and curating their following networks. Storey et al. [2014] found in a survey of developers using social media platforms, which includes transparent environments such as GitHub, that a majority of developers surveyed felt overwhelmed and distracted by such tools. The survey also found that key challenges that developers faced included both potentially missing important information and filtering out low-quality content.

## 2.4 Signaling Theory

Signaling theory is a useful lens for understanding how developers use information made visible by transparent environments. *Signals* are observable pieces of information that are used by a receiver to infer an unobservable quality of the signaler. In Spence [1973]’s classic example, potential employees indicate their unobservable ability level to employers through the signal of educational credentials.

Transparent environments enable a new class of potentially useful, unintentional signals that are derived from observing work. While the majority of signals that are studied in literature are positive, intentional signals, unintentional signals still convey important information [Janney and Folta, 2003]. For positive, intentional signals, the incentive to signal is to affect the decision of the receiver [Spence, 1973]. In transparency, some useful signals are unintentional and a product of development work, merely performing the task produces the signal. Therefore the incentive of producing such a signal becomes aligned with performing the development task.

Signaling theory offers key constructs [Connelly et al., 2010] for describing the relationships between signals, unobservable qualities, signalers, and receivers. Signals vary

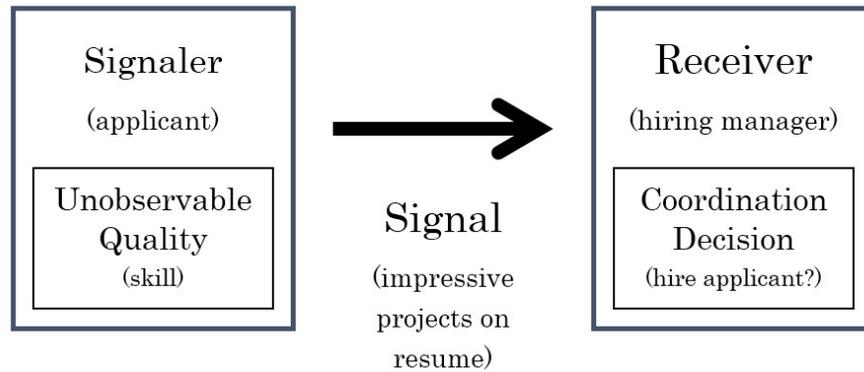


Figure 2.1: Diagram of Signaling Process

in their *cost* to produce, often with an assumption that signalers possessing the unobservable quality are better suited to absorb these costs than others. Signals may also vary in *fit*, the correlation between the signal and the unobservable quality it indicates. A related concept is *honesty*, the extent to which the signaler actually possesses the signaled quality. For example, a resume listing involvement in an impressive-sounding project may not reflect meaningful skill for a particular task. If listing impressive-sounding projects is not correlated to the needed skill, then that signal has a poor fit. If such projects *are* correlated to skill but the candidate in question does not possess that quality, the signaler is dishonest. Receivers, especially those that need to attend to the environment such as in transparent development environments, may need to actively scan the environment for signals. *Receiver attention* is the extent to which receivers attend to the environment for signals. Once receivers receive a signal, the translation from signal to unobservable quality is *receiver interpretation*. During this translation, receivers may apply their own weights or even meanings to signals. For example, one hiring manager may weight impressive-sounding projects much more highly than another manager.

In online communities such as open source software projects, participants draw signals from the environment to infer qualities of both people and projects. Donath [2007] finds rich patterns of signaling and deception in online communities to infer member identities. Many of the studies of the inferences that developers make in GitHub described in the previous subsection 2.3 explore potential signals that developers use. For example, to evaluate a developer, information in the environment such as activity traces and past discussions are used as signals to infer unobservable qualities such as coding ability and personality [Marlow et al., 2013]. Signals are not limited to developers but may include projects. For example, activity traces may be signals for project properties such as qual-

ity, collaborative environment, and member commitment [Dabbish et al., 2012]. Outside of GitHub, Scaffidi et al. [2010] found in their study that users use signals of previous successful authorship and mass appeal for web macro scripts to decide whether to reuse a script.

## **2.5 Summary**

Transparent development environments hold much promise for assisting developers in facing the coordination challenges that are present in online communities such as open source software projects. Transparent environments also possess their unique challenges such as dealing with information overload and audience pressures. While prior studies are starting to find that developers use inferences derived from transparency information to address some of these challenges, a principled understanding of these inferences and the mechanisms behind this inference process may enable improvements that directly address challenges developers in these environments face. I use signaling theory as a theoretical lens to further our understanding of how developers use information in a principled manner by reasoning about this process in terms of qualities and signals. In the following chapters, I use signaling theory to inform studies on specific coordination tasks such as evaluating code contributions or evaluating projects and the qualities and signals that developers use to inform these tasks.



# Chapter 3

## Research Context: GitHub

GitHub is a popular software project-hosting site started in 2008 that brands itself as "Social Coding." The site offers both free open source project hosting and paid private hosting and was home to almost twenty million code repositories in 2016.<sup>1</sup> Some of the more popular open source software projects that GitHub hosts include *Ruby on Rails* and *jQuery*.

I selected GitHub as my research setting for the studies described in the following chapters because it is a very widely used transparent development environment. While traditional open source environments make low-level activities of a particular project visible, GitHub includes many transparency features that make low-level development activities visible at a much larger scale. Specifically, GitHub makes explicit social relationships and links together activity information via these relationships. For example, a normal open source project has a freely visible codebase and commit log. For a project on GitHub, not only is the commit log visible but also visible are all of the commits performed, bug reports filed, and comments made across all projects in the GitHub ecosystem for each committer.

This chapter describes features in GitHub that are relevant for the studies described in the following chapters. These features include transparency features and methods of contributing to projects in GitHub.

<sup>1</sup><https://octoverse.github.com/> (accessed March 2017)

## 3.1 Transparency Features in GitHub

GitHub includes transparency features that link together activity traces via social relationships and broadcast activity information across social networks. GitHub's features involving social relationships are the ability for users to "follow" other members in the community and to "star" or "watch" repositories. The social networking-style features that aggregate activity information for consumption are user profiles and the activity feed.

GitHub provides support for explicitly establishing user-to-user relationships via the "following" feature and user-to-project relationships via the "star" and "watch" features. Following directs events about a developer's actions to the participant's news feed. Highly respected developers can be "followed" via their profile page as in Figure 3.2 to see their development activity across all of their projects, perhaps to learn how they code or to identify trendy projects [Dabbish et al., 2012]. Much of the followed participants' social activity is also visible in the feed, including changes to the set of users that person is following. "Watching" a repository is a similar action to following a user. Events about that repository are directed to the participant's news feed. "Starring" a repository works similarly to a bookmarking system, adding the starred project to a list of projects for a particular user. As a note, before 2012, "starring" did not exist as a feature on GitHub as "watching" included all of the bookmarking features that stars currently implements.<sup>2</sup> At this time, stars seems to have supplanted watchers as a common visible signal of attention for a project as described by Dabbish et al. [2012]. Stars and watchers are highly visible for projects, both in project search results and individual project pages, as seen in Figure 3.1.

GitHub provides social networking-style features that aggregate activity information. Each GitHub user has a profile page as shown in Figure 3.2 that lists personal information, activity-related information such as the repositories they own and have starred, and an activity feed that displays recent actions such as commits, pull requests, or comments. Each user also has a personal activity feed as shown in Figure 3.3 which is the default page shown when logging on to the GitHub site. This feed aggregates recent events from the projects the user is involved with or watches. These events include contributions such as commits, pull requests, or comments. The personal feed also includes events from users followed such as the repositories they create or star.

<sup>2</sup><https://github.com/blog/1204-notifications-stars> (accessed March 2017)

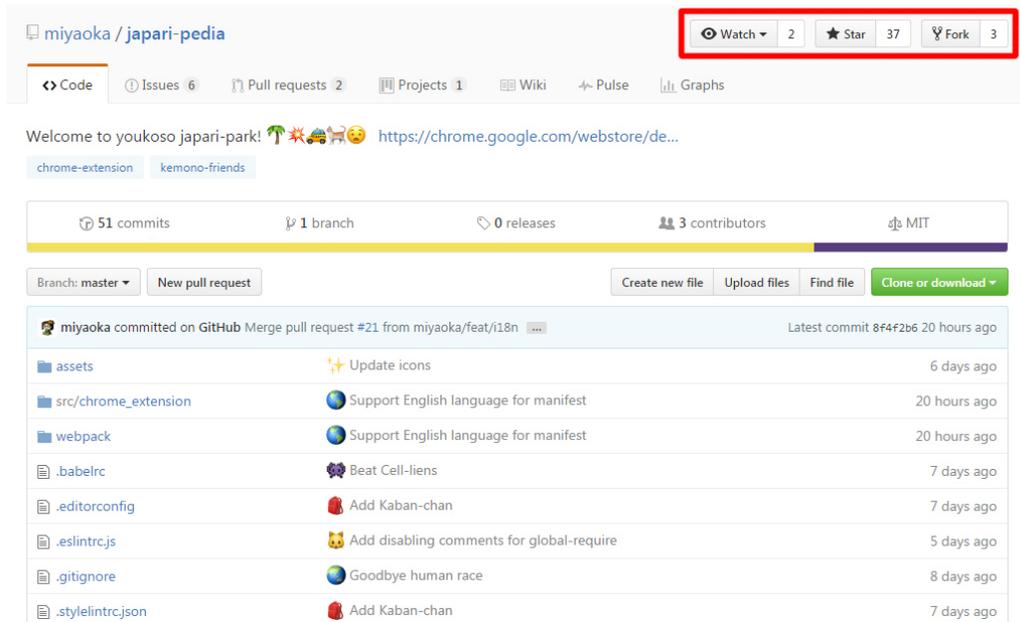


Figure 3.1: Example of GitHub Project Page with Watch, Star, and Fork Buttons Highlighted

## 3.2 Contributions in GitHub: Pull Requests, Issues, Comments

GitHub provides features that standardize and streamline offering contributions to a project. Offering code contributions or patches to projects in GitHub is done through “pull requests.” GitHub also provides its own bug tracker system where users can file “issues” for a project. For both of these contribution mechanisms and for specific lines of code, users may discuss the contribution through comments.

Offering code contributions in GitHub involves “forking” a project and then sending a “pull request” to that project. GitHub and its underlying version control system Git allows any user to “fork” any public project as shown in Figure 3.1. “Forking” creates a personal copy of any public project where the user can then make changes to, add, or alter functionality, without disturbing the code in the original project. This user can then send a “pull request” to the original project as shown in Figure 3.4 to request that some or all of their changes to the code base be reintegrated into the original project. The project manager has several options to “close” the pull request, including accepting the offered contribution and merging it into the project’s code base or rejecting the contribution. At

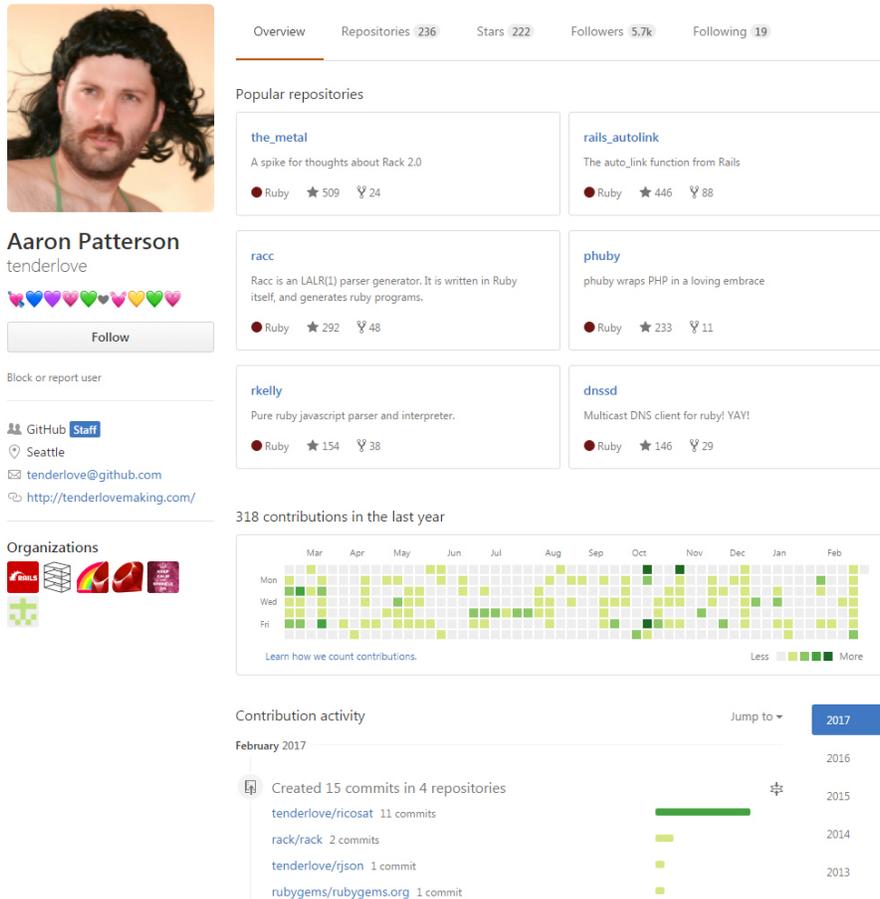


Figure 3.2: Example of GitHub User Profile Page

the same time, managers and other interested users may comment on the pull request, perhaps to suggest improvements or negotiate over the code change. Of course, project managers may also ignore the contribution, leaving the pull request "open."

GitHub also provides a bug tracker for each project that allows users to file "issues." Any user is able to submit an issue as seen in Figure 3.5 to a public project, often to either report a bug or request a feature. Like other bug tracking systems, project managers are able to assign labels to issues such as "security" in Figure 3.5. Similar to pull requests, managers are also able to "close" issues as they are resolved or simply ignore issues and leave them "open." Also similar to pull requests, project managers and interested users are able to discuss issues via comments.

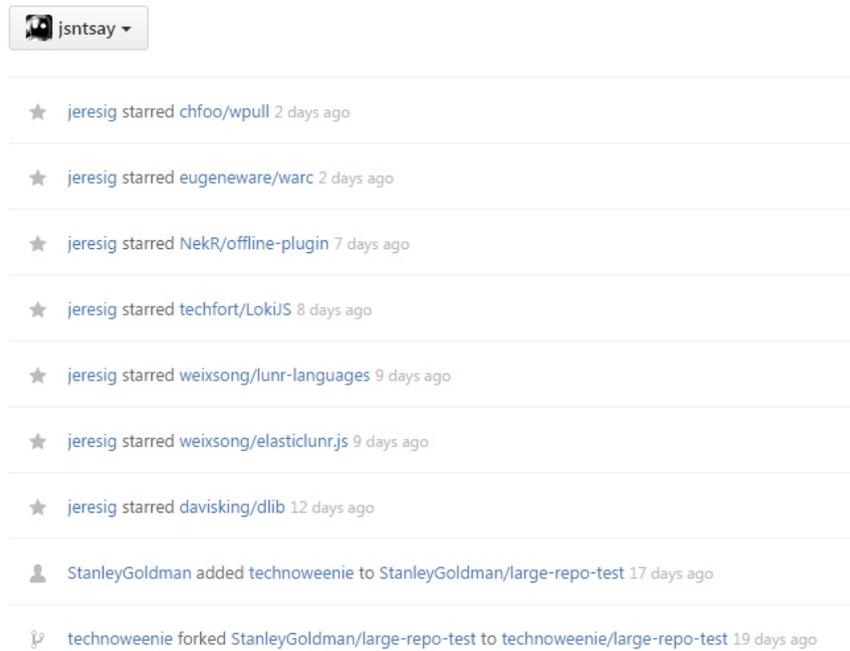


Figure 3.3: Example of GitHub Activity Feed

GitHub provides commenting as a means for users to discuss contributions. As described earlier, both pull requests and issues allow for any interested user to leave comments as shown in Figure 3.4 and 3.5. In these comments, specific users may be notified using the “@” symbol. For example, if octocat is a GitHub user, a comment of “@octocat please look at this” will notify that user. For pull requests and specific commits, users may also leave comments for specific lines of code as shown in Figure 3.6.

jruby / activerecord-jdbc-adapter

Watch 42 Star 411 Fork 321

Code Issues 65 Pull requests 4 Projects 1 Wiki Pulse Graphs

## Don't barf on the datetime2 data type when connecting to MS SQL 2008 #233

Closed trak3r wants to merge 1 commit into jruby:master from trak3r:master

Conversation 14 Commits 1 Files changed 1

trak3r commented on Sep 12, 2012

Don't barf on the datetime2 data type when connecting to MS SQL 2008

don't barf on datetime2 when connecting to ms sql server 2008 dd5b577

bwalsh commented on Dec 4, 2012

Running into this problem. Is there anything blocking this pull? Anything I can do to help?

gregors commented on Dec 28, 2012

I'm also running into this issue. Also this is a duplicate of #116 - are the build failures related to this pull request? What do we need to do to get this moving? A failing test for 2008?

gregors commented on Jan 8, 2013

I guess ms sql doesn't get any love around here?

kares commented on Jan 17, 2013

taking quite a while. I'll try to manage but it's hard without a MS-SQL DB ... a (specific) tests would be great

Reviews: No reviews

Assignees: No one assigned

Labels: None yet

Projects: None yet

Milestone: 1.3.0

Notifications: [Subscribe](#)

You're not receiving notifications from this thread.

6 participants

Figure 3.4: Example of GitHub Pull Request

rails / rails Stop ignoring 2,404 Unstar 34,612 Fork 14,113

<> Code **Issues 598** Pull requests 671 Projects 0 Pulse Graphs

## New Secrets feature uses poor cryptography #28135 New issue

**Open** stouset opened this issue 2 days ago · 8 comments

**stouset** commented 2 days ago · edited Contributor

```

def new_cipher
  OpenSSL::Cipher.new("aes-256-cbc")
end

def cipher(mode, data)
  cipher = new_cipher.public_send(mode)
  cipher.key = key
  cipher.update(data) << cipher.final
end

```

Unauthenticated CBC mode with a constant IV. This is not up to the minimum standards of how a modern cryptosystem should be designed. Please use an authenticated mode (GCM, or if unavailable, derive an authentication key and HMAC the ciphertext) and generate an unpredictable, random IV *each and every time* an encryption is performed.

👍 27

**cliochris** commented 2 days ago Contributor

I filed this and a related issue as #208496 on the bounty program page at HackerOne, though I believe it's not publicly visible. The link is [https://hackerone.com/bugs?report\\_id=208496](https://hackerone.com/bugs?report_id=208496)

**rafaelfranca** added this to the 5.1.0 milestone 2 days ago

**Assignees**  
No one assigned

**Labels**  
attached PR  
railties  
security

**Projects**  
None yet

**Milestone**  
5.1.0

**Notifications**  
  
You're not receiving notifications from this thread.

**6 participants**

Figure 3.5: Example of GitHub Issue

<> **pangratz** commented on the diff on Jan 9, 2012

packages/ember-states/lib/view\_state.js View full changes

```

... .. @@ -16,6 +23,13 @@ Ember.ViewState = Ember.State.extend({
16 23   exit: function(stateManager) {
17 24     var view = get(this, 'view');
18 25
26 +   if (typeof view === 'string') {
27 +     view = getPath(view);
28 +     if (view) {
29 +       this.set('view', view);

```

**pangratz** on Jan 9, 2012 Member  
why are you setting the view again when it is removed a few lines below?

**tchak** on Jan 9, 2012 Member  
For caching purpose. If later I reenter the state, the reference for view is there

Figure 3.6: Example of Inline Code Comment in GitHub



# Chapter 4

## Signals for Evaluating Contributions in GitHub

### Chapter Summary

Evaluating code contributions is a key coordination task that developers must perform in order to ensure the integrity of their projects. Transparent development environments enable developers to use information such as technical value and social connections as signals when evaluating contributions. This chapter describes a study on potential signals that developers use when evaluating pull requests in GitHub. The study analyzes the association of various technical and social signals with the likelihood of contribution acceptance. I found that while signals for good technical contribution practices were associated with acceptance, the effect of the social connection between the submitter and project manager was much higher. Pull requests with many comments were much less likely to be accepted, moderated by the submitter's prior interaction in the project. Well-established projects were more conservative in accepting pull requests. These findings provide evidence for potentially useful signals for evaluating pull requests. The findings also suggest that social signals in this case may be more useful than technical signals, perhaps due to differences in assessment costs.

---

<sup>0</sup>For the full paper describing this study, please see Tsay et al. [2014a], published in the International Conference on Software Engineering (ICSE) 2014.

The open contribution model of open source software projects that enables any developer the potential to submit code in many ways characterizes the movement of open source software (OSS) itself. While open contribution enables a variety of people with diverse expertise – the “long tail” of contributors – to add their unique value to a project, openness also brings the danger of integrating changes with errors which may expose serious vulnerabilities such as with the infamous Heartbleed bug in the OpenSSL project that compromised secret keys.<sup>1</sup> This danger of accidentally accepting problematic code requires the core developer team for projects to carefully evaluate submitted code contributions to ensure their quality and to maintain technical integrity for the project. Evaluating contributions is a key coordination task for software developers that I examine in this study.

The advent of transparency in development environments may enable project managers to better evaluate code contributions by using the vast amount of information regarding code contributions and submitters that transparency makes visible. As opposed to traditional open source software projects which make visible all of the code and changes for a project, transparent environments also make explicit the relationship between users and work artifacts or other users. This relationship also links together information to surface potentially useful signals for project managers. For example, project managers are able to view all of the prior projects that a newcomer might have participated in and evaluate them as signals of developer skill before deciding whether or not to accept a contribution. With information available from potentially millions of developers and millions of repositories in these transparent work environments, what information do software developers actually use as signals when evaluating software contributions?

Identifying the type of signals that project managers make use of when evaluating contributions furthers our understanding of both how developers evaluate projects and use information in transparent environments. Traditional perceptions on open source software characterize projects as places where evaluations are based solely on technical merit [Scacchi, 2007] as “code is king” while literature on open source suggests the existence of a complex social structure around contribution to projects [Ducheneaut, 2005]. Transparency makes visible a number of these potential technical and social signals for project managers to make use of when evaluating contributions. The choice of type of signal that project managers use during evaluation may also give insight into the code contribution evaluation process itself. For example, for a project where “code is king,” only technical signals should be used when evaluating submissions. The choice of signals to use also may inform how developers use transparency information in general. For example, perhaps social signals are much easier for project managers to use than technical signals, suggesting that the cost to interpret a signal is an important factor.

<sup>1</sup><http://heartbleed.com/> (accessed March 2017)

To further our understanding of what signals project managers use to evaluate contributions, I performed a quantitative analysis of pull request acceptance from thousands of open source projects on GitHub. I calculate a multi-level set of measures for potentially useful technical and social signals from the pull request, submitter, or project level. The potential signals are grounded on literature on open source software and online communities. I perform a regression analysis that associates each signal with pull request acceptance. Signals that are strongly related to pull request acceptance may also suggest which types of signals are most likely to be used by project managers when deciding to accept a contribution.

I found that both technical and social signals had strong associations with contribution acceptance. In particular, social signals of the connection between the submitting user and the user managing the contribution were especially associated with contribution acceptance. Contributions with many associated comments were much less likely to be accepted, perhaps due to contention between the submitter and the core project team. The negative influence of comments was moderated, however, by the submitter's prior interaction with the project.

The findings of this study provide evidence for possible signals that project managers use for the coordination decision of evaluating contributions. In particular, the strong association of social signals such as social distance compared to technical signals such as test inclusion suggests that these signals are more useful for project managers. The difference in usefulness is perhaps attributable either to the qualities that these signals indicate or the assessment costs in actually using the signals. The negative association of extended discussions on contributions and the moderating effect of the submitter's prior interaction with this negative association raise questions about the nature of these discussions. These discussions may also be an important coordination mechanism for developers and are explored further in Chapter 5.

In the following sections I consider related research on the contribution practices of open source software projects and online communities in order to generate hypotheses, describe the study's multi-level logistic model of pull request acceptance, report the results of the analysis, and discuss the implications of the study's findings.

## **4.1 Contributions in Transparent Development Environments**

I ground this work in prior literature on participants in open source software projects and online communities making contributions. Informed by this work, I generate a set of potential signals and hypotheses for each signal to test in the analysis of contributions in GitHub.

### **4.1.1 Contributions in Open Source Software**

Literature on the contribution process for open source software projects suggests that accepting contributions, especially from unknown developers, is a complex process. von Krogh et al. [2003] found in their study of the contribution process in the Freenet open source project that there are “joining scripts” that successful newcomers follow before offering contributions. These joining scripts involve participating in prior activity such as lurking on the project’s mailing list, participating in technical discussions, and reporting bugs. Developers that offered technical contributions without following this joining script tended not to have their contributions accepted into the project. Ducheneaut [2005] made a similar observation in the Python project, noting a progressive socialization process that requires both displaying technical skills and creating the right social relations. In order for contributions to be accepted into the project, the contribution must both be technically sound and be vetted by core members of the project. For successful and complete socialization (becoming an “insider”), a developer needs to recruit core members of the project as a network of “allies.”

Shah [2006] observed that this contribution process also leads to evolution for a developer’s level of participation in the project. Most developers make simple initial contributions such as bug fixes in order to fulfill some need. A number of these developers choose to continue to participate in the project, evolving from a need-based participation to a hobbyist. Often, these developers will also gain committer rights or the right to freely commit their changes directly into the project.

As open source software projects evolve, their contribution needs tend to change as the project matures. Nakakoji et al. [2002] observed that as open source projects evolve, the communities around the project co-evolve along with the open source software system. Contributions to the project influence the transformation of both the software system and the community. Nakakoji et al. also defined at least three different classes of open source projects that evolve into each other. These project classes each have their own unique

contribution needs and selection criteria. For example, a Service-Oriented OSS system like PostgreSQL tends to be very conservative in terms of accepting contributions due to a need for stability. Stewart and Gosain [2006b] also found that project maturity in open source software projects on SourceForge moderates both objective and subjective performance outcomes. For example, the effect of task completion on perceived effectiveness is more positive for more mature projects.

### **4.1.2 Contribution in Online Communities**

To survive and thrive, online communities face the challenge of attracting and evaluating contributions. Kraut and Resnick [2012] claim that when dealing with newcomers, successful online communities must meet a number of challenges: attracting newcomers, selecting among the newcomers, retaining newcomers, socializing newcomers, and protecting existing members from potential problems newcomers may bring. When evaluating newcomers, communities will often screen potential members by using signals of whether or not a newcomer is a good fit. In the community of Wikipedia, Bryant et al. [2005] found that some newcomers will transition from making peripheral contributions to specific articles into core users that help maintain Wikipedia and its community as a whole. Newcomers learn the conventions and contribution rules of the Wikipedia community through observation (lurking) and direct mentoring from more experienced users. Related to this concept of mentoring is a community-wide norm of “don’t bite the newcomers.” The nature of users’ contributions also tends to change as newcomers become more socialized, from purely making edits in articles to also participating in community discussions, administrative duties, and “meta” tasks.

Iriberry and Leroy [2009] found that online communities have multiple lifecycle stages with different contribution needs. For example, during the earlier Growth stage, communities are more concerned with attracting new members and supporting interactions while the later Maturity stage, communities may prefer to recognize contributions and increase visibility of certain members.

### **4.1.3 Evaluating Contributions in Transparent Development Environments**

Previous qualitative research on GitHub [Dabbish et al., 2012] showed that project managers, especially those in popular projects that received many contributions (pull requests) per day, would make use of inferences about the quality of code contributions and sub-

mitter competence. Marlow et al. [2013] found that when GitHub developers engage in information-seeking behaviors, they use signals in the environment to form impressions of users and projects. For example, impressions of general coding ability could be gleaned from the contents of a GitHub user’s profile. Signals of whether or not a developer possesses specialized project-relevant skills were embedded in the user’s activity log. Project managers would often account for uncertainty when evaluating contributions, straightforward and easily verifiable changes were often accepted “as is” whereas complicated, uncertain changes would require discussion before acceptance. In these cases, project managers would often engage in discussion with the submitter in order to negotiate the change. In these cases, where the value of the contribution was uncertain, project managers would make use of both code-based factors and person-based factors. For example, a project manager may weigh the cost of fixing a contribution against the benefit of recruiting a new member to the project.

Pham et al. [2013] found in their study of the testing culture in GitHub that project managers would demand that contributions include tests in certain cases. For example, contributions that introduced new features were expected to include tests. On the other hand, contributions that involved existing code, especially if the change was small like a bug fix, may or may not require tests. Also, if the project manager trusted the submitting developer, the contribution tended to be evaluated more leniently. Many submitters would include tests as a method for highlighting the value of their contribution to the project manager.

#### **4.1.4 Hypotheses Development**

I use the above prior works to derive a set of potentially useful signals for evaluating contributions in transparent development environments.

##### *Technical Contribution Norms*

Prior work about GitHub suggests that there are certain contribution norms that signal a technically well-prepared contribution. For example, project managers see an urgent need for automatic testing in their projects in order to maintain quality as the number of peripheral developers scales [Pham et al., 2013]. So, project managers tend to value contributions that include test cases more highly. Another example of such a signal is the community norm of having legible, easy-to-evaluate pull requests [Dabbish et al., 2012]. Contributions that display these signals of technical value may indicate a well-thought out technical submission that is also easier for a project manager to evaluate [Dabbish et al., 2012].

**H1:** Contributions that show signs of following technical contribution norms are more likely to be accepted.

#### *Social Connection*

In traditional open source software projects, newcomers often need to “recruit” core members of a project in order to have their contributions accepted [Ducheneaut, 2005]. This process involves knowing who the key core members are and being able to convince them of the usefulness of the contribution, especially if the code contribution is complex. Often, these key members expect newcomers to have previously participated in technical discussions and other peripheral actions in order to learn project-specific norms and prove suitability before submitting contributions [von Krogh et al., 2003]. In GitHub, these kinds of social connections are visible and made explicit, perhaps making social connections between submitters and project managers more salient.

**H2:** Contributions from submitters with a stronger social connection to the project are more likely to be accepted.

#### *Highly Discussed Contributions*

Certain contributions raise uncertainty about their value for a project and subsequently generate more discussion [Marlow et al., 2013]. Changes that required high amounts of discussion tend to be more closely scrutinized by more members of the site, as GitHub users would look at discussion on a contribution as a signal of controversy. These contributions may be less technically sound, more complicated to evaluate, or simply controversial in terms of project direction or implementation strategy. Due to the high degree of uncertainty, project managers may then be less willing to accept the contribution.

**H3:** Contributions with a high amount of discussion are less likely to be accepted.

#### *Decision-Making for Highly Discussed Contributions*

When the value of a contribution is uncertain, project managers may employ different standards when evaluating the contributions [Marlow et al., 2013]. In the cases of contributions with high amounts of discussion, I expect both the tone of the discussion and the degree of uncertainty to change depending on differences between the technical nature of the contribution and the social relationship between the submitter and the core project team. These different social and technical factors should then moderate the uncertainty in highly discussed contributions.

**H4:** Acceptance of highly discussed contributions will be moderated by both social and technical factors.

#### *Submitter’s General Community Standing*

Previous research on GitHub has found that developers often use inferences about developers and software projects to evaluate them [Marlow et al., 2013]. This research suggests the identity of the submitter and/or the software project may affect how contributions are evaluated. Members of the GitHub community regard certain members as being at a higher standing. Some prolific developers are even considered “coding rockstars” by the overall community [Dabbish et al., 2012]. Project managers who receive contributions from higher standing submitters may then be more willing to accept them based on the submitter’s status.

**H5:** Contributions from submitters with a high status in the general community are more likely to be accepted.

#### *Submitter’s Status in Project*

With open source software projects, there often is a structure of “core” and “periphery” developers, with core developers being the few central developers who implement most of the code changes and make important project direction decisions and peripheral developers being the “many eyes” of the project that make small changes such as bug fixes [Mockus et al., 2002]. Core developers who make contributions to their own project may then be more likely to have their contributions accepted by fellow project managers.

**H6:** Contributions from submitters that hold higher status in a specific project are more likely to be accepted.

#### *Project Establishment*

As open source software projects progress through their lifecycle, their needs tend to differ from less mature projects [Nakakoji et al., 2002]. The development stage of an open source project also tends to moderate its performance outcomes [Stewart and Gosain, 2006b]. As projects evolve, their contribution needs may also co-evolve [Nakakoji et al., 2002]. More established projects may be more service-oriented with many downstream dependencies. Project managers are often aware that their projects are depended on by other, perhaps more high profile projects. For example, certain popular websites may depend on a particular library on GitHub, so a broken release may also break the popular website [Dabbish et al., 2012]. Project managers of established projects may then be much more conservative when accepting contributions in light of these dependencies.

**H7:** Contributions to established projects are less likely to be accepted.

## 4.2 Methods

To investigate signals for evaluating contributions, I created and analyzed a dataset from the popular open source software hosting site GitHub. I selected a sample of pull requests on GitHub and gathered information on the pull requests, the submitting users, and the project the pull request was submitted to. From this dataset, I fit a statistical model that associates social and technical contribution signal measures with the likelihood of pull request acceptance. In this section I present our data collection procedures, signal measure calculation, and analysis technique.

### 4.2.1 Pull Request Selection

I create a dataset of pull requests and the users and repositories associated with each pull request through sampling for active, collaborative projects on GitHub. The dataset comprises information gathered from the GitHub Application Programmer Interface (API). First, I drew a sample of repositories from the GitHub Archive dataset<sup>2</sup> on July 17, 2013 with the following sampling criteria:

1. Excluded forks, developer-specific copies of repositories often meant for interim development work, in order to avoid double-counting contributions in my model.
2. Excluded repositories that have not had at least one event of activity within one week prior to data collection, July 10, 2013 in order to avoid inactive projects.
3. Excluded repositories that do not use the GitHub issue tracker, as I also use the issue tracker as a source of data.

This selection included 185,342 repositories. I further refined the selection using the GitHub API to retrieve more detailed information about each repository with the following criteria:

1. Removed each repository that did not contain at least one closed pull request due to using closed pull requests as a base unit of analysis.
2. Excluded repositories with less than three unique contributors in order to ensure that the project has received some outside contributions.

<sup>2</sup><https://www.githubarchive.org/>

After this second phase of filtering, the sample included 12,482 projects.

I used pull requests as a base unit of analysis. From these 12,482 projects, I extract all closed pull requests from the API. As this study is concerned with the decision of whether or not to accept a pull request, I excluded all open pull requests. In total, this includes 659,501 pull requests across the 12,482 projects. This dataset also gathered information about each unique GitHub user associated with the set of pull requests. This set of user information includes 95,270 unique GitHub user accounts. I also used the API to gather information on all issues and comments for each repository.

## 4.2.2 Signal Measures

Using the created dataset, I generated measures for potential signals for evaluating contributions. Each signal is based on prior literature on GitHub, traditional open source software communities, and online communities (see Table 4.1 for a descriptive summary of the measures).

Table 4.1: Descriptives of Contribution Evaluation Signal Measures (Pre-transformation)

Measure	mean	median	stdev	skew
Test Inclusion*	0.151	0	0.358	1.95
Commit Size (lines)	1456	25	27799	61.876
Files Changed	13.265	2	165.46	67.691
Social Distance*	0.096	0	0.295	2.74
Prior Interaction	200.583	22	566.388	8.184
Comments	2.664	1	6.656	19.198
Followers	35.972	7	177.082	22.965
Collaborator Status*	0.435	0	0.496	0.261
Repo Maturity**	2.104	1.956	1.188	0.568
Collaborators	20.203	8	42.808	6.063
Stars	1981	293	4095	2.977
Pull Req Acceptance*	0.723	1	0.447	-0.999

\*Dichotomous variables

\*\*In years as of July 17, 2013

## Outcome Measure

The main outcome measure was whether or not a pull request is accepted. Pull request acceptance in this context means that the code contributions included in the pull request were merged into the project's code base. Pull request acceptance is a dichotomous variable.

## Pull Request-level Measures

For the base level of measurement, I collected information unique to each closed pull request in our dataset. Each measure for the pull request represents a social or technical signal about the contribution that may factor into the acceptance decision.

### *Technical Contribution Norms*

I use three measures to operationalize different dimensions of valued technical contribution norms for a pull request.

**Test Inclusion** – This measure was a dichotomous variable indicating whether or not the pull request included test cases. The prior work of Dabbish et al. [2012] on GitHub suggests that when core members evaluate pull requests, they look for the inclusion of test cases as a signal of the thoroughness of the contribution. To measure this, I looked at the file pathnames in each pull request and looked for the word “test”. If the pull request included such a pathname, then the pull request is labeled as including tests. This is due to most test cases either residing in a test folder (i.e. project/test/...) or the filenames including the word “test” (i.e. test\_numberformat.java). To verify, a simple spot-check was performed on forty randomly chosen pull requests, twenty labeled as having tests, twenty labeled as not having tests. All checked pull requests were found to be correctly labeled. Of course, this measure is probably conservative, with unfound false negatives.

**Commit Size** – This measure is the number of lines changed in the pull request. Along with number of files changed, I included the number of lines changed in a pull request as a signal of a pull request's legibility. Pull requests that change large portions of the code base at a time are much harder for project managers to understand and evaluate.

**Number of Files Changed** – This measure is the number of files changed in the pull request. Along with the commit size, we use these measures to indicate how legible a particular pull request is. Pull requests that touch a large number of files tend to be much harder to understand and evaluate for project managers [Marlow et al., 2013].

### *Social Connection*

To represent two different dimensions of the social connections in GitHub, I used a mea-

sure for social distance and another for prior interaction.

**Social Distance** – This measure was a dichotomous variable indicating whether or not the submitter follows the user that closes the pull request. I use this as a proxy of the social closeness between the submitter and the closer in a particular pull request.

**Prior Interaction** – Prior work on GitHub by Dabbish et al. [2012], indicates that core members for a project, especially when attempting to recruit new members, use prior contributions as a signal of the trustworthiness of a contributor and contribution. To measure prior interaction, I counted the number of events before a particular pull request that the user has participated in for this project. Events include participating in issues, pull requests, and commenting on various GitHub artifacts.

#### *Highly Discussed Contributions*

**Comments on Pull Request** – Marlow et al. [2013] found that uncertain pull requests tended to require negotiation and/or explanation. Pull requests with lots of comments also tended to signal controversy [Dabbish et al., 2012]. To measure the level of discussion, I counted the number of comments in the closed pull request.

### **User-level Measures**

As each pull request has a submitting user that may submit multiple pull requests to a project, I grouped pull requests by the submitting GitHub user account and collected information about each GitHub submitter.

#### *Submitter's General Community Standing*

**Followers** – This measure is the number of followers a GitHub user has at time of data collection. The number of followers a GitHub user possesses is used as a signal of standing [Dabbish et al., 2012] within the community. For example, users with lots of followers were treated as local celebrities. **Submitter's Status in Project**

**Collaborator Status** – This signal is a dichotomous variable for the user's collaborator status within the project. In GitHub, a collaborator for a project has direct commit access to the repository. Therefore, they do not need to perform the pull request process in order to merge code contributions into the project. However, interviews with GitHub users indicate that many collaborators opt to create pull requests for code contributions despite having commit status. Often, this is done to allow other users to review changes before accepting the code contribution.

## Repository-level Measures

I further grouped the dataset by grouping each set of submitters into a repository and collected information about each repository.

### *Project Establishment*

I used three different measures to represent three dimensions of establishment for the project receiving the pull request.

**Repository Age** – This measure is a continuous variable representing the project’s age how long a project has existed on GitHub since the time of data collection. I use this as an indicator of the repository’s maturity.

**Collaborators** – This measure is the number of collaborators on a project. I use the number of collaborators as a proxy for the relative size of the development team involved in a particular GitHub project.

**Stars** – This measure is a continuous variable for the number of stars on a project. When evaluating projects, GitHub users make use of the number of stars as a signal for community interest in the project [Dabbish et al., 2012]. As stars were indications of attention from a user to a particular project, more stars indicate more users interested in the project. Measures such as the number of forks and the number of contributors to a particular GitHub project were highly correlated with this measure and were omitted to avoid collinearity.

## 4.2.3 Analysis

Using these pull request-level, submitter-level, and repository-level measures, I create a model that predicts the likelihood of pull request acceptance. I fit a multi-level mixed effects logistic regression model to our data because our outcome variable (acceptance) is dichotomous and our dataset nested in multiple levels. I chose a logistic regression approach in order to better predict our dichotomous outcome variable. To account for the three-level nesting of the dataset from pull requests to users to repositories, I created a mixed model where our contribution measures are fixed effects and the unique user and repository intercepts are represented as random effects. I used a R [R Core Team, 2013] package [Bates et al., 2013] that accounts for cross-classification of data, as 28,880 out of 95,720 users appear in multiple projects in our dataset. None of the measures had pairwise correlations above 0.6 suggesting no multicollinearity problems [Dormann et al., 2013]. To ensure normality, each of the continuous variables in the model was log transformed and then centered such that the mean of each measure is 0 and standard deviation is 1.

## 4.3 Results

My analysis suggests that both technical and social contribution measures are highly associated with acceptance. First, I examine our hypotheses and how each predictor variable associates with acceptance. I also consider signals that cut across pull requests such as user-level and repository-level measures. I report measure associations with contribution in odds ratios, which are the increase or decrease of the odds of acceptance occurring per “unit” of the measure. In this case, a “unit” of each measure is one standard deviation from the log-transformed for continuous variables or the presence of a dichotomous variable. Odds ratios provide a convenient way to compare association strengths across measures. A summary of the models is presented in Table 4.2.

### 4.3.1 Pull Request-Level Measures

#### *Technical Contribution Norms*

**H1:** Contributions that follow technical contribution norms are more likely to be accepted.

I tested H1 by examining the association of test case inclusion, commit size, and files changed with contribution acceptance. The inclusion of test cases was positively associated with pull request acceptance, with acceptance likelihood increased by 17.1% when tests are included. Lines changed had a stronger effect but negative, with each unit of lines changed decreasing the chance of acceptance by 26.2% compared to 7.3% with each unit of files changed. As I expect contributions that include test cases and are more legible are more likely to be accepted, so I find support for H1.

#### *Social Connection*

**H2:** Contributions from submitters with a stronger social connection to the project are more likely to be accepted.

I tested H2 by examining the association of social distance and prior interaction with contribution acceptance. I find support for H2 as both of our social connection measures were positively associated with pull request acceptance. Our measure of social distance had the strongest influence on likelihood of acceptance as compared with other pull-request level factors, increasing acceptance by 187% when the submitter follows the project manager. Prior interaction was also positively associated with acceptance, increasing acceptance likelihood by 35.6% per unit.

#### *Highly Discussed Contributions*

**H3:** Contributions with a high amount of discussion are less likely to be accepted.

Table 4.2: Multi-level Mixed Effects Logistic Model for Pull Request Acceptance

Factor	Variable	Pull Request Level			Pull+Submitter Level			Pull+Submitter+Repo Level		
		Model I	Model II	Model III	Model I	Model II	Model III	Model I	Model II	Model III
	(Intercept)	2.934***	2.898***	2.845***	2.934***	2.898***	2.845***	2.934***	2.898***	2.845***
Technical Contribution Norms (H1)	Test Inclusion	1.059***	1.023*	1.114***	1.059***	1.023*	1.114***	1.059***	1.023*	1.114***
	Commit Size	0.849***	0.834***	0.736***	0.849***	0.834***	0.736***	0.849***	0.834***	0.736***
	Number of Files Changed	1.165***	1.152***	0.970***	1.165***	1.152***	0.970***	1.165***	1.152***	0.970***
Social Connection (H2)	Social Distance	1.345***	1.461***	3.636***	1.345***	1.461***	3.636***	1.345***	1.461***	3.636***
	Prior Interaction	1.423***	1.362***	1.207***	1.423***	1.362***	1.207***	1.423***	1.362***	1.207***
High Discussion (H3)	Comments	0.481***	0.480***	0.414***	0.481***	0.480***	0.414***	0.481***	0.480***	0.414***
	<i>Test Inclusion x Comments</i>		1.057***	1.092***		1.057***	1.092***		1.057***	1.092***
Decision-Making for High Discussion (H4)	<i>Commit Size x Comments</i>		1.101***	1.166***		1.101***	1.166***		1.101***	1.166***
	<i>Files Changed x Comments</i>		1.017***	1.043***		1.017***	1.043***		1.017***	1.043***
	<i>Social Distance x Comments</i>		0.806***	0.792***		0.806***	0.792***		0.806***	0.792***
	<i>Prior Interaction x Comments</i>		1.106***	1.246***		1.106***	1.246***		1.106***	1.246***
Status in Community (H5)	Followers			1.060***			1.060***			1.060***
Status in Project (H6)	Collaborator Status			3.904***			3.904***			3.904***
Project Establishment (H7)	Repository Age			0.820*			0.820*			0.820*
	Collaborators			0.954*			0.954*			0.954*
	Stars			0.648*			0.648*			0.648*
AIC:		633600	630879	506850	633600	630879	506850	633600	630879	506850
				461077			461077			461077

To test H3, I examined the association between pull request comment count and acceptance. Pull requests with longer discussion, as indicated by higher counts of comments, were less likely to be accepted, supporting H3. This is the second strongest effect among the pull request-level factors, with the likelihood of acceptance decreasing by 54.6% with each unit of comment count.

#### *Decision-Making for Highly Discussed Contributions*

**H4:** Acceptance of highly discussed contributions will be moderated by both social and technical factors.

To test H4, I added an interaction term to the model, interacting number of comments with each pull request-level measure in order to investigate how social and technical factors moderated the decision-making process for highly discussed contributions. I found that all five interactions with social and technical factors were significant, indicating support for H4.

I provide charts detailing the direction of the interactions in Figures 4.1, 4.2, and 4.3. The associations of test inclusion, number of files, commit size, and social distance all significantly moderate the influence of discussion on contribution acceptance, though with a small effect. Prior interaction most strongly moderates the relationship between discussion and acceptance, with number of comments having almost no influence on acceptance for previous contributors. I discuss later the implications of these interactions for how evaluating highly discussed contributions may differ from more standard contributions.

### **4.3.2 User-Level Measures**

#### *Submitter Status in General Community*

**H5:** Contributions from submitters with a high status in the general community are more likely to be accepted.

To test H5, I examined the association of follow count with pull request acceptance. I find a positive association, supporting H5. Having followers increases the likelihood of acceptance by 18.1% per unit of followers. This suggests that submitters with higher community standing are more likely to have their pull requests accepted.

#### *Submitter Status in Project*

**H6:** Contributions from submitters that hold higher status in a specific project are more likely to be accepted.

I tested H6 by examining the association of collaborator status with contribution accep-

tance. Perhaps unsurprisingly, when submitters with commit access choose to create pull requests instead of directly merging code, their pull requests are more likely to be accepted than non-collaborators, supporting H6. Being a collaborator on a project increases the likelihood of contributions being accepted by 63.6%.

### 4.3.3 Repository-Level Measures

#### *Project Establishment*

**H7:** Contributions to established projects are less likely to be accepted.

I test H7 by examining the association of our project establishment measures (the age of the project, number of users with commit status, and popularity of the project) with contribution acceptance. All three of our project establishment dimensions have negative associations with pull request acceptance, so I find support for H7. Number of collaborators, used as a proxy for project team size, has the smallest influence on acceptance likelihood out of the three establishment measures, decreasing acceptance by 4.6% per unit of collaborator count. Somewhat surprisingly, this suggests that project “size” does not have as strong an influence on pull request acceptance as compared with age or popularity. The older a project, used here as a proxy for maturity, the less likely it is to accept pull requests, with acceptance decreasing by 18.0% per unit of project age. Popularity had the strongest negative influence on acceptance, with projects 35.2% less likely to accept pull requests per unit of increase in stars.

## 4.4 Discussion

In this section, I summarize the results and discuss the implications of the hypotheses in terms of prior literature.

### 4.4.1 Technical Norms and Social Connection

From conventional wisdom on open source software projects, I expect to see some evidence that “code is king” and that technical contribution norms should reign over other signals when considering contributions [Scacchi, 2007]. However, in the transparent environment of GitHub, I also expect contributors to make use of the social connections that the environment makes salient. The analysis suggests that while following technical

contribution norms for pull requests is associated with acceptance, the social connections behind pull requests have even stronger associations.

In terms of technical contribution norms, I found that pull requests more consistent with community-wide pull request practices like inclusion of test cases and small commit sizes [Dabbish et al., 2012] were more likely to be accepted. Code contributions that did not follow technical norms were less likely to be accepted, perhaps due to the higher assessment costs required by the project manager.

I also find that social connections increase likelihood of contribution acceptance, even when controlling for compliance with technical contribution norms. In traditional open source software projects, contributors are often expected to participate in more social aspects of the project such as participating in mailing list technical discussions before making code contributions in order to learn project-specific norms and ease socialization [von Krogh et al., 2003]. In the case of GitHub however, this expectation may be less prevalent because the pull request system standardizes the contribution process. The pull-request process also lowers the barriers for contribution, meaning many developers will make one-off contributions to projects or “drive-by commits” [Pham et al., 2013]. However, I still found that a contributor that has prior interaction with a project also has a higher likelihood of pull request acceptance. I also find that submitters socially closer to project managers tend to have their contributions accepted. This social distance association is also the strongest in the model. Similar to evaluating technical contribution norms, stronger social connections may indicate qualities such as trust, which may lower the project manager’s assessment cost. For example, if the submitter is trusted to make good contributions, project managers may be more lenient in their evaluations [Pham et al., 2013].

While both technical contribution norms and social connections were associated with pull request acceptance, our measures for social contribution had much stronger associations than our technical contribution norm measures. This difference in signaling theory is related to the construct of [Connelly et al., 2010]. Receivers, developers in this case, may apply weights to signals based on their own preconceived notions of importance. One possible explanation for the difference in weighting is that when project managers are evaluating pull requests, when the evaluation cost is too high, they may decide to outright reject the contribution. Whereas pull requests that follow technical norms such as legible code changes and test cases make the pull request much easier to evaluate, a strong social connection between the project manager and submitter may allow the project manager to bypass much of the evaluation process. Pull requests from unknown developers may be subject to much more thorough and costly evaluations from project managers than pull requests from known contributors [Pham et al., 2013]. For example, a familiar developer may be expected to already have run the contribution through the test suite, allowing for

a project manager to bypass that phase of the evaluation, increasing the likelihood of acceptance. Similarly, members of the project with commit rights, perhaps an explicit form of trust, also have positive associations with acceptance. This may be similar to the effect of familiarity in distributed software development, where team familiarity is associated with team performance, especially for geographically dispersed teams (as many GitHub projects are) [Espinosa et al., 2007]. One explanation for the familiarity finding is that teammates with high familiarity know whom to contact for queries and resources, making coordination much more efficient. Perhaps submitters with a strong social connection also lower the coordination costs required to use the contribution. For example, project managers familiar with the submitter may not bother to look for project-specific coding style norms, knowing that the submitter already should know them.

Future research should examine in more detail how technical and social signals influence evaluation cost during pull request acceptance. If technical norm signals are harder to evaluate, then perhaps future collaborative software tools should focus on lowering the evaluation cost of a software contribution. At the same time, if developers are using social signals to evaluate contributions, then perhaps those signals should be made more visible during evaluation tasks, assuming that these signals are optimal for decision-making. Future research should also examine whether these evaluation decisions are optimal or what leads to optimal acceptance decisions.

#### **4.4.2 Decision-Making and Highly Discussed Contributions**

Next to social distance, the amount of discussion around a pull request had the strongest influence on likelihood of acceptance. The more highly discussed the contribution, the less likely the contribution would be accepted. This by itself is not too surprising, given the high degree of uncertainty present in such pull requests [Marlow et al., 2013]. However, I also hypothesized that highly discussed pull requests differ in both the tone of the discussion and the degree of uncertainty in the contribution being discussed. These differences in the nature of the discussion around evaluating a pull request also reflect differences in the decision-making process for project managers. When discussing a pull request in order to evaluate the value of the contribution, project managers may be using different kinds of information. The model finds (see Table 4.2) that both technical contribution norms and social connection measures moderated the effect of discussion on contributions.

For highly discussed contributions, the social and technical pull request-level measures moderate the negative association of discussion amount on acceptance. For most of our factors, however, regardless of being social or technical in nature, the moderating effect is too small to affect the very negative influence of having a large amount of discussion in a

contribution. In Figure 4.1 for example, the negative effect of high discussion overwhelms the positive technical effect of test inclusion, reducing the likelihood of acceptance by about 30% regardless of test inclusion. Even for the variable with the largest association with acceptance, social distance, having a high amount of discussion still reduces the likelihood of acceptance by about 25% regardless of whether or not following occurs as seen in Figure 4.2. This small moderating effect suggests that for most pull requests, project managers are much less willing to accept the contribution, regardless of whether or not technical contribution norms are followed or a social connection of the submitter to the project manager exists. This may indicate that regardless of the tone of the discussion or nature of the contribution, high amounts of discussion on a pull request indicates a high degree of uncertainty for the value of the contribution.

However, a submitter's prior interaction on the project significantly changes the influence of discussion on acceptance as seen in Figure 4.3. Surprisingly, there is a positive association between discussion and acceptance likelihood for participants with prior interaction. This may indicate that when experienced submitters are working on a project, the nature of the discussions around their pull requests is different in some way than submitters who do not have this prior experience. Discussions where the submitter has high amounts of prior interaction may be less focused on evaluating a contribution's value and more focused on optimizing the code. Conversely, the discussion around a contribution from a submitter with no prior interaction on the project may focus more on evaluating whether the pull contribution is worth accepting. For example, a submitter with no prior interaction may be unaware of a project's submission practices and the resulting discussion would be focused on ensuring the pull request matches the project's standards.

Interestingly, the moderation effect of prior interaction is at odds with the effect of social distance despite both variables being used for our social connection measure in the analysis model. This may suggest that when discussing contributions, project managers will turn to prior interaction rather than social distance as a signal to use during evaluation of pull requests. Perhaps this occurs because prior interactions are a more trustworthy signal than the social distance signal of the submitter following the project manager. To demonstrate prior interaction, a user has to actively participate in discussions, bug reports, and other forms of contribution on the project. Prior interaction may act as an assessment signal, where the signal of prior interactions cannot easily be generated without actual participation [Donath, 2005]. Prior interaction is a reliable signal of social connection because participation cannot be easily faked. On the other hand, social distance via following may indicate a social connection between two users through convention. This signal is less reliable because a submitter can follow a project manager without actually creating a social connection with the project manager. When discussing how to evaluate contributions,

the convention of following users does not replace familiarity built from actual prior interaction.

I examine these highly-discussed pull requests in more detail in Chapter 5.

Future research should examine how we can design tools that assist in deliberation by highlighting certain information. Future tool design may assist developers during software change evaluation discussions by making certain signals more or less visible. Future tools may even dynamically change the visibility of different signals depending on the tone of the discussion.

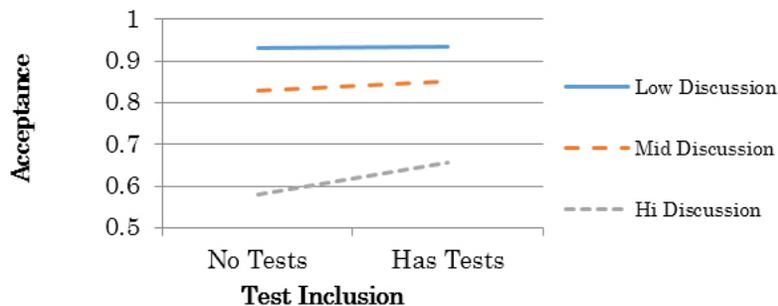


Figure 4.1: Interaction Plot of Test Inclusion and Contribution Discussion

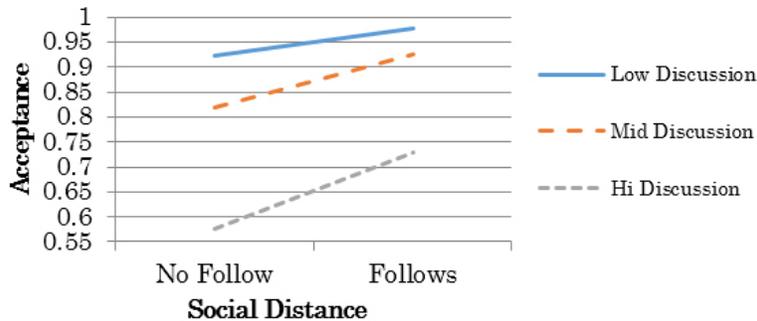


Figure 4.2: Interaction Plot of Social Distance and Contribution Discussion

### 4.4.3 Audience Pressures

While social and technical features of pull requests had important associations with acceptance, the model also suggests that the type of submitter and the type of project that the pull request is submitted to also influences acceptance likelihood.

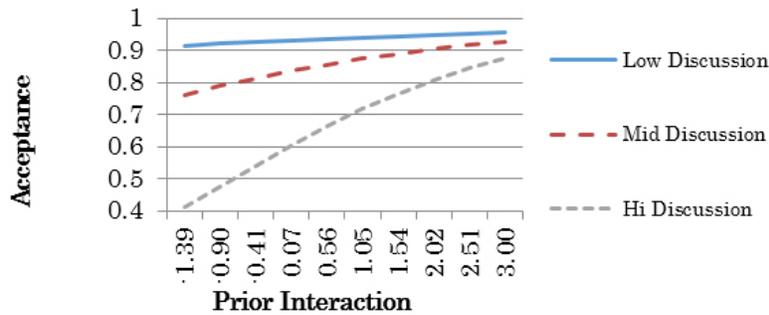


Figure 4.3: Interaction Plot of Prior Interaction and Contribution Discussion (prior interaction values are standardized)

Pull requests from submitters who have commit rights, known as collaborators in GitHub, were associated with acceptance. Pull requests from collaborators seem to be special cases of contribution because these users are not required to undergo the pull request process in order to have their changes merged into the project, unlike other developers.

Well-established projects were negatively associated with acceptance on all three dimensions. In particular, the popularity of a repository has the strongest negative association out of the three. Number of stars, a proxy for project popularity, is used by members of the GitHub community as a signal for project quality, which project managers are aware of [Dabbish et al., 2012].

The contrasting associations between popular projects and collaborators may indicate that audience pressure is a factor when project managers evaluate pull requests. For popular projects, the transparent nature of GitHub means project managers are aware, at least in part, of the identity of users of their project [Dabbish et al., 2012]. Knowing that hundreds or thousands of users, some highly visible, depend on a particular project may discourage project managers from accepting risky or uncertain code contributions. Conversely, collaborators, who possess the ability to accept pull requests into the project, may be immune to these audience pressures.

The effect of audience pressure on software contribution evaluation is not well understood. Future research may investigate more thoroughly how audience pressures affect both contributors and core members of projects. Signals used to evaluate contributions may differ depending on whether or not core members feel pressure from the audience. For example, core members may be much more concerned about managing uncertainty when they are aware that millions of potential users are watching and depending on the project being stable.

#### **4.4.4 Limitations**

One of the main limitations of this study is that most of the data is of a cross-sectional nature. At the same time, some of the measures are more robust to reverse-causality because of timing inherent in the pull request process. Prior interaction, test inclusion, and number of lines and files changed, are all variables whose value is determined prior to any consideration of acceptance of the pull request. Other variables, however, are cross-sectional at the time of data collection, such as follower count. Without performing a true longitudinal analysis, the direction of causality cannot be determined for these latter variables using our dataset. Future work should perform longitudinal analyses on contribution measures in order to make stronger inferences about causality.

### **4.5 Conclusion**

This study examines potential social and technical signals available through the transparent environment of GitHub and their relationship with contribution evaluation decisions. I created a statistical model analyzing the association of different pull request, submitter, and repository measures of contributions with the likelihood of the contribution being accepted. The study found that project managers made use of information involving both the technical contribution practices of a pull request and the strength of the social connection between the submitter and project manager when evaluating pull requests. Highly discussed pull requests were much less likely to be accepted, however, the submitter's prior interaction in the project moderated this effect. The discussions around these pull requests are examined in more detail in Chapter 5. Well-established projects were more conservative when evaluating pull requests, perhaps due to audience pressures.

This study's findings inform ways that software developers and project managers might make use of information in transparent environments and imply a variety of ways that transparency feature can support software development. This study identifies signals with strong associations to pull request acceptance that are potentially useful to guide both contribution submissions and evaluations. Surfacing strongly associated signals such as social connection and prior interaction in future tools may help guide project managers in making evaluation decisions. The strong association of social signals may also indicate receiver interpretation [Connelly et al., 2010] or preferential weighting of these signals by project managers. This weighting may suggest that social signals are more useful, for better or worse, for this coordination task.



# Chapter 5

## Negotiating Contributions through Discussion in GitHub

### Chapter Summary

Extended discussions around contributions are a key coordination mechanism to both evaluate the contribution and ensure that the needs of other developers and projects are met. These discussions have important implications for project management regarding contributors and the evolution of project requirements and direction. This chapter describes a study of the coordination that occurs between developers in transparent environments as they discuss pull requests. This study analyzes a sample of extended discussions around pull requests and supplements the data with interviews with GitHub developers. I found that developers raised issues around contributions over both the appropriateness of the problem that the submitter attempted to solve and the correctness of the implemented solution. Both core project members and third-party stakeholders discussed and sometimes implemented alternative solutions to address these issues. Different stakeholders also influenced the outcome of the evaluation by eliciting support from different communities such as dependent projects or even companies. The level of a submitter's prior interaction on a project changed how politely developers discussed the contribution and the nature of proposed alternative solutions. These findings suggest the potential importance of political signals for projects, such as the influence of stakeholders and their associated projects or companies.

<sup>0</sup>For the full paper describing this study, please see Tsay et al. [2014b], published in Foundations of Software Engineering (FSE) 2014.

---

Though the open contribution model of open source software allows many people with diverse expertise to add their unique value to a project, this openness also brings the risk of destabilizing a project with error-ridden or even malicious code contributions. When contributions are deemed unsuitable or threaten technical integrity, a negotiation between contributor and project members often ensues [Mockus et al., 2002]. This negotiation often takes place in the form of extended discussions around the contribution. While the average GitHub contribution generates little if any discussion and involve very few developers [Gousios et al., 2014], cases of extended discussions occur as project members and the broader community work together to understand the implications of the suggested change [Marlow et al., 2013]. These extended discussions are a key coordination mechanism in open source software projects for both evaluating contributions and allowing participants to voice their concerns.

The coordination that occurs during these extended discussions often involves negotiations between the participants as changes to a project often have major impacts on other related projects. These related projects operate independently and may have their own, sometimes conflicting, goals [Crowston et al., 2008]. For example, a submitter may offer a contribution to a project that modifies the behavior of some feature. Even if this change is useful, downstream projects may rely on the old behavior. In the discussion around this contribution, the submitter's goal may be to convince the core team that this change is beneficial [Ducheneaut, 2005] while the core team's goal may be to ensure compatibility with downstream projects. These problems and issues identified during negotiations reveal unique challenges of coordinating with a heterogeneous diffuse set of contributors that may possess their own goals. In an open setting, project members have little or no formal authority over contributors and vice versa [Lerner and Tirole, 2002]. Left unchecked, these challenges may cause undesirable outcomes for discussions and the contributions they are tied to. For example, conflicting goals between developers may escalate to emotional disagreements or affective conflict which may surface as ugly personal attacks [Arazy et al., 2013]. The consequences may be especially far-reaching in transparent environments, as contributions and discussions are visible to all, including any such emotional conflicts.

By better understanding the methods and means participants use to negotiate problems in contribution discussions, I may be able to inform policies and tools that enable developers to better manage open projects, coordinate between projects, and resolve potential conflicts. For example, if problems frequently arise around contributions where developers have conflicting goals, then collaborative software development environments can

incorporate specialized conflict management tools [van Wendel de Joode, 2004]. Notification mechanisms in transparent environments (e.g., watching, following, mailing list subscriptions) can be more precisely targeted to developers in downstream projects whose interests are potentially affected.

To understand the negotiations that occur in extended contribution discussions, I studied discussions on pull requests in GitHub. Average-case pull requests typically generate little to no discussion [Gousios et al., 2014] and therefore are not very informative about the reasons for acceptance or rejection. For this reason, I chose to focus on only pull requests with extended discussions, which often reveal the developers' reasoning process. Using a grounded theory approach [Corbin and Strauss, 2008] I focused on the phenomenon of extended contribution discussions (rather than interactions around contributions more broadly) because these discussions are an important aspect of the open collaboration process and a key place where core and peripheral members negotiate around aspects of project evolution and direction.

I explored the kinds of issues core developers raised and the arguments they over both the appropriateness of the problem that submitters attempted to solve, and the correctness of the implemented solution in a submitted code contribution. Due to the open nature of the software projects, other stakeholders from outside the project observed and participated in these extended discussions, sometimes attempting to influence the outcome of the contribution through rallying support of the audience or leveraging project or company communities. I also found that non-member contributions were more likely to be rejected following a long discussion. However, although core teams rejected new submitter's contributions more often, they almost always satisfied the submitter's technical goal by implementing an alternative solution. Core members interacted more politely with new submitters and in cases of conflict, were more likely to implement alternative solutions for these newcomer submitters rather than simply suggest them.

While this study did not specifically explore signals that developers used during these extended discussions, I found that participants used signals in their environment to infer qualities about other participants. These qualities tended to be political in nature, such as inferring the influence of a project or company participating in a discussion. These political qualities and their signals may be useful for developers who wish to evaluate projects or changes. For example, a future transparent environment might display to a submitter what impact their change may have on downstream projects.

In the next sections I motivate my research questions based on previous research, describe the study methodology, present the results of the discussion analysis, and discuss the implications of the findings.

## 5.1 Contribution and Discussion in Online Work

Previous research on suggests complex social and technical processes for evaluating contributions in online communities and open source software projects. In transparent development environments such as GitHub, developers augment these processes by making use of the information transparency makes visible. Informed by previous work, I develop research questions around how developers discuss contributions in open environments.

### 5.1.1 Discussions around Contributions in Online Communities

Successful online communities rely on members contributing their unique resources to the community, such as users uploading videos on YouTube or posting pictures or comments on reddit. In peer-production communities such as Wikipedia, conversation is used to organize work. Viégas et al. [2004] found that in Wikipedia editors primarily used edit comments to coordinate article edits. Feedback and discussions around contributions may encourage future contributions from newcomers. Kraut and Resnick [2012] analyzed challenges that online communities face when trying to encourage contribution: matching users to contributions needed, making requests to members, using intrinsic and extrinsic motivators, and grouping users together. They review evidence showing that constant feedback to members, whether it be character levels in World of Warcraft or community comments in YouTube, motivates members to create more contributions. Similarly, Burke and Kraut [2008] found that a higher perceived politeness increases reply rates in online communities.

However, newcomers may sometimes come into conflict with experienced users. In particular, Kittur et al. [2007a] identified "elite" and "common" classes of users in Wikipedia where much of the work is shifting from experienced users to newcomers in the "common" class. Along with shifts in workload, Forte and Bruckman [2008] found that governance in creating and enforcing policy is also shifting from a small group of administrators to a more decentralized model. They found that Wikipedia in particular relies on policy as one of the main governance mechanisms. With an explicit community norm against voting, creating policy requires discussion to build consensus. Similarly, although administrators hold both technical and social authority, decisions by administrators are not enforced without widespread support from the larger community. As consensus is often required to resolve conflicts, Kriplean et al. [2007] found that users will engage in political maneuvering (termed "power plays" in the work) to influence the larger community. These political strategies include arguing or redefining the scope of the article, referencing past policy or consensus in other articles, pointing to past work as an appeal to legitimacy, and disputing

the legitimacy of sources. "Elite" users such as administrators made use of their technical or social authority in some cases by threatening sanctions through formal sanctioning or arbitration mechanisms, overriding competing interpretations through greater authority, and threatening to leave articles. During these conflicts, newcomers and experienced users often resolve conflicts through discussions. The nature of the discussions and feedback may discourage newcomers. In a study of newcomer contribution on Wikipedia, Halfaker et al. [2011] found that reverts decreased motivation for newcomers. Reverts from experienced editors were the most demotivating, suggesting that certain interactions around contributions may have a particularly negative influence on motivation to contribute to a project.

### **5.1.2 Discussions around Contributions in Open Source Software**

As open source software often relies on the contributions of a diverse group of software developers [Crowston et al., 2008], members of software project teams must evaluate and discuss contributions to ensure the integrity of the software project.

Literature on the contribution process for open source software projects suggests that evaluating contributions, especially from unknown developers, is a complex social process. von Krogh et al. [2003] found in their study of the contribution process in the Freenet open source project that successful newcomers must follow "joining scripts" before submitting a contribution. These joining scripts involve participating in prior activity such as lurking on the project's mailing list, participating in technical discussions, and reporting bugs. They also found differences in the tone of discussion between developers who were invited to join the project versus developers who were not. For example, the detail and specificity of feedback given was much more general for non-joiners. Ducheneaut [2005] noted that developers looking to make successful contributions to the Python project needed to undergo a progressive socialization process. Core members on a project would vet contributions to ensure the code changes were technically sound. Successful socialization allowed potential submitters to learn project norms and to identify key members of the core project team. In order to successfully start the contribution evaluation process, a submitting developer needed to "recruit" core members of the project as a network of "allies".

When evaluating code contributions for technical correctness, core project members often use a peer review process. Rigby et al. [2008] found in their examination of different peer review processes in the Apache server open source project that early and frequent reviews of small contributions from the core team were effective in finding defects in contributions. In particular, the usage of the project mailing list allowed for self-selection

of expert core members and a more open discussion between members. Ko and Chilana [2011] found that discussions around bug reports established scope, proposed ideas, identified design dimensions, defended claims with rationale, moderated the process, and finally made a decision. The most powerful factors in decision-making around a bug report were the participant's authority (developers over users) and actions taken (writing a patch).

### **5.1.3 Social Signals in Transparent Development Environments**

While evaluating contributions is a key process in all popular open source projects, the information that transparency makes visible allows developers to make use of additional signals during the evaluation process. Chapter 4 presents some examples of possible signals that are explored in more depth in this study, such as highly discussed contributions that are much less likely to be accepted and the moderating effect of prior interaction on highly discussion contributions [Tsay et al., 2014a].

Marlow et al. [2013] found that when evaluating a developer's contributions, project managers would look to the submitter's other projects in order to better understand how much assistance or extra effort the submitter would require in order to accept their contribution. Questionable changes required back-and-forth discussion between the submitter and core members in order to explain why the contribution could not be automatically accepted and negotiate the outcome. In these cases, project managers would make use of information about the code contribution and the submitter, to decide how accommodating towards the submitter they should be. For example, a project manager may weigh the cost of fixing a contribution against the benefit of recruiting a new member to the project.

The literature on deliberation in online communities suggest that members engage in discussion to both encourage and evaluate contributions to the community [Dabbish et al., 2012, Kraut and Resnick, 2012, von Krogh et al., 2003]. Open source software projects, needing to ensure the technical integrity of code contributions, engage in complicated social processes and peer technical reviews [Mockus et al., 2002]. Often, developers would also engage in discussion in order to socialize themselves when joining a project [Dabbish et al., 2012, Kraut and Resnick, 2012]. Transparent work environments such as GitHub have developers using information made visible due to transparency to make inferences about projects and other developers when evaluating contributions [Crowston et al., 2008, Lerner and Tirole, 2002]. However, we still know relatively little about the kinds of issues that arise, and the nature of discussions developers have when evaluating contributions.

## 5.1.4 Development of Research Questions

Our examination of literature on discussion around contributions in online environments suggests a number of research questions to advance our knowledge of how software developers coordinate in transparent development environments through extended discussions.

From online communities such as Wikipedia [Viégas et al., 2004], we see that editors engage in discussion over conflicts in article direction. In open source software, developers discuss problems in bug reports, making and justifying arguments when discussing the design of a solution [Ko and Chilana, 2011]. For more uncertain changes in GitHub, core members engage in back-and-forth discussion to justify the value of the contribution [Marlow et al., 2013]. However, it is not well understood what issues around open source software development need to be worked out in these discussions. By better understanding the issues, arguments, and criteria raised in contribution discussion, we can identify challenges in coordinating with a diffuse set of contributors in an open environment. This leads to the first research question:

**RQ1:** What are the different kinds of issues raised around code contributions?

In online communities, members use intrinsic and extrinsic motivators when making requests to encourage compliance [Kraut and Resnick, 2012]. One tactic identified in open source software is that submitters recruit core members to assist in the evaluation process [Ducheneaut, 2005]. In this way, submitters are able to influence the outcome of the evaluation process. What is not well understood is the full range of methods that different stakeholders may use to influence the evaluation process. By better understanding the decision-making process and how influence is brought to bear, we also gain insight into what motivates software developers to accept changes. The influence tactics used in an environment where there is little formal authority [Lerner and Tirole, 2002] also reveal what constitutes power in open collaboration. This leads to the second research question:

**RQ2:** How do participants try to influence the decision process in code contributions?

In online communities, the outcome of a contribution evaluation may be farther-reaching than simply whether the contribution itself is accepted. For example, the outcome of a contribution and the identity of the evaluating editor in Wikipedia has an impact on the motivation of a new editor to contribute again [Halfaker et al., 2011]. In open source software environments such as GitHub, project managers may decide to accept less desirable code contributions in order to recruit new members [Marlow et al., 2013]. The following research question investigates the less obvious impacts of outcomes of code contribution evaluations:

**RQ3:** What are the different outcomes for proposed code contributions?

In Chapter 4, I found that a submitter’s level of prior experience on a project has an association with contribution acceptance and seems to moderate the negative effect of discussion [Tsay et al., 2014a]. This suggests that a submitter’s prior experience (or lack of) changes the nature of discussion around code contributions and may influence the outcome of the evaluation. In online communities, new submitters tend to interact differently. In Wikipedia, new editors tend to make peripheral, specific edits to articles [Bryant et al., 2005] and tend to be sensitive to reverts from experienced editors [Halfaker et al., 2011]. In open source software, new contributors tend to follow “joining scripts” before making successful contributions [von Krogh et al., 2003]. By understanding how a submitter’s prior experience impacts discussion, we may better understand how projects manage both new and experienced contributors. This leads to the fourth research question:

**RQ4:** Is discussion different when the submitter has prior experience with a project?

## **5.2 Method**

To study the coordination that occurs in discussions around contributions, I created and analyzed a dataset of both interview data and contribution discussions from the social open source software hosting site GitHub. The interviews explored the practices of a relatively broad sample of developers, while the content analysis of entire collections of comments for specific pull requests allowed analysis of complete exchanges and their outcomes in some depth. In this section, I present our data collection methods and analysis technique.

### **5.2.1 Data Collection**

The dataset consists of both a set of 423 comments from 115 developers, embedded in extended pull request discussions and interviews with 47 users of GitHub.

#### **Highly Discussed Pull Request Sample**

From a larger dataset of 659,501 pull requests across 12,482 GitHub projects [Tsay et al., 2014a], I created a sample of highly discussed pull requests (see Table 5.1). For this study, “highly discussed” is defined as pull requests where the number of comments is one standard deviation (6.7) higher than the mean (2.6) in the dataset, filtering out all pull requests with less than 9 comments in the discussion. Each pull request includes both discussion comments on the pull request itself and code-level inline comments. From

this reduced sample, 20 highly discussed pull requests were randomly selected from 20 different software projects. The list of pull requests is provided in Appendix A. From these 20 pull requests, a total of 423 comments from 115 developers were analyzed. As the analysis reached theoretical saturation, I drew no more pull requests from the sample. The sample is stratified to include both accepted and rejected pull requests, as well as submitters with varying levels of prior interaction with the project (see Table 5.2).

As the research questions are concerned with the outcomes of contributions, I ensured a roughly equal representation of both accepted and rejected pull requests. As this study also investigates how a submitter’s prior experience on a project changes the discussion around a contribution, I also ensured a roughly equal distribution of new submitters (no prior interaction) and experienced submitters (see Table 5.2).

Table 5.1: Description of Pull Request Sample

Number of pull requests	20
Total number of comments	423
Mean number of comments	21.1
Total number of participants	115
Mean number of participants	5.75

Table 5.2: Distributions of Pull Request Sample

	Yes	No
Pull Request Accepted?	9	11
Submitter Has Prior Interaction on Project?	11	9

## Interview Data

To supplement the sample of extended discussions, I also conducted a series of semi-structured interviews with 47 GitHub users [Dabbish et al., 2012]. The goal in these interviews was to document and understand in more detail the different ways GitHub functionality was used by our participants, including how pull requests were created and managed. Participants were solicited via email. Interviews were conducted in person or via phone or Skype. Remote participants shared their screen during the interview using Adobe Connect so users could demonstrate their activities on the site. Participants were asked to walk through their last session on GitHub, describing how they interpreted information displayed on the site as they reviewed earlier work activities. For this study, I focused

on participants describing their last pull request sent to a project and the last pull request received for their own project. None of the pull requests mentioned in interviews were included in the sample of pull request discussions. Interviews lasted approximately 45 minutes to one hour overall. These interviews were then transcribed verbatim to support further analysis.

## **5.2.2 Data Analysis**

This study uses a grounded theory approach to analyze how developers evaluate contributions in transparent environments for the sample of pull request discussions [Strauss et al., 1990]. I first identified instances of how developers evaluated code contributions in the comments of five pull request discussions. For each instance analyzed, I identified the participants involved, information made available by GitHub that is used by developers, the type of comment, what portion of the code contribution is referred to, and the higher-level goal of the participant in regards to the contribution. Then, open coding was performed on these examples, grouping examples into categories that were conceptually similar. This process revealed different categories of interaction between different types of participants for a code contribution. This first set of categories was used to code the remaining pull request discussions, revealing additional categories. I used an iterative process until the discussions no longer revealed new interactions not captured in the existing set of categories (theoretical saturation). During this process, I also identified similar interactions in the interview data and supplemented our findings using these examples.

## **5.3 Results**

The analysis found that both core and peripheral developers in a project engaged in discussion in order to resolve issues around both the problem that the contribution is attempting to solve and the solution that the contribution implements. Different stakeholders such as third parties and audience members sometimes attempted to influence the outcome of discussion. I found different outcomes for contributions and discussions around them. I also found that the submitter's level of prior interaction on the project changed the discussion around the code contribution.

### **5.3.1 Issues Raised Around Code Contributions**

**RQ1:** What are the different kinds of issues raised around code contributions?

Contributions to projects in the form of pull requests sometimes generated issues that the submitter and the core members must resolve through discussion. Core members raised different issues over the appropriateness of the problem that the contribution was attempting to solve. Developers also discussed how to optimize the solution that the contribution implements with various levels of involvement. At times, core members in the project also disagreed amongst each other over these issues.

### **Disapprove of the Problem Being Solved**

One main issue that developers discussed was whether the problem that the code contribution was trying to solve is appropriate. Core members would sometimes discuss whether the pull request belonged in their project while other times would ask contributors to prove the value of their contribution through explicit use cases.

#### *Project Appropriateness*

Core members questioned whether the submitter was using the project in the intended manner. Sometimes, contributions offered by submitters using the project inappropriately would attempt to implement features that were not in the intended scope of what the project was meant to do (P2, P7, P14, P15, P16, P19, P20). In these cases, core members offered alternative solutions to the submitters outside of their own project. One example had the project owner offer to assist the submitter in learning how to use the project correctly offline, in a local hackathon (P15). If the contribution was outside the scope of the project, core members sometimes suggested that the contribution actually be made to an upstream or downstream project (P20).

To prevent submitters from wasting time on inappropriate contributions, core members expected submitters to propose their contributions before implementation to get feedback on its appropriateness (P7). In cases when core members inadvertently accepted inappropriate earlier contributions into the project, later contributions would be necessary to revert the inappropriate change (P20), further increasing the time and effort wasted for both submitters and core members.

*“The idea of proposals issues before-hand is to see the likelihood of something getting merged, so you don’t feel you’ve wasted all your time if it doesn’t.” (P7)*

Some GitHub developers explained in interviews that submitters would sometimes inadvertently solve inappropriate problems because the project would move in a different direction unknown to the submitter. For example, a core developer’s planned changes to a project made a submitter’s contribution obsolete.

*”So, yeah. Not sure what’s gonna happen with this off the top of my head, if it’s gonna*

*get landed or– I mean because some of the things that we’re doing with this refactor of the master branch make this whole thing a little unnecessary now.”*

#### *Value Proposition Request*

In order to explore whether the contribution truly had value for the project, core members asked submitters to provide specific use cases or test cases (P1, P2, P3, P4, P13, P14, P20). Core members used this requirement as a way to confirm that the specific problem submitters were trying to solve in their contribution was appropriate. In some discussions, core members refused to continue the evaluation process until use cases were presented (P1, P3, P4, P14).

In response, submitters offered use cases or test cases to demonstrate the problem their contribution solved. For the contributions in this category, submitters that provided code examples or references to downstream projects (P2) tended to successfully prove the appropriateness of their problem and also tended to have their contributions accepted. Occasionally, third parties from the audience also jumped in, offering their own use cases when core members asked for them (P13). When submitters were unable to satisfactorily demonstrate use cases that their contribution solves, the contribution tended to be rejected. In some cases, core member simply closed the pull request until a test case was provided (P1). In one case, the submitter, uncertain about their own use case due to discussion, decided to close their own pull request until a better use case was presented (P14).

*“I think it may be better to close the pull and the associated issue unless I’m overlooking a real use case... I’m going to close this for now, if someone comes up with a good use case we can reopen.”* (P14)

#### **Disapprove of the Solution**

When core members and third parties from the audience questioned the solution that the pull request implemented, developers offered a gradient of responses to questionable solutions from passively questioning the submitter’s approach to actively suggesting alternative solutions to offering their own solutions to the problem.

#### *Question Solution Approach*

Core members raised objections to the way the submitter chose to implement the solution in the contribution. Most often, developers raised questions about the submitter’s approach to implementing the solution in the pull request (P2, P4, P8, P9, P10, P11, P17, P18, P20). In some cases, core members asked about design decisions that the submitter made when implementing the pull request (P2, P4, P8, P10, P18, P20). These decisions ranged from the forming of dependencies (P10, P20) to more elegant code (P4) to even

best commenting practices (P8). Other cases had core members act as testers for the code change, reporting bugs with the pull request (P11, P17). In one case, the core member actually reported the bug after the pull request was accepted (P17).

*“Wow. Don’t you think you’re going a little bit overboard with this many comments? Or is this just for my benefit when checking on your code?”* (P8)

#### *Suggest Alternative Solutions*

Some core members and third parties from the audience took a more active approach when the contribution’s implemented solution was suspect and suggested alternative solutions to the submitter’s implemented solution, often with the expectation that the submitter would implement the suggestion (P2, P7, P8, P9, P11, P15, P19). Many of the suggestions given were technical in nature, suggesting ways to improve the code through optimization (P9) or better practices (P11, P15) or avoiding bugs (P7, P8). Others were more stylistic in nature, suggesting changes to conform to best practices or project norms (P19).

*“I would suggest having an array of possible node locations and loop through them in order using `fileExists` to determine if it’s available.”* (P15)

Submitters sometimes followed and sometimes ignored suggested alternative solutions. In one pull request, a core member made a suggestion for an alternative solution that the submitter accepted and attempted to implement. Not being able to implement the suggestion, the submitter decided to leave the pull request as-is (P11). In some cases, the submitter actively rejected the suggested alternative solution. In one case, a third party developer from the audience suggested an alternative solution that the submitter, a core member, rejected with an explanation why (P9). With these two examples, regardless of the outcome, submitters addressed suggestions from the core or third parties. In one case, a new submitter even preemptively addressed an obvious alternative solution, explaining why it would not be appropriate for the particular problem that the contribution was trying to solve (P2).

*“You might ask: “why don’t you install the suggested `rb-inotify` gem to avoid getting that [...] warning?” The reason is that such a task can only be performed by the end user who uses my scripts; I have no control over their machines”* (P2)

#### *Advertise Own Solution*

As a response to issues in how the solution in the contribution was implemented, some core members or third party developers from the audience took the initiative to implement their own alternative solutions to the problem presented in the contribution and then advertise their own solution in the pull request discussion (P2, P3, P13, P15). The actual form of the alternative solution varied widely from case to case. Interested third party developers

from the audience gave examples of solutions to similar problems that were implemented in outside projects that the developers previously worked on and provided hyperlinks to that project (P2, P3). One third-party developer from the audience, in response to problems in the implemented solution, made suggestions for an alternative solution and then implemented the solution in another pull request (P3). This case created a competing solution to the same problem that the original contribution attempts to solve. In another case, a core member sent a pull request to the submitter's personal fork of the project that made code changes to the contribution, effectively making a contribution on a contribution (P13).

*"@[submitter] I sent you a PR([link to pull request]) that accounts for once in the callback, avoiding a potential infinite loop. Test included too."* (P13)

### **Disagreement among the Core**

In almost a third of cases in our sample of highly discussed pull requests, core members disagreed amongst themselves in regards to the best way to approach a problem or what is the best possible solution for a contribution (P2, P3, P6, P8, P10, P11, P12). In these cases, core members often showed deference to more senior core members, often project owners or project creators (P2, P3). On the other hand, more senior core members used the opportunity to instruct or even admonish other core members (P2, P3). In some cases, senior core members even handed down edicts to what the project will do about a particular problem.

*"@[not-as-core developer] The point is that we need to allow this kind of integration (that's part of the interoperability we try to promote)."* (P3)

Core members, when disagreeing with each other, used various techniques to hedge their arguments. In many cases, disagreeing developers used humor and emoticons to soften their arguments (P2, P8, P10).

*"Hey guys, sorry I'm a bit late but I don't feel comfortable with writing what's not diagnostic to me to STDERR. [...] We're losing control guys!!!!!! :P"* (P2)

### **5.3.2 Methods of Influencing the Decision Process for Code Contributions**

**RQ2:** How do participants try to influence the decision process in code contributions?

Various stakeholders involved in the code contribution employed different methods to influence the outcome of the contentious pull request. Third-party stakeholders in the audi-

ence at times applied pressure to core members to accept code contributions.

## **Audience Pressures**

Third party developers in the audience held stakes in particular code contributions, often needing a particular change for their own usage. These interested audience members applied pressure to core members in order to influence their evaluation decision. Developers in the audience were able to pressure core members through rallying support from other developers and projects or companies.

### *Community Support*

Outside developers in the audience with a stake in a code change commonly demonstrated support for a particular contribution by making comments in pull request discussions indicating that they needed the change (P1, P2, P3, P5, P7, P9, P13, P14, P16, P19). Most commonly, audience members indicated their support in the form of a “+1” or a “+1” emoticon (P2, P5, P13, P16).

*“@[submitter] +1. It’s very convenient for setting off one time operations that need to respond once to a recurring event, such as a set up operation.” (P13)*

Other than simply indicating support, audience members also commented that they were experiencing the same problem as what the code contribution fixes, increasing the perceived number of users that needed the change (P1, P7). In some cases, core members also indicated their support for a particular code change to other core members (P2, P3, P14, P19).

*“Just to confirm that this issue still exists in master. . . The fix in the pull request works for me. Please consider merging.” (P1)*

Interviewed GitHub members explained that when they perceived that their community needed a feature through feedback, they were motivated to implement those features.

*“I get feedback from those people and kind of think about and think, oh gosh, it looks like what they really need is this feature and this will work for them and I’ll do the design.”*

Other interviewees complained about such practices, citing the noise that such community support brought when trying to discuss issues around code contributions.

*“I mean it’s kind of difficult to have a productive conversation about something like that when you get a million people coming in and just saying plus one, plus one, plus one, plus one”*

### *Project and Company Support*

In some cases, rather than simply indicating a need for the change, third party developers in the audience cited their own projects or companies that would benefit from the contribution in question (P3, P6, P13, P14, P16, P18). In these cases, developers intensified their stake in the code contribution, demonstrating that other projects or even companies were relying on the change to be accepted.

*“@[core member] if you are still interested in finding a solution for this problem i can give you any details you need just ask. I’m very interested in solving this because we are investing a lot on [project] in my company but every single of our applications uses saml for authentication [sic].”* (P3)

Developers also seemingly leveraged their own user base in order to exert influence on the contribution decision process. In one example, the submitter mentioned that the contribution was actually meant to solve a problem on behalf of one of the submitter’s users (P3). Another case had the submitter mention that many users of the project switched to the submitter’s fork of the project in order to avoid a particular bug that the contribution also fixes (P16).

*“Several people have started using this fork in order to get around the issues reported in [issue link].”* (P16)

## **Alerting the Core**

In order to engage particular core members in the contribution discussion, both the submitter and core members made use of the @mention feature in GitHub, which notifies specific developers who are mentioned during discussion. (For example, @octocat sends an email notification to the developer with username “octocat”.) Developers used the @mention feature to alert core members who are key to the evaluation process for the contribution in question (P2, P3, P13, P14, P16, P18, P19). Submitters or other core members @mentioned core members to start the code review process (P13, P16). Occasionally, the reverse also occurred, with core members @mentioning the submitter to continue the review process (P14). Often, core members @mentioned other core members to solicit feedback from more qualified core developers (P2, P3, P18 [not an @mention but still a solicitation of feedback from fellow core members]). In one case, a core member alerted the rest of the core team before merging a code contribution to give other core developers an opportunity to comment.

*“I think these two sketches look good, anyone see any issues with merging?”* (P19)

In some cases, third party developers also @mentioned core members to attempt to influence their decision regarding the contribution (P13).

“+1 @[core member] please re-open this for consideration. .once does not provide the same functionality” (P13)

#### *Submitter Asks Core About Evaluation Status*

After periods of inactivity in the discussion around a contribution, submitters often asked the core team about the status of the evaluation process for the pull request (P6, P13, P16). The periods of silence before a submitter asked about status ranged from 18 days (P13) to 2 months (P16). Inactivity, as discussed in a later section, caused developers to fear that their contributions were ignored by the core team.

“Anything I can do to get this merged? @[core member] @[core member] ?” (P13)

### **5.3.3 Outcomes for Proposed Code Contributions**

**RQ3:** What are the different outcomes for proposed code contributions?

Chapter 4 on signals for evaluating contributions found that highly discussed contributions tended to be rejected while submitters prior interaction on a project tended to have their contributions accepted [Tsay et al., 2014a]. This chapter is an opportunity to examine in detail what types of discussions result in rejected or accepted pull requests. With the different issues raised around code contributions, we also saw different methods of resolution and different behaviors after a pull request is resolved.

#### **Rejection and Meeting Technical Goals**

One finding from the analysis of discussions is that while many of the highly discussed pull requests examined were rejected, the core team would often still meet the underlying technical goal of the submitter (P3, P7, P13, P15). For example, in a few contributions, the core team realized during the discussion around the contributions that the underlying problem that the submitter was attempting to solve was much more complicated than originally thought. After discussing the contributions, the core team decided to implement their own, more complete, solution to the original problem (P3, P13). In this way, although the submitter did not have their contribution accepted, the core team fulfilled the submitter’s technical goals. In one case, the submitter had submitted a malformed pull request, leading to its rejection. Rather than resubmitting the contribution, the submitter instead asked a core member to implement the bugfix. In this case, the submitter was more interested in meeting personal technical goals than having “credit” for having an accepted pull request (P7).

*“So I’m going to let [core member] decide what he want to do with it. It’s an easy search-replace action, so it doesn’t have to be this PR.” (P7)*

## **Contribution Outcomes**

When submitters or third-party stakeholders exerted influence through audience pressures, the pull requests examined suggested that they were no more likely to be accepted by core members (4 rejected and 7 accepted). However, with the exception of one pull request (P1), whenever the audience influenced the outcome, the technical goal of the submitter was met, either through the contribution being accepted or the core team implementing their own solution to the problem in the contribution (P2, P3, P5, P6, P7, P9, P13, P16, P18, P19).

In cases where the problem that the contribution was attempting to solve was suspect, especially when the project usage or scope was inappropriate, the contribution tended to be rejected (P1, P3, P4, P7, P13, P14, P15, P20). In the two exceptions (P2, P16), where the problem the contribution solved was suspect yet the pull request was accepted, the core team disagreed amongst each other, engaging in extended discussions about the contribution.

## **Future Contributions Advertised**

After contributions were resolved, submitters often advertised future changes in the discussion (P2, P4, P7, P11, P18, P19). Even if the contribution was rejected, submitters sometimes offered suggestions on similar changes in the same direction as the offered contribution (P4, P7, P11).

*“I’m closing this for now, as this needs more testing. I would also like to investigate whether we can support multi-monitor configurations better than today.” (P11)*

In changes that were accepted, some submitters indicated future changes that were incoming (P18, P19).

*“Let us know about syntax, formatting etc. on these 4. We 50 more in the pipe passing internal peer review. [...] We’re upping our schedule to get more pages. Out we have a ton in the works but need to sign off on them internally before handing them over.” (P18)*

### 5.3.4 Submitter's Prior Experience

**RQ4:** How does a submitter's prior experience with a project change the discussion?

Chapter 4 also found that a submitter's prior interaction had an influence both on whether pull requests and highly discussed pull requests were accepted [Tsay et al., 2014a]. This study examines how a submitter's experience changed the nature of discussions around their contributions.

#### Core Thanking New Submitters

When submitters were new to the project, core members almost always made sure to politely engage with the new submitter regarding their contribution. For new submitters, core members thanked the submitter for their contribution as their first comment (P2, P5, P6, P11, P12, P13, P18, P19). In other cases, core members apologized to new submitters for delays in responding (P1, P5, P6, P10, P12, P13, P18, P20). Often, developers in GitHub interpreted delays in response as the core project team ignoring a contribution and use this information as a signal for poor project management. Often, the first comment to a new submitter combined the two, both thanking a new submitter for their contribution and apologizing for a delay in response at the same time.

*"Looks impressive. Since I'm a bit busy with some other stuff I'll made a review in a week or something. Please be patient. And thank you for contribution :)"* (P5)

Interviewed GitHub developers were aware of the value of being courteous in regards to accepting pull requests.

*"I mean if there's a problem with the library you don't want to rush in and say, "Your library sucks, and it's wrong in the following ways and I'll fix it for you. You need to merge it," or whatever. It really comes off badly [...]. But if you come at it from another direction and say, "This is a great library. Thanks for providing it. I do have one or two little changes that I'd like to make. I think it'd help the library as a whole. What do you think?" That generally comes off much, much better."*

Interviewees also explained that they would be polite to new submitters to try to encourage contributions.

*"For smaller things, does it help people to contribute? So I think that this is kind of entirely an issue of how do you handle it [...]. Not to say that you're always squashing their ego or putting them down when you're making these changes for them. I think that you can say, "Hey, thank you for the pull request. There were some issues here, here, here that I fixed up and then I merged it. In the future try and make sure that you do this. Thanks again,*

*though, for the code.” Usually people respond pretty positively to that.”*

### **Alternative Solutions for New Submitters**

When the submitted contribution’s implemented solution was suspect, depending on the level of the submitter’s prior interaction on the project, core members and third parties had different responses when offering alternative solutions. In general, regardless of the submitter’s experience on the project, other developers questioned the approach of the contribution’s implemented solution. However when discussing alternative solutions to the contribution, the prior experience of the submitter seemed to change how developers offered their alternative solutions. Submitters with experience on the project tended to receive suggestions on alternative solutions to solve the contribution’s problem (P7, P8, P9, P11, P15) while new submitters to a project tended to receive implemented alternative solutions from core members and third party developers (P2, P3). In both pull requests, multiple alternative solutions were advertised, ranging from competing contributions (P3) to similar solutions in other projects (P2, P3).

## **5.4 Discussion**

This study finds that developers were very aware of the different stakeholders when discussing contributions. Developers also had multiple methods of influencing the evaluation process, including influencing power relationships in the project. The findings also suggest that core members and submitters defined and evolved project requirements during discussions around code contributions.

### **5.4.1 Stakeholders Influencing the Outcome**

One of the side effects of open collaboration is that the environment allows for third party developers to participate in discussions around evaluating contributions. In open environments, a project’s dependencies are not fully known to the core members. Any developer can independently use any library. Notification mechanisms, such as GitHub alerts, make developers in the audience aware of important changes that may affect them.

While prior work on GitHub has suggested that the presence of a perceived audience itself pressures developers into behaving differently [Dabbish et al., 2012], this study suggests that the audience takes on a much more active role when evaluating contributions.

Similar to developers overhearing discussions in collocated software teams [Teasley et al., 2000], this study finds in our sample of pull requests with extended discussions that developers in the audience would often jump into discussions where they may have stakes in the outcome. Gousios et al. [2014] found in their sample of pull requests that discussion participants who have never committed to the repository are rare. I found, however, that extended discussions tended to draw developers who were not directly related to the pull request, i.e., were neither the submitter nor core members. Most of these third party developers made some peripheral contribution to the project at some point.

The ability of third party developers to independently join the discussion around any contribution may influence how core members and submitters evaluate and discuss contributions. Submitters received suggestions from both core members and third party developers from the audience and would often need to justify their design decisions. In some cases, submitter's solutions even competed with alternative contributions from third party developers that solved the same problem as the submitted contribution. The extra negotiation required due to suggestions from the audience may raise the cost for core members to evaluate a pull request, reducing its chances of acceptance [Tsay et al., 2014a]. At the same time, this exploration of alternative solutions by the audience seems to be a form of decentralized experimentation. So while core developers may be less willing to make risky experimental code changes while being watched, third parties from the audience may be willing to take on the risk.

This study suggests that political signals may be useful for developers. "Political" signals in this case refer to indications of the influences of different stakeholders such as users or developers from dependent projects. The findings suggested that core members tended to fulfill the technical goal of the submitter whenever the audience applied pressure. This may indicate that core members may wish to infer which changes are needed by users or dependent projects. Implementing a feature similar to the "+1" comments (essentially stars for changes) would be a very direct method of creating a signal of which changes have the most influence from users. Similarly, given that stakeholders are referencing projects and companies in an attempt to influence core members, perhaps creating a signal that indicates which downstream projects are affected by a change would be useful for project managers.

Software development environments with pervasive notification mechanisms such as GitHub allow developers the affordance of staying aware of projects where they may be stakeholders but not necessarily core members. This awareness has the side effect of creating an audience that may actively attempt to influence the development of a software project through participating in discussions or developing experimental code changes. Future research should explore how notification mechanisms enable developers to be action-

ably aware of projects they may have stakes in. A better understanding of how developers act on awareness notifications would inform the design of tools that better notify developers when to participate in relevant discussions in dependent projects and allow core members to effectively manage experimentation from third party stakeholders.

## 5.4.2 Power Relationships in Evaluating Contributions

Discussions around contributions had three types of participants: submitters, project core members, and third party audience members. These three groups of developers appeared to have implicit power relationships.

The closer the developer was to the project's core, the more influence the developer seemed to wield. For example, a third party developer's suggestion had much less weight to the submitter than one from a core member. Core members had the ultimate power to accept or reject a code contribution due to their commit access. The degree of influence also varied within the core, with certain core members showing deference to more senior core members such as project owners or veteran contributors. Submitters, having implemented a solution, demonstrated investment in the project. Third-party stakeholders had not demonstrated such investment.

To help determine power relationships in a project, developers used information present in the environment to make inferences on the expertise of other developers [Marlow et al., 2013]. Core members and submitters may attribute less influence to the comments of a third party due to inferences made using cues in the environment. For example, a third party developer with no connection to the project may be seen as a certain "type of person" who only reports problems but does not actually contribute code [Marlow et al., 2013] and therefore may have less of a stake in the technical discussion about the contribution. The submitter's prior experience on the project was also used as information to infer the submitter's expertise. Prior experience may be an indicator of the degree of socialization a developer has undergone for the project [Ducheneaut, 2005]. Socialized developers, possessing knowledge of the core team and project-specific norms, may be less likely to create risky contributions or contributions of uncertain value.

While third party and submitting developers may wield less power than core members during the contribution evaluation process, these developers were able to leverage their own communities to influence the core team on a project. Developers would cite their own projects and companies to intensify their perceived stake in the code contribution, perhaps increasing their influence on a change through pressure [Kraut and Resnick, 2012]. Leveraging user bases in this way to influence the core was often effective because core

members understood that their authority is closely tied to keeping users satisfied [Lerner and Tirole, 2002]. In some cases, stakeholder communities would actually cause a submitter to create the contribution in the first place. For example, if a user was experiencing a bug in a certain project, the project owner implemented and submitted a bug fix to an upstream project [Dabbish et al., 2012]. This suggests a chain of influence across the upstream and downstream dependencies in software projects. The pressure to contribute to an upstream project may have benefits to the technical integrity of both projects due to ensuring that a code change resides in the most appropriate location in terms of architecture. For example, if a bug goes unfixed in an upstream project, multiple downstream projects may all have to implement the same workaround or bug fix.

Online community literature finds similar power relationships between "elite" and "common" users in Wikipedia [Kittur et al., 2007a]. However, "elite" Wikipedia users such as administrators wield power and authority very differently than project managers in GitHub. The affordances of the environments give GitHub project managers much more technical authority than Wikipedia administrators. Project managers in GitHub have the final authority to accept or reject changes. In contrast, Wikipedia administrators have special administrative powers such as "protecting" pages and blocking users but lack definitive authority to decide what edits stay or go. Additionally, many de facto "owners" of articles are not administrators but regular users, deriving their authority entirely through social rather than technical means [Forte and Bruckman, 2008]. Social authority through community consensus is present in both GitHub and Wikipedia, as seen through attempts by the community to influence outcomes. Some of the political strategies for influencing the outcome are similar across both GitHub and Wikipedia. For example, arguing about the scope of a change and appealing to other projects or pages Kriplean et al. [2007]. Differences in influencing strategies correspond to differences in community norms between Wikipedia and GitHub. For example, Wikipedia has a strong emphasis on using policy for governance, often relying on past policy decisions to resolve conflicts Kriplean et al. [2007] while GitHub emphasizes open source ideologies such as aiding other open source developers and projects [Stewart and Gosain, 2006a]. This may explain why Wikipedia users attempt to influence discussions through appealing to past policy decisions while GitHub users cite other projects and companies.

How these power relationships between open source developers as well as the incentives and decision rights that are present support good decision-making in terms of evaluating code contributions is not well understood. Future research should investigate these relationships in more detail, in order to determine what factors allow developers to wield more influence than others when making evaluation decisions. Environments that make these factors such as expertise visible or allow for different notification capabilities may

have an impact on these power relationships and the outcome of code contribution evaluations.

### **5.4.3 Developing Software Requirements through Discussion**

Core members and third party developers from the audience often raised issues around a contribution, either about the appropriateness of the problem solved in the pull request or the correctness of the implemented solution. In cases where the contribution's problem was suspect, submitters and core members often engaged in extended discussions about the appropriateness of the code change. For example, the submitter may have attempted to implement a feature that is outside the scope of what the software project should be able to do. This discussion over whether the problem to solve was appropriate was actually a negotiation over the requirements of the software project.

Open source software projects tend to not have formal requirements documents that are created through a formal elicitation process [Scacchi, 2002]. Instead, requirements in open source projects tend to emerge in forms such as mailing list messages or forum posts as a byproduct of the community discussing the direction and assignment of future code contributions [Scacchi, 2004]. This study saw a similar method for evolving the requirements of the software project when submitters and core members discussed whether a particular code contribution was appropriate for the software project. In other words, core members assessed whether the problem that the submitter was trying to solve was within the scope of the project's projected feature set.

Besides submitters and core members, other stakeholders such as the third party developers in the audience were also able to participate in evolving the requirements of the software project by participating in the discussion. This is somewhat similar to how community members in traditional open source projects will communicate their needs through bug reports or feature requests [Mockus et al., 2002]. Due to transparency in the environment however, perhaps a wider variety of stakeholders were able to influence the requirements of the software project through discussion.

Future research should examine this connection between software requirements and contribution discussions in more detail. Future tool design may explicitly recognize when requirements are being evolved during discussions and may archive these discussions in a more visible way for the benefit of core members.

## 5.5 Conclusion

In this work I examined the coordination that occurs between open source developers during discussions of extended contributions. I found that when developers raised issues with either the problem the submitter was attempting to solve or the solution that was implemented in the pull request, it provided an occasion to discuss alternative solutions or negotiate requirements. Different stakeholders also attempted to influence the outcomes of contributions through pressuring the core or directly alerting them. The transparent environment provides specific mechanisms for stakeholders in the audience who are outside of the submitter and project core team to participate in the evaluation process.

I found unexpected outcomes for contributions where though a submitter may have their pull request rejected, the core team still fulfilled the technical goals of the submitter in some other way. I also found that the submitter's level of prior interaction on the project changed how core and audience members interacted with the submitter during discussions around contributions.

The findings of this study inform the design of notification and discussion mechanisms for large-scale collaboration where a wide variety of stakeholders participate in evaluation discussions around code contributions. Findings may also inform how distributed developers negotiate software requirements during code contribution evaluation discussions. This study also suggests a number of potentially useful political signals, such as signals indicating the influence of users or downstream dependencies. Future work should investigate how different kinds of event notification mechanisms influence participation in contribution discussions. Ideally, all legitimate interests should be able to enter the discussion, with notification mechanisms alerting third party stakeholders of relevant discussions. Since submitters also rally support as an effective tactic, more systematic ways of showing support for a change, and perhaps helping to prioritize it relative to other possible changes might also prove useful. Finding ways to identify when conflict resolution mechanisms might also facilitate better and less disruptive ways to handle difficult decisions. Finally, since social relationships seem to have an impact, various mechanisms for visualizing these connections or making them more salient might also impact these negotiations.



# Chapter 6

## Signals for Evaluating Projects for Use or Contribution

### Chapter Summary

Evaluating projects before making the decision to use or contribute is a key coordination task that open source software developers regularly perform. Transparent development environments provide information about projects and users that assists developers in making more informed evaluation decisions. Identifying the information that developers use as signals to evaluate projects may inform the design of tools and environments and provide insight into what qualities of projects developers value when deciding to use or contribute to a project. This chapter describes an exploratory sequential mixed methods study of the signals and qualities developers use towards the tasks of deciding to use or contribute to a project. The first phase is a qualitative exploratory interview study of signal usage by developers on GitHub. The second phase is a quantitative validation analysis that uses a larger sample of developer and project data from GitHub to determine which of the signals that interviewees perceive as useful, and that signaling theory suggests as informative, are actually used by developers at large. The study finds that developers evaluating projects for usage used signals related to working dynamics such as commit volume to indicate liveliness and personal utility such as code churn to indicate maturity. For evaluating projects for contribution, developers used signals for working dynamics by inferring responsiveness via time to close pull requests and personal utility through the accessibility of contributions via issues. These findings suggest that costlier signals resulting from development work or extra work by the core team are potentially more useful to developers. The usefulness of low-cost community signals such as stars count is an open question and

is examined in more detail in the Chapter 7.

---

The widespread availability of reusable open source libraries and frameworks greatly increases the efficiency of creating software systems at the cost of having to evaluate these software projects [Ajila and Wu, 2007]. Implementing a wide range of functionality is often as simple as establishing dependencies with the right projects. Especially with the popularity of package managers such as Ruby’s RubyGems and node.js’s npm, creating complex software systems may be as easy as running a few commands to install libraries and writing some “glue code” that ties together these libraries. However, for many developers the choice of establishing a dependency with a project is a common yet potentially impactful coordination decision. Choosing the wrong project may lead to future difficulties or extra work. For example, using unstable or error-prone dependencies may result in propagated errors that are difficult to correct. Developers then must evaluate potential dependencies in order to mitigate future problems. This evaluation process, while common for developers, is relatively unknown and understanding this process in more detail may inform both the design of future tools and environments and software project management.

Transparent development environments make visible information that assists developers in evaluating projects and provides opportunities to better understand the evaluation process. Transparent environments make visible development activity such as bug reports and discussions but also make explicit relationships between developers and projects. These relationships link together information such as the activities of the developers who work on projects, the projects these developers work on, and so on [Dabbish et al., 2012]. This additional information is also an opportunity to examine the evaluation process in more detail. Transparent environments allow the study of previously unavailable archival information about multiple aspects of software development. For example, the explicit relationships that transparent environments establish may indicate social aspects of development. Determining what information developers specifically use to evaluate projects may in turn give insights into what aspects of development are important for the evaluation process.

I use signaling theory as a lens to understand how developers use information in their environment. Signals are observable pieces of information that indicate some unobservable quality of the person or entity that generated the signal [Connelly et al., 2010]. Signaling theory is a rich lens to understand how developers make inferences from information to inform decisions. Transparency offers a new class of “honest” signals that are derived from development work rather than intentionally broadcast by the signaler. For example,

developers evaluating other developers for recruitment use commits to high-status projects as a signal to infer the unobservable quality of coding ability [Dabbish et al., 2012]. Understanding what pieces of information are useful signals for developers may inform how future transparent environments avoid overwhelming developers with information. A risk of transparent environments is information overload, creating challenges for developers in these environments to manage and interpret their information feeds [Singer et al., 2014, Storey et al., 2014]. For some developers, the high cost of managing and interpreting activity traces is so high that it leads to them not using transparency features [Singer et al., 2014]. Discovering and validating useful signals would also inform the design of tools and development environments that are able to visualize exactly the information developers need to inform decisions that arise during software development.

To determine how developers use information in their environment as signals, I performed an exploratory sequential mixed methods study [Creswell, 2013]. As the usage of transparency information by developers is not well-understood, the first phase is a qualitative exploratory interview study that explores perceptions of what signals are useful during development-related tasks. Signaling theory gives guidance to which signals should be more informative. In some cases, interviewee's perceptions are contrary to what theory suggests. I then test interviewee's perceptions of usefulness against theory through the second phase, a quantitative validation study. By using a dataset of developers and projects on GitHub, I determine how strongly individual signals relate with decision outcomes.

The findings for this study suggest that signals for working dynamics and personal utility are more useful for informing developers' usage or contribution decisions. Signaling theory suggests that the usefulness of these signals lies in the relatively high cost in generating these signals. For example, commit volume is a signal for project liveliness that is highly associated with project usage and is a signal that cannot be generated without developers in a project actually performing numerous commits. In contrast, community signals such as stars count have relatively low costs and the signaling *fit* is an open question. This concept of fit, whether a signal actually indicates an unobservable quality, is explored for the signal of stars in Chapter 7. This chapter also provides a potentially useful methodology for eliciting and validating signals for a specific coordination task in transparent environments. Future work may examine whether a similar mixed-methods approach can be applied to other tasks, domains, or environments. For example, this study's methodology could potentially be used to understand what signals writers use when evaluating prose and the results of such a study may feed back into designing collaborative writing tools.

In the following sections I consider related research on open source software projects, transparent development environments, and signaling theory to motivate the research ques-

tions for this study, describe the exploratory interview study of GitHub developers, describe the quantitative validation study that relates a set of signal metrics with outcomes of project-related tasks, and discuss implications of the findings.

## **6.1 Open Source Software and Signaling**

I ground this work in prior literature in awareness in open source software and transparent development environments. I also review concepts of signaling theory that are relevant for this study. Informed by prior work, I develop research questions to further our understanding of the information developers use to evaluate projects.

### **6.1.1 Awareness and Open Source Software**

As open source software tends to require distributed software developers to coordinate their efforts [Mockus et al., 2002], open source software developers seek out information about their fellow developers in order to stay aware of their work activities. Gutwin et al. [2004] found that developers in open source software projects sought awareness information such as who is working on what part of the project from simple text communication such as mailing lists and text chat to stay aware of the work of other developers on the project. They found that work awareness information from these simple communication tools was enough to satisfy most project coordination needs. Newcomer developers to an open source software project also needed to seek similar awareness information from text communication tools [Ducheneaut, 2005] in order to “recruit” core project developers towards supporting their contributions. Rigby and Storey [2011] found in their study of peer review on open source software projects that developers on a project used similar work awareness information from the mailing list in order to select code contributions to review. In their study, developers also suffered from “too much awareness” and needed filtering techniques to manage the information overload.

### **6.1.2 Signal Usage in Transparent Development Environments**

While open source developers use awareness information to coordinate efforts with other developers within a project, transparent development environments provide new actionable information regarding outside developers and projects.

Previous qualitative research on GitHub, a popular transparent development environment, shows that developers make a variety of subtle inferences about other developers and projects using signals from the environment. Dabbish et al. [2012] found that open source software developers use these inferences in practical ways, for instance to help manage external contributions to projects, discover project user needs, and recruit developers. Developers in this study used signals of community attention to a project or event in the feed to determine if a project was worth using or a discussion was worth reading. However, developers also struggle to manage information in their environment. Singer et al. [2014] in their study of how open source developers use Twitter found that developers dealt with challenges in how to manage consuming large amounts of information through developing strategies such as filtering tweets and curating their following networks.

As examples of signals derived from transparency that developers used, Marlow et al. [2013] found that GitHub developers used information in the environment in order to form impressions of users and projects. Pham et al. [2013] found that project managers used signals for quality and risk when assessing how much testing a contribution requires, such as the size of the change, the type of contribution, and how much they trusted the submitter. Chapter chapter4 about signals for evaluating contributions suggests that project managers use social and technical information about contributions and their submitters, such as following status and comments on contributions, as signals when evaluating submitted contributions [Tsay et al., 2014a].

### 6.1.3 Signaling Theory as a Theoretical Lens

Signaling theory is a useful lens for understanding how information made visible by transparent environments is used for evaluating projects. Signals are observable pieces of information that are used by a receiver to infer an unobservable quality of the signaler. Transparent environments enable a new class of potentially useful, unintentional signals that are derived from observing work. While the majority of signals that are studied in literature are positive, intentional signals, unintentional signals still convey important information [Janney and Folta, 2003]. For positive, intentional signals, the incentive to signal is to affect the decision of the receiver [Spence, 1973]. In transparency, some useful signals are unintentional and a product of development work, merely performing the task produces the signal. Therefore the incentive of producing such a signal becomes aligned with performing the development task.

Signaling theory offers key constructs [Connelly et al., 2010] describing the relationship between signal and signaler. Signals vary in their *cost* to produce, often with an assumption that signalers possessing the unobservable quality are better suited to absorb

these costs than others. Signals may also vary in *fit*, the correlation between the signal and the unobservable quality it indicates. A related concept is *honesty*, the extent to which the signaler actually possess the signaled quality. For example, a resume listing involvement in an impressive-sounding project may not reflect meaningful skill for a particular task. If listing impressive-sounding projects is not correlated to the needed skill, then that signal has a poor fit. If such projects *are* correlated to skill but the candidate in question does not possess that quality, the signaler is dishonest.

In online communities such as open source software projects, participants draw signals from the environment to infer qualities of both people and projects. Donath [2007] finds rich patterns of signaling and deception in online communities to infer member identities. In GitHub, information in the environment such as activity traces and past discussions are used as signals to infer qualities about developers such as coding ability and personality [Marlow et al., 2013]. Activity traces are also used as signals for project properties such as quality, collaborative environment, and member commitment [Dabbish et al., 2012]. Outside of GitHub, Scaffidi et al. [2010] found in their study that users use signals of previous successful authorship and mass appeal for web macro scripts to decide whether to reuse a script.

#### 6.1.4 Research Questions Development

While previous research on GitHub has established that developers use information in their environment to make useful inferences, we do not yet have a systematic understanding of the relationship between the information developers use as signals, the unobservable qualities they infer from these signals, or how the inferences inform decisions. The first step towards this systematic understanding is to use signaling theory as a lens to identify unobservable qualities and signals that are useful to developers for informing tasks. Through an exploratory stage of interviews with developers on GitHub (subsection 6.2.2), I discovered that a key task developers perform is to evaluate software projects for usage as a dependency or for contributing to the project. This study focuses on these two project-related tasks and the qualities that developers need to infer about projects in order to perform these tasks.

**RQ1:** What kinds of qualities do developers infer when using signals to decide whether to use a project as a dependency or submit a contribution to a project?

Qualitatively identifying signals that developers perceive as useful also provides the opportunity to quantitatively compare these perceptions against the amount of usage and contributions a software project actually receives. If qualitative analysis suggests that developers

use signal A to make a decision B regarding a project, then there should be a statistical association between the presence of signal A and the outcome of decision B. Comparing the strength of statistical associations (see Table 6.6 and Table 6.7) to task outcomes would suggest which signals are actually used by developers for the task in question. There also may be signals that interviewees indicate but are not reflected in statistical associations. Identifying these disparities may indicate signals that developers perceive as important but are not actually used for informing decisions.

**RQ2:** What are the strengths of statistical associations of signals with the key decisions of using and contributing?

## 6.2 Qualitative Exploratory Interview Study

In the first phase of the mixed methods study, I investigated the research questions qualitatively by interviewing a sample of developers from the popular social coding software hosting site GitHub. The goal of the interviews was to discover what specific signals developers used for two different software project-related tasks: (1) deciding whether to use a project by incorporating it as a dependency, and (2) deciding whether to offer a code contribution to a project. This section describes the interview study methodology and its findings.

### 6.2.1 Methods

#### Data Collection

To investigate how developers make usage and contribution decisions, I conducted a series of semi-structured interviews with 47 GitHub users. In order to gather a wide variety of practices, I sampled users from the most popular projects in multiple programming languages on GitHub. From these projects, I sampled both peripheral developers and heavy users with more than 80 “stars” on at least one project. Participants were solicited via email, and interviews were conducted in person, via phone or Skype. Remote participants shared their screen during the interview using Adobe Connect so users could demonstrate their activities on the site. Participants were asked to walk through their last session on GitHub, describing how they interpreted information displayed on the site as they reviewed earlier work activities.

Interviews were performed in two stages: 1) an exploratory stage with 24 developers that included questions about project management and how they used different GitHub

features in their work, and 2) a second stage with 23 developers focused on project-related coordination decisions. These project-related decisions include deciding to form a dependency with a project and sending pull requests or managing received pull requests. Developers were asked to describe specific coordination instances such as pull requests sent and decisions made leading up to and during these interactions.

## **Data Analysis**

I applied a grounded theory approach [Corbin and Strauss, 2008] to analyze how developers make use of signals about projects while choosing to use or to contribute to projects. I first identified 75 instances where participants reported performing these tasks. The analysis includes both stages of interviews, as the exploratory interviews include instances of developers performing relevant tasks. For each instance, I identified the decision performed, the project involved, what visible project information the developer looked at to make the decision, and what unobservable quality about the project the developer inferred based on this information. I then conducted open coding on a random sample of 50 of these instances where examples were first grouped into the task performed then into categories of inferences that were conceptually similar. This process revealed several categories of unobservable qualities made about the software project in question such as liveliness, ease-of-use, and community size. I used this first set of categories to code the remaining 25 interview excerpts which no longer revealed new interactions not captured in the existing set of categories, indicating theoretical saturation [Corbin and Strauss, 2008]. I then performed axial coding, identifying three overarching categories of unobservable qualities that developers inferred: working dynamics, personal utility and community evaluation. Codes and axial codes are summarized in Table 6.1.

### **6.2.2 Results**

Interviewed developers used information broadcast by both projects and members of the projects as signals to infer different unobservable qualities about software projects. Through the first stage of interviews, I discovered and chose to focus on the key tasks of using and contributing to projects. The focused interviews found that across the two tasks, developers inferred certain types of unobservable qualities. I describe the qualities of projects that interviewed developers infer to inform the tasks of using and contributing to projects and the signals used to infer these qualities.

## Exploratory Stage Interviews

By exploring in the first stage of interviews how developers make use of transparency features in GitHub, I discovered two key project-related decisions to focus on: using a project as a dependency and contributing to a project. During this stage, I discovered that developers used information in their environment to assist in a number of coordination tasks (see Dabbish et al. [2012] for more detail on the methods and findings from these initial interviews) such as evaluating projects for use, contributing to projects, recruiting developers, identifying user needs, managing code contributions, and managing dependencies with other projects. I chose to focus on the tasks of evaluating projects to *use* by incorporating as a dependency (such as including a library) and *contributing* code to a project (usually by pull request). I choose to focus on these tasks because participants identified them as important coordination-related decisions. Moreover, how developers evaluate projects for usage or contribution is not well understood. Similar tasks such as managing code contributions [Tsay et al., 2014a] or recruiting developers [Marlow et al., 2013] in transparent environments have been well-studied. The current study is an opportunity to both learn how developers use information in transparent environments and how developers evaluate projects.

## Types of Unobservable Qualities of Software Projects

Across the software development tasks of using and contributing to software projects, I found that developers used signals to infer three general types of unobservable qualities about projects. The types of qualities are summarized in Table 6.2. These three types of unobservable qualities correspond to axial codes developed during the coding process and answer the first research question: *What kinds of unobservable qualities do developers infer when using signals to perform tasks?*

**Project working dynamics** - Developers used signals to make inferences about the project's working style and direction. These inferences include how internal activity happens, such as development by core members, and reactions to external activity, such as core members responding to outside contributions such as pull requests. The signals often used were work artifacts and activity traces from the project such as the commit log.

**Personal utility** - Developers used signals to estimate their personal utility from investing time and effort into using or participating in a project. These inferences also include estimating the cost of involvement. Developers made these inferences to determine if the benefits of participating outweigh the costs. The signals developers often used were both work artifacts and the visible “extra work” [Trainer et al., 2015] core developers perform

to enhance a project’s accessibility.

**Community evaluation** - Developers used signals to evaluate the community around a software project. Often these inferences allowed developers to determine the degree of community support or participation in the project. The signals developers often used were personal social networks or the networks of project members, both of which are made visible and explicit in GitHub.

Table 6.1: Interview Study Summary

Task	Type of Unobservable Quality (axial codes)	Inferred Unobservable Quality (codes)
Usage	Project Dynamics	Project Activity and Liveliness
	Personal Utility	Technical Utility
	Community Evaluation	Community Interest
Contribution	Project Dynamics	Responsiveness of Core Team
	Personal Utility	Accessible Contribution Opportunity
	Community Evaluation	Community Benefit

Table 6.2: Unobservable Quality Types Summary

Unobservable Quality Type	Definition
Project Working Dynamics	Project working style and direction
Personal Utility	Potential cost and benefit from choosing project
Community Evaluation	Community around project

## Signals for Using Projects

Relevant instances (out of 75) of developers performing tasks are included as (*In*) to help the reader understand the characteristics of the sample. I make no claims, of course, that these numbers are representative of the larger population. I also describe the signals in terms of signaling theory constructs [Connelly et al., 2010] of *intentionality*, *honesty*, *cost*, and *fit* which are described in more detail in subsection 6.1.3.

### *Dynamics: Project Activity and Liveliness*

Developers making the decision to use a project inferred the unobservable quality of a project’s working dynamics in order to determine the degree of future support and development. Many projects on GitHub, as on other hosting services, are “dead” projects with

little to no chance for active development by the original core developers [Kalliamvakou et al., 2014]. Developers used a project’s commit activity as a signal for the project’s liveliness (I48, I50, I55, I57, I62) in that projects with more commits were seen as more desirably lively projects with active development teams who are more likely to support the project in the future. Developers also used more nuanced signals such as the volume, velocity, and recency of a project’s commits (I50, I55, I57) and the size and diversity of the project’s core (I50) to infer the degree of active development by the project’s core team.

*“Usually, it’s part of my research for, like, what is a good C library, I will go look at the commit history and see if it’s actually actively being worked on [...] are there changes going in? Like, how recent are changes? Like, weeks, months, years? Another kind of metric is what’s the mix of [developers]? Is it all one person? Is it all people in a single organization? Is it a wide group of people? Kind of the wider the net there’s more likely any issue that I might run into is actually— has already been identified and the patch probably got [done by] somebody else.” (I50)*

Signals that developers used to infer project dynamics such as commit activity and size of the core team are unintentional signals that are products of transparency. As these signals are used to infer qualities of how the core team works, signals derived from the core team and their development activities are the most direct indicators. The directness of these signals should result in a high *fit* to the qualities indicated. These signals are also expected to have high *honesty* and *cost*, as generating these signals is a product of development work rather than intentionally broadcast.

#### *Utility: Technical Utility*

Developers inferred the potential costs and benefits of using a forming a dependency with a project in order to avoid future problems or wasted effort. Developers used commit activity to infer a project’s stability through its maturity (I61, I65). Developers perceived mature projects as having more code contributions that are smaller in scope than large feature implementations, or colloquially, a lower “code churn” (I65). Mature projects also tended to have formal numbered releases compared to newer projects (I61).

Developers used signals from the project to gauge its ease-of-use and potential cost to integrate. Developers used the existence of a project’s README file that documents a GitHub project as a signal that the project is easy for other developers to use (I63). Similarly, developers used signals such as the number of required upstream dependencies and coding style to infer the difficulty of using a project (I54, I72). Developers also preferred to use GitHub projects that are also in official package managers such as RubyGems for Ruby (I63) as they tended to be much easier to install compared to installing from source files on GitHub.

*“If there’s a choice, then I go with whichever one is nicest for me to use, as a programmer, so if [...] there’s one project, and you need to set up [multiple things] to actually use it, versus this project where you include this in your thing, and then you do the thing that makes sense, and it all just magically works. I’m like, well, I’m going to use the second one.” (I72)*

Signals developers used to infer personal utility such as code churn are unintentional while other signals such as the README are intentional signals by the core team. As mentioned earlier, unintentional signals made visible by transparency are expected to have high *honesty*, *cost*, and *fit* due to how closely tied these signals are to development work. Intentional signals produced by the core team are cases where core members are performing extra work [Trainer et al., 2015], making these signals costly. *Costly* signals tend to also be *honest* and with a high *fit*, assuming that signalers who possess the quality in question are better suited to absorb the costs of producing the signal [Spence, 1973]. However whether this assumption holds is still an open question. For example, whether a project is easy to use may have no connection with the costs of creating a useful README file.

#### *Community: Community Interest*

As open source software projects often rely on a large user community for technical support [Lakhani and von Hippel, 2003], system testing, and problem reporting [Mockus et al., 2002], developers used signals about the size and level of interest of a project’s community to infer the level of support should a problem with the project arise. Developers used a project’s watcher/star and fork count to infer how many other developers were interested in the particular project (I64, I66). For developers deciding whether to use a certain project, these metrics for a project’s popularity were signals for the project’s overall quality. Developers tended to choose projects with interested communities as these projects also tended to be better supported by their users as well as tended to stay alive longer (I66). Developers also used information about a project’s commit activity to infer the degree of interest of the project’s development community (I53, I62). For example, a project with a single developer has a very different commit log as well as ability to support its users than a project with a wide range of core developers.

*“You know and it shows the last activity, the number of watchers, the number of forks I think. The number of people watching a project or people interested in the project, obviously it’s a better project than versus something that has no one else interested in it.” (I64)*

Signals developers used to infer projects’ community such as stars count are a different type of unintentional signal than signals related to development work. Signals such as stars and forks count are an aggregation of the activities of users of the GitHub ecosystem

rather than project activities, similar to online reviews and ratings. Because these user-aggregation signals are generated by users rather than the core team, such signals should have high *honesty* [Flanagin and Metzger, 2013]. The *cost* of starring or forking a project is also extremely low, involving little more than the click of a button. It is an open question of the *fit* of these signals to qualities indicated.

## Signals for Contributing to Projects

### *Dynamics: Responsiveness of Core Team*

As contributions to a project in GitHub usually require the evaluation of the core project team, developers making code contributions inferred the responsiveness of the project's core team towards new contributions. Potential contributors looked at how other contributions were managed in a project as signals for the responsiveness of the project's core team. Pull requests that have not been closed by the core team were common signals for a lack of responsiveness (I37, I41, I47). Developers held expectations that responsive project core teams reviewed contributions and then close the pull request, so projects with open pull requests were seen as unresponsive.

*"I can see, and so he just didn't even accept this pull request, and just kind of sitting there, and now I haven't talked to this dude, and I don't know if this project is moving forward, but since he doesn't care, I decided I didn't care, and I'm working on other things now."* (I47)

Similarly, developers also used comments on pull requests from the core team as a signal of responsiveness (I40, I45). In particular, developers expected the core team to indicate some degree of interest; pull requests without at least one comment signaled an unresponsive core team (I40). Some core teams would be extremely responsive in order to elicit contributions from their communities (I39, I46). Expectations for responsiveness were also mediated by the size of the project in that larger projects were expected to take longer to respond to contributions than smaller projects (I41, I47).

Signals that developers used to infer the work dynamics projects have towards contributions were unintentional and made possible through transparency. As discussed earlier, due to their close relation to the work performed by the core team towards evaluating contributions, signals such as time to close pull requests and comments are expected to have high *honesty*, *cost*, and *fit*.

### *Utility: Accessible Contribution Opportunities*

Developers looking for contribution opportunities to improve their skills [Lakhani and von Hippel, 2003] or potentially join a project [von Krogh et al., 2003] made inferences

about the potential cost of contributing to a project. In particular, these developers used signals for how accessible a project is towards contributions through visible contribution opportunities. Some developers identified projects that were mature enough to easily offer additional features through outside contributions (I40, I41, I42, I43) as immature projects may require additional effort to contribute to (I43). In these cases, the offered contributions tended to widen a project's feature set to make the project "more applicable with more people" (I43). For example, a mature database connector might get support for more database types from outside contributors (I42).

To find clear opportunities for contribution, developers looked through the project's issue tracker (I38) or through ongoing discussions (I73) or on the project wiki (I75). In other cases, core members to projects maintained a list of contribution opportunities for newcomer contributors (I74, I75, I76). In one case, core members marked open issues with labels that indicate contribution opportunities such as "easy fix" or "needs code" (I76).

*"When I log in I have a newsfeed and I'm checking [...] any ongoing comments or discussion regarding Rails that's going on. Mainly what I'm looking for there, certainly if there's some low-hanging fruit that I can contribute to"* (I73)

Signals that developers used to infer the accessibility to contributing to a project were both unintentional in the case of qualities such as maturity and intentional by the core team in qualities such as clear contribution opportunities. As discussed earlier, signals tied to development work are expected to have high *honesty*, *cost*, and *fit*. Also discussed earlier, signals intentionally emitted by the core team to indicate easy opportunities to contribute involve extra work [Trainer et al., 2015] and are expected to be *honest*, *costly*, and high *fit* only if the costs align with the quality in question.

#### *Community: Community Benefit*

Before making a contribution to a project, developers made inferences about the potential impact their change would have on the project's community. Due to the fork and pull request model that GitHub provides, developers first implement their code changes on a personal copy of a particular repository (fork), then decide whether to keep this change solely on their personal fork or to offer the change back to the original repository via pull request. If a particular change is very specific to a personal use case, then the change tended to be left in the personal fork. However, if the change could potentially benefit other people, then the developer shared the change with the original repository by making a contribution (I36). The code changes contributed tended to generalize the project's functionality in order to make the project "more applicable with more people" (I43).

*"It really depends on if it's a bug that I know is going to affect other people or if it's something that's just very specific to the way we're doing it and it's probably not going to*

*be very helpful for anyone else.” (I36)*

Signals that developers used to infer how a project’s community are based on their perception of the project’s user base. As discussed earlier, user-generated signals are expected to have high *honesty* with a low *cost* to the user and an unknown *fit*.

## Results Overview

As this study follows an exploratory sequential mixed methods study approach [Creswell, 2013], the goal of the first phase interview study is to explore how developers use information to infer unobservable qualities about projects when performing project-related tasks. The interview study finds that developers make inferences about three types of unobservable qualities (RQ1): the project working dynamics, personal utility, and community evaluation. Interviewees identified signals that they use to infer these qualities for each task.

While interviewed developers identify signals that they individually used to inform tasks of using or contributing to projects, the signals that interviewees perceive as useful may not actually be used by developers within the ecosystem of GitHub. Signaling theory also gives insight into what kinds of signals may potentially be more or less useful. The constructs of *honesty*, *cost*, and *fit* are related to each other [Connelly et al., 2010] but the usefulness of a signal to a receiver is a function of both *honesty* and *fit* [Davila et al., 2003]. High *costs* are often related to both high *honesty* and *fit* and therefore usefulness [Spence, 1973], assuming that signalers possessing the quality in question are better positioned to absorb the cost of producing a high cost signal. For example, if wearing expensive jewelry is a high cost signal for wealth, then signalers not possessing the quality of wealth will be unable to produce that signal. In this case, the high costs should make deceptive signaling difficult, allowing for higher honesty. The *fit* should also be high, as the signal is only producible by those possessing the quality in question. For example, obtaining a certification is a signal with a high cost. Signalers who are skilled should have a considerably easier time obtaining the certification than those who are not. *Fit* and *honesty* are related but distinct concepts in that *fit* describes the signal’s relationship with the quality it indicates. *Honesty* describes the signaler and their relationship to the unobservable quality in question.

The types of unobservable qualities also correspond with three types of signals: unintentional signals derived from transparency, intentional signals derived from extra work by the core team, and community signals derived from aggregate user actions. Each type of signal is associated with its own costs, fit, and honesty which is summarized in Table 6.3.

Signaling theory suggests a relationship between *cost* and usefulness that we expect to observe in the signals identified by interviewees. High cost signals such as extra work by core members should be more useful than low cost signals such as stars count. Similarly, we expect signals with a clearly high *fit* and *honesty* to be more useful than signals where the fit is unknown. This study provides the opportunity to quantitatively compare the perceptions of usefulness among interviewed developers to what signaling theory suggests as useful.

Table 6.3: Summary of Signal Types and Characteristics

Signal Type	Example	Cost	Fit	Honesty
Unintentional	Commit Activity	High	High	High
Extra Work	README	High	High*	High*
Community	Stars Count	Low	Unknown	High

\*if cost is aligned with quality

The following section describes the second phase of the study, which is a quantitative validation study. If developers on GitHub use signals and make decisions in the way our interviewees described, we should be able to observe a number of statistical associations with decision outcomes in the repository data. For example, if developers care about a project’s responsiveness when deciding whether to contribute code to it, and they use the time to close pull requests as a signal of responsiveness, then we should observe a negative correlation between time to close and the number of pull requests submitted to a project (controlling for other factors influencing the number of pull requests submitted). By seeing if these associations exist, are statistically significant, and have meaningful strength, I can identify which of the signals in our qualitative study that interviewees perceived as useful are actually used. I can also compare these results against the types of signals that signaling theory suggests are more or less useful (RQ2).

### 6.3 Quantitative Validation Analysis

In the second phase of the study, I performed a quantitative validation analysis to investigate the relationship between metrics representing signals used by developers in our exploratory interview study and project task outcomes. The task outcomes in the quantitative phase are related to the two work decisions of interest investigated in the qualitative phase: 1) deciding to use a project by incorporating it as a dependency and 2) deciding to offer a code contribution to a project. In this section, I describe the creation of the dataset,

regression analysis, and the findings from the analysis.

### 6.3.1 Methods

#### Data Collection

I created a dataset of software projects on GitHub and the users and activities associated with each project through sampling for active, collaborative projects on GitHub. As Creswell [2013] points out, it is essential for an exploratory mixed methods study that different participants be used for the qualitative and quantitative phases. Therefore I restricted the project sample to Ruby gems, self-contained libraries for the Ruby programming language. I chose to focus on Ruby gems as Ruby is one of the most popular programming languages on GitHub (third most popular at the time of writing) and users develop Ruby gems for a wide variety of purposes, ranging from web application frameworks such as Ruby on Rails to command line interfaces such as Thor. Restricting the project sample to Ruby gems allows for computing dependencies in a straightforward manner as well as eliminating variation due to differences in programming languages while still representing a wide variety of types of software projects.

The dataset comprised of information gathered from the GitHub and RubyGems Application Programmer Interface (API). I used GitHub projects as the unit of analysis. First, I drew a sample of repositories from the GitHub Archive dataset<sup>1</sup> on April 21, 2013 that excluded forks (to avoid double-counting), repositories without at least one event of activity within one week prior to data collection (avoid inactive projects), repositories that did not use the issue tracker (issues are used as data), and non-Ruby projects. I further refined the selection to remove non-collaborative projects (projects without at least one pull request and three unique contributors) and projects that were not Ruby gems or registered on the RubyGems<sup>2</sup> hosting service. After this second phase of filtering, the final sample included 1,862 Ruby gem projects.

#### Outcome Measures

I selected two outcome measures to correspond with the two project-related tasks: using and contributing to projects. Table 6.4 describes all measures used in the analysis.

**Using – *Dependent Projects*:** This measure indicates the number of downstream projects

<sup>1</sup><https://www.githubarchive.org/>

<sup>2</sup><https://rubygems.org/>

Table 6.4: Descriptives of Project Evaluation Measures (Pre-transformation)

Type	Metric	mean	median	stdev	skew
Outcome	Downstream Dependencies	57.73	1	636.74	24.58
	Pull Requests	42.89	13	118.45	11.6
Control	Project Age (days)	824.04	719.37	571.71	0.71
	Project Size (kb)	3946.53	827	15246.38	11.04
	Project Contributors	10.96	7	9.34	1.12
Signals	Commit Velocity (commits/day)	0.52	0.21	1.39	17.58
	Commit Volume	89.99	37	192.63	9.13
	Project Versions	26.48	16	37.19	5.36
	Recent Code Churn (lines/commit)	70.49	9.67	733.28	22.77
	Upstream Dependencies	6.02	5	5.12	1.75
	Stars	307.72	46	928.45	10.97
	Time to Close Pull Request (days)	19.35	6.23	45	9.4
	Project Issues	64.06	10	416.87	30.12

that have established the given project as a dependency. I use this as a measure of actual project usage as opposed to indirect usage metrics such as download count.

**Contributing** – *Pull requests*: This measure indicates the total number of pull requests submitted to a given project, including open and closed pull requests. Both accepted and rejected closed pull requests are included in this measure, as both reflect decisions to contribute code to a project.

### Control Measures

I developed a set of control measures to ensure that observed associations between project characteristics were not due to correlations with these control variables that may influence our dependent variables. As I have chosen to restrict the sample to Ruby gems, the sample also inherently controls for programming language.

**Project Age** - Age of the project is used as a way to control for temporal effects in the dataset. Projects that have existed for longer may have more users or contributions than newer projects simply because they have had more time to attract developers.

**Project Size** - Server-side size of the project repository in kilobytes is used as a way to control for differences in breadth of project functionality.

**Number of Contributors** - This control measure is the total count of contributors, or developers who have successfully landed a commit to a given repository. I used this measure to control for the size of the project's core team.

## Signal Measures

I developed operationalized measures for signals identified in the qualitative phase as predictor variables for the statistical models. The signal measures are divided into two statistical models: *Using* and *Contributing* and further divided into the three types of unobservable qualities: Project Dynamics, Personal Utility, and Community Evaluation. For each measure, I developed a hypothesis for the measure's relationship with the outcome based on interview findings. Hypotheses are summarized in Table 6.5.

### *Using Model measures*

**Project working dynamics** – The first phase found that when interviewees evaluated projects for usage, they made inferences about the liveliness of the project in order to find projects with active development. I develop two metrics for project liveliness that represent two dimensions of a project's commit activity. The first metric is the *volume of a project's commits* as the sum total of commits in the project. The second metric is the *velocity of a project's commits* as the average number of commits per day for the past year of commit activity. For both metrics of project liveliness, I expect that the association with usage is positive; a project with a higher volume and/or velocity should be more active or "alive" and therefore have more usage as represented by number of downstream dependencies (H1, H2).

**H1:** Volume of commits will be positively associated with number of users.

**H2:** Velocity of commits will be positively associated with number of users.

**Personal utility** – When interviewees evaluated projects for usage, they made inferences regarding the potential cost of creating a dependency to a project in terms of both maturity and ease-of-use. For maturity, I use two measures to represent different dimensions of project stability. The first metric is a count of the *number of released versions* available for a project. The second metric is the *recent code churn* for a project as represented by the average number of lines changed per commit for the most recent week of commit activity. I expect that a project with more versions should be more mature and should have a more usage (H3). I expect that projects with larger recent commits should be less mature and should have less usage (H4).

For ease-of-use, I use the count of *required upstream dependencies* for a given project according to the RubyGems hosting service. I expect that a project that requires more

upstream dependencies should be harder to use and should have less usage (H5).

**H3:** Number of released versions will be positively associated with number of users.

**H4:** Recent code churn will be negatively associated with number of users.

**H5:** Number of upstream dependencies will be negatively associated with number of users.

**Community evaluation** – The first phase also found that interviewees inferred the size of the project’s community in order to gauge the potential for support in using the project. I use *number of stars* for a given project as a metric for a project’s community size. I expect that a project with more stars should have more usage (H6).

**H6:** Number of stars will be positively associated with number of users.

#### *Contributing Model measures*

**Project working dynamics** – The first phase found that when interviewees made the decision whether to contribute to a project, they made inferences about how responsive the project core team would be towards their potential contribution. I develop a metric for responsiveness as the average *time to close pull requests* for a given project. I expect that a project that takes longer to close pull requests should have fewer contributions as represented by the number of pull requests a project receives (H7).

**H7:** Average time to close pull requests will be negatively associated with number of pull requests.

**Personal utility** – The first phase found that interviewees inferred the accessibility for making contributions to a given project. I use both the *number of versions* and the *number of issues* in a project as metrics for these accessible contribution opportunities. Number of versions is used as a metric for a project’s maturity in that more mature projects were easier to offer contributions to. Number of issues is used as a metric as I found that developers focused on the project’s issue tracker in order to find these opportunities. For both measures, I expect the association with contributions to be positive; a project with more versions and/or issues should be more accessible to contributions and therefore have more contributions (H8, H9).

**H8:** Number of released versions will be positively associated with number of pull requests.

**H9:** Number of issues will be positively associated with number of pull requests.

**Community evaluation** – The first phase also found that interviewees inferred the potential impact their contribution would have on the project’s community. I create two metrics for potential impact in a particular project. The first is the *number of stars* for a project as an indication of the users interested in a particular project. The second is the *number*

of downstream dependencies for a project as an indication of the user population for a project. I expect both of these metrics to have a positive association with contributions; a project with more stars and/or downstream dependencies should have a higher potential impact and more contributions (H10, H11).

**H10:** Number of stars will be positively associated with number of pull requests.

**H11:** Number of downstream dependencies will be positively associated with number of pull requests.

Table 6.5: Quantitative Hypotheses for Project Evaluation Signals

Model	Unobservable Quality	Inference	Metric	Hypothesis
Usage	Project Dynamics	Project Liveliness	Commit Volume Commit Velocity	H1: Positive H2: Positive
	Personal Utility	Project Maturity	Project Versions Recent Code Churn	H3: Positive H4: Negative
		Ease-of-use	Upstream Dependencies	H5: Negative
	Community Evaluation	Community Interest	Stars	H6: Positive
Contribution	Project Dynamics	Responsiveness	Time to Close	H7: Negative
	Personal Utility	Accessible Opportunities	Project Versions Project Issues	H8: Positive H9: Positive
		Community Evaluation	Community Benefit	Stars Downstream Dependencies

## Data Analysis

I created separate negative binomial regression models for the Using and Contributing outcome measures for analysis and comparison. I elected to use negative binomial regression, as the criterion variables were based on counts, and therefore highly skewed. Negative binomial regression is well suited for outcomes based on counts, especially in the presence of overdispersion such as in this dataset [Jewell and Hubbard, 2010]. As the predictor variables varied widely in scale, each of the non-binary predictor variables was transformed using Poisson scaling (each variable is divided by its root mean square). This

allows for convenient comparisons between the relative magnitudes of each variable's association with the outcome variable. I performed a multicollinearity diagnostic for each model. The variance inflation factor (VIF) analysis reported factors no higher than 2.56, therefore none of the predictor or control measures were removed from the model due to collinearity [O'brien, 2007].

I use a hierarchical modeling approach that first creates control models that use the control variables of project age, project size, and contributor size to predict the outcome variables for usage (downstream dependencies) and contributing (pull requests). Then I create models that add the predictor variables of interest and compare against the control model. For both the Using and Contributing models, I find evidence that the models with predictor variables are a better fit than the control models. For the Using model, the Akaike information criterion (AIC) decreases from 10828 for the control model to 10391 for the predictor model. For the Contributing model, the AIC decreases from 15082 for the control model to 15021 for the predictor model. In both cases, this indicates an effectively 0.0% relative likelihood that the control model will perform better than the predictor model.

### **6.3.2 Results**

I describe the results for the Using and Contributing regression models. The statistical significance and incident rate ratios (IRR) is reported for each variable for both models. An IRR greater than 1 indicates a positive relationship between a predictor and the outcome measure, while a ratio less than one indicates a negative relationship. Incident rate ratios provide a convenient method for comparing the association between different measures and outcomes. Comparing associations answers the second research question: *What are the strengths of statistical associations of signals with the key decisions of using and contributing?*

#### **Using Model**

For the model of developers choosing to use projects, usage is predicted through the outcome variable of number of downstream dependencies for a project. The results for the Using model are summarized in Table 6.6.

For metrics representing project working dynamics, I find that both commit velocity and volume are highly statistically significant. However, whereas commit volume is highly positively associated with usage through number of downstream dependencies as expected (H1), commit velocity is unexpectedly highly negatively associated with usage (H2). For

each standard deviation of volume, usage has an IRR of 15.97 or an expected 1496.7% increase in downstream dependencies (7.8% per commit). On the other hand, velocity has an IRR of 0.10 or an 89.52% decrease in usage per standard deviation (64.29% for each commit per day).

For metrics representing personal utility signals, I find that number of versions, recent code churn, and number of required upstream dependencies are all highly statistically significant. The associations for each of these metrics also follow expectations for direction (H3, H4, H5). Number of versions is highly positively associated with an IRR of 1.91 or an expected 91.29% increase in usage by downstream dependencies per standard deviation of versions (2.93% increase per version). Recent code churn is highly negatively associated with an IRR of 0.34 or a 66.31% decrease in usage per standard deviation of recent code churn (9.04% per 100 lines of code changed). Number of required upstream dependencies is also highly negatively associated with an IRR of 0.54 or a 45.54% decrease in usage per standard deviation of upstream dependencies (8.89% per upstream dependency).

Lastly, the metric representing community evaluation signals, number of stars, is surprisingly not statistically significant (H6). Though the qualitative study and prior work [Dabish et al., 2012] has found that developers report using a project’s number of stars as a signal for community interest in a project, this result suggests that despite the popularity of stars as a signal, developers in GitHub do not actually use stars to inform their usage decisions.

Table 6.6: Usage Model Analysis Summary

Unobservable Quality	Inference	Metric	IRR	Hypothesis Support?
Project Dynamics	Project	Commit Volume	15.967***	H1: Support
	Liveliness	Commit Velocity	0.105***	H2: No Support (opposite)
Personal Utility	Project	Project Versions	1.913***	H3: Support
	Maturity	Recent Code Churn	0.337***	H4: Support
	Ease-of-use	Upstream Dependencies	0.545***	H5: Support
Community Evaluation	Community Interest	Stars	1.085	H6: No Support
Controls		Project Age	9.557***	
		Project Size	0.667***	
		Contributors	3.518***	

## **Contributing Model**

For the model of developers choosing to contribute to projects, contributions is predicted through the outcome variable of total number of pull requests for a project. The results of the Contributing model are summarized in Table 6.7.

For the metric representing project working dynamics, I find that time to close pull requests is highly statistically significant. As expected, time to close pull requests is negatively associated with contributions with an IRR of 0.90 or an expected 9.56% decrease in contributions by a pull request per standard deviation of time to close pull requests (2.12% per additional 10 days to close) (H7). This suggests that developers are in fact discouraged to contribute to a project that fails to respond to a contribution in a timely manner.

For metrics representing personal utility, I find that number of versions and issues are both highly statistically significant. The associations for both of these metrics also follow our expectations for direction (H8, H9). Number of versions is positively associated with contributions with an IRR of 1.19 or a 19.27% increase in contributions by a pull request per standard deviation of versions (5.18% per 10 versions). Number of issues is also positively associated with contributions with an IRR of 1.24 or a 23.69% increase in contributions per standard deviation of issues (5.71% per 100 issues). These results suggest that developers value accessibility for contribution opportunities through issues and versions, perhaps more than working dynamics due to the increase of association magnitude (9.56% vs. 19.27% or 23.69%).

Lastly, for metrics representing community evaluation, I find that number of stars has a marginal statistical significance at a p-value of 0.057 whereas number of downstream dependencies is surprisingly not significant (H10, H11). Number of stars is positively associated with contributions with an IRR of 1.06 or a 6.26% increase in contributions per standard deviation of stars (0.67% per 100 stars). The marginal statistical significance for community evaluation suggests that either our measures are inappropriate approximations for community impact or that developers do not value community impact when making the decision to contribute.

## **6.4 Discussion**

### **6.4.1 Evaluating Projects and Signal Fit**

The combined results of the study suggest that work-related signals are more associated with task outcomes of using or contributing to projects. The exploratory qualitative in-

Table 6.7: Contribution Model Analysis Summary

Unobservable Quality	Inference	Metric	IRR	Hypothesis Support?
Project Dynamics	Responsiveness	Time to Close	0.904***	H7: Support
Personal Utility	Accessible	Project Versions	1.193***	H8: Support
	Opportunities	Project Issues	1.237***	H9: Support
Community Evaluation	Community Benefit	Stars	1.063	H10: Marginal
		Downstream Dependencies	0.988	H11: No Support
Controls		Project Age	0.871***	
		Project Size	1.299***	
		Contributors	3.979***	

interview study identifies three broad types of unobservable qualities that developers infer about projects: project working dynamics, personal utility, and project community evaluation (RQ1). The signals indicating these qualities differ in terms of honesty, cost, and fit as discussed in subsection 6.1.3. The validation quantitative study finds that project working dynamics signals have the strongest relationship with usage and personal utility signals have the strongest relationship with contributions. However, I also find in both models that community evaluation signals were not well represented. This finding suggests that signals indicating working dynamics or personal utility may have stronger relationships to task outcomes than community evaluation (RQ2). This suggests that while developers may perceive community signals as informative, in practice, these signals are not used in informing tasks.

There are a number of possible explanations for the community evaluation signals having no effect in the models. This may be due to selecting unrepresentative signals to represent the signal type, or an inappropriate operationalization of the signal, or that the community-generated signals used to indicate community evaluation have a low signal *fit* [Connelly et al., 2010]. The first point seems unlikely as multiple interviewees mentioned using stars and forks to indicate qualities of the project’s community. This result is also found in prior work of GitHub developers [Dabbish et al., 2012]. The second point also seems unlikely as interviewees explicitly stated using the count of stars and forks for a project as a signal. On GitHub, stars and forks count are highly visible for each project, making their usage highly likely.

The last point, the lack of fit, is an explanation supported by theory. While the community-generated nature of these signals implies that these signals are *honest* in that manipulating

these counts is difficult for project owners, these signals may have a low *fit*, not actually indicating any community evaluation-related qualities regarding the project. Literature on signaling theory finds that poor signal fit similarly results in poor correlation with outcomes of interest, such as stock market response [Yan Zhang, 2009] or venture exit performance [Busenitz et al., 2005]. One possible explanation offered by signaling theory for why community-generated signals identified in this study have low fit is that the *cost* of producing these signals is low, since the act of “starring” a GitHub project only requires a user to click the “star” button on the project page. In contrast, signals made possible via transparency such as commit activity often share signaling costs with the cost of performing development work. Similarly, intentional signals identified by developers such as README files and clearly labeled issues often involve extra work [Trainer et al., 2015] on the part of project managers. In both cases, the cost of project managers to generate the signal is relatively high.

Lack of signal fit also explains the surprising negative association of commit velocity with usage (H2). While interviewees perceived commit velocity as a signal of the quality of project activity, we do not find support for that relationship in our quantitative model. In fact, the highly significant and strong negative association suggests that while commit velocity and project activity may have poor fit, perhaps commit velocity actually signals a different quality. The conceptual similarity of the measures (commit velocity and volume of recent commits) and similar highly negative associations with usage (H4) suggest that high code velocity actually signals project instability or immaturity rather than liveliness. (Although commit velocity and code churn are conceptually similar, they are not collinear with a pairwise correlation of 0.02.)

Future work should examine in more detail this relationship between signal cost, signal fit, and informing decisions. Although the low-cost community-generated signals in this study were not associated with task outcomes, it is possible that other community-generated signals are more useful. Similarly, low-cost signals that were not explored in this study may still be informative. For example, Donath [2005] discusses conventional signals such as wedding rings that are low-cost yet still informative due to societal conventions and norms. There may exist similar conventional signals in transparent environments such as GitHub that are enforced by community norms. For example, a GitHub user may mark themselves as “hireable” which is a low-cost signal of hiring availability. However, if every user marks themselves as “hireable” (or forgets to unmark the field after being hired), then the signal fit and honesty is too low to be useable. A community norm of only indicating “hireable” if actually looking for a job must be enforced for that signal to be useful. It is also possible that community-generated signals such as stars count are much more useful after a certain volume. User-generated movie ratings have higher credibil-

ity and influence compared to expert ratings only when the volume is high [Flanagin and Metzger, 2013]. Similarly, perhaps stars count and similar signals are informative only at high volumes.

The fit of community signals, specifically stars count, is examined in more detail in Chapter 7.

## 6.4.2 Implications for Transparent Development Environments

The findings suggest implications for the study and design of transparent development environments. The primary implication of our study is theoretical and methodological. I suggest that signaling theory is a useful lens to understand how developers use information in transparent environments. I use signaling theory to break down the information usage process into signals and inferred unobservable qualities. I further break down signals into unintentional, intentional, and community-generated and identify attributes of honesty, fit, and cost for each of the signal types. I also relate these signals to task outcomes. Although this study focuses on how developers evaluate projects for usage or contribution, the study's methodology is potentially applicable to other software development tasks. For example, what qualities do developers need to infer to recruit other developers [Marlow et al., 2013]? What signals indicate these qualities [Dabbish et al., 2012, Marlow et al., 2013]? Which signals are most related to recruitment? Additionally, this methodology may be applicable to other domains or transparent environments. For example, what signals and qualities do writers use when evaluating prose?

This study identifies actionable signals for developers which may have implications for how users of transparent environments manage the often overwhelming information flow that transparency provides. If most of the information available to developers is not relevant to their current task, then transparent environments should filter non-actionable information to prevent overload. This study gives direct suggestions to which signals should be displayed on transparent environments such as GitHub for developers evaluating projects for usage or contribution. Perhaps community evaluation signals such as stars do not need to be prominently displayed. This is in direct contrast to how projects are currently displayed on GitHub where stars count is one of the most prominent signals displayed for each project. Instead, signals for working dynamics such as commit volume or personal utility such as recent code churn should have higher visibility. Upstream dependencies for a project is a useful personal utility signal identified by interviewees but not directly visible on GitHub. Similarly, the average response time to a pull request is a useful working dynamics signal that is not currently directly visible on GitHub.

The findings also reveal gaps between unobservable qualities developers wish to determine and actionable signals. These gaps are potential opportunities to improve how and what signals transparent environments display. While interviewees used signals to evaluate a project's community for both tasks, this study found no relationship between community evaluation signals and task outcomes. For example, while interviewees reported using a project's stars count to infer community interest, this signal was not related to actual usage in our model. This suggests opportunities to create reliable signals to replace unreliable community evaluation signals. For example, aggregating community interactions for a project may be a much more reliable signal for community interest as this signal is aggregated from costly community interactions rather than simple clicks of the star button. Creating such signals allows for the design of environments that more directly and accurately answer questions developers have. Future work may also investigate approaches more supplicated than data aggregation for representing unobservable qualities that developers wish to know. For example, machine-learning techniques may be useful for creating signals that do not currently exist in quantitative forms, e.g. language models representing coding style norms for projects [Hellendoorn et al., 2015].

Chapter 7 explores this idea of creating more reliable signals. Specifically, I attempt to improve on the low-cost signal of stars count by replacing it with an aggregation of more costly community support actions.

### **6.4.3 Limitations**

One of the primary limitations in this study is the cross-sectional nature of the quantitative analysis performed. Without a longitudinal dataset, it is difficult to definitively draw causal conclusions from quantitative analyses. However, the interview phase of the study does suggest a causal direction to many of the signals that are analyzed. Interviewees identified signals that they use to inform decisions. As the signals and their associations with outcomes in the quantitative phase of the study were predicted from the interview study, it seems reasonable to suggest a causal interpretation, without claiming to have definitively demonstrated one. I want to be clear, however, that a causal interpretation of the correlations relies entirely on the qualitative study. Another potential limitation is that some of the measures chosen may not have been satisfactory operationalizations of signals studied, in particular community evaluation signals. In the regression analysis, some of the measures were not statistically significant in the model. The non-significance is very difficult to interpret. It may indicate that the signal is not widely used in the community, or that it has only weak effects, or that the signal's fit is inherently low.

## 6.5 Conclusion

This study uses signaling theory as a theoretical lens to understand how software developers in transparent development environments use information in their environment to evaluate projects. The results further our understanding of how open source developers evaluate software projects for usage and contributions. I found that developers use signals to infer unobservable qualities about projects: working dynamics, personal utility, and evaluations of the project's community. I also found specific signals of qualities such as maturity and responsiveness that are highly associated with outcomes of usage or contribution. The study also furthers our understanding of how developers use information in transparent environments. I found that certain types of signals were strongly associated with desired outcomes while others such as stars count were not. This suggests that while developers may claim to use a variety of signals, only certain types actually inform decisions. I turn to signaling theory to explain this disparity, that signals related to development work made visible by transparency such as commit activity are more used than low-cost community-generated signals such as stars count. The methodology of this study is potentially applicable to understanding how developers perform other software development tasks in transparent environments. Future work may identify the signals and unobservable qualities that are useful for tasks such as recruiting developers and evaluating code contributions. A more complete understanding of what signals developers use to inform their work creates opportunities to assist developers in managing the overwhelming information transparent environments provide. Identifying which signals are most useful to developers for their current tasks may enable future transparency features that visualize the most actionable signals for developers and filter out less useful information.



## Chapter 7

# Evaluating and Creating Signals for Community Support in Software Projects

### Chapter Summary

To survive and thrive, open source software projects often rely on their community members for support and maintenance. Projects with more community support should be more successful and useful for other developers. This concept of community support for projects while useful, is relatively unexplored with little known of what indicates or affects support. To further our understanding of community support, I perform a mixed-methods study. The first phase is a preliminary interview study with developers on GitHub to examine how they perceive and infer community support. The interviews find that stars count is used to infer degree of community support for projects and that the project's current life-cycle state and size of their core team may affect support needs. These findings inform the main longitudinal quantitative study of predictors for community support. The findings also inspire the design of a potentially improved signal for community support called "supportiveness" which measures the degree that developers participate in support actions for any project in the GitHub ecosystem. I create a longitudinal statistical model that uses stars count and supportiveness as predictors for community support in projects over time and include measures to represent the project's state. This model is replicated for three programming languages: JavaScript, Ruby, and Python. The analysis finds that while both stars count and supportiveness predict community support for projects, supportiveness has

a stronger and more robust association. I also find evidence that both the project's current lifecycle state and size of their core team may affect how stars (but not supportiveness) indicates support.

---

In many ways, the ability of community members of an open source project to support and give back to the project is what characterizes open source software. In open source software projects, many of the “mundane but necessary” tasks that keep projects running such as providing field support [Lakhani and von Hippel, 2003], discussing problems [Tsay et al., 2014b], and reporting and fixing bugs [Lerner and Tirole, 2002] are handled by volunteers [Lakhani and von Hippel, 2003, Lakhani and Wolf, 2005, Lerner and Tirole, 2002]. Open source software projects with supportive communities are then expected to be more successful and more useful for other developers.

Improving our ability to measure and predict the relatively unexplored concept of community support will assist developers in evaluating projects and guide project managers in practices to garner more support for their project. While gauging community support is widely regarded as important for helping developers make better decisions about what projects to use [Dabbish et al., 2012], it is a somewhat fuzzy concept with little known about which projects have more or less support and why. Analyzing the development activity and social relationship information that transparency makes visible may reveal what aspects of projects and their community members affect supportiveness in projects. For example, do projects that have large core development team sizes [Mockus et al., 2002] or have high popularity [Dabbish et al., 2012] tend to also have high community support? Taking a step further and examining what *predicts* community support may lead to a better understanding of what affects support in projects. For example, if a project first becomes popular and then receives a high amount of support from its community, then we can reason that popularity may drive support. Identifying these predictors may enable the design of tools or even future transparent environments that inform developers of the present and future community support for a project. Evaluating the support of a project may inform many project-related coordination tasks for developers, such as deciding which project to establish a dependency with. Understanding predictors for community support may also inform practices for project managers that wish to ensure their project is supported by their community. For example, if popularity is a predictor for support, then drawing additional support may involve first increasing the publicly visible indicators of popularity.

Signaling theory is a useful theoretical lens for identifying what information in transparent environments are useful predictors of community support. Signals are observable

pieces of information that indicate some unobservable quality of the person or entity that generated the signal [Connelly et al., 2010]. This study identifies what signals software projects emit that indicate the unobservable quality of community support. Signaling theory provides useful constructs regarding signals such as *cost* and *fit* [Connelly et al., 2010]. Signals which are more costly tend to have better fit, which is the extent that signals correlate to the unobservable quality they indicate. For example, the high cost of owning expensive jewelry makes it a costly signal to emit. Therefore, expensive jewelry also has a high fit with the quality of wealth as only wealthy people can afford to own expensive jewelry. Signaling theory and its constructs are useful to reason about why certain pieces of information are more useful than others in indicating community support.

For this study, rather than focus on signals and their associations with task outcomes like in previous chapters, I examine the concept of *fit* regarding the unobservable quality of community support. Though Chapter 6 finds that interviewees stated that they valued community support as a useful quality of projects, stars count as a signal for community support did not associate with usage or contribution. As discussed in Chapter 6, this lack of association may be due to a low *fit* between stars count and community support. This low fit may be due to the low *cost* of producing the signal of stars count. So rather than examining the relationship between signals and coordination tasks, this study examines how accurately signals of differing costs may indicate the unobservable quality of community support.

To further our understanding of community support for software projects, I present a mixed-methods empirical study of predictors for support. As the concept is fairly unexplored, I first perform a preliminary interview study with developers on GitHub to examine how they perceive and infer community support. I learn from these interviews that community support is a quality developers care about when evaluating projects and that stars count is used to infer degree of community support. Additionally, the project's current state in their lifecycle and the size of their core team may affect the degree of support it receives. For example, a project that is growing may have differing support needs than one that is declining. I use these findings to inform the main quantitative study which uses a longitudinal dataset of GitHub projects to analyze predictors for community support. Inspired by signaling theory, I also design a potentially improved signal for community support called "supportiveness". Supportiveness is a measure of the degree that open source developers participate in support actions for any project across the GitHub ecosystem. This designed measure is novel in that I am taking advantage of transparency to predict a characteristic of a project by aggregating behaviors of the developers connected to that project. I compare the predictor that interviewees identified, a project's stars count, against my designed signal of supportiveness and include measures to represent the project's current

state. The analysis finds that while both stars count and supportiveness predict community support for projects, supportiveness has a stronger and more robust association. I also find evidence that both the project's current state and its core team size may moderate the association between stars and community support (but does not affect supportiveness).

The findings for this study suggest that my designed signal of community support, supportiveness, has a higher *fit* and robustness than the commonly used signal of stars count. Based on signaling theory, this suggests that higher cost signals do have higher *fit* and perhaps are more robust. Combined with Chapter 6, my work suggests a general methodology for designing improved signals for tools or environments. Chapter 6 describes how to identify useful qualities for tasks and the signals that indicate these qualities. Should any of the signals identified have low fit due to low cost, then improving that signal is a matter of designing a signal with a higher cost. Future work should attempt to apply this methodology for other tasks, domains, or environments. For example, what would a signal for community support in Wikipedia articles look like?

In the next sections I ground this study on previous research on open source software and online communities, describe a preliminary interview study that motivates the research questions, describe the longitudinal quantitative analysis study's methodology, present the results of the analysis, and discuss the implications of the findings.

## **7.1 Community Support in Open Source**

I ground this study by examining prior work in how open source software projects and online communities attract community members and encourage members to become more involved in the project or community. I also examine how information made visible in transparent development environments may enable the creation of useful signals and predictors for developers.

### **7.1.1 Community Involvement in Open Source Software**

Open source software relies on a variety of contributions from a diverse group of mostly volunteer software developers Crowston et al. [2008]. These “mundane but necessary” contributions are not necessarily code but include maintenance actions such as providing field support [Lakhani and von Hippel, 2003] and reporting bugs [Lerner and Tirole, 2002]. Volunteers provide support for often intrinsic motivations, such as learning through reading questions or giving answers, expecting reciprocity, helping the “cause” of open source

software, and potentially gaining reputation in the community.

Zhou and Mockus [2012] suggest that the difference between one-time contributors and more involved long-term contributors is their “willingness” to help the project. They suggest that the nature of support actions these contributors participate in are a measure for willingness. For example, the low cost of reporting an issue through a tool may be less involved than the higher cost of applying for an account in Gnome Bugzilla, creating a report, and filling in the bug reproduction template. We see an opportunity to measure this “willingness” for the members of a project’s community for it may also predict the degree of support a project receives.

Despite the importance of community involvement for open source software projects, Crowston et al. [2003] point out that user involvement as a success measure is unexplored. Regarding software project success measures, Stewart and Gosain [2006b] suggest that the project lifecycle stage may act as a moderator for the effect of developer team size on performance measures. We intend for this study as a step towards exploring user involvement in regards to project success and also take into account both developer team size and project lifecycle stage.

## **7.1.2 Community Involvement in Online Communities**

Open source projects are a form of online community and face similar challenges of community building. To survive and thrive, online communities must attract new contributions and encourage commitment from community members. Kraut and Resnick [2012] claim using the collective effort model [Karau and Williams, 1993] that commitment to an online community increases willingness to contribute. They also claim that encouraging commitment is a combination of affective commitment (attachment to the group or project), normative commitment (obligations to the community), and needs-based commitment. Normative commitment may be particularly relevant in open source software as Raymond [2004] claims that open source developers have a moral obligation to help each other. Stewart and Gosain [2006a] find these collaborative open source values positively impact team effectiveness measures such as cognitive and affective trust and communication quality. In the community of Wikipedia, Bryant et al. [2005] found that some newcomers will transition from making peripheral contributions to specific articles into highly-involved core users that help maintain Wikipedia and its community as a whole. The support actions members participate in become more varied as newcomers become more involved, from purely making edits in articles to also participating in community discussions, administrative duties, and “meta” tasks. Besides member involvement, Iriberry and Leroy [2009] found that the lifecycle stage of the online community itself also changes the na-

ture of support actions community members perform. For example, during the earlier Growth stage, communities are more concerned with attracting new members and supporting interactions while the later Maturity stage, communities may prefer to recognize contributions and increase visibility of certain members.

### **7.1.3 Developers Using Information in Transparent Environments**

Transparent development environments such as GitHub that make visible low-level development activities are increasingly popular. Previous qualitative research finds that developers are able to use the information transparency makes visible as signals to infer a variety of otherwise unobservable qualities about other developers and projects. The transparency information that developers make use of may also inform the creation of useful predictors for desired qualities of software projects on GitHub. Kikas et al. [2016] use a number of issue and project features to predict the probability of an issue closing at various points in its lifetime. Vasilescu et al. [2015] use a longitudinal dataset to create gender and tenure diversity predictors for productivity. They find that both predictors are positive and significant. I continue this line of work of using transparency information in GitHub to create predictors that may guide developers performing coordination-related tasks. Helping developers to accurately and easily infer the degree of community support in projects may assist in project-related tasks such as establishing a dependency or contributing to a project.

## **7.2 Preliminary Interview Study**

As community support in open source software projects is still relatively unexplored, I first conducted an interview study with developers on GitHub to identify their perceptions of community support and potential signals they use to infer support. I use prior literature and the results of these interviews to inform potential predictors for community support to analyze. The interviews discovered how developers value community support when evaluating a project and that stars and project state are useful signals they used to infer support.

### **7.2.1 Interview Methodology**

To investigate how developers make coordination decisions regarding projects, I conducted a series of semi-structured interviews with 47 GitHub users. In order to gather a wide vari-

ety of practices, I sampled users from the most popular projects in multiple programming languages on GitHub. From these projects, I sampled both peripheral developers and heavy users with more than 80 “stars” on at least one project. Participants were asked to walk through their last session on GitHub, describing how they interpreted information displayed on the site as they reviewed earlier work activities. Developers were asked to describe specific project-related coordination instances such as pull requests sent and decisions made leading up to and during these interactions. Interviews lasted approximately 45 minutes to one hour.

I applied a grounded theory approach [Corbin and Strauss, 2008] to analyze the signals that developers used when coordinating with projects. I performed open coding using coordination instances that interviewees identified as the unit of analysis. The codes correspond to unobservable qualities about projects developers inferred and the signals that developers used to infer these qualities. For this work, I report on coordination instances and signals related to the unobservable quality of community support. Please see Chapter 6 for more detail on the methods and findings from these interviews.

## **7.2.2 Community Support in Open Source Software Projects**

I discovered from interviewed developers that they considered a project’s community as an important quality for evaluating a project. Specifically, developers evaluated a project’s community to assist in making the decision whether to use a project as a dependency or contribute to a project. As open source software projects often rely on a large user community for technical support [Lakhani and von Hippel, 2003], system testing, and problem reporting [Mockus et al., 2002], developers used signals about the size of a project’s community to infer the level of interest and likelihood of support should a problem with the project arise.

To infer the degree of community support a project receives, developers used signals related to a project’s star count and its core developer team. Developers used a project’s watcher/star and fork count to infer how many other developers were interested in the particular project. For developers deciding whether to use a certain project, these metrics for a project’s popularity were signals for the project’s degree of user interest. Developers tended to choose projects with interested communities as these projects were better supported by their users as well as tended to stay alive longer. Developers also used the size and diversity of a project’s core development team as signals to infer the nature of support that is provided for the project. For example, an interviewee mentioned that the larger the group of core developers on a project, the “wider the net” and the more likely issues have been identified and fixed. Conversely, a project with a single developer would have a

limited ability to support their users.

The findings from the preliminary qualitative study suggest opportunities to determine which of the signals interviewees perceived as useful actually indicate community support for projects. Interviewees seemed to primarily use popularity-related signals such as the stars count of a project in order to infer the degree of community support for a project. However, such popularity-related signals may not necessarily be useful in making project-related decisions such as using or contributing to a project due to their low cost to produce. Signaling theory suggests that signals with a low cost to produce may have unknown fit or relation with the unobservable quality [Connelly et al., 2010]. In this case, starring a project requires little to no cost from a user, so users may star a project for any number of reasons with no intention to support a particular project. Therefore, the fit of stars as a signal, whether popularity actually indicates the unobservable quality of a project's degree of community support, is an open question. Additionally, there is the opportunity to create a signal which may be more accurate than stars count by designing a signal such that the costs are relatively high and involve supporting open source projects. The interview findings also suggest that the size of the project's core developer team may also indicate something about the nature of support provided for the project. While core developer team size may also be a signal for community support, it is also possible that projects with small or large developer teams may have differing support needs. For example, perhaps a smaller project is more concerned with getting "more eyes" for their bugs while a larger project benefits more from enthusiastic, highly-involved developers to fix more complex bugs. This suggests that perhaps the signals that indicate community support may differ depending on the core team size due to the differing support needs.

### **7.2.3 Research Question Development**

Based on prior work on open source software communities and findings from the preliminary interview study, I identify opportunities to improve the measurement and prediction of community support in transparent environments such as GitHub. By doing so, I hope to both identify useful predictors for developers making project-related decisions and further our understanding of how community support functions. As mentioned in the prior subsection, while interviewees identified stars count as a signal for community support, stars may not be reliable as a signal. Specifically, this study determines if stars count indicates future community support. I focus on future community support for two reasons. Interviewees wished to infer whether a project will be supported by its community in the future as they believe these projects stay alive longer. Also, it is possible that performing support actions for a project increases the project's popularity with other developers. Focusing on

future community support allows for suggesting a causal relationship between increases in star counts and increases in support.

**RQ1:** Does stars count predict future community support for a project?

Given the potential weaknesses of stars count as a signal for community support, then what would more strongly indicate support? Signals with high *fit* tend to also have high *fit* as long as the costs are aligned with the unobservable quality in question [Connelly et al., 2010]. For example, if owning expensive jewelry has a high cost, then it is a good signal for the quality of wealth. However, if the quality in question is coding ability, then the high monetary costs of owning jewelry are not informative.

As we mentioned earlier, Zhou and Mockus [2012] suggest that long-term contributors are characterized by “willingness” to help the project. As they measure “willingness” by looking at the support actions a developer performs, perhaps I can also measure “willingness” in a similar manner. As opposed to the low cost of starring a project, a measure of “willingness” that involves the relatively high cost of performing actual support actions for projects may be a much more accurate signal for community support than stars count.

**RQ2:** Is it possible to design a signal that more strongly predicts community support than stars count?

Interviewees and prior work in open source software and online communities suggest that the state of the project in question is likely to have some effect on its support needs from the community. In particular, prior literature suggests that in both open source software [Stewart and Gosain, 2006b] and online communities [Iriberry and Leroy, 2009], the lifecycle stage of the project or community affects the nature of support that is needed. As a project becomes more well-established, it may be less concerned with attracting support through new community members [Iriberry and Leroy, 2009]. In prior literature [Stewart and Gosain, 2006b] and interviews, I find that the size of the developer core team may also affect support needs. The larger the core team, the greater their ability to support their project and perhaps they require less support from their community as a result. If project state, specifically project lifecycle and core team size, changes the support needs for a project, then the signals that predict community support may also change along with project state. For example, if projects with smaller core teams wish to have “more eyes” for their projects, then stars count may be a stronger predictor for these projects due to its connection with popularity.

**RQ3:** Does project state affect how signals predict community support?

## 7.3 Community Support Modeling

To further our understanding of community support in transparent environments, I perform a longitudinal quantitative analysis of predictors for community support of GitHub projects. I create three datasets of JavaScript, Ruby, and Python projects and develop measures for community support, predictors, and project state. I create a longitudinal statistical model relating predictors to future community support and replicate this model across the three datasets.

### 7.3.1 Dataset Collection

I created a longitudinal dataset of software projects on GitHub and the users and support activities associated with each project, including data from January 1, 2015 to July 14, 2016. GitHub projects are used as the unit of analysis. I created three datasets for the purposes of replication, each focusing on a specific programming language: JavaScript, Ruby, and Python. These languages are among the most popular programming languages on GitHub (first, third, and fourth at the time of writing). For each dataset, I further restricted the project sample to self-contained packages, specifically those managed by the language’s respective package manager such as Ruby Gems or Python packages. In these languages, users develop packages for a wide variety of purposes, ranging from web application frameworks such as Ruby on Rails to command line interfaces such as Thor to build utilities such as Rake. Restricting the project sample to these self-contained packages eliminates variation due to differences in programming languages or types of projects while still representing a wide variety of both purposes of software projects as well as developers on GitHub.

The dataset comprised of information gathered from a combination of the GitHub and Application Programmer Interface (API), GHTorrent [Gousios and Spinellis, 2012], and the GitHub Archive<sup>1</sup>.

First, I drew a sample of repositories from GHTorrent that excluded forks (to avoid double-counting), repositories without at least one commit or at least one support action (issue, pull request, or comment) since January 1, 2015 (to avoid projects that were inactive for the entire time period of interest), had less than five stars (to exclude “code dump” projects), and projects that were not written in JavaScript, Python, nor Ruby. At this phase, I also removed non-package projects from the sample. This selection included 45,530 JavaScript projects, 16,442 Ruby projects, and 7,715 Python projects.

<sup>1</sup><https://www.githubarchive.org/>

To create both outcome and predictor measures for the longitudinal dataset, I also aggregated all support events (issue, pull request, or comment) for the time period of 1/1/2015 to 7/14/2016. From this sample of repositories, I used the GitHub Archive to aggregate support (issue, pull request, or comment) and starring events (as stars are used as a predictor) for a repository for the time period of interest (1/1/2015 to 7/14/2016). In total, 11,878,304 events for JavaScript projects, 3,080,129 for Ruby projects, and 2,323,364 for Python projects were collected. To create the designed supportiveness predictor, I also required all support events for each project member outside of the project in question. From gathering a list of 1,143,410 developers from the aggregated project support events from all 3 languages, support events across the GitHub ecosystem were collected for each developer through the GitHub Archive for the time period of 1/1/2015 to 7/14/2016, a total of 2,080,233,294 events. For this dataset, a “community member” is defined as any developer who has participated in a support action for the project during the particular 30 day time period, regardless of commit access. A sensitivity analysis restricting community members to only those with commit access (core members) or only those without commit access (peripheral members) does not significantly change the results, so I report on total membership for simplicity.

The longitudinal dataset is divided into periods of 30 days for a total of 16 time periods. Time periods of 15 and 70 days were also tested with no significant changes in the results. The measures in the following section were calculated for each project for each time period. For example, the stars count of Project A is the number of stars Project A receives for the specific 30 day time period rather than the aggregated stars count. To filter out projects with mostly inactive communities, projects where over half the time periods had no support actions whatsoever were removed. Additionally, as a limitation of the R package used [Croissant et al.], datasets must be balanced such that each project spans the full 16 time periods. The final filtered dataset for our analysis contains 7,806 JavaScript projects, 3,112 Ruby projects, and 1,708 Python Projects.

### **7.3.2 Measure Development**

Outcome, predictor, project state, and control measures are developed for the predictive models and summarized in Table 7.1.

#### **Outcome Measure – Community Support of Open Source Software Projects**

For the outcome measure in our statistical models, I aggregate support actions performed by project community members for a software project in GitHub. I define the following

Table 7.1: Descriptives of Community Support Measures (Pre-transformation) (JavaScript Dataset)

Type	Measure	mean	median	stdev	skew
Outcome	Community Support	30.11	5	123.38	16.56
Control	Project Age (days)	1079.63	961.7	434.36	1.13
	Project Size (kb)	9777.06	725	53787.06	20.85
Project State	Collaborators	0.06	0	0.34	16.93
Predictor	Stars	19.98	4	86.23	34.36
	Supportiveness	1889.12	136	10456.08	21.48

actions as support actions: reporting a bug (opening an issue), fixing a bug (opening a pull request), and discussing problems in the project (comments on issues, pull requests, or commits). To calculate this measure, I aggregate these three actions for each project over 30 day time periods. The Cronbach’s alpha of aggregating the three actions is 0.85, suggesting an acceptable internal consistency for this outcome measure.

### Predictor Measures

**Stars** – As suggested by interviewees and other GitHub studies [Dabbish et al., 2012], stars are a widely-used signal for evaluating the community around a GitHub project. This measure is specifically the count of stars a project receives during a 30 day time period.

**Supportiveness** – I design this potentially improved predictor for community support inspired by assessment signals [Donath, 2005] in signaling theory and “willingness” [Zhou and Mockus, 2012]. Assessment signals require the signaler to possess the quality in question in order to produce the signal. If the quality desired is support for a particular project, then I make the assumption that developers who possess the willingness to support other projects will also support the project in question. I base this assumption off of observations from prior literature that the open source software community values helping each other out [Stewart and Gosain, 2006a], to the point where it is a moral obligation [Raymond, 2004]. As this signal requires developers to perform actual support actions, it has a high signaling cost. Signaling theory suggests that this signal should also have high fit [Connelly et al., 2010] due to the cost and alignment with the unobservable quality in question.

To calculate this measure, I aggregate support actions as defined in the outcome mea-

sure for each project member. However, instead of aggregating support actions for a focal project, I aggregate the support actions a developer performs for all projects except the focal project for the 30 day period.

## **Project State Measures**

These measures account for the state of the project itself in the statistical models.

**Project Lifecycle** – This measure represents the project’s state in their lifecycle, specifically in terms of community growth or decline. Specifically, I create two dummy variables. “Positive” for a project where the community is growing and “negative” for a project where the community is in decline. The variables are mutually exclusive but the absence of both indicates a project that is well-established and neither growing nor declining. To define “positive” or “negative”, I calculated the general trend of the community support outcome measure per project over the entire time period of the dataset (1/1/2015 to 7/14/2016) using a linear regression. If the project has a slope greater than 1, then it is “positive”. Less than -1 is “negative”.

**Number of Collaborators** - This project state measure is the total count of collaborators, or developers who have commit access to the repository. I use this measure to represent the size of the project’s core developer team.

## **Control Measures**

The control measures for these models were chosen to ensure that observed associations between contributor characteristics were not due to correlations with these control variables that may influence our dependent variables. As I have chosen to restrict each dataset to a particular programming language, each dataset also inherently controls for programming language.

**Project Age** - I use the age of the project as a way to control for temporal effects in the dataset. Projects that have existed for longer may have more users or contributions than newer projects simply because they have had more time to attract developers.

**Project Size** - The server-side size of the project repository in kilobytes was used as a way to control for differences in breadth of project functionality.

### 7.3.3 Analysis

I performed a random-effects two-way panel analysis across the three datasets. Panel analysis was chosen as it fits our multi-dimensional dataset of multiple projects across multiple time periods. For each 30-day time period, we predict the future outcome measure of community support at  $t+30$  using predictor, project state, and control measures at  $t$  for all projects. I then repeat this analysis for all time periods in the dataset, predicting community support at  $t+60$  using predictors at  $t+30$  and so on. A random-effects model allows for including time-invariant measures such as project state and control measures [Wooldridge, 2015] and is preferred over the fixed-effects model for my data by the Hausmann Test. The two-way model accounts for individual (per project) and time effects [Croissant et al.]. Measures are normalized via the z-transform to allow for comparisons. The variance inflation factor (VIF) analysis reported factors no higher than 1.19, therefore none of the predictor, project state, or control measures were removed from the model due to collinearity [O'Brien, 2007]. Woolridge's first-difference test suggests that the models are autocorrelational and require usage of robust estimators for the covariance matrix of the regression coefficients [Croissant et al.]. Regression coefficients are reported using these autocorrelation-robust covariance estimators.

## 7.4 Results

I describe the results from the predictive models for community support. Measures are transformed and normalized in order to compare relative association strengths. I find that both predictors are significantly associated with the outcome measure for community support. I also find that project state affects stars as a predictor but not supportiveness. The JavaScript, Python, and Python models are summarized in Table 7.3.

### 7.4.1 Model Fit

I use a hierarchical modeling approach in order to determine the improvement in modeling fit from including the predictor measures. I first create control models that use the project state measures (project lifecycle and number of collaborators) and control measures (project age and project size) to predict the outcome measure of community support. Then I create models that add the predictor measures (stars and supportiveness) and compare against the control model. For all three languages, I find evidence that the models with predictor measures are a better fit than control models. Similarly, for each model, I add

interactions that further increase the fit. For the largest dataset, JavaScript, the adjusted R-squared for the control model is 0.022, the model with predictors included is 0.339, and the model with interactions included is 0.400. The three models are summarized in Table 7.2. The hierarchical panel models are replicated across the three datasets.

Table 7.2: Comparison of Fit for Community Support Hierarchical Models (JavaScript Dataset)\*

Type	Measure	Control Model	Predictor Model	Interaction Model
Controls	Project Size	0.196*	0.180*	0.138**
	Project Age	-0.065***	-0.049***	-0.045***
Project State	“Positive”	0.510***	0.336***	0.353***
	“Negative”	0.417***	0.255***	0.254***
	Collaborators	0.055***	0.034***	0.033***
Predictors	Stars		0.076***	0.074***
	Supportiveness		0.391***	0.337***
Project State Interactions	Collabs x Supportiveness			0.035
	Collabs x Stars			0.005
	“Positive” x Supportiveness			-0.004
	“Positive” x Stars			0.022
	“Negative” x Supportiveness			0.030**
	“Negative” x Stars			0.027***
Model Fit	Adj. R2	0.022	0.339	0.4

\* Values reported are regression coefficients (beta weights) for normalized measures.

## 7.4.2 Community Support Predictors

I created models to predict future community support using predictor measures of stars and supportiveness. This model is replicated across three datasets each representing a programming language community: JavaScript, Ruby, and Python. Finding a statistically significant association of stars count with future community support answers the first research question: *Does stars count predict future community support for a project?* Comparing the strengths of association between stars count and the designed signal of supportiveness answers the second research question: *Is it possible to design a signal that more strongly predicts community support than stars count?*

I find across all three models that both predictors are highly statistically significant and have positive associations with the outcome of community support. Across all three models, the association of our designed predictor of supportiveness is higher than stars. For the largest dataset of JavaScript, stars has an association of 0.074 while supportiveness has an association of 0.337. For the replicated datasets of Ruby and Python, stars have associations of 0.070 and 0.064 while supportiveness have associations of 0.214 and 0.148 respectively.

### 7.4.3 Project State Measures

To examine how project state may affect how signals predict community support, I added interactions between the project state measures and predictors. In total, I added 6 interactions, one for each combination of project state measures (positive state, negative state, and number of collaborators) and predictors (stars and supportiveness). Including project state measures and interactions in the model answers the third research question: *Does project state affect how signals predict community support?*

Across all three models, the “positive” and “negative” project lifecycle state measures are highly statistically significant and have positive and fairly strong associations with the outcome of community support. For the dataset of JavaScript, growing projects have an association of 0.336 while declining projects have an association of 0.255. Number of collaborators is only significant for the JavaScript dataset with a relatively small association of 0.034.

I find that across all three models, supportiveness had no significant interactions with any project state measure. Stars significantly interacts with each project state measure for most (but not all) datasets. Stars interacts negatively with number of collaborators for Ruby and Python, indicating that stars affects community support less for projects with more collaborators. Stars interacts positively with positive lifecycle state for all three datasets. Stars also interacts positively with negative lifecycle state for the Python and JavaScript datasets. This indicates that stars affects community support more for projects that either have a positive or negative trend in their lifecycle.

Table 7.3: Summary of Predictive Models for Community Support\*

Type	Measure	JavaScript	Ruby	Python
Predictors	Stars	0.076***	0.061***	0.083***
	Supportiveness	0.391***	0.247***	0.150***
Controls	Project Size	0.187**	0.279***	0.280*
	Project Age	-0.009	-0.01	-0.014
Project State	“Positive”	0.336***	0.366***	0.435***
	“Negative”	0.255***	0.158***	0.321***
	Collaborators	0.034***	0.014	0.032
Project State Interactions	Collabs x Supportiveness	0.005	0.011	0.013
	Collabs x Stars	-0.004	-0.017**	-0.022***
	“Positive” x Supportiveness	0.022	0.077	-0.057
	“Positive” x Stars	0.030**	0.029*	0.056***
	“Negative” x Supportiveness	-0.009	-0.041	0.025
	“Negative” x Stars	0.027***	0.042	0.049*

\* Values reported are regression coefficients (beta weights) for normalized measures.

## 7.5 Discussion

### 7.5.1 Predicting Community Support

The results of the predictive model suggest that both stars and the designed signal of supportiveness are significant predictors for community support across all of the datasets. I also find evidence that the designed signal of supportiveness has a stronger association with community support and is more robust than stars count.

The robustness of supportiveness is due to its strong association with community support regardless of the state of the project. Specifically, while stars count interacts with a subset of the project state measures per dataset, there are no such interactions for supportiveness across any of the three datasets.

I use signaling theory [Connelly et al., 2010] for a possible explanation for the differences between predictors, specifically the differing costs to produce each predictor or signal. A potential issue with stars count as a predictor for community support is that the cost of starring a project is both too low and unaligned with the quality of community support. GitHub users star projects by simply clicking the highly visible “star” button on each project page regardless of the particular user’s intent to actually support the project.

Signaling theory provides a concept of both cost and fit for signals where signal fit is how well a signal (stars or supportiveness) correlates to the unobservable quality (community support). Signals with high costs tend to also have high fit [Connelly et al., 2010]. In developing the designed supportiveness signal, I attempted to design a signal where the costs were both relatively high but also aligned with supporting the project. Developers are not able to produce the supportiveness signal without incurring the cost of actually supporting other projects. If the quality to indicate is support for a particular project, then I assume that developers who support other projects also support the focal project. Therefore, I also assume that our signal should also have high fit with community support. The results also suggest that the supportiveness signal also indicates “willingness” [Zhou and Mockus, 2012] of developers. Projects that draw developers who are willing to support tend to be better supported in the future. The results from the statistical model suggest that both assumptions hold for supportiveness.

However, the significant result for stars count also suggests that while the cost is low, the fit is still sufficiently high for this signal to indicate community support. In combination with results from the previous study, this suggests two possibilities for why stars count as a signal for community support is not associated with usage or contribution. One possibility is that the unobservable quality of community support, though perceived by multiple interviewees as useful, is not actually used by developers when evaluating projects. The other possibility is that while stars count is a signal for community support, the fit or robustness is not high enough to be useful in general cases. For example, the results of this study suggest that stars is more indicative of community support for projects that are growing or declining. This may suggest that stars count is not a useful signal for well-established project. Future work should examine in more detail this relationship between community support and usefulness for tasks such as usage and contribution.

## **7.5.2 Project State Affecting Community Support**

The predictive models find that project state interacts with stars count (but not supportiveness) for some of the datasets. Specifically, “positive” lifecycle projects interact positively with stars count across all three datasets and “negative” lifecycle projects interact positively for Python and JavaScript. This suggests that for projects in a non-stable lifecycle state, whether growing or declining, stars has a stronger association with community support. While the positive interaction of “negative” projects with stars is somewhat surprising, perhaps it indicates that both growing and declining projects have similar community-related support needs. Particularly for “positive” projects as it is the only interaction present through all three datasets.

Core team size interacts negatively with stars count for Ruby and Python. This suggests that for projects with smaller core teams, stars has a stronger association with community support. Given the similar moderation effect on stars, perhaps this finding indicates that projects with small core teams have similar support needs as growing and declining projects.

Growing, declining, and small core team projects may all have support needs that are better served by drawing more attention from developers compared to more well-established projects. As stars is a signal for the attention a project receives from the overall GitHub community [Dabbish et al., 2012], perhaps these interactions indicate the effectiveness of drawing attention towards solving a project's support needs. For example, perhaps projects in these states have a breadth of simple issues that need to be addressed but with a lack of users willing to help. For these projects, drawing attention from developers increases the pool of users willing to help with these simple tasks. In contrast, perhaps well-established projects have reached a critical mass of users [Iriberry and Leroy, 2009] that are willing to work on simple tasks but have more complex issues that need to be addressed.

### **7.5.3 Implications for Software Engineering**

The findings suggest opportunities to enable developers to more easily evaluate community support in software projects and project managers to encourage support from their community members. I found in the interviews that developers evaluating projects as potential dependencies find community support as a useful quality to infer. The results from the quantitative analysis suggest ways to assist developers in this evaluation through the use of signals. In particular, the analysis suggests that the designed supportiveness signal has a stronger and more robust relationship with community support than stars count which developers seem to currently use. Tools that assist developers in coordinating with software projects then could use the supportiveness signal to easily indicate the degree of support a project may receive. For example, a tool that assists developers in evaluating and picking projects may use the supportiveness signal to compare the likelihood of future support for potential dependencies. Similarly, the results may inform designing future transparent environments. If community support is a useful quality of projects for developers, then perhaps supportiveness should be displayed as a signal when enumerating or comparing projects. This is in contrast to how GitHub currently displays projects with stars count as one of the most prominent signals. The effect of project state on predictors for community support also suggests that depending on the project in question, perhaps different signals should be visualized. For example, perhaps displaying stars is not as

informative for well-established projects but is more useful for projects that are growing.

The results also suggest practices that project managers for open source software projects to encourage community support. There is the possibility that the predictors of stars and supportiveness may represent differing goals for projects. Stars count may represent a project's popularity or attention from the community [Dabbish et al., 2012]. Supportiveness in some ways represents how good of an open source software citizen [Raymond, 2004] a developer is via the extent they support projects across the entire GitHub ecosystem. If these representations are reasonable, then the results suggest goals for project managers if they wish to attract support from the community depending on their project's state. In particular, if the project is either growing, declining, or has a small core team, then perhaps attracting support means drawing attention in general (stars count). On the other hand, attracting good open source citizens (supportiveness) is always highly beneficial regarding community support regardless of the project state.

#### **7.5.4 Limitations**

One of the main limitations of this study is construct validity [Shadish et al., 2002]. In particular, the method of measuring the unobservable quality of community support is by no means comprehensive for all possible maintenance actions for a software project on GitHub. For example, discussions on mailing lists or IRC channels by community members regarding how to fix bugs would not be represented. Similarly, I make no claims that the chosen signals are comprehensive for predictors of community support in projects. Future work improving either the measurement or predictors for community support would greatly further our understanding of the concept. The longitudinal nature of this analysis allows for suggestions of causal directions from predictors to future community support. However, without a true controlled experiment, it is difficult to make absolute claims of causality.

Similarly, this study's use of mixed methods allows for suggestions of how developers interpret signals through interviews. However, without a true controlled experiment, it is difficult to make absolute claims of how developers interpret specific signals. Specifically, although I establish an association between stars and support, it is difficult using this study's data to absolutely explain why this association exists and why it is weaker than supportiveness. For example, I assume in the discussion that stars is an indication of the attention a project receives. The more attention a project receives, the more people it draws, hence it receives more support. Alternatively, stars count could actually indicate (more weakly than supportiveness) willingness to support the project. Future work should examine precisely how developers interpret such signals in regards to intent of providing

support to a project.

## **7.6 Conclusion**

In this work I explored the concept of community support for open source software projects in GitHub and predictors for the degree of support a project receives from its community. Grounding the analysis through prior literature and a preliminary interview study, I performed a longitudinal analysis of community support for the predictors of a project's stars count and a designed signal of supportiveness and project state measures of lifecycle state and core developer team size. I found that while both predictors were associated with community support, the designed signal of supportiveness had a stronger and more robust relationship with support. Projects in either growing or declining states as well as small core teams saw higher effects of stars on community support.

The findings of this study further our understanding of the relatively unexplored concept of community support in open source software projects. I find that project state affects both which projects garner support and stars as a predictor for support. This study validates stars as a signal for support but also designs an improved signal called "supportiveness." This improved signal may enable developers to easily evaluate which projects are well-supported which may be useful in the design of coordination tools or future transparent environments such as GitHub. The findings also suggest practices for project managers who wish to attract more support for their project.



# Chapter 8

## Future Work

The findings from the studies described in previous chapters suggest opportunities for improving signals, tools, and environments as well as performing similar studies in other domains and transparent environments. Table 8.1 provides a summary of the studies and associated with coordination tasks. While each chapter contains discussions of future work specific to the corresponding study, this chapter discusses broader opportunities for future work.

Table 8.1: Overview of Signaling Implications From Dissertation Studies

Coordination Task	Highly Associated Types of Signals	Example Signals
-------------------	------------------------------------	-----------------

Evaluating Contributions	Social Signals	Social Distance
Negotiating Contributions	Political Signals	”+1” comments
Evaluating Projects	Working Dynamics	Commit Volume
Inferring Community Support	Costly Signals	”Supportiveness”

### 8.1 Methodology for Eliciting and Improving Signals

My work, in particular Chapter 6 and Chapter 7, suggests a general methodology for eliciting and potentially improving useful signals for a specific coordination task in transparent development environments. Following the mixed-methods design of Chapter 6, the methodology would include the following steps: 1) interview users to explore what signals and qualities they perceive as useful in informing the coordination task of interest, 2) create measures for signals identified and the task in question, 3) create a statistical model

relating signals measures to task outcomes to verify which signals are actually used across the environment, 4) identify signals with low fit and design an improved signal if necessary. This methodology is potentially useful for both better understanding how developers perform a specific coordination task and designing tools or environments that assist developers in a given task. As seen in Chapter 6, identifying the qualities and signals that developers use to inform a task gives insight into how the task is performed. For example, Chapter 6 suggests that developers find signals of the project's working dynamics as useful when evaluating projects. Identifying these useful signals also may accordingly inform the design of environments or tools by surfacing these signals. For example, Chapter 6 suggests that a tool that helps developers evaluate projects may benefit from prominently displaying working dynamics signals such as commit volume. Implications for designing tools and environments are discussed in more detail in subsection 8.3. Subsection 8.2 discusses in more detail how this methodology might also improve signals with low fit or correlation to the unobservable quality they indicate [Connelly et al., 2010].

An open question that future work should address is whether this methodology is applicable to other domains and environments. For example, this study's methodology could potentially be used to understand what signals editors in Wikipedia use when evaluating articles and the results of such a study may reveal insights into the process of evaluating articles or inform the design of article-management tools or Wikipedia itself. Even software development in a different transparent environment from GitHub may result in different signals found for a given task due to differences in affordances [Gaver, 1991] that the environment provides. For example, developers that work in an environment that deemphasizes commit activity may find working dynamics signals such as commit volume less useful when evaluating projects compared to developers in GitHub. As I only study coordination tasks in the GitHub environment, how these affordances affect signals used is an open question that future work should address. A more extreme case that future work should explore is how participants in transparent environments outside of general software development use signals. For example, members of the scientific software community may have very different goals when evaluating software projects compared to general open source software developers [Trainer et al., 2015]. A scientific software developer may wish to prioritize reproducible research and therefore attend to signals indicating how accessible the project is to reusing data structures or algorithms such as variety of data format wrappers.

Future work should also examine whether tasks that require heterogeneous roles affect the signals needed to inform a task. An assumption made throughout the studies described in the previous chapters is that only software developers are involved in the coordination tasks studied. Relaxing this assumption, as many software projects often include the

efforts of non-developers such as designers, data scientists, user experience researchers, domain experts, managers, and so on, may affect what signals are needed to inform this task. Multiple roles may complicate the signal elicitation methodology described earlier as each role may find different signals and qualities useful. For example, while a software developer may wish to know how easy a project is to incorporate as a dependency, a data scientist may wish to know how accessible a project is to reusing data sets. This concept is reflected in signaling theory as receiver interpretation [Connelly et al., 2010] or that receivers may weight signals differently. Discovering these weightings may inform the design of tools or environments that accommodate multiple roles. In fact, GitHub projects in industry settings find that non-developers struggle with using GitHub, often resulting to using external tools [Kalliamvakou et al., 2015]. Identifying qualities and signals that specific roles need to know in order to perform a task may enable the design of less overwhelming environments that are easier to use for non-technical roles.

## **8.2 Designing Improved Signals for Software Developers**

Findings from the studies described in previous chapters suggest opportunities to use signaling theory to improve signals that developers use in transparent environments. The community support study in Chapter 7 and its designed signal of supportiveness suggest an approach to creating improved signals driven by signaling theory: aggregating activity to create a high cost signal. Specifically, “improved” here refers to the concept of a higher signal fit or the correlation of a signal to the unobservable quality it indicates [Connelly et al., 2010]. This approach is novel because it is only possible in transparent environments where work activities are made visible. The activities aggregated also need to be related to the unobservable quality desired. For example, in Chapter 7 I aggregate support actions across projects for developers to indicate the quality of community support for a project. Signaling theory suggests that signals with high costs are often related to both high honesty and fit and therefore usefulness [Spence, 1973], as long as the costs are aligned with the quality in question. This approach effectively creates a high cost signal from aggregating related activities that should also have high honesty and fit. The cost is high because it requires developers to actually perform the activities in question in order to produce this signal. Another benefit of this approach is that the signals generated from activities are unintentional, the incentives to produce the signal is aligned with performing development work. In contrast, most signals studied in signaling theory literature are positive and intentional, and the incentive to produce the signal is to affect the decision of the receiver [Spence, 1973]. The unintentional nature of these designed signals may make them more robust to deception, especially given the transparent nature of the environment.

For example, “gaming” an aggregated signal would involve creating false development activities which would also be visible in transparent environments. This approach and concept is explored in detail in Chapter 7 for the task of evaluating projects but is perhaps applicable to improving low fit signals in general.

Future work may attempt to apply this approach of aggregating related activities to designing improved signals for other coordination tasks. For example, the study of potential signals for evaluating contributions in Chapter 4 found that technical signals were surprisingly not as strongly associated with acceptance as social signals [Tsay et al., 2014a]. Assuming that this difference in effect size is due to signal fit, a potentially improved technical signal for informing the evaluation of contributions may aggregate technical signals across multiple pull requests for a submitter. Doing so may create a signal that indicates the willingness of a submitter to follow technical norms in general which may be more informative to a project manager than looking at a specific contribution. Also, providing such signals through tooling may reduce the assessment cost for project managers. I discuss in Chapter 4 that the discrepancy between social and technical signal usage may be due to differences in assessment cost that the manager incurs when using technical signals. Making aggregated technical signals easily visible in a tool should greatly reduce such costs.

While I explore the effect of signal cost and fit in previous studies, future work may examine the potential of other signaling concepts and types of signals. The concept of honesty [Connelly et al., 2010], or the correlation between signaler and unobservable quality, is discussed in Chapter 6 but not explored. One reason for this is that many of the signals explored in the studies in previous chapters are products of performing work in transparency. These signals are expected to be honest as the incentive for producing the signal is aligned with performing actual development work rather than influencing the receiver. However, it is possible that future work may identify potentially useful signals that are also potentially deceptive. Similarly, future work may examine other coordination tasks where useful signals are potentially deceptive. For example, users on GitHub may be motivated to emit deceptive signals that indicate the quality of coding ability. If projects owned and programming languages used are signals for coding competency [Marlow et al., 2013], then developers aiming to deceive potential recruiters may attempt to create dummy projects in multiple languages. For this coordination task of recruiting developers, the honesty of signals may be more relevant than fit as the possibility of “cheating” lessens the usefulness of signals [Connelly et al., 2010]. Future work may also explore the usefulness of “political” signals that are discussed in Chapter 5. “Political” signals in this case refer to indications of the influences of different stakeholders. For example, the collective “+1” comments on a pull request is a signal for users that desire for the change to be accepted.

Such signals are relatively unexplored in both software engineering and signaling theory literature. The findings of the study in Chapter 5 suggests that core members were sensitive to audience pressure and tended to fulfill technical goals if influenced by users. This suggests that core members may find these political signals useful for deciding how to prioritize pull requests. Future work should examine the potential usefulness of political signals for other coordination tasks or even environments. For example, editors managing articles of controversial topics in Wikipedia may benefit from signals of the influence of different “sides” on a topic.

### **8.3 Designing Developer Tools and Transparent Development Environments**

Using transparent development environments comes with the risk of information overload which may be alleviated by guiding users through visualizing useful signals. Due to the social networking-like features in transparent environments, many developers in these systems find it challenging to effectively consume the sheer amount of content, developing ad-hoc strategies to filter and skim [Singer et al., 2014] or even refusing to participate in social aspects of development. The qualities and signals identified in the studies of the previous chapters offer opportunities to target information that developers need to know in order to perform specific coordination tasks. Future tools or transparent environments may choose to prioritize visualizing these signals in order to prevent overwhelming developers with information. This is in contrast to the design of many current transparent environments which tend to make all or most information visible using dashboards or feeds [Treude and Storey, 2010].

The findings of the previous chapters suggest implications for designing future transparent environments through what signals to display to users. Specifically, the previously described studies identify signals that GitHub or future environments may display to assist developers in performing specific coordination tasks. For the coordination task of evaluating contributions, the study in Chapter 4 suggests that social signals for social distance and prior interaction are potentially useful for project managers [Tsay et al., 2014a]. This suggests that perhaps GitHub or future environments should implement some indication of social distance or experience for the submitter when project managers are evaluating contributions. When projects and developers coordinate via extended contribution discussions, the study in Chapter 5 suggests that “political” signals may be useful for project managers [Tsay et al., 2014b]. This suggests possible visualizations for project managers in GitHub or future environments that display the competing influence of various stake-

holders. Also, a simple “political” signal for GitHub specifically would be to formally implement the “+1” comment as a feature. The amount of “+1”s in a discussion would be a simple signal for the audience pressure to address a particular contribution. For the coordination task of evaluating a project for usage, Chapter 6 suggests that signals for project liveliness such as commit volume, signals for maturity such as project versions, and signals for ease-of-use such as required downstream dependencies may be useful. Similarly, when evaluating a project for contribution, developers tend to use signals for responsiveness through time to close pull requests and signals for accessibility to contribute through issues. These signals identified are mostly not easily visible when searching for projects or examining project pages in the current implementation of GitHub. Also, in contrast to how GitHub currently prominently displays stars throughout the site, the results of Chapter 6 and Chapter 7 suggest that de-emphasizing stars in favor for other signals such as supportiveness may be more useful for developers evaluating projects. Future work should implement some of these signals in tools or environments and examine if developers actually benefit from using these signals in the field.

An open question that future work should address are the human-computer interaction concerns with how to best display useful signals to developers. Although the studies described in previous chapters suggest which signals may be useful to display, how to effectively visualize this information to developers is still unexplored. For example, while some of the signals identified seem straightforward to visualize, such as the average time to close pull requests, more complicated signals such as supportiveness or political signals for influence may be more challenging to display. Using influence as an example, one option is to simply display a number for the users or projects that may potentially be impacted by a contribution. A more informative but costly to interpret signal may be to display the network of users or projects that may be impacted. The former signal may be potentially misleading as most of the projects impacted may be minor or even dead projects. The latter signal would indicate the relative importance of projects or users impacted but may be so costly to interpret that users ignore the visualization when evaluating contributions.

## **8.4 Dynamic Signals for Tasks, Projects, and Users**

The previous chapters’ findings suggest the potential usefulness of dynamic signals that adapt according to changes in current task, project, or user. Previous chapters suggest that the interpretation of signals may be affected by the current task at hand and the state of the user or project emitting the signal. The study of signals developers use to evaluate projects for usage or contribution in Chapter 6 suggests that even for similar signals, the

task at hand affects the interpretation. Specifically, while project versions is in both Usage and Contribution models, the association with Usage is much higher. This may suggest that developers evaluating a project for use as a dependency may find the signal of previous versions more useful than one who is evaluating a project for contribution. Similarly, Chapter 4 and Chapter 7 find that the state of the user or project emitting the signal may moderate how receivers interpret the signal. In the study of evaluating contributions in Chapter 4, the submitter's prior interaction moderates the influence of the discussion on acceptance [Tsay et al., 2014a]. In the study of predictors for community support in Chapter 7, the project's current lifecycle state and core team size moderates the influence of stars on community support. Both of these findings suggest that the usefulness of signals may change depending on the state of the user or project emitting the signal. Therefore, a future tool that visualizes signals for developers may benefit from dynamically altering what signals are shown depending on the current task or state of the user or project.

Future work should examine the potential of dynamic signals and the human-computer interaction concerns with implementing such signals. As discussed in the previous subsection, how to effectively visualize signals is an open question. Dynamic signals in particular may be challenging to visualize effectively due to multiple states and the potential combinatorial explosion. For example, a naïve implementation of dynamic signals for a tool that assists developers in evaluating projects may display different sets of signals depending if the user is looking for projects to use or to contribute to. The same tool may display different signals depending on the state of the project evaluated, such as their current state in the lifecycle and core team size. Even when making the simplifying assumption that lifecycle state and core team size are binary, this implementation requires 8 different sets of signals per project. Including more states of interest increases the signal sets required exponentially.



# Chapter 9

## Conclusions

By understanding how developers make use of information made visible by transparent development environments, we have the opportunity to assist developers in coordinating in these environments. In this dissertation, I have used signaling theory as a theoretical lens to model developers' usage of information for the specific coordination tasks of evaluating contributions in Chapter 4, negotiating contributions through discussion in Chapter 5, and evaluating projects in Chapter 6. The findings of these studies identify signals that developers use to inform these tasks. These signals give insight into how these coordination tasks are performed, suggest implications for the design of future developer tools and transparent environments, and may establish the foundation for exploring signals in other domains or environments. I have also used signaling theory to design an improved signal called supportiveness for the quality of the community support for a software project that is stronger and more robust than the signal developers currently use in Chapter 7. These studies describe a general approach of using signaling theory to understand and potentially improve how developers use information during coordination tasks.

The novel application of signaling theory to understand how software developers use information to coordinate is my primary contribution to the field of software engineering. Coordination in current software engineering literature largely focuses on tasks within projects such as task assignment or modularity [Crowston et al., 2008] and less on tasks involving other projects or developers. My work not only examines these tasks but also the decision-making process behind each task that developers perform. My approach of using signals to model how developers use information is particularly well-suited for information-rich environments such as transparent development environments. Understanding the unobservable qualities developers wish to know and signals used to indicate these qualities is particularly useful for coordination tasks that involve other projects or

developers. Conceptually separating transparency information into signals and qualities is a strength of using signaling theory as a lens. Identifying unobservable qualities that developers actually wish to know gives insight into the underlying decision-making process and allows for differentiating between poor-quality and less-useful signals. For example, a naïve version of the study in Chapter 6 may determine that stars and other such community signals in general are not useful. However, modeling the process as signals and qualities reveals that while community qualities are useful, currently used community signals such as stars count may not be. Signaling theory also gives useful constructs such as cost and fit to reason about why certain signals may be more or less useful.

My approach of using signaling theory also possesses a number of limitations and potential open questions related to assumptions made in the described studies. A general weakness of my work is that the tasks in question must involve coordination with other dependencies such as projects or other developers. Similarly, my work assumes that the environment is both transparent and information-rich. My approach is not able to study tasks where neither of these conditions are true, such as a developer implementing software features with predetermined dependencies or developers in traditional industry environments that may be highly compartmentalized. Another weakness of using signaling theory is that unobservable qualities must be codifiable. For example, tasks that may rely on intuition, like evaluating the aesthetics of potential designs, would not benefit from applying signaling theory. The studies in my dissertation also make assumptions about the environment and participants. It is an open question to how my approach may work in non-GitHub transparent environments or tasks that involve participants other than software developers. Another limitation of my work is its external validity. The signals studied are hardly comprehensive, potentially useful or latent signals may exist. In particular, I focus mostly on social tasks and signals. Even in technical tasks such as evaluating code contributions, I do not study the use of technical signals such as code complexity measures or call trees. The usefulness of my approach for these technical tasks and signals is an open question.

An open question to the usefulness of my approach of using signaling theory is whether signals are a good proxy for software quality. Can signals tell developers everything they need to know about the quality of a code contribution or software project? Evaluating software quality is inherently difficult to reason about, as software quality as a concept is complex and multidimensional [Crowston et al., 2003]. The studies described in this dissertation also suggest that the concept of software quality may vary quite a bit depending on the current task at hand. For example, Chapter 6 suggests that the signals developers use to evaluate projects significantly differ depending on if the developer is looking to use or contribute to a project. A limitation of the studies in this dissertation is that absolute “quality” of coordination tasks are not examined. Using the study of evaluating projects

again as an example, though I examine how developers think when deciding which projects to use and connections between signals and usage, I do not examine whether or not these usage decisions were “correct.” This is a limitation of the study design. Examining the quality of coordination decisions that use signals would require a long-term study. For example, issues with establishing a dependency with a project may take not manifest for months. Without examining the “quality” of coordination decisions that use signals, it is difficult to absolutely claim whether signals are a good proxy for software quality. However, though software quality is multidimensional, the studies described in this dissertation do examine some of the social dimensions of software quality. My approach of using signaling theory, particularly in differentiating between signals and qualities, may also help in reasoning about specific dimensions of quality. For example, Crowston et al. [2003] identify a number of potential open source software success measures. Some of the social measures such as the process of how development happens (project working dynamics in Chapter 6), community measures, and popularity (both as community evaluation in Chapter 6). As discussed earlier, technical dimensions of software quality are not examined in the studies in this dissertation. Future work may examine the usefulness of technical signals for technical qualities regarding code such as maintainability, testability, and usability [Crowston et al., 2003]. So while it is difficult to absolutely claim that signals are a good proxy for software quality, signaling theory may be a good method for reasoning about multiple dimensions of quality.

My work of using signaling theory in transparency may inform the design of future tools and environments. The studies described in the dissertation have implications for immediate potential changes for signals in current transparent development environments. For example, Chapter 5 suggests that displaying “political” signals such as “+1” comments or potentially impacted projects may be useful for problem-solving discussions around contributions. Another potential implication of my work is a generalizable methodology for understanding and potentially improving how transparent environments or tools display signals to inform specific tasks. Taking the coordination task of recruiting a developer as an example, developing a tool to assist in this process may first interview developers to determine qualities and signals they use to inform this decision. A quantitative study would then determine which of these qualities or signals are actually used. Any potential disparities between the interviews and quantitative study may offer opportunities to design improved signals for this tool, such as with “supportiveness” in Chapter 7. Implementing signals in this way may also have unintended implications. For example, a common concern in signaling theory literature is the potential of signalers to “game” the signaling environment by sending dishonest signals [Connelly et al., 2010]. However, transparency may mitigate some of these effects as creating dishonest signals may also be visible in the environment. For example, gaming the supportiveness signal designed in Chapter 7

would involve performing a large number of false support actions. Transparency would make these false support actions visible, exposing attempts to “game” the system. Another unintended implication which is also related to a current limitation of the studies is that focusing on the signals that most developers use (particularly in quantitative studies) may systematically bias against certain minority populations. For example, it may be possible that using social relationship signals like in Chapter 4 also tend to bias against minority developers because they are less likely to possess the social relationship. However, signaling theory may mitigate this issue through differentiating between the signal and the desired unobservable quality. Using the previous example, if the unobservable quality that project managers actually wish to infer is trust, then there is the opportunity to design a signal that is resistant to biasing against certain populations. Future work may examine how to overcome these unintentional implications of signaling, perhaps through using transparency or signaling theory.

# Appendix A

## Pull Request Extended Discussion Sample

List of pull requests with extended discussions that were analyzed for Chapter 5.

1. <https://github.com/jruby/activerecord-jdbc-adapter/pull/233>
2. <https://github.com/guard/guard/pull/156>
3. <https://github.com/symfony/symfony/pull/5248>
4. <https://github.com/emberjs/ember.js/pull/365>
5. <https://github.com/gitlabhq/gitlabhq/pull/3351>
6. <https://github.com/tastejs/todomvc/pull/120>
7. <https://github.com/laravel/framework/pull/720>
8. <https://github.com/Quicksilver/quicksilver/pull/219>
9. <https://github.com/playframework/playframework/pull/469>
10. <https://github.com/OpenImageIO/oio/pull/82>
11. <https://github.com/linuxmint/Cinnamon/pull/1003>
12. <https://github.com/owncloud/apps/pull/1>

13. <https://github.com/jashkenas/backbone/pull/697>
14. <https://github.com/numpy/numpy/pull/3306>
15. <https://github.com/appium/appium/pull/793>
16. [https://github.com/gregbell/active\\_admin/pull/1952](https://github.com/gregbell/active_admin/pull/1952)
17. <https://github.com/drothlis/stb-tester/pull/9>
18. <https://github.com/xapi-project/xen-api/pull/957>
19. <https://github.com/cfengine/design-center/pull/14>
20. <https://github.com/mxcl/homebrew/pull/3403>

# Glossary

- community evaluation** community around a software project. 86, 91, 101, 139
- community support** degree to which a software project's community will support the project through performing maintenance actions. 7, 107, 108, 112, 137
- coordination** management of dependencies between activities [Malone and Crowston, 1994], where dependencies may include software projects and developers. 1, 9, 27, 51, 52, 77, 129, 137
- cost** cost for signaler to produce a signal. 5, 16, 77, 79, 81, 87–91, 101–104, 109, 110, 114, 118, 123, 131, 132, 138
- fit** correlation between the signal and the unobservable quality it indicates. 5, 16, 79, 82, 87–91, 101–103, 109, 110, 114, 115, 124, 131, 132, 138
- honesty** extent to which the signaler actually possesses the signaled quality. 5, 16, 82, 87–91, 101, 103, 132
- personal utility** estimation of personal utility and/or cost of using or participating in a project. 85, 91, 101
- project working dynamics** project's working style and direction, how the core team works. 85, 91, 101, 139
- receiver attention** extent to which receivers attend to the environment for signals. 16
- receiver interpretation** translation from signal to unobservable quality by the receiver, who may apply their own weights or meanings to signals. 16

**signal** piece of information that indicates an unobservable quality about the signaler. 5, 6, 15, 27, 28, 32, 35, 51, 53, 71, 77, 78, 81, 103, 108, 112, 123, 129, 137, 138

**transparency** accurate observability of an organization's low-level activities, routines, behaviors, output, and performance [Bernstein, 2012]. 2, 13, 19, 20, 28, 56, 78, 91, 108, 144

**transparent development environment** software development environment that implements transparency features. 2, 6, 9, 13, 19, 27, 54, 57, 77, 78, 80, 112, 129, 133, 137

# Bibliography

- Samuel A Ajila and Di Wu. Empirical study of the effects of open source adoption on software development economics. *Journal of Systems and Software*, 80(9):1517–1529, 2007. ISSN 0164-1212. doi: <http://doi.org/10.1016/j.jss.2007.01.011>. URL <http://www.sciencedirect.com/science/article/pii/S0164121207000076>. 1, 6
- Ofer Arazy, Lisa Yeo, and Oded Nov. Stay on the Wikipedia task: When task-related disagreements slip into personal and procedural conflicts. *Journal of the American Society for Information Science and Technology*, 64(8):1634–1648, 2013. ISSN 1532-2890. doi: 10.1002/asi.22869. URL <http://dx.doi.org/10.1002/asi.22869>. 2.1, 5
- Douglas Bates, Martin Maechler, and Ben Bolker. *lme4.0: Linear mixed-effects models using S4 classes*, 2013. URL <http://r-forge.r-project.org/projects/lme4/>. 4.2.3
- Ethan S Bernstein. The Transparency Paradox: A Role for Privacy in Organizational Learning and Operational Control. *Administrative Science Quarterly*, 57(2):181–216, 2012. doi: 10.1177/0001839212453028. URL <http://asq.sagepub.com/content/57/2/181.abstract>. 1.1, 2.3, A
- Susan L Bryant, Andrea Forte, and Amy Bruckman. Becoming Wikipedian: transformation of participation in a collaborative online encyclopedia. In *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work, GROUP '05*, pages 1–10, New York, NY, USA, 2005. ACM. ISBN 1-59593-223-2. doi: 10.1145/1099203.1099205. URL <http://doi.acm.org/10.1145/1099203.1099205>. 2.1, 4.1.2, 5.1.4, 7.1.2
- Moira Burke and Robert Kraut. Mind Your Ps and Qs: The Impact of Politeness and Rudeness in Online Communities. In *Proceedings of the 2008 ACM Conference on*

*Computer Supported Cooperative Work*, CSCW '08, pages 281–284, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-007-4. doi: 10.1145/1460563.1460609. URL <http://doi.acm.org/10.1145/1460563.1460609>. 5.1.1

Lowell W Busenitz, James O Fiet, and Douglas D Moesel. Signaling in Venture Capitalist—New Venture Team Funding Decisions: Does It Indicate Long-Term Venture Outcomes? *Entrepreneurship Theory and Practice*, 29(1):1–12, 2005. ISSN 1540-6520. doi: 10.1111/j.1540-6520.2005.00066.x. URL <http://dx.doi.org/10.1111/j.1540-6520.2005.00066.x>. 6.4.1

Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. Developer Onboarding in GitHub: The Role of Prior Social Links and Language Experience. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 817–828, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786854. URL <http://doi.acm.org/10.1145/2786805.2786854>. 2.3

Marcelo Cataldo, Patrick A Wagstrom, James D Herbsleb, and Kathleen M Carley. Identification of coordination requirements: implications for the Design of collaboration and awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 353–362, New York, NY, USA, 2006. ACM. ISBN 1-59593-249-6. doi: 10.1145/1180875.1180929. URL <http://doi.acm.org/10.1145/1180875.1180929>. 1.2

B. L. Connelly, S. T. Certo, R. D. Ireland, and C. R. Reutzel. Signaling Theory: A Review and Assessment. *Journal of Management*, 37(1):39–67, dec 2010. ISSN 0149-2063. URL <http://jom.sagepub.com/cgi/content/abstract/37/1/39>. 1.2, 1.3, 2.4, 4.4.1, 4.5, 6, 6.1.3, 6.2.2, 6.2.2, 6.4.1, 7, 7.2.2, 7.2.3, 7.3.2, 7.5.1, 8.1, 8.2, 9

Juliet Corbin and Anselm Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage, 2008. 5, 6.2.1, 7.2.1

John W. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. SAGE publications, 4th edition, 2013. 6, 6.2.2, 6.3.1

Yves Croissant, Giovanni Millo, and Others. Panel data econometrics in R: The plm package. 7.3.1, 7.3.3

Kevin Crowston, Hala Annabi, and James Howison. Defining open source software project success. In *ICIS 2003. Proceedings of International Conference on Information Systems 2003*, 2003. 7.1.1, 9

- Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. Free/Libre open-source software development: What we know and what we do not know. *ACM Comput. Surv.*, 44(2):7:1—7:35, mar 2008. ISSN 0360-0300. doi: 10.1145/2089125.2089127. URL <http://doi.acm.org/10.1145/2089125.2089127>. 1.2, 2.2, 5, 5.1.2, 5.1.3, 7.1.1, 9
- 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, CSCW '12*, pages 1277–1286, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1086-4. doi: 10.1145/2145204.2145396. URL <http://doi.acm.org/10.1145/2145204.2145396>. 1.1, 1.2, 1.3, 2.3, 2.4, 3.1, 4.1.3, 4.1.4, 4.2.2, 4.2.2, 4.2.2, 4.4.1, 4.4.3, 5.1.3, 5.2.1, 5.4.1, 5.4.2, 6, 6.1.2, 6.1.3, 6.2.2, 6.3.2, 6.4.1, 6.4.2, 7, 7.3.2, 7.5.2, 7.5.3
- Antonio Davila, George Foster, and Mahendra Gupta. Venture capital financing and the growth of startup firms. *Journal of Business Venturing*, 18(6):689–708, 2003. ISSN 0883-9026. doi: [http://dx.doi.org/10.1016/S0883-9026\(02\)00127-1](http://dx.doi.org/10.1016/S0883-9026(02)00127-1). URL <http://www.sciencedirect.com/science/article/pii/S0883902602001271>. 6.2.2
- Judith Donath. *Signals, truth, and design*. MIT Press, Cambridge, MA, 2005. 1.3, 4.4.2, 6.4.1, 7.3.2
- Judith Donath. Signals in Social Supernet. *Journal of Computer-Mediated Communication*, 13(1):231–251, 2007. ISSN 1083-6101. doi: 10.1111/j.1083-6101.2007.00394.x. URL <http://dx.doi.org/10.1111/j.1083-6101.2007.00394.x>. 2.4, 6.1.3
- Carsten F Dormann, Jane Elith, Sven Bacher, Carsten Buchmann, Gudrun Carl, Gabriel Carré, Jaime R García Marquéz, Bernd Gruber, Bruno Lafourcade, Pedro J Leitão, and Others. Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, 36(1):27–46, 2013. 4.2.3
- Nicolas Ducheneaut. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Computer Supported Cooperative Work (CSCW)*, 14(4):323–368, 2005. ISSN 0925-9724. doi: 10.1007/s10606-005-9000-1. URL <http://dx.doi.org/10.1007/s10606-005-9000-1>. 2.2, 4, 4.1.1, 4.1.4, 5, 5.1.2, 5.1.4, 5.4.2, 6.1.1
- J Alberto Espinosa, Sandra A Slaughter, Robert E Kraut, and James D Herbsleb. Familiarity, Complexity, and Team Performance in Geographically Distributed Soft-

- ware Development. *Organization Science*, 18(4):613–630, 2007. doi: 10.1287/orsc.1070.0297. URL <http://orgsci.journal.informs.org/content/18/4/613.abstract>. 4.4.1
- Andrew J Flanagin and Miriam J Metzger. Trusting expert- versus user-generated ratings online: The role of information volume, valence, and consumer characteristics. *Computers in Human Behavior*, 29(4):1626–1634, 2013. ISSN 0747-5632. doi: <http://dx.doi.org/10.1016/j.chb.2013.02.001>. URL <http://www.sciencedirect.com/science/article/pii/S0747563213000575>. 6.2.2, 6.4.1
- A Forte and A Bruckman. Scaling Consensus: Increasing Decentralization in Wikipedia Governance, 2008. 2.1, 5.1.1, 5.4.2
- William W Gaver. Technology Affordances. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 79–84, New York, NY, USA, 1991. ACM. ISBN 0-89791-383-3. doi: 10.1145/108844.108856. URL <http://doi.acm.org/10.1145/108844.108856>. 8.1
- G Gousios and D Spinellis. GHTorrent: Github's data from a firehose. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 12–21, jun 2012. doi: 10.1109/MSR.2012.6224294. 7.3.1
- 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*, ICSE 2014, pages 345–355, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568260. URL <http://doi.acm.org/10.1145/2568225.2568260>. 5, 5.4.1
- Carl Gutwin, Reagan Penner, and Kevin Schneider. Group Awareness in Distributed Software Development. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, CSCW '04, pages 72–81, New York, NY, USA, 2004. ACM. ISBN 1-58113-810-5. doi: 10.1145/1031607.1031621. URL <http://doi.acm.org/10.1145/1031607.1031621>. 2.2, 6.1.1
- Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. Supporting Developers' Coordination in the IDE. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, pages 518–532, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2922-4. doi: 10.1145/2675133.2675177. URL <http://doi.acm.org/10.1145/2675133.2675177>. 2.2

- Aaron Halfaker, Aniket Kittur, and John Riedl. Don't Bite the Newbies: How Reverts Affect the Quantity and Quality of Wikipedia Work. In *Proceedings of the 7th International Symposium on Wikis and Open Collaboration, WikiSym '11*, pages 163–172, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0909-7. doi: 10.1145/2038558.2038585. URL <http://doi.acm.org/10.1145/2038558.2038585>. 2.1, 5.1.1, 5.1.4
- Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. Will They Like This?: Evaluating Code Contributions with Language Models. In *Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15*, pages 157–167, Piscataway, NJ, USA, 2015. IEEE Press. URL <http://dl.acm.org/citation.cfm?id=2820518.2820539>. 6.4.2
- Alicia Iriberry and Gondy Leroy. A life-cycle perspective on online community success. *ACM Comput. Surv.*, 41(2):11:1—11:29, feb 2009. ISSN 0360-0300. doi: 10.1145/1459352.1459356. URL <http://doi.acm.org/10.1145/1459352.1459356>. 4.1.2, 7.1.2, 7.2.3, 7.5.2
- Jay J Janney and Timothy B Folta. Signaling through private equity placements and its impact on the valuation of biotechnology firms. *Journal of Business Venturing*, 18(3):361–380, 2003. ISSN 0883-9026. doi: [http://dx.doi.org/10.1016/S0883-9026\(02\)00100-3](http://dx.doi.org/10.1016/S0883-9026(02)00100-3). URL <http://www.sciencedirect.com/science/article/pii/S0883902602001003>. 2.4, 6.1.3
- N. P. Jewell and A. Hubbard. *Analysis of Longitudinal Studies in Epidemiology*. Chapman and Hall, New York, 2010. 6.3.1
- Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*, pages 92–101, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597074. URL <http://doi.acm.org/10.1145/2597073.2597074>. 6.2.2
- Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, Leif Singer, and Daniel M German. Open Source-style Collaborative Development Practices in Commercial Projects Using GitHub. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, pages 574–585, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL <http://dl.acm.org/citation.cfm?id=2818754.2818825>. 2.3, 8.1

- Steven J Karau and Kipling D Williams. Social loafing: A meta-analytic review and theoretical integration. *Journal of Personality and Social Psychology*, 65(4):681–706, 1993. ISSN 1939-1315(Electronic);0022-3514(Print). doi: 10.1037/0022-3514.65.4.681. 2.1, 7.1.2
- Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. Using Dynamic and Contextual Features to Predict Issue Lifetime in GitHub Projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 291–302, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4186-8. doi: 10.1145/2901739.2901751. URL <http://doi.acm.org/10.1145/2901739.2901751>. 2.3, 7.1.3
- A Kittur, E H Chi, B A Pendleton, B Suh, and T Mytkowicz. Power of the few vs. wisdom of the crowd: Wikipedia and the rise of the bourgeoisie. *25th Annual ACM Conference on Human Factors in Computing Systems (CHI 2007)*, 2007a. URL <http://www.parc.com/research/publications/details.php?id=5904>. 2.1, 5.1.1, 5.4.2
- Aniket Kittur, Bongwon Suh, Bryan A Pendleton, and Ed H Chi. He Says, She Says: Conflict and Coordination in Wikipedia. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 453–462, New York, NY, USA, 2007b. ACM. ISBN 978-1-59593-593-9. doi: 10.1145/1240624.1240698. URL <http://doi.acm.org/10.1145/1240624.1240698>. 2.1
- Andrew J Ko and Parmit K Chilana. Design, Discussion, and Dissent in Open Bug Reports. In *Proceedings of the 2011 iConference*, iConference '11, pages 106–113, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0121-3. doi: 10.1145/1940761.1940776. URL <http://doi.acm.org/10.1145/1940761.1940776>. 2.2, 5.1.2, 5.1.4
- Robert E Kraut and Paul Resnick. *Building Successful Online Communities: Evidence-Based Social Design*. MIT Press, Cambridge, MA, 2012. 2.1, 4.1.2, 5.1.1, 5.1.3, 5.1.4, 5.4.2, 7.1.2
- Travis Kriplean, Ivan Beschastnikh, David W McDonald, and Scott A Golder. Community, Consensus, Coercion, Control: Cs\*W or How Policy Mediates Mass Participation. In *Proceedings of the 2007 International ACM Conference on Supporting Group Work*, GROUP '07, pages 167–176, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-845-9. doi: 10.1145/1316624.1316648. URL <http://doi.acm.org/10.1145/1316624.1316648>. 2.1, 5.1.1, 5.4.2
- Karim R Lakhani and Eric von Hippel. How open source software works: “free” user-to-user assistance. *Research Policy*, 32(6):923–943, 2003. ISSN 0048-7333. doi: 10.1016/

S0048-7333(02)00095-1. URL <http://www.sciencedirect.com/science/article/pii/S0048733302000951>. 6.2.2, 6.2.2, 7, 7.1.1, 7.2.2

Karim R. Lakhani and Robert Wolf. Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects. In Joe Feller, Brian Fitzgerald, Scott Hissam, and Karim R. Lakhani, editors, *Perspectives on Free and Open Source Software*, pages 3–22. MIT Press, Cambridge, MA, 2005. 7

Josh Lerner and Jean Tirole. Some Simple Economics of Open Source. *The Journal of Industrial Economics*, 50(2):197–234, 2002. ISSN 1467-6451. doi: 10.1111/1467-6451.00174. URL <http://dx.doi.org/10.1111/1467-6451.00174>. 5, 5.1.3, 5.1.4, 5.4.2, 7, 7.1.1

Thomas W Malone and Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Comput. Surv.*, 26(1):87–119, 1994. ISSN 0360-0300. doi: 10.1145/174666.174668. URL <http://doi.acm.org/10.1145/174666.174668>. 1, 1.2, A

Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. Impression formation in online peer production: activity traces and personal profiles in github. In *Proceedings of the 2013 conference on Computer supported cooperative work, CSCW '13*, pages 117–128, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1331-5. doi: 10.1145/2441776.2441792. URL <http://doi.acm.org/10.1145/2441776.2441792>. 2.3, 2.4, 4.1.3, 4.1.4, 4.2.2, 4.4.2, 5, 5.1.3, 5.1.4, 5.4.2, 6.1.2, 6.1.3, 6.2.2, 6.4.2, 8.2

Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, jul 2002. ISSN 1049-331X. doi: 10.1145/567793.567795. URL <http://doi.acm.org/10.1145/567793.567795>. 2.2, 4.1.4, 5, 5.1.3, 5.4.3, 6.1.1, 6.2.2, 7, 7.2.2

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, IW-PSE '02*, pages 76–85, New York, NY, USA, 2002. ACM. ISBN 1-58113-545-9. doi: 10.1145/512035.512055. URL <http://doi.acm.org/10.1145/512035.512055>. 4.1.1, 4.1.4

Robert M O'brien. A caution regarding rules of thumb for variance inflation factors. *Quality & Quantity*, 41(5):673–690, 2007. 6.3.1, 7.3.3

- Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. Creating a shared understanding of testing culture on a social coding site. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 112–121, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL <http://dl.acm.org/citation.cfm?id=2486788.2486804>. 2.3, 4.1.3, 4.1.4, 4.4.1, 6.1.2
- R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. URL <http://www.r-project.org/>. 4.2.3
- Eric S Raymond. The Jargon File, version 4.4.8, 2004. URL <http://www.catb.org/jargon/index.html>. 7.1.2, 7.3.2, 7.5.3
- Peter C Rigby and Margaret-Anne Storey. Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 541–550, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985867. URL <http://doi.acm.org/10.1145/1985793.1985867>. 2.2, 6.1.1
- Peter C Rigby, Daniel M German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*, pages 541–550. ACM, 2008. 2.2, 5.1.2
- Walt Scacchi. Understanding the requirements for developing open source software systems. In *Software, IEE Proceedings-*, volume 149, pages 24–39. IET, 2002. 5.4.3
- Walt Scacchi. Free and open source development practices in the game community. *Software, IEEE*, 21(1):59–66, 2004. 5.4.3
- Walt Scacchi. Free/Open Source Software Development: Recent Research Results and Methods. In Marvin V Zelkowitz, editor, *Architectural Issues*, volume 69 of *Advances in Computers*, pages 243–295. Elsevier, 2007. doi: [http://dx.doi.org/10.1016/S0065-2458\(06\)69005-0](http://dx.doi.org/10.1016/S0065-2458(06)69005-0). URL <http://www.sciencedirect.com/science/article/pii/S0065245806690050>. 4, 4.4.1
- Chris Scaffidi, Chris Bogart, Margaret Burnett, Allen Cypher, Brad Myers, and Mary Shaw. Using traits of web macro scripts to predict reuse. *Journal of Visual Languages & Computing*, 21(5):277–291, 2010. ISSN 1045-926X. doi: <http://dx.doi.org/10.>

1016/j.jvlc.2010.08.003. URL <http://www.sciencedirect.com/science/article/pii/S1045926X10000443>. 2.4, 6.1.3

William R Shadish, Thomas D Cook, and Donald Thomas Campbell. *Experimental and quasi-experimental designs for generalized causal inference*. Wadsworth Cengage learning, 2002. 7.5.4

Sonali K Shah. Motivation, Governance, and the Viability of Hybrid Forms in Open Source Software Development. *Manage. Sci.*, 52(7):1000–1014, jul 2006. ISSN 0025-1909. doi: 10.1287/mnsc.1060.0553. URL <http://dx.doi.org/10.1287/mnsc.1060.0553>. 1, 4.1.1

Leif Singer, Fernando Figueira Filho, Brendan Cleary, Christoph Treude, Margaret-Anne Storey, and Kurt Schneider. Mutual Assessment in the Social Programmer Ecosystem: An Empirical Investigation of Developer Profile Aggregators. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work, CSCW '13*, pages 103–116, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1331-5. doi: 10.1145/2441776.2441791. URL <http://doi.acm.org/10.1145/2441776.2441791>. 2.3

Leif Singer, Fernando Figueira Filho, and Margaret-Anne Storey. Software Engineering at the Speed of Light: How Developers Stay Current Using Twitter. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 211–221, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568305. URL <http://doi.acm.org/10.1145/2568225.2568305>. 1.2, 2.3, 6, 6.1.2, 8.3

Michael Spence. Job Market Signaling. *The Quarterly Journal of Economics*, 87(3):355–374, 1973. ISSN 00335533, 15314650. URL <http://www.jstor.org/stable/1882010>. 1.3, 2.4, 2.4, 6.1.3, 6.2.2, 6.2.2, 8.2

Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. Social Barriers Faced by Newcomers Placing Their First Contribution in Open Source Software Projects. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing, CSCW '15*, pages 1379–1392, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2922-4. doi: 10.1145/2675133.2675215. URL <http://doi.acm.org/10.1145/2675133.2675215>. 2.2

Katherine J Stewart and Sanjay Gosain. The Impact of Ideology on Effectiveness in Open Source Software Development Teams. *MIS Quarterly*, 30(2):291–314, 2006a. ISSN

02767783. URL <http://www.jstor.org/stable/25148732>. 5.4.2, 7.1.2, 7.3.2

Katherine J Stewart and Sanjay Gosain. The moderating role of development stage in free/open source software project performance. *Software Process: Improvement and Practice*, 11(2):177–191, 2006b. ISSN 1099-1670. doi: 10.1002/spip.258. URL <http://dx.doi.org/10.1002/spip.258>. 4.1.1, 4.1.4, 7.1.1, 7.2.3

Margaret-Anne Storey, Leif Singer, Brendan Cleary, Fernando Figueira Filho, and Alexey Zagalsky. The (R) Evolution of Social Media in Software Engineering. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 100–116, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2865-4. doi: 10.1145/2593882.2593887. URL <http://doi.acm.org/10.1145/2593882.2593887>. 2.3, 6

Anselm L Strauss, Juliet Corbin, and Others. *Basics of qualitative research*, volume 15. Sage Newbury Park, CA, 1990. 5.2.2

Stephanie Teasley, Lisa Covi, M S Krishnan, and Judith S Olson. How Does Radical Collocation Help a Team Succeed? In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW '00, pages 339–346, New York, NY, USA, 2000. ACM. ISBN 1-58113-222-0. doi: 10.1145/358916.359005. URL <http://doi.acm.org/10.1145/358916.359005>. 5.4.1

W Ben Towne, Aniket Kittur, Peter Kinnaird, and James Herbsleb. Your Process is Showing: Controversy Management and Perceived Quality in Wikipedia. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW '13, pages 1059–1068, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1331-5. doi: 10.1145/2441776.2441896. URL <http://doi.acm.org/10.1145/2441776.2441896>. 2.1

Erik H Trainer, Chalalai Chaihirunkarn, Arun Kalyanasundaram, and James D Herbsleb. From Personal Tool to Community Resource: What's the Extra Work and Who Will Do It? In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, pages 417–430, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2922-4. doi: 10.1145/2675133.2675172. URL <http://doi.acm.org/10.1145/2675133.2675172>. 6.2.2, 6.2.2, 6.2.2, 6.4.1, 8.1

C Treude and M Storey. Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 365–374, may 2010. doi: 10.1145/1806799.1806854. 8.3

- Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 356–366, New York, NY, USA, 2014a. ACM, ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568315. URL <http://doi.acm.org/10.1145/2568225.2568315>. 1.4, 0, 5.1.3, 5.1.4, 5.2.1, 5.3.3, 5.3.4, 5.4.1, 6.1.2, 6.2.2, 8.2, 8.3, 8.4
- Jason Tsay, Laura Dabbish, and James Herbsleb. Let’s Talk About It: Evaluating Contributions through Discussion in GitHub. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pages 144–154, New York, NY, USA, 2014b. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635882. URL <http://doi.acm.org/10.1145/2635868.2635882>. 1.4, 0, 7, 8.3
- Ruben van Wendel de Joode. Managing conflicts in open source communities. *Electronic Markets*, 14(2):104–113, 2004. 5
- Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark G J van den Brand, Alexander Serebrenik, Premkumar Devanbu, and Vladimir Filkov. Gender and Tenure Diversity in GitHub Teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI ’15*, pages 3789–3798, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3145-6. doi: 10.1145/2702123.2702549. URL <http://doi.acm.org/10.1145/2702123.2702549>. 2.3, 7.1.3
- F B Viegas, M Wattenberg, J Kriss, and F van Ham. Talk Before You Type: Coordination in Wikipedia. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, page 78, 2007. doi: 10.1109/HICSS.2007.511. 2.1
- Fernanda B Viégas, Martin Wattenberg, and Kushal Dave. Studying Cooperation and Conflict Between Authors with History Flow Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI ’04*, pages 575–582, New York, NY, USA, 2004. ACM. ISBN 1-58113-702-8. doi: 10.1145/985692.985765. URL <http://doi.acm.org/10.1145/985692.985765>. 5.1.1, 5.1.4
- Georg von Krogh, Sebastian Spaeth, and Karim R Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241, jul 2003. ISSN 0048-7333. doi: [http://dx.doi.org/10.1016/S0048-7333\(03\)00050-7](http://dx.doi.org/10.1016/S0048-7333(03)00050-7). URL <http://www.sciencedirect.com/science/article/pii/S0048733303000507>. 2.2, 4.1.1, 4.1.4, 4.4.1, 5.1.2, 5.1.3, 5.1.4, 6.2.2

- Jeffrey M Wooldridge. *Introductory econometrics: A modern approach*. Nelson Education, 2015. 7.3.3
- Margarethe F Wiersema Yan Zhang. Stock Market Reaction to CEO Certification: The Signaling Role of CEO Background. *Strategic Management Journal*, 30(7):693–710, 2009. ISSN 01432095, 10970266. URL <http://www.jstor.org/stable/20536072>. 6.4.1
- Y Yu, H Wang, V Filkov, P Devanbu, and B Vasilescu. Wait for It: Determinants of Pull Request Evaluation Latency on GitHub. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 367–371, 2015. doi: 10.1109/MSR.2015.42. 2.3
- Minghui Zhou and Audris Mockus. What Make Long Term Contributors: Willingness and Opportunity in OSS Community. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 518–528, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL <http://dl.acm.org/citation.cfm?id=2337223.2337284>. 2.2, 7.1.1, 7.2.3, 7.3.2, 7.5.1