Software Supply Chain Development and Application

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Yuxing Ma

August 2020

To my friend, ...

Acknowledgments

First and foremost, I would like to thank my advisor and mentor, Dr. Audris Mockus for providing me with the opportunity to work on this project and introducing me to the field of open source software development and evolution. I also thank my committee for their time, support and guidance as I stepped into the unknown: Dr. Russell Zaretzski, Dr. Bruce MacLennan and Dr. Hairong Qi. Further, I would also like to thank Dr. Randy Bradley and Dr. Bogdan Bichescu for their continuous support with all of the knowledge of supply chain management and brilliant suggestions throughout this project. In addition, I would like to thank Dr. James Herbsleb and Dr. Christopher Bogart their all the extensive domain knowledge, experience and guidance during the research time. I would also like to thank the National Institute for Computational Sciences (NICS) at the University of Tennessee for granting us access to various HPC clusters which have provided us with a vast amounts of computational resources and network bandwidth without which this research would not have been possible. I am grateful to have performed this research in a department that goes above and beyond to create a positive environment for graduate students. Thanks to my friends and peers in the department: Tapajit, Sadika, Sara and many others who have contributed to my work and development in so many ways. Finally, I would like to thank my parents — I am where I am today because of their love and support.

Note: This work was supported by the National Science Foundation ([Grant No. 1633437]). All opinions, findings, conclusions, or recommendations expressed in this document are those of the author(s) and do not necessarily reflect the views of the sponsoring agency.

Abstract

Motivation: Free Libre Open Source Software (FLOSS) has become a critical component in numerous devices and applications. Despite its importance, it is not clear why FLOSS ecosystem works so well or if it may cease to function. Majority of existing research is focused on studying a specific software project or a portion of an ecosystem, but FLOSS has not been investigated in its entirety. Such view is necessary because of the deep and complex technical and social dependencies that go beyond the core of an individual ecosystem and tight inter-dependencies among ecosystems within FLOSS.

Aim: We, therefore, aim to discover underlying relations within and across FLOSS projects and developers in open source community, mitigate potential risks induced by the lack of such knowledge and enable systematic analysis over entire open source community through the lens of supply chain (SC).

Method: We utilize concepts from an area of supply chains to model risks of FLOSS ecosystem. FLOSS, due to the distributed decision making of software developers, technical dependencies, and copying of the code, has similarities to traditional supply chain. Unlike in traditional supply chain, where data is proprietary and distributed among players, we aim to measure open-source software supply chain (OSSC) by operationalizing supply chain concept in software domain using traces reconstructed from version control data.

Results: We create a very large and frequently updated collection of version control data in the entire FLOSS ecosystems named World of Code (WoC), that can completely cross-reference authors, projects, commits, blobs, dependencies, and history of the FLOSS ecosystems, and provide capabilities to efficiently correct, augment, query, and analyze that data. Various researches and applications (e.g., software technology adoption investigation) have been successfully implemented by leveraging the combination of WoC and OSSC.

Implications: With a SC perspective in FLOSS development and the increased visibility and transparency in OSSC, our work provide potential opportunities for researchers to conduct wider and deeper studies on OSS over entire FLOSS community, for developers to build more robust software and for students to learn technologies more efficiently and improve programming skills.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Object of Study

As society increasingly relies on open source software (OSS), so does the need to better understand the behaviour of a highly interconnected network of OSS developers who are scattered all over the world. With a large number of software projects (over 100M [39]) and a rapid growth, it becomes more and more challenging to study how the entire OSS ecosystem functions, despite various studies on individual OSS projects [17, 64]. This thesis is investigating ways to conceptualize OSS as a software supply chain (SSC), to measure the SSC of the entire of OSS ecosystems, to provide insights to this operation, and to suggest approaches to reduce several types of the risk that participants are exposed to in this decentralized environment.

Supply chain (SC) concept has been widely studied and explored in traditional industries for many years. In business and commerce, a successful supply chain management [8], i.e., planning, organizing, and controlling in business logistics, plays an important role for a company to sustain competitiveness in product marketing [104]. Software development, especially for OSS, encompasses a number of properties that provide the potential for being viewed and explored from the perspective of supply chain:

- Developers are distributed and scattered all of the world geographically, but cooperate on the development of software products through virtual internet.

- A majority of software products are built on top of one or more mature software products by directly reusing source code of other projects, following successful design in other projects, involving core developers from other projects, etc.

Supply chain management has been successful for helping businesses manage risks from the distributed nature of production. We want to leverage existing findings and transformative insights from SC and apply to software domain.

After an exploration of existing SSC related works (see details in Chp. 2), we found that there was no definition of SSC contextualized for OSS where it may help with distributed decision making, and there was no measurable definition of SSC.

We define SSC by making analog of components in traditional SC. SSC has developers and groups (companies), corporate backers supporting these developers or groups ("financing"), relationships among software projects or packages representing the "chain" of the flow, and changes to source code (e.g. files, modules, frameworks, or entire distributions) representing products or information.

With a supply chain perspective in OSS development, developers can better evaluate software components on various properties, e.g., risks, maintenance and quality, thus being enabled to make more wise decisions on the selection of downstream packages to use. Besides, as the increment of transparency and visibility in SSC, developers are enabled to seek for more talented software engineers to cooperate with, which leads to a faster growth in knowledge and learning experience for developers, more mature and successful software products, and meanwhile, speeds up the evolution of software ecosystems.

With the potentially large amount of benefits of SSC, we move forward to measure SSC. In order to assist empirical researches and investigations in SSC with need for software development data in large, we propose to construct an infrastructure for mining software development data from open source forges, reorganizing data to feed the analytic need, and providing free service (API query) for scholars to conduct their researches. We notice that several large-scale software mining efforts (e.g., Software Heritage [30], GHTorrent [50, 47, 48, 52, 51], Boa [34]) already exist and provide services to the public, however, none of them provide a complete census of OSS with an analysis engine. We, therefore, decide to build our own infrastructure, World of Code (WoC). WoC periodically

(every three months) retrieves projects with new development activities from popular open source forges, e.g., GitHub, BitBucket, extracts and stores new development data to our core distributed database. Subsidiary services are also updated to provide the latest development data to researchers. Currently, WoC[1] contains a collection of more than 123M software projects, 42M developer IDs and 2B commits. So far, we provide three types of APIs (Python, Perl, Shell Script) to meet different users' needs with various backgrounds. Initially, WoC is designed to serve the queries among software development entities, e.g., given a developer ID, returns all of his/her project name. According to the definition of SSC, these entities are the components in SSC networks, and any SSC network can be constructed simply by integrating a series of queries. Besides such basic query service, we develop various subsidiary services to further assist researchers:

- Language based software ecosystem data source is gathered by extracting projects, developers and source files that are related to specific language. These data source can reduce the efforts that researchers otherwise have to spend on conducting researches in a specific language domain.

- Fork detection [101] is enabled to achieve a more clean set of software projects by deduplicating projects that are essentially clones. Data set encompassing forks/clones might introduce bias to research analytics and might weaken corresponding findings.

- Developer ID deduplication [4] is enabled to cluster multiple developer IDs (owned by a single developer) to a single real developer. A software developer may have multiple accounts and use different IDs when contributing source code. By correctly identifying and clustering multiple IDs attached to a single developer, we can achieve more accurate SSC networks (authorship network), and better evaluate developer's reputation and contribution.

Once the WoC infrastructure is ready, we begin to evaluate SSC from two perspectives: Insights and risk reduction. More specifically, we want to learn if SSC could bring new insights to software development, and what new insights and analysis it could bring.

[1] https://bitbucket.org/swsc/overview/src/master/

Furthermore, we want to see if SSC can help in reducing various risks in both software development and OSS community.

We found that SSC brings insights in various aspects.

After leveraging the joining power of SSC and WoC infrastructure, researchers successfully conducted various types of researches and analysis:

- Historical trend of usage of programming languages are captured
- A couple of cross-ecosystem comparison studies (file cloning and developer migration) are implemented
- Python package sustainability analysis

Another insight is the operationalization of SSC, i.e., what are the components and networks in SSC? In software domain, we have three major entries: software project, developer, and source code. Based on the types of node and edge (relationship) in SSC , we propose three different networks to describe SSC: dependency network, code reuse network and knowledge network. Dependency network represents the dependency relationships among software projects and packages; Code reuse network describes the source code spread across software projects on a finer level; Knowledge flow network represents knowledge transformation within developers through cooperation in software development. Each of the networks represents a unique type of relationship, and we hope these networks are able to capture crucial relationships in software development. Note that there might be other types of networks in SSC based on different relationships, which can be added in to further improve the completeness of SSC measurement, but we do not discuss it in this thesis.

SSC is helpful in mitigating risks embedded in the development and usage of OSSs. To exemplify the benefit of SSC for risks reduction, we investigated the risk of abandonment from the perspective of a user downstream and the risk of low adoption from the perspective of the producer upstream.

Among other potential research topics, the phenomenon of software technologies spread draws our attention. We investigate what combination of attributes are driving the adoption of a particular software technology, and hope that developers seeking to increase the adoption

rate of their products can benefit from our findings. We look into the adoption of a pair of competitive packages (tidy and data.table) in R, derive 11 variables (e.g., issue response, overall deployments, etc.) from social contagion theory and SSC to model the adoptions, leverage WoC infrastructure to extract the adoption related data in OSS community, and train a choice model (regression model) over the collected dataset to measure the influence of each factor. We find that a quick response to raised issues, a larger number of overall deployments, and a larger number of high-score StackExchange questions are associated with higher adoption. Decision makers tend to adopt the technology that is closer to them in the technical dependency network and in author collaborations networks while meeting their performance needs. To gauge the generalizability of the proposed methodology, we investigate the spread of two popular web JavaScript frameworks `Angular` and `React`, and achieve a similar result. We hope that our methodology encompassing social contagion that captures both rational and irrational preferences and the elucidation of key measures from large collections of version control data provides a general path toward increasing visibility, driving better informed decisions, and producing more sustainable and widely adopted software.

In short, a supply chain perspective on OSS development leads to novel research questions, insights, and practical applications. In this thesis, we firstly define SSC, propose a number of crucial networks to measure SSC, describe the construction of an infrastructure (WoC) which provides free access to an almost entire collection of OSS development data for study and research, show how various empirical engineering applications could benefit from our infrastructure, and in the end report an investigation on the spread of software technologies from the perspective of SSC. We conclude that SSC networks are vital in understanding how OSS ecosystem functions, and play an important role in helping mitigate the risks of relying on the OSS system. Ultimately, we hope that the discoveries enabled by SSCs would allow the rapid innovation in OSS along with its growing size and complexity.

1.2 Problem Statement

As OSS communities flourish, software industry is increasingly dependent on OSS. Giant companies in software domain often open source inner projects to attract developers worldwide to contribute and increase their market share. Meanwhile, OSSs are always attractive to commercial companies to use for its free-of-cost nature. Although researchers have been spending large amount of efforts in investigating OSS development, most of such investigations are focused on a specific software project or groups of projects within one ecosystem [17, 64]. As lacking of evaluation over the entirety of OSS, the findings from these researches might be limited to specific software projects or ecosystems, posing risks to scenarios where the findings are applied but not fit.

In the process of a typical OSS development, developers that are scattered all over the world cooperate together remotely, and make contributions to a software repository hosted on one of open source platforms, where version control system (VCS) is embedded to ensure that conflicts can be resolved and synchronisation are maintained, along with the capability to roll back to previous versions as all historical changes are saved inside VCS. Inspired by the concept of SCM in traditional industries, where supply chain design and maintenance are optimized to retain great commercial profit while various potential risks are reduced, software development, especially for open source software, can be viewed as a process of SCM. More specifically, source code as product in software development is generated by software developers and shipped to the host (software repository), and then shipped to other developers to integrate their newly generated piece of source code. From the perspective of dependency, a software may depend on other software projects and packages, i.e., downstream packages or projects, and each software project serves as an upstream or/and downstream from the global view. The risks in one software project will influence its upstream projects, posing the potential to leverage existing SCM findings on risk mitigation in OSS domain.

The very first research question need to be addressed for SSC is:

(RQ1) How to define SSC?

We extensively explored the existing published related studies on SSC, and found that there was neither proper contextualized SSC definition for OSSs, nor a measurable definition. We define SSC by carefully making analog of every key component from traditional SC, so that vast amount of existing findings in traditional SC can be tested and applied to SSC in an effortless manner.

Once we have the proper definition of SSC, the next step is to measure SSC. So we ask:

(RQ2) How to measure SSC for OSS?

In order to measure SSC, we need to have an infrastructure for providing access to software development data in large. There are already a number of data mining platforms available: Software Heritage [30], GHTorrent [50, 47, 48, 52, 51], Boa [34], etc. However, none of these platforms can meet our high demand of large data query on relationships discovery and analysis over the entirety of OSS (see Sec 3.2 in Chapter 3 for reference).

We aim to build an infrastructure to facilitate the discovery of various relationships in OSS community and make the related analytics efficiently. To meet such a requirement, we leverage HPC resources to collect large amount of OSS from various open source platforms, extract and refine the development data, create distributed key-value databases to store these components and relationships, and serve queries directly from these core databases. Meanwhile, we pre-calculate basic statistics for different entities and store then into MongoDB databases to meet the potential queries for direct analysis usage. The details of the infrastructure and services are reported in Chapter 3. Note that our infrastructure (WoC) is evolving and more services are expected to be provided in the future.

With such an infrastructure ready, we would like to evaluate SSC. More specifically, we evaluate SSC from two perspectives: new insights/analysis and risk mitigation in OSSs. From the perspective of new insights and analysis, we ask:

(RQ3) What can we learn about SSC in OSS?

We conduct two types of studies on top of SSC using WoC infrastructure to verify if our infrastructure is capable of supporting research tasks in SSC, and to see what new insights and analysis can be implemented in SSC.

To achieve this, we implement several basic research tasks that require the entirety of FLOSS data as a part of the investigation. Furthermore, we recruit three researchers external

to our group to either conduct investigations of their own utilizing WoC and SSC, or to provide us with their research problems that can only be solved by using WoC. We found that the combination of SSC and WoC infrastructure can enable and efficiently support various domain specific and cross-ecosystem researches. Furthermore, we plan to move our services to cloud platform to attract more users, and many more promising insights and analysis would be expected in the future.

Besides these insights on specific research tasks, the operationalization of SSC networks itself might be interesting. In software development, developers, projects, source code and tacit knowledge are primary entities. SSC networks should represent the complicated relationships within and across these entities. Based on the types of components and relationships, we propose three SSC networks (dependency network, code reuse/spread network and knowledge flow/authorship network, see details in Chapter 4). We are aware that these three networks can not cover all relationships inside OSS ecosystem, and we look for additional types of networks in the future to make the operationalization of SSC towards becoming complete.

From the perspective of the value of SSC in risk mitigation in OSS, we ask:

(RQ4) How can we reduce the risks through SSC in OSS?

To exemplify the value of SSC on reducing OSS risks, we look into a specific research domain, the analysis of software technologies adoption among developers. Software technologies spread or migration is vital for both software developers and software companies. On one hand, being able to recognize and pay particular attention to the influential factors of software adoptions can help software developers maintain the attractiveness of their software products and keep users loyalty. On the other hand, being able to correctly evaluate a software's competitiveness and predict the adoption trend would help software companies to make wise decisions on which technology to support and which to abandon. We leverage social contagion theory and statistical modeling to identify, define, and test empirically measures that are likely to affect software adoption. More specifically, we construct a software dependency chain for a specific set of R language source-code files. We formulate logistic regression models, where developers' software library choices are modeled, to investigate the combination of technological attributes that drive adoption among competing data frame

(a core concept for a data science languages) implementations in the R language: `tidy` and `data.table`. We quantify key project attributes (e.g., response times to raised issues, overall deployments, number of open defects, etc.) and find that a quick response to raised issues, a larger number of overall deployments, and a larger number of high-score StackExchange questions are associated with higher adoption. Decision makers tend to adopt the technology that is closer to them in the technical dependency network and in author collaborations networks while meeting their performance needs. See Chapter 5 for more details.

1.3 Motivations and Goals

The key output of software development activity is the source code. Therefore software development is reflected in the way the source code is created and modified. Although various individual and groups of projects have been well studied, it only gives partial results and conclusions on the propagation and reuse of source code in the large. For example, Cédric Teyton studied library migration for Java projects in Apache[106], Foutse Khomh studied release cycles relation with software quality in Mozilla Firefox[63], Michael W. Godfrey investigated the evolution of open source software on Linux source code[111]. Moreover, little is known about intensive interactions and complicated relations among software projects and software developers, which could bring risks to one's system. For example, an 11 lines NPM package called left-pad with only 10 stars on GitHub was unpublished and it broke some of the most important packages on all of NPM, and Facebook stopped functioning. Besides, no systematic architecture or approach has been created to discover and analyze these underlying relations.

We propose to bridge these gaps through the lens of software supply chain. As in traditional supply chains, the developers in FLOSS make individual decisions with some cooperative action, hence the analytical findings from traditional supply chains may help in FLOSS. Second, we have a complicated network of technical dependencies with code and and knowledge flows akin to traditional supply chains, making the analogies less complicated. Third, the emerging phenomena, for example, the lack of transparency and visibility, appear to be as, or more, important in FLOSS as in traditional supply chains. Fourth, unlike

traditional supply chains, FLOSS has very detailed information about the production and dependencies. We, therefore, hope that detailed data with supply chain analytical framework may bring transformative insights not just for FLOSS supply chains, but for all supply chains generally. We, therefore, would like to systematically analyze the entire network among all the repositories on all source forges, revealing upstream to downstream relations, the flow of code and the flow of knowledge within and across projects.

1.4 Contribution

Firstly, we developed an approach (SSC) to enable systematic analysis of open source community, revealing underlying complicated relations among software projects and developers. As the transparency and visibility increase in SSC, software developers are enabled to have a more comprehensive knowledge and estimation of other software projects, make wise decisions when choosing from other software projects to integrate (as downstream components) through the evaluation of a combination of developer reputation and software project maintenance. Furthermore, risks in software development can be mitigated by an early detection of bug/vulnerability propagation in source code snippets (code reuse network), downstream packages (dependency network), and submissions from inexperienced or malware developers (knowledge flow/authorship network).

Secondly, we built an infrastructure(WoC) to provide broad data access and facilitate SSC network construction and related analysis. We enable wide data access to collected data source by providing a tool built on top of the infrastructure, which scales well with completion to query in linear time. Furthermore, we implement ways to make this large dataset usable for a number of research tasks by implementing targeted data correction and augmentation and by creating data structures derived from the raw data that permit accomplishing these research tasks quickly, despite the vastness of the underlying data. In a nutshell, WoC can provide support for diverse research tasks that would be otherwise out of reach for most researchers. Its focus on global properties of all public source code will enable research that could not be previously done and help to address highly relevant challenges of open source ecosystem sustainability and of risks posed by this global software supply chain.

Lastly, by leveraging SSC networks, we investigated the adoption of technologies in software domain, discovered factors that are influential to decision makers and provide suggestions to software developers on making one's product popular. The contribution consists of proposing a method to explain and predict the spread of technologies, to suggest which technologies are more likely to spread in the future, and suggest steps that developers could take to make the technologies they produce more popular. Developers can, therefore, reduce risks by choosing technology that is likely to be widely adopted. The supporters of open source software could use such information to focus on and properly allocate limited resources on projects that either need help or are likely to become a popular infrastructure. In essence, our approach unveils previously unknown critical aspects of technology spread and, through that, makes developers, organizations, and communities more effective.

1.5 Thesis Organization

This thesis is divided into six chapters. The remaining of this thesis is organized as follows:

Chapter 2 describes the origin and success of supply chain in traditional industries, the background of SSC in OSS community, the definition of SSC, the importance of transparency and visibility properties in SSC.

Chapter 3 dives into the details of WoC infrastructure construction to meet queries for software development data in large and assist various analytics implemented by researches in open source domain. The whole framework are divided into four stages: project discovery from open source platforms, project clone to local machines, development data extraction, decomposition and store, and database update. This chapter also discusses a novel update mechanism through which overall data transmission demand are largely reduced and the total time cost for a full update significantly decreases. Besides the details of infrastructure design, this chapter describes various APIs provided for users to run and customize the queries. Furthermore, performance benchmark of different APIs and suggestions for choosing proper API for different tasks are provided for assistance. Finally, it discusses the potential limitations on data incompleteness and data update frequency, and poses a list of services that are expected to be implemented in the future.

Chapter 4 discusses the evaluation of WoC's capability on being supportive for a wide range of applications. It reports researches that include both ecosystem-wise and cross-ecosystem analysis which leverage WoC for data resources. It also provides sample code (multiple approaches) that are used in one of the applications as a guidance for users to follow. Lastly it discusses the operationalization of SSC and various types of SSC networks.

Chapter 5 reports a study of software technology adoptions among developers to show how OSS risks can be reduced by leveraging SSC. It describes an investigation on what combination of attributes are driving software technology adoption by modeling (choice model) a group of key properties (derived from social contagion and SSC) on a pair of competitive R packages (data.table and tidy). It also includes a second case study in Javascript domain (Angular VS. React) to verify the external generalizability of the previous findings.

Chapter 6 concludes by reiterating over the research questions and the answers we found, as well as the summary of contributions made in this thesis.

Chapter 2

Conceptualization of Software Supply Chain[1]

2.1 Overview

In this chapter, we discuss how integrating the concept of supply chains [87] with OSS development and leveraging the existing knowledge on traditional supply chains, big data, and data science can lead to a more comprehensive understanding, the formulation of new research problems, and practical applications. More specifically, we start by introducing SC concept and the rise of SSC. Then, we explore the historical studies on SSC. After that, we give our definition of SSC in an measurable manner. Finally, we discuss two important properties (visibility and transparency) in SC and SSC, and emphasize how we can benefit in the context of OSS.

[1]This chapter, in part, is a reprint of the material as it appears in 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 458-459, titled "*Constructing supply chains in open source software*" (2018). Authors: **Yuxing Ma** and Audris Mockus. The dissertation author was the primary investigator and author of these two papers. Copyrights of both papers are held by **Yuxing Ma** and Audris Mockus.

2.2 The rise of SC and SSC

2.2.1 Traditional Supply Chain and Management

Supply chains (SC) were originally defined as encompassing all activities associated with the flow and transformation of goods from raw materials through to the end user, as well as the associated information flows. For example, the supply chain of a cheesecake factory includes cheese factories (upstreams) where cheese (raw material) were made and shipped to cheese factory, and cheesecake stores (downstreams) where cheesecakes were put on sale to meet shoppers/customers (end users).

The term of "Supply Chain Management" (SCM) was invented by Keith Oliver [87] in 1982. However, it did not draw much attention until late 1990s, when Robert published a book pertaining to "Supply Chain Management" [91] and further defined it as the integration of supply chain activities through improved supply-chain relationships to achieve a competitive advantage. The ultimate goal of SCM is to ensure that merchandise produced and distributed at the right quantities, to the right locations, and at the right time, in order to minimize system-wide costs while satisfying service level requirements.

2.2.2 Software Supply Chain

In last century, software was mainly developed separately within each software company, and few intermediate products (source code) were able to be seen/utilized by other software engineers outside, which indicates a loss of their full value. In 1998, the open-source software movement came into the scene and greatly shocked traditional software building process by introducing the core idea "free and open software". Under the influence of this idea, millions of free software supporters create and share software projects on popular open-source platforms such as GitHub and BitBucket, where most projects are public and every piece of source code is available.

With this futile land for growing software, we are enabled to view software domain from a new perspective: Software Supply Chain (SSC).

Supply chain concept of a system of organizations, people, activities, information, and resources involved in moving a product or service from supplier to customer has been tremendously successful for helping businesses manage risks that arise from the distributed nature of production. OSS production is also distributed with decision making that is not centralized and done by a variety of individuals and groups. With a variety of risks affecting software development, it is reasonable to assume that the supply chain concept will benefit open source community and the achieved experience and findings in traditional supply chain for mitigating various risks will apply in OSS.

2.3 Literature Review

Before we devote into the study of SSC, let's look into the historical efforts on this topic.

"software supply chain management" was firstly proposed by Jacqueline in 1995 [56] to replace 'system development life cycle' to integrate with business. However, SSC was not clearly defined explicitly, and did not draw much attention in software research domain. In 2003, Jack Greenfield investigated the concept of product line [53], claiming that software components are created by software factories, and software supply chain is used to create standards to ease the alignment and assembling process downstream. In 2005, Aparna [19] proposed a research question (how to select from multiple suppliers) in software focused supply chain. In 2010, Robert Ellison [40] reused SC concept, discussed several ways how risks can be introduced in coding, control management, deployment and operations.

In short, we did not find much existing efforts on the investigation on SSC. Furthermore, there is no definition of SSC contextualized for OSS where it may help with distributed decision making, and there is no measurable definition of SSC proposed in before.

2.4 Define SSC

We decide to conceptualize SC concept in software domain. We define SSC by carefully making analog of components in traditional SC. SSC has developers and groups ("companies") , corporate backers supporting these developers or groups ("financing"), relationships

among software projects or packages representing the "chain" of the flow, and changes to source code (e.g. files, modules, frameworks, or entire distributions) representing products or information.

2.5 Visibility and Transparency

Traditional SCM literature highlights the critical importance of Visibility and Transparency for managing supply chains. The concept of transparency more specifically refers to the level of detail that a willing individual can grasp in a certain entity or activity within the supply chain. Generally speaking, the tenets of OSS postulate the need to have all decisions recorded publicly via commits, online discussions, or issue trackers. Unlike traditional supply chains where intermediaries may not care to reveal their upstream sources or business strategies, OSS presumably has much more transparency. However, some important aspects, e.g. the affiliation and the motivation of contributors may not be known. More importantly, the transparency of continued support, that contracts in traditional supply chains provide, is often completely absent in OSS.

In other words, transparency is defined as information that developers share with their consumers about the inputs, processes, sources and practices used to bring the product to the consumer. It is more outwardly focused and more from the consumer perspective than visibility.

The second critical aspect of the SCM is visibility. This, in contrast to transparency, reflects the ability of developers who are users of other software. It relates to the ability of the user of a package, of a pull request, an issue, or of a potential collaborator to asses the qualities of the object (or subject) to be used (engaged). Transparency and Visibility are distinct concepts and increasing each may require different approaches. The massive size of the OSS ecosystem makes it very difficult for any particular consumer to understand the practices and qualities of every open source software and developer, no matter how transparent the activities of all other projects and developers may be. We must therefore increase visibility by making it easier for any specific stakeholder to discover and understand the entities that are most relevant to them in a straightforward manner.

In other words, visibility is defined as information that developers have about the inputs, processes, sources and practices used to bring the product to consumers/market. This includes complete supply chain visibility including traceability for the entire supply chain. Visibility is, generally, inwardly/developer focused.

In summary, to increase transparency we must create general summaries or extract salient properties of the nodes and links in the supply chain that are not directly reflected in the operational data and need to be estimated via modeling or other exercises. To increase visibility, however, it is necessary to enable potential users to find, assess, and use the elements they may need. As the visibility and transparency increase in SSC networks, risks in OSS community are mitigated.

In next chapter, we describe an research infrastructure, world of code (WoC), as a software development data platform for measuring SSC.

Chapter 3

Research Infrastructure[1]

In this chapter, we present the work of constructing an research infrastructure to assist the analysis of SSC and the implementation of related applications in OSS community. Open source software (OSS) is essential for modern society and, while substantial research has been done on individual (typically central) projects, only a limited understanding of the periphery of the entire OSS ecosystem exists. For example, how are tens of millions of projects in the periphery interconnected through technical dependencies, code sharing, or knowledge flows? To answer such questions we a) create a very large and frequently updated collection of version control data for FLOSS projects named World of Code (WoC) and b) provide capabilities to efficiently correct, augment, query, and analyze that data. Our current WoC implementation is capable of being updated on a monthly basis and contains over 24B git objects. To make WoC more easily usable in a wide variety of research scenarios, we have designed an architecture to help simplify, support, and evaluate the implementation of research tasks.

[1]This chapter, in part, is a reprint of the material as it appears in 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 143-154, titled "*World of code: an infrastructure for mining the universe of open source VCS data*" (2019). Authors: **Yuxing Ma**, Audris Mockus, et al. The dissertation author was the primary investigator and author of these two papers. Copyrights of both papers are held by **Yuxing Ma** and Audris Mockus.

3.1 Overview

Tens of millions of software projects hosted on GitHub and other forges attest to the rapid growth and popularity of Free/Libre Open Source Software (FLOSS). These online repositories include a variety of software projects ranging from classroom assignments to components, libraries, and frameworks used by millions of other projects. Such large collections of projects are currently archived in public version control systems, and, if made available for analysis conveniently, would represent a unique opportunity to study FLOSS at large and answer both theoretical and practical questions that rely on the availability of the entirety of FLOSS data. In particular, this infrastructure, referred to as World of Code (WoC) and described below, allows researchers to conduct a census of open source software that would provide types and prevalence across projects, technologies, and practices and serve as a guide to setting policies or creating innovative services. Our infrastructure facilitates the discovery of key technical dependencies, code flow, and social networks that provide the basis to understand the structure and evolution of the relationships that drive FLOSS activities and innovation. Such a large database of software development activities can serve as a basis for "natural experiments" that evaluate the effectiveness of different software development approaches. If preserved, it will also facilitate future anthropological studies of software development [30].

Our objective in the current study is to describe a prototype of an infrastructure that can store the huge and growing amount of data in the entire FLOSS ecosystem and provide basic capabilities to efficiently extract and analyze that data at that scale. Our primary focus is on the types of analyses that require global reach across FLOSS projects. A good example is a software supply chain where software developers correspond to the nodes or producers, relationships among software projects or packages represent the "chain", and changes to the source code represent products or information (that flow along the chain) with corporate backers representing "financing."

Several formidable obstacles obstruct progress towards this vision. The traditional approaches for obtaining the repository of a project or a small ecosystem does not scale well and may require too many resources and too much effort for individual researchers or

smaller research groups. Thus, the community needs a way to scale and share the data and analytic capabilities. The underlying data are also lacking in context necessary for meaningful analysis and are often incorrect or missing critical attributes [82]. Keeping such large datasets up-to-date poses another formidable challenge.

In a nutshell, our approach is a software analysis pipeline starting from discovery and retrieval of data, storage and updates, and transformations and data augmentation necessary for analytic tasks downstream. Our engineering principles are focused on using the simplest possible techniques and components for each specific task ranging from project discovery to fitting large-scale models. The result is a conceptual implementation loosely following the microservices architecture [85], where the design and performance of the loosely coupled components can be independently evaluated, each service can utilize a database that is optimal for its needs, and the most computationally-intensive components are extremely portable to ensure they run on any high-performance platform. Specifically, our prototype appears to capture almost the entirety of the publicly available source code in version control systems and the latency of updates on the existing hardware platform does not exceed one calendar month, which is pretty fast given the size of the dataset and the complexity of the task (See Sections 3.3.1 and 3.3.2 for more details). Furthermore, we built a tool on top of the infrastructure and provided two types of API to enable wide data access for users.

We begin with an overview of related work in Section 3.2. The architecture of the prototype implementation of the infrastructure is discussed in Section 3.3. We facilitate wide access to the large data collection by developing a tool on top of our infrastructure, which is described in Section 3.4, along with an evaluation of query performance. We discuss various ways of improving the existing infrastructure in Section 3.5, discuss a few existing limitations in Section 3.6, and conclude our paper in Section 3.7.

3.2 Literature Review

While we are not aware of a complete census of FLOSS with an analysis engine, several large-scale software mining efforts exist and may be roughly subdivided into attempts at preservation, data sharing for research purposes, and construction of decision support tools.

Software development is a novel cultural activity that warrants preservation as a cultural heritage. The software source code, the only representation of software that contains human readable knowledge, needs to be preserved to avoid permanent loss of knowledge [30]. Software Heritage [30] is a distributed system involved in collecting and storing large amount of open source development data from various open source platforms and package hosts. It currently has software from GitHub, GitLab, Debian, PyPI, etc., and contains 73M projects, 1.7B commits, and 15.6B source files. The main drawback of this particular effort is the lack of focus on enabling applications to software analytics. The API provided allows for quick query of every historical particle in a software project and meets the preservation need, however, it does not grant the access to the full relationships (e.g., the set of projects containing a given commit) among these particles across entire collection of software. Quick access to these relationships is crucial in conducting software analytics such as identification of dependencies among artifacts and authors as well as code spread in the open source community.

One potential value of archiving software lies in the reuse of software artifacts. For example, Nexus[2] repository manager, allows developers to share software artifacts in a standard way and provides support for building and provisioning tools (e.g. Maven) to access necessary components such as libraries, frameworks and containers.

Commercial efforts, such as BlackDuck or FOSSID[3] have proprietary collections they use to determine if their clients have included open source software within their proprietary software code. It is generally not clear how complete these collections are nor if the companies involved might consider opening them for research purposes.

In addition to source code and binaries, large scale collection of other software development resources could be integrated with the source code data. For example, GHTorrent [50, 47, 48, 52, 51] attempts to record every event for each repository hosted on GitHub and provides multiple approaches (SQL request and MongoDB data dump) for data access. The primary limitation is that the collected metadata is specific to GitHub and it does not include the underlying source code as well. Therefore, obtaining dependencies

[2] https://www.sonatype.com/nexus-repository-oss

[3] blackducksoftware.com,fossid.com

encoded within the source code cannot be accomplished. FLOSSmole [58] collects open source metadata from various forges as a base for academic research but only focuses on software project metadata.

Another platform is Candoia [108, 107, 112, 113] which provides software development data collections abstraction for building and sharing Mining Software Repository (MSR) applications. In particular, Candoia contains many tools for artifact extraction from different VCSs and bug databases and it also support projects written in different languages. On top of these artifacts, Candoia created its general data abstraction for researchers to implement ideas and build tools upon. This design increased portability and applicability for MSR tools by enabling application on software repositories across hosting platforms, VCSs and bug recording tools. The approach is focused on the design and benefits of creating a specialized software repository mining language. While it abstracts a number of repository acquisition tasks, it also makes it more difficult to handle operational data problems that tend to occur at much lower levels of abstraction and tend to be too idiosyncratic for generalized abstraction. The main drawbacks of Candoia are that it only supports limited programming language (JS and Java) based projects, and ecosystem-wide research might be difficult to implement since Candoia relies on users to provide software related data (e.g., targeted software repository URL) and eco-system wide compliance is generally low.

Other platforms are aimed at improving reproducibility by providing a repository of datasets for researchers to share their data. These include PROMISE Repository [99], Black Duck OpenHub[4], and SourcererDB [88]. PROMISE Repository is a collection of donated software engineering data. It was created to facilitate generations of repeatable and verifiable results as well as to provide an opportunity for researchers to extend their ideas to a variety of software systems. Black Duck OpenHub is a platform that discovers open source projects, tracks the development and provides the functionality of comparison between softwares. Currently, it is tracking 1.1M repositories, connecting 4.2M developers and indexing 0.4M projects. SourcererDB is an aggregated repository of 3K open source Java projects that are statically analyzed and cross-linked through code sharing and dependency. On top of

[4] https://www.openhub.net/

providing datasets, it also provides a framework for users to create custom datasets using their projects.

Apart from providing datasets (repository) for potential users, platforms such as Moose [32], RepoGrams [95], Kenyon [12], Sourcerer [7], and Alitheia Core [49] are more focused on facilitating building and sharing MSR tools. Moose is a platform that eases reusing and combining data mining tools. RepoGrams is a tool for comparing and contrasting of source code repositories over a set of software metrics and assists researchers in filtering candidate software projects. Kenyon is a data platform for software evolution tools. It is restricted to supporting only software evolution analysis. Sourcerer is an infrastructure for large scale collection of open source code where both meta data and source code are stored in a relational database. It provides data through SQL query to researchers and tool builders but is only focused on Java projects. Alitheia Core is a platform with a highly extensible framework and various plug-ins for analyzing software on a large database of open source projects' source code, bug records, and mailing lists.

Furthermore, there were efforts to standardize software mining data description for enhanced reproducibility [65]. None of the listed platforms focus on both collection and analysis of the dependencies of the entirety of FLOSS source code version control data. Further, they contain either limited collections (e.g. only GitHub, no source code, have only donated data, or do not contain an analysis engine). For example, it is not possible to answer simple questions such as "In which projects has a file been used?", "What projects/codes depend on a specific module?", "What changes has a specific author made?" etc.

Some large companies have devoted substantial effort to develop software analysis platforms for the entire enterprise, aiming to improve the quality of software they build and to help the enterprise achieve its business goals by providing recommendations to software development organizations/teams, monitoring software development trends, and prioritizing research areas. For example, Avaya, a telecommunications company, built a platform [54], which collects software development related data from most of its software development teams and third parties and enabled systematic measurements and assessments of the state of software. CodeMine [24], is a software platform developed by Microsoft that collects a variety of source code related artifacts for each software repository inside Microsoft. It is

designed to support developer decisions and provide data for empirical research. We hope that similar benefits can be realized with the WoC platform targeted to the entire FLOSS community.

Large scale software mining efforts also include domain specific languages. Robert Dyer et al. developed Boa [34, 37, 33, 89, 36, 35], both as a domain specific language and as an infrastructure, to ease open source-related research over large scale software repositories. The approach is focused on the design and benefits of an infrastructure and language combination. However, the lack of explicit tools to deal with operational data problems make it of limited use to achieve our aims. Their collection procedures -discovery, retrieval, storage, update, and completeness issues (for example, only certain languages are supported)- are not the primary focus of this effort. The tools to deal with operational data problems common in version control data are also lacking in Boa.

The system described in this paper is loosely modeled after a system described a decade ago [81, 79]. In comparison, at that time, git was just beginning to emerge as a popular version control system, but now it dominates the FLOSS project landscape. The number of software forges and individually hosted projects was much larger then in contrast to the consolidation of forges and the overwhelming dominance of GitHub. Furthermore, the scale of the FLOSS ecosystem is more than an order of magnitude larger now and it continues to experience very rapid growth. WoC could not, therefore, reproduce that design closely and, instead, is focused on preserving the original git objects and on creating a design that enables both efficient updating of this huge database and ways to cross-reference it so that the complete network of relationships among code and people is readily available.

3.3 Architectural Considerations

The process of mining individual git repositories is complex to begin with [14], but becomes even more difficult on a large scale [45]. More specifically, using operational data from software repositories requires resolution to three major problems [82]: the lack of context, missing attributes or observations, and incorrect data. This makes critical tasks such as debugging and testing complex and time consuming. To cope with these big data challenges

we employed both vertical and horizontal prototyping [93, 2, 71, 15]. Most big data systems use the layered data approach where initial layers approximate raw data and later layers include cleaned/augmented data.

In this section we present a prototype WoC implementation. It has four stages: project discovery, data retrieval, correction, and reorganization as shown in Figure 3.1.

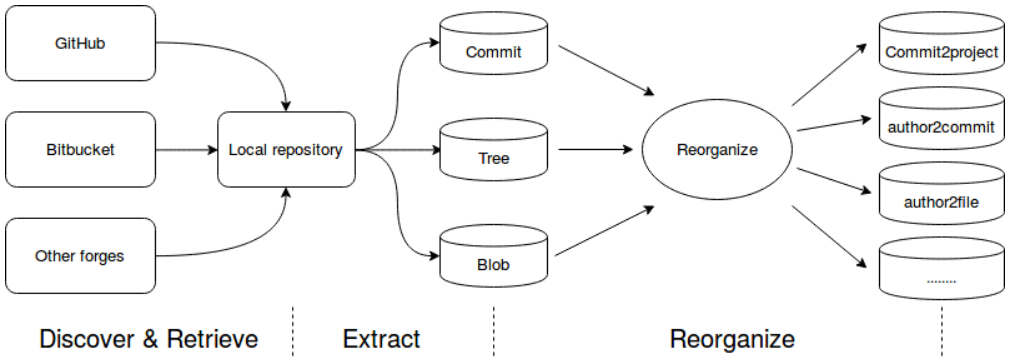

Figure 3.1: Overarching data flow

3.3.1 Project Discovery

Millions of projects are developed publicly on popular collaborative platforms/forges such as GitHub, Bitbucket, GitLab, and SourceForge. Some of the FLOSS projects can be identified from the registries maintained by various package managers (e.g., CRAN, NPM) and Linux distributions (e.g., Debian, Fedora). Other project repositories, however, are hosted in personal or project-specific sites. A complete list of FLOSS repositories is, therefore, difficult to compile and maintain since new projects and forges are created and older forges disappear. There is a tendency for the FLOSS repositories to migrate to (or be mirrored on) several very large forges [74]. A number of older forges provide convenient approaches to migrate repositories to other viable forges before being shut down. This consolidation has alleviated some of the challenge of discovering all FLOSS projects [81], though the task remains

nontrivial. We discuss several approaches to project discovery below. To package our project discovery procedure we have created a docker container[5] that has the necessary scripts.

Using Search API: Some APIs may also be used to discover the complete collection of public code repositories within a forge. The APIs are specific to each forge and come with different caveats. Most APIs tend to be rate limited (for user or IP address) and the retrieval can be sped up by pooling the IDs of multiple users.

Using Search Engine: Search engines (e.g., Google or Bing) can supplement the discovery of FLOSS project repositories on collaborative forges when the forge does not provide an API, or when the API is broken. The primary drawback is the incompleteness of the repositories discovered.

Keyword Search: Some forges provide keyword based search of public repositories, which is a complementary approach when a forge does not provide APIs for the enumeration of repositories and the results returned from search engines are lacking.

Using these and other opportunistic approaches helps ensure that they complement each other in approximating the publicly available set of repositories though it does not guarantee the completeness. We expected that various ways of crowdsourcing the discovery (with incentives to share a project's git URL) would help increase the coverage in the future.

3.3.2 Project Retrieval

This data retrieval task can be done in parallel on a very large number of servers but requires a substantial amount of network bandwidth and storage. The simplest approach is to create a local copy of the remote repositories via git clone command. As of December 2018, we estimate over 62M unique repositories (excluding GitHub repositories marked as forks, repositories with no content and private repositories). A single thread shell process on a typical server CPU (we used Intel E5-2670) with no limitations on network bandwidth clones randomly selected 20K to 50K repositories (the time varies dramatically with the size of a repository and the forge) in 24 hours. To clone 60M repositories in one week would, therefore, require from two to four hundred servers. We do not possess dedicated resources of such size and, therefore, optimize the retrieval by running multiple threads per server

[5]https://github.com/ssc-oscar/gather

and retrieving a small subset of the repositories that have changed since the last retrieval. Specifically, we use five Data Transfer Nodes of a cluster computing platform[6].

3.3.3 Data Extraction

Code changes are organized into commits that typically change one or more source code files within the project. Once the repository is cloned as described above, we extract Git objects[7] from each repository and store these git objects in a single database.

Data Model

Git [18] is a content-addressable filesystem containing four types of objects. The reference to these objects is a SHA1[8] [38] calculated based on the content of that object. **commit** is a string including the SHA1's of commit parent(s) (if any), the folder (tree object), author ID and timestamp, committer ID and timestamp, and the commit message. **tree**: A tree object is a list that contains SHA1's of files (blobs) and subfolders (other trees) contained in that folder with their associated mode, type, and name. **blob**: A blob is the compressed version of the file content (the source code) of a file. **tag**: A tag is the string (tag) used to associate readable names with specific versions of the repository.

Fig. 3.2 illustrates relationships among objects described above. The snapshot at any entry point (commit) is constructed by following the arrows from left side to right side.

Object Extraction

While a standard Git client allows extraction of raw git objects, it displays them for manual inspection. For the bulk extraction need, first we list all objects within the git database, categorize them, and create bulk extractor based on a portable pure C implementation of *libgit2*[9]. We run listing and extraction using 16 threads on each of the 16-CPU node on

[6]No. node: 300, Bandwidth up to 56 Gb/s
[7]https://git-scm.com/book/en/v2/Git-Internals-Git-Objects
[8]https://en.wikipedia.org/wiki/SHA-1
[9]https://libgit2.org/

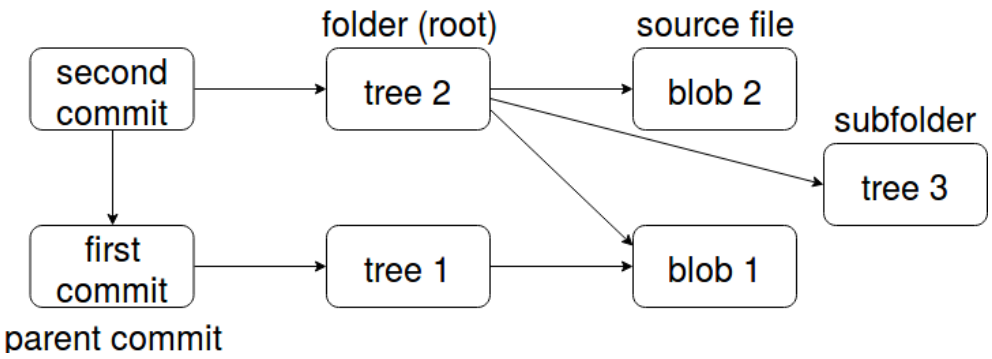

Figure 3.2: Git objects

a cluster[10]. The process takes approximately two hours for a single node to process 50K repositories. The extraction procedure represents a microservice.

3.3.4 Data Storage

The collection of public Git repositories as a whole replicate the same git object hundreds of times [81]. Without removing this redundancy, the required storage for the entire collection exceeds 1.5PB, and it also makes analytics tasks virtually impossible without extremely powerful hardware. Many reasons for this redundancy exist, such as pull-based development, usage of identical tools or libraries, and copying of code.

To avoid redundancy of git object among repositories, we store all git objects into a single database. The database is organized into four parts corresponding to each type of git object. Each part is further separated into a cache and content. The cache is used to rapidly determine if the specific object is already stored in our database and is necessary for data extraction described above. Furthermore, the cache helps determine if a specific repository needs to be cloned. If the heads (the latest commits in each branch in .git/refs/heads) of a repository are already in our database, there is no need to clone the repository altogether.

[10]CPU: E5-2670, No. node: 36, No. core: 16, Mem size: 256 GB

Cache database is a key-valued database, with the twenty byte Git object SHA1 being the key and the packed integer (indexing the location of the object in the corresponding value database) being the value. The value database consists of an offset lookup table that provides the offset and the size of the compressed git object in a binary file (containing concatenated compressed git objects). While this storage allows for a fast sweep over the entire database, it is not optimal for random lookups needed, for example, when calculating diffs associated with each commit. For commits and trees, therefore, we also create a key value database where key is SHA1 of the git object and value is the compressed content of the said object. Cache performance is relatively fast: a single thread on Intel E5-2623 is capable of querying of 1M git objects in under 6 seconds, or over 170K git objects per second per thread. This can be multi-threaded and run on multiple hosts, thus reaching any desired speeds with expanded hardware.

Needless to say, with 12B objects occupying over 80TB we need to use parallel processing to do virtually anything. Thankfully, we can use SHA1 itself to split the database into pieces of similar size. We, therefore, split each of the database into 128 slices based on the first seven bits of Git object SHA1. This results in 128 key-offset cache databases for all four types of objects, 128 content databases as flat files for the four types of objects, and 128 key value databases for commits and trees: 128*(4+4+2) databases with each capable of being placed on a separate server to speed up parallel tasks. The individual databases containing content range from 20MB for tags up to over 0.5TB for blobs. The largest individual cache databases are over 2Gb for tree object SHA1s.

Databases are fragile and may get corrupted due to hardware malfunction, internet attack, pollution/loss by unrecoverable operation, etc. To enhance the robustness and reliability and to avoid permanent data loss, we maintain three copies of the databases: two copies on two separate running servers and one copy on a workstation that is not permanently connected to Internet. In the future, we will consider keeping a copy using a commercial cloud service.

Furthermore, due to the size of the data and complexity of the pipeline, some of the objects may have been missed or have been retrieved but are not identical to originals. Techniques to validate the integrity of the data at every stage of the process are necessary.

We therefore, include numerous tests to ensure that only valid data gets propagated to the next stage.

In particular, the errors when listing and extracting objects are captured and the operation is repeated in case a problem occurs. The extracted objects are validated to ensure that they are not corrupt and also to ensure that they are not going to damage the database or the analytics layer. To validate correctness, the object is extracted per git specifications and recreated from scratch. The SHA1 signature is compared to ensure it matches that of the original object. A substantial number of historic objects have issues due to a bug in git that has since been fixed. Furthermore, a much smaller number of objects also had issues that we assume are either caused by problematic implementations of git or problems in operation (zero-size objects that may be occasionally created when git runs out of disk space during a transaction).

Despite the scrubbing and validation efforts, some of the data may still be problematic or missing, therefore a continuous process of checking the database for missing or incorrect data is needed. We plan to add missing object recovery service that identifies missing commits, blobs, and trees, and retrieves and stores them (in case they are still available online).

3.3.5 Update

The process of cloning all GitHub repositories takes an increasing amount of time with the growth in size of existing repositories and the emergence of new ones, given fixed hardware. Currently, to clone all git repositories (over 90M including forks), we estimate the total time to require six hundred single-thread servers running for a week and the result would occupy over 1.5PB of disk space. Fortunately, git objects are immutable and we can leverage that to simplify and speed up the updates. More generally, to get acceptable update times, we use a combination of two approaches:

- Identify new repositories, clone and extract Git objects
- Identify updated repository and retrieve only newly added Git objects

The work flow is illustrated in Fig. 3.3.

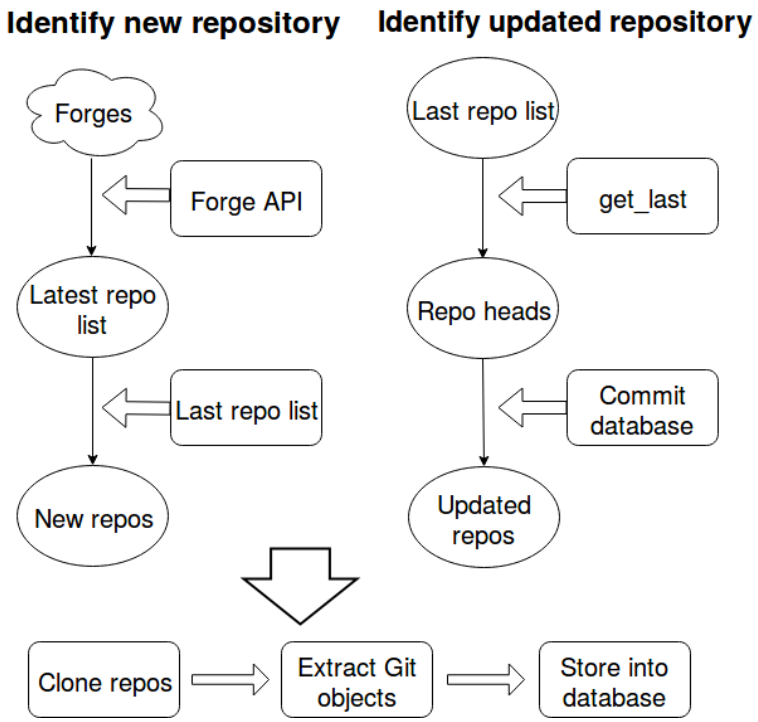

Figure 3.3: Update workflow

In fact, only approximately three million new projects were created and an additional two million updated during Dec, 2018.

Procedures for new repositories

Forge-specific APIs are utilized to obtain the complete list of public repositories as described above. A comparison with prior extract yields new repositories. The list may include renamed repositories and forks. We can exclude forks for GitHub, since it is an attribute returned by GitHub API. Other forges contain fewer repositories, so the forks are not large enough to be a concern.

Procedures for updated repositories

First we need to identify updated repositories from the complete list of repositories. Since we are not sure how GitHub determines the latest update time for a repository, we use a forge-agnostic way of identifying updated repositories. We modified the *libgit2* library so

that we can directly obtain the latest commit of each branch in a Git repository for an arbitrary Git repository URL, without the need to clone the repository. If any of the heads contain a commit that is not already in our database, the repository must have had updates and needs to be obtained.

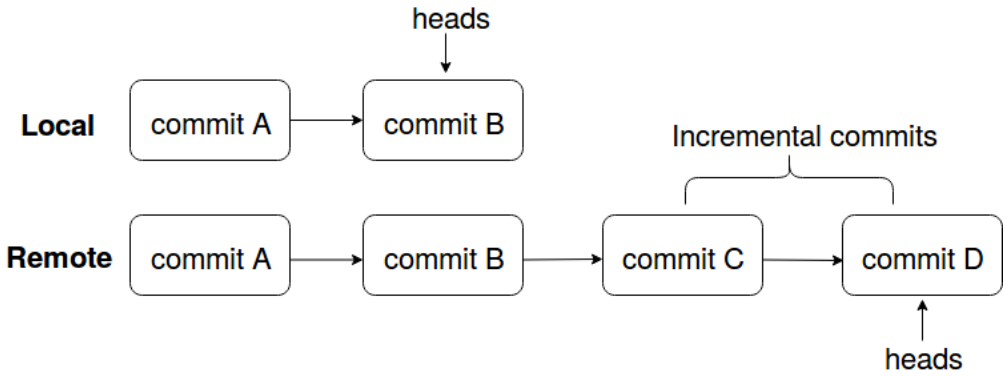

Figure 3.4: Incremental commits

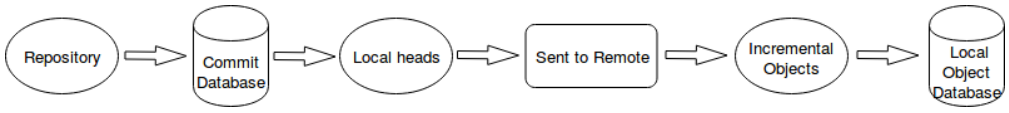

Figure 3.5: Future workflow

We are working on a strategy to reduce the amount of bandwidth needed to do the updates. Instead of cloning an updated repository, we'd like to retrieve only incremental Git objects (see Fig. 3.4) that are generated during the time gap between two consecutive updates. This can be easily done via git fetch for a git repository, but since we do not keep the original git repository and it is time consuming to prepopulate it with git objects, we plan to customize git fetch protocol by inserting additional logic in order to use our database backend that comprises git objects from all repositories. The procedure consists of two steps:

1. Customize git fetch protocol[11] to work without git's native database.

2. Keep track of the heads for each project that we have in our database so that we can identify latest commits to the modified git fetch.

For the second step, the database backend will use the project name as input and provide the list of heads for the project. These heads are then sent to the remote so that the set of latest commits (and related trees/blobs) will be calculated out and transferred back as illustrated in Figure 3.5. By following this strategy, we could drastically speed-up mining incremental Git objects from repositories in each update.

3.3.6 Data Reorganization for Analytics

Objects in Git are organized in a way for fast reconstruction of a repository at each commit/revision. In fact even the seemingly simple operation of identifying what files changed in a commit is computationally intensive. Furthermore, there is no consideration for the projects, files, or authors as first-class objects. This limits the usability of the git object store for research and suggests the need for an alternative data design. Since our objective is to obtain relationships among projects, developers, and files, we have created an alternative database that allows both a rapid lookup of these associations and sweeps through the entire database that make calculations based on such relationships.

Analytic Database

The scale of the desired database limits our choices. For example, a graph database [12] like neo4j would be extremely useful for storing and querying relationships, including transitive relationships. However, it is not capable (at least on the hardware that we have access to) of handling hundred's of billions of relationships that exist within the entire FLOSS. In addition to neo4j, we have experimented with more traditional database choices. We evaluated common relational databases MySQL and PostgreSQL and key value databases or NoSQL [69] databases MongoDB, Redis, and Cassandra. SQL like all centralized

[11] git fetch downloads only new objects from the remote repository

[12] a database that uses graph structures for semantic queries with nodes, edges and properties to represent and store data

databases [1] has limitations handling petabyte datasets [97, 72]. We, threfore, focus on NoSQL databases [83] that are designed for large scale data storage and for massively parallel data processing across a large number of commodity servers [83].

For the specific needs of the cache database and for key value stores for the analytics maps we use a C database library called TokyoCabinet (similar to berkeley_db) using a hash-indexed as described above, to provide approximately ten times faster read query performance than a variety of common key value databases such as MongoDB or Cassandra. Much faster speed and extreme portability lead us to use it instead of more full-featured NoSQL databases.

Maps

Apart for the general requirement to be able to represent global relationships among code, people, and projects, we also consider the basic patterns of data access for several specific research tasks as use cases in order to design a database suitable for accomplishing research tasks within a reasonable time frame. The specific use cases are:

1. Software ecosystem research would need the entire set of repositories belonging to a specific FLOSS sub-ecosystem, e.g., the set of all repositories that use Python language.

2. Developer behavior research would need to identify all projects that a specific developer worked on, the files they authored, and software technologies they used.

3. Code reuse research would need to identify all projects where a specific piece of code occurs and determine how it got there.

To support the first task, a mapping from file names to project names would be necessary. The second task would require author to project, file, and to content of the versions of the file authored by that developer (in order to access the source code and identify what components or libraries were employed). The last task would require a map between blobs (that contain snippets of code) and projects. It would also require a map between blobs and commits in order to identify the time when the specific piece of code was introduced.

We have identified a number of objects and attributes of interest here: projects, commits, blobs, authors, files, and time. The complete set of possible direct maps for an arbitrary

pair is 30. Since author and time are properties of the commit and are not properties of projects, blobs, or files, it makes sense to place commit at the center of this network database. The author-to-file map can then be constructed as a composition of author-to-commit and commit-to-file maps; and author-to-project map can be constructed via author-to-commit and commit-to-project maps. We also need to associate file names with the corresponding blobs since a single commit may create multiple files. Out of the 12 maps[13], only 10 need to be instantiated because commit-to-author and commit-to-time maps are embedded as the properties of the commit object.

In addition to having the commit at the center, for certain tasks we also needed to have a blob-to-file map as well. For example, we want to identify module use in Python language files. First, we need to identify relevant files via suitable extension (e.g., .py), then we can determine all the associated commits via file to commit map. These commits, however, may involve other files and if we use commit to blob map to identify associated blobs, we would get blobs not just for python, but also for all files that were modified in commits that touched at least one python file. The file-to-blob map allows us to reduce the number of blobs that need to be analyzed dramatically.

In addition to these basic maps we create additional maps, such as the author ID to author ID map for IDs that have been established to belong to the same person (see Section 4.3), and project to project maps to adjust for the influence of forking. Project-to-project maps are based on the transitive closure of the links induced between two projects by a shared commit. Explicit forks that can be obtained as a GitHub project property do not generalize to other forges and, even on GitHub, represent only a fraction of all repositories that have been cloned from each other and then developed independently. Project-to-project map also handles instances where repositories exist on multiple forges or when they are renamed.

As with the original data we utilize multiple databases and use compressed files for sweep operations and TokyoCabinet for random lookup. We separate maps into 32 instead of 128 databases we use for the raw objects since maps tend to be much smaller in size than, for example, blobs. For commits and blobs we use the first character of SHA1 for database

[13]bidirectional maps between the commit and five objects/attributes and between file and blob

identification. For authors, files, and projects, we use the first byte of FNV-1a Hash [14]. Both approaches yield quite uniform distribution over bins.

As noted above, the maps from commit to meta data are not difficult to achieve because meta data are part of the content of a commit object. However, git blobs introduced or removed by a commit are not directly related to the commit and need to be calculated by recursively traversing trees of the commit and its parent(s). A Git commit represents the repository at the-state-of-world and contains all the trees (folders) and blobs (files). To calculate the difference between a commit and its parent commit, i.e., the new blobs, we start individually from the root tree that is in the commit object, traverse over each subtree and extract each blob. By comparing two sets of blobs of each commit, we obtain the new blobs for the child commit. This step requires substantial computational resources, but the map from the commit to the blobs authored in a commit is used in numerous research scenarios and, therefore, is necessary. On average, it takes approximately one minute to obtain changed files and blobs for 10K commits in a single thread. With 1.5B commits, the overall time for a single thread would take 104 days, but it needs to be done only on approximately 20-40M new commits generated each month.

3.4 Architecture for research workflows

To make WoC more easily usable in a wide variety of research scenarios, we have designed an architecture to help simplify, support, and evaluate the implementation of research tasks. This section describes that architecture, along with critical performance benchmarks to inform the users on the computational tasks for alternative implementations.

3.4.1 Architecture

The research workflow architecture is illustrated in Figure 3.6. The figure shows the application layer, built on top of the three lower layers:

Application Layer: This layer is where the research tasks are implemented by use of WoC. We provide a library of applications to illustrates various types of research analyses that can

[14] http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-1a

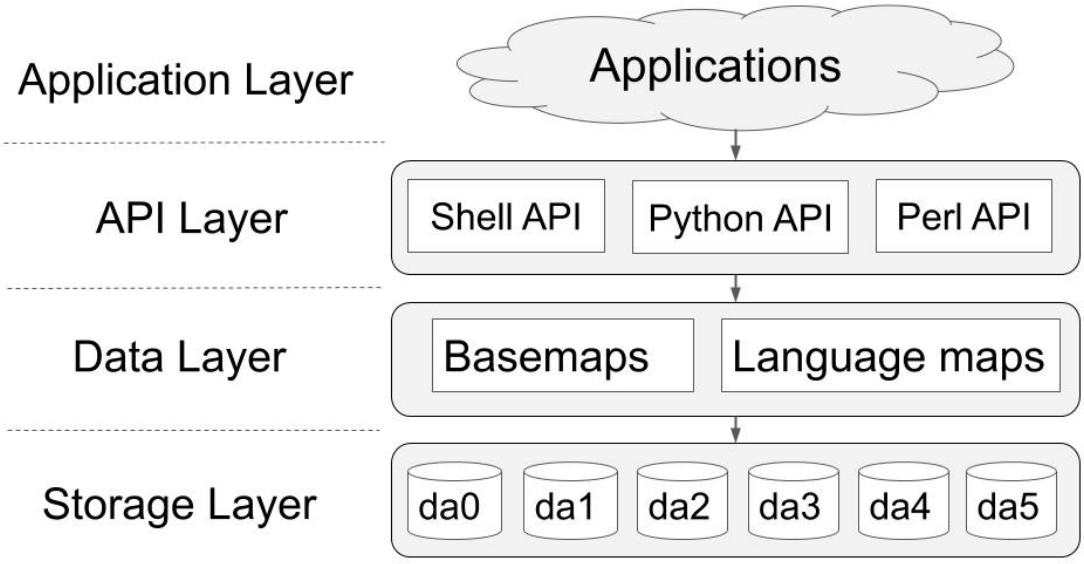

Figure 3.6: Architecture of the Tool

be implemented using WoC.

API Layer: The applications may use Shell or Python API, or may reuse or modify Perl files (used to support Shell API) to access and process the WoC data. A more detailed description of the Shell and Python APIs can be found in Section 3.4.2.

Data Layer:

As described above, to be able to identify the relationships rapidly we constructed several types of relationships (or basemaps) that cross-reference the git objects and other properties. In particular, we treat *project, commit, blob, author, file name* as the first class objects and map them to their properties (e.g., time, parent commit, head commit, child commit, etc.). In addition to these basemaps, we also construct technical dependencies that are derived from importing external dependencies for several languages (Language Maps). These dependencies are calculated based on each version of a file. The data is described in more detail in Section 3.4.3.

Storage Layer All the data are hosted on six servers, which are connected to each other through NFS (network file system). Users can login to any of the servers (da0 to da5) and and run their applications on multiple servers.

3.4.2 API

We support three primary APIs for WoC users to access the dataset: Shell, Python, and Perl. Presently, running an application requires being logged in to one of the hosting servers.

Shell API

For the lowest level access we provide Shell API that is modeled after core philosophy of Unix[15]: have a set of specialized commands that are connected in a workflow through their standard input and output and via creation of files, E.g., according to Doug McIlroy: "Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features". The entire application workflow can be built using exclusively shell and standard Unix utilities such as 'join', 'sort', 'cut', 'uniq', 'sed', and 'wc' with added specialized commands to extract data from key-value databases. The key information for this API is the knowledge of how to use shell and standard Unix command and the description of the databases. To enable this approach we also provide all databases as key-sorted (and compressed) text files that can be used with 'grep', 'join', or 'sort' to produce any desired queries. We also add a random lookup operation `getValue mapname` to access values of a key object in the provided `mapname`. In addition, we add the command `showCnt type` to access the content of each git object given in the standard input where `type` is one of `tag, tree, commit, blob`. A few examples are listed below:

- Checking the content of a Git object given a SHA:

```
1  # (on da3) e.g., show a commit SHA's content:
2  echo e4af89166a17785c1d741b8b1d5775f3223f510f | showCnt commit
3  # Output Formatting:
4  # Commit SHA;Tree SHA;Parent Commit SHA;Author;Committer;Author Time;
       Commit Time
5  e4af89166a17785c1d741b8b1d5775f3223f510f;
       f1b66dcca490b5c4455af319bc961a34f69c72c2;
       c19ff598808b181f1ab2383ff0214520cb3ec659;Audris Mockus <audris@utk.
       edu>;Audris Mockus <audris@utk.edu>;1410029988 -0400;1410029988 -0400
```

[15] https://en.wikipedia.org/wiki/Unix_philosophy

- Given an object, check its related objects:

```
# (on da3) e.g., show the names of the projects associated with a given
    commit SHA:
# ‘‘getValue" command takes a database name as an argument and keys
    presented as standard input and produces key-value pairs as output.
echo e4af89166a17785c1d741b8b1d5775f3223f510f | getValue /da0_data/
    basemaps/c2pFullP
# Output Formatting: Commit SHA;ProjectNames
e4af89166a17785c1d741b8b1d5775f3223f510f;W4D3_news;chumekaboom_news;
    fdac15_news;fdac_syllabus;igorwiese_syllabus;jaredmichaelsmith_news;
    jking018_news;milanjpatel_news;rroper1_news;tapjdey_news;taurytang_
    syllabus;tennisjohn21_news
```

Python API

At the top level of abstraction, we provide Python API via package **oscar**[16] that implements the key notions of author, file, project, commit, blob, and tree as the corresponding classes. The enumeration below describes Python classes that were created by wrapping up data objects 3.1. Each of the classes has a couple of methods attached to access corresponding properties. For the methods that contain slash(/), the method before slash returns actual data in string, while the one after return a generator of corresponding python instances. E.g. `Author.commit_shas()` returns a list of the SHAs of commits that the person authored; `Author.commits()` returns a generator of Commit objects built from those SHAs.

1. Author(‘...’) - initialized with a combination of name and email, e.g. “Albert Krawczyk <pro-logic@optusnet.com.au>”

    - .commit_shas/commits - all commits by this author
    - .project_names - all projects this author has committed to

[16] https://github.com/ssc-oscar/oscar.py

2. Blob('...') - initialized with SHA of blob
    - .commit_shas/commits - commits creating or modifying (but not removing) this blob
3. Commit('...') - initialized with SHA of commit
    - .blob_shas/blobs - all blobs in the commit
    - .child_shas/children - the commit that follows this commit
    - .changed_file_names/files_changed
    - .parent_shas/parents - the commit that this commit follows
    - .project_names/projects - projects this commit appears in
4. Commit_info('...') - initialized like Commit()
    - .head
    - .time_author - the commit time and its author
5. File('...') - initialized with a path, starting from a commit root tree. This represents a filename, regardless of content or repository; e.g. File(".gitignore") represents all .gitignore files in all repositories.
    - .commit_shas/commits - All commits that include a file with this name
6. Project('...') - initialized with project name/URI
    - .author_names - all author names in this project
    - .commit_shas/commits - all commits in this project

Perl APIs

While the Python API provides high level of abstraction, it is not very computationally efficient. In order to provide an intermediate level of efficiency between that of Python and Shell APIs, we also provide a way to implement applications or their components in Perl language. For example, the shell commands `getValue` and `showCnt` are both implemented

in Perl. The Perl API instead of creating classes of objects as in Python, it handles the maps directly. To support writing WoC workflows in Perl we provide a variety of utility functions in package 'WoC.pm.' We also have, over the course of evolving WoC, created a number of applications that can be used as templates and modified by the users for their needs. For example, we can parse the content of the commit to obtain its tree, parent commit, author, and time:

```
use WoC;
my ($tree, $parent, $authName, $authEmail) = ("","","","");
my ($pre, @rest) = split(/\n\n/, $code, -1);
for my $l (split(/\n/, $pre, -1)){
  $tree = $1 if ($l =~ m/^tree (.*)$/);
  $parent .= ":$1" if ($l =~ m/^parent (.*)$/);
  ($authName, $authEmail) = gitSignatureParse($1) if ($l =~ m/^author (.*)$/);
}
($auth, $ta) = ($1, $2) if ($auth =~m/^(.*)\s(-?[0-9]+\s+[\+\-]*\d+)$/);
$parent =~ s/^:// if defined $parent;
```

We also have examples on how to parse, for example, a python source code to obtain the dependencies defined by the import statements (a segment is shown below):

```
for my $l (split(/\n/, $code, -1)){
 if ($l =~ m/^\s*import\s+(.*)/) {
  my $rest = $1;
  $rest =~ s/\s+as\s+.*//;
  my @mds = $rest =~ m/(\w[\w.]*[\,\s]*)*/;
  for my $m (@mds) { $matches{$m}++ if defined $m};
 }
 if ($l =~ m/^\s*from\s+(\w[\w.]*)\s+import\s+(\w*)/) {
  if ($2 ne ""){ $matches{"$1.$2"} = 1; }
  else{ $matches{$1} = 1; }
 }
}
```

For more detail please refer to the tutorial page of our repository[17].

3.4.3 Description of the WoC Data

We use abbreviated object names for WoC data and basemaps as shown in Table 3.1. As noted above, types of basemaps are created to represent relationships among these objects, which are illustrated in Figure 3.7. Notice that some maps are missing in Figure 3.7, because initially we built maps with commit being the core, and other maps were built as certain research tasks the users were attempting to do would benefit from them. The basemaps are stored in TokyoCabinet databases for random queries and key-sorted compressed text files of these basemaps are also created to enable quick sweeps over the whole dataset and to enable the shell API.

In addition to the basemaps, programming language based maps are created to enable language oriented analytic and applications. These contain mappings that list repositories, and the modules they depended on, at a given UNIX timestamp under a specific commit. The format of each entry in these maps are like the following, where `module1;module2;...` represent the modules that repository depended on at the time of that commit:

```
commit;repository_name;timestamp;author;blob;module1;module2;...
```

So far, 12 maps are ready including C, C#, Java, JavaScript, Python, R, Rust, Go, Swift, Scala, and Fortran. It is likely that more language maps will be added in the future.

Table 3.1: Objects Terminology

Object Abbreviation	Annotation	Entity Type
a	author	string
b	blob	SHA
c	commit	SHA
f	file name	string
p	project	string

[17]https://bitbucket.org/swsc/lookup/src/master/
[18]'File' in this figure refers to 'File name'

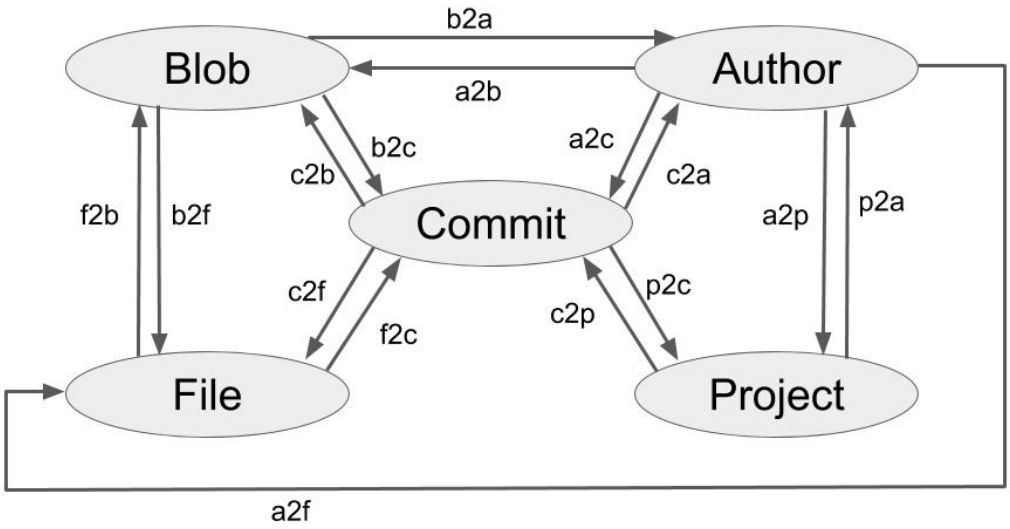

Figure 3.7: Maps between primary objects[18](Basemaps)

3.4.4 Performance Benchmark

The anticipated workflow of a specific research task involves a set of queries that proceed from selecting an initial sample of interest such as a set of files related to a specific language, a set of projects or authors with certain properties or other collection. This is typically followed up by one or more network operation such as identifying blobs associated with the selected files, projects associated with the initial set of developers and so on. These tasks can typically be implemented in numerous ways, each leading to different computer memory, disk IO, and computational overheads. To help users decide upon the the best way to proceed and, more generally, to gauge the time needed for their desired workflow, we set up experiments to test our WoC infrastructure performance on such queries. Our existing basemaps should meet users' need in most cases by a query of a single map (e.g. author to commit). However, in cases where a map is not ready (e.g. file to project in Figure 3.7), users might need to combine/join two or more maps to achieve their goal. We, therefore, tested the performance of both single map queries and combined map queries, and present the results below.

Since the file[19] to project map is not pre-computed, we can start from the file to commit map to test single map query performance and then join the results with the commit to project map to test the combined map query. We randomly selected 100, 1K, 10K, 100K, and 1M file names from our dataset, and used the Python and Shell APIs without any other task being run on the server to find the corresponding commits in which the files were modified and the projects those commits belong to. We collected the time it took to run each test and show them in Figure 3.8 for the single map queries, and Figure 3.9 for the combined map queries.

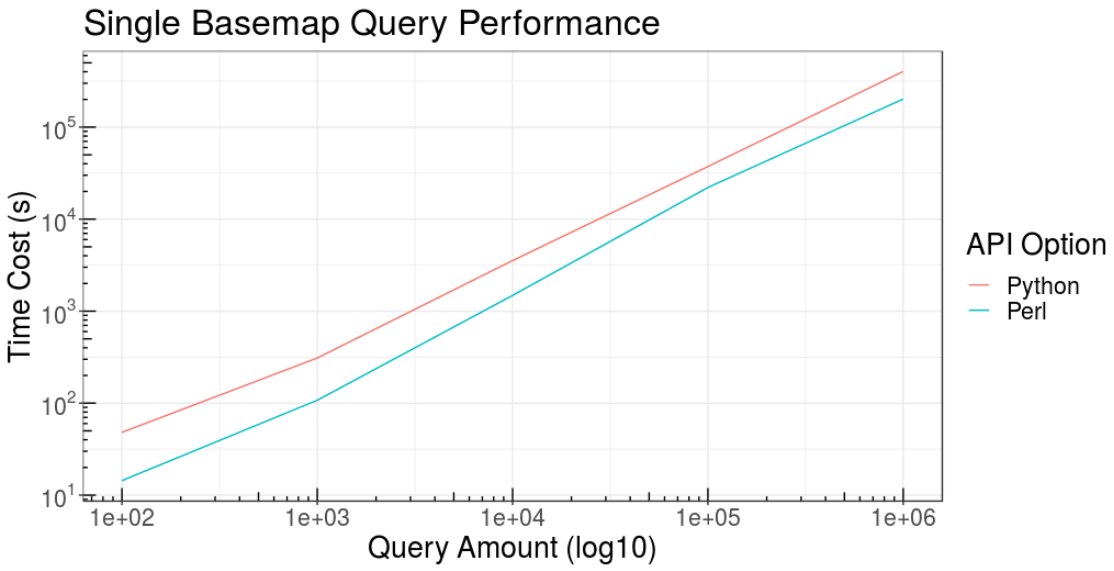

Figure 3.8: Single Map Query Performance

From Figures 3.8 and 3.9, we see that the run time increases linearly as the task size increased, highlighting the scalability of the WoC infrastructure. We also found Shell API to be three to four times faster than Python API (Figure 3.8 and right part of Figure 3.9), for the same query. One hypothesis is the interpreted nature of Python. Specifically, the data access parts of Shell API are implemented in Perl. While Perl is an interpreted language just

[19]By file, we refer to the file name (including folder path) in the rest of our paper.

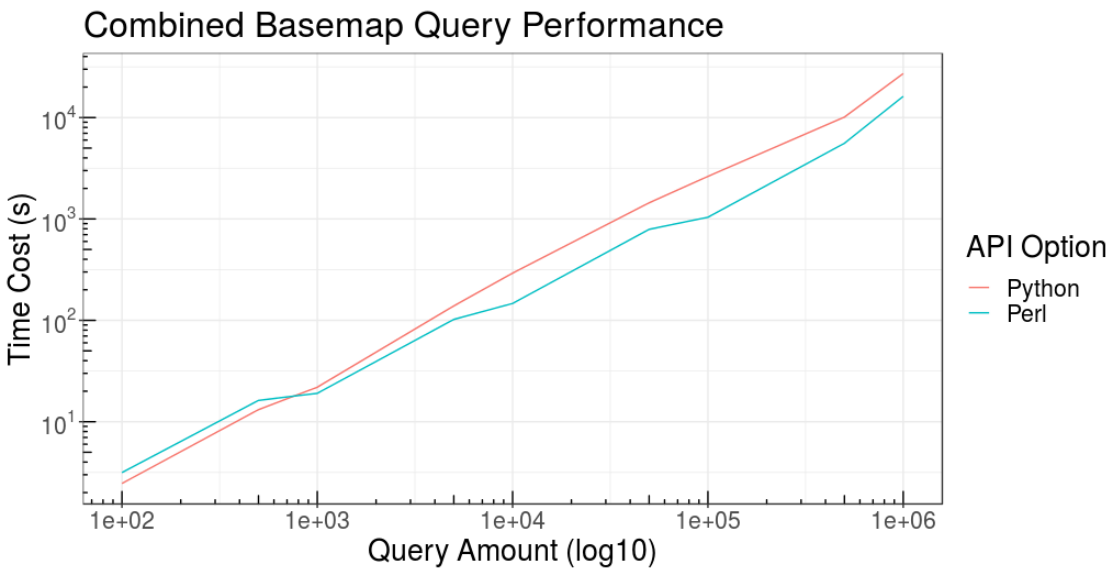

Figure 3.9: Combined Maps Query Performance

as Python, many of the functions are implemented natively in C language, while in Python more performance-critical code is interpreted.

It is worth noting that the x-axis on Figure 3.9 represents the number of queries, which in this scenario is the sum of the number of file to commit queries and the number of commit to project queries.

We tested the performance of the tool for 100 to 1M queries. If a research workflow involves the initial sample of objects for a very large part of the WoC database, we recommend leveraging the database in the form of compressed text for key-value basemaps instead, because as the number of random access queries increases, it exceeds the time it takes to sweep the entire database using efficient shell commands such as grep. In fact, a single sweep of the file to commit compressed data only takes 38 hours while 1M queries of the file to commit basemap takes 56 hours using Shell API.

3.5 Future work

To have an impact on research practice, the WoC prototype needs to be exposed via reliable services that help with research and do not overwhelm the platform. Currently, we only

have Python and Perl API available. However, more languages will be supported in the future. Comparatively small pre-extracted relations will be stored into relational database to extend our accessibility to users who are used to SQL. WoC should also accommodate additional data and computational procedures needed for discovering, correcting, cleaning, augmenting, and modeling the underlying data. Processing hundreds of terabytes of data on powerful clusters may be out of reach for most research groups. Therefore, to accommodate massive queries WoC would require more powerful hardware. Such hardware can be obtained from cloud vendors, but the costs of hosting and analyzing data on these platforms might be high. An alternative might be a few high-throughput services that work on the hardware we currently employ.

The differentiating features of WoC are the completeness of the collection and access to global relationships. Specifically, two basic services would be difficult to replicate outside WoC, yet be capable of high throughput on the limited hardware. First, a reporting service that considers prevalence of certain features, such as languages, tools, and other technologies as well as the information about contributors might provide services akin to those provided by a population census. The second basic service would focus on identifying all entities linked to a specific entity, such as files modified by a developer, all repositories containing a specific code, or all files that use a specific module or technology. These two capabilities, in conjunction with MSR technology already in use, would provide both, population-level data and complete links within entire FLOSS ecosystem. It would then be up to researchers to retrieve additional data on individual projects based on the stratified samples from the first service or derived from the relationships obtained from the second service.

3.6 Limitations

We tried to make the assumptions and rationale for specific decisions clear within each section but it is important to reiterate at least some of the limitations. Despite a large size (the collection contains over 1.45B commits), there is no guarantee it closely approximates the entirety of public version control systems as the project discovery procedure is only an approximation. Our focus on git (due to the simplified global representation) excludes older

version control systems that have not been converted to git yet. We regularly identify issues with data being incomplete due to collection, cleaning, or processing and we are working on an approach to continuously validate and correct it. The particular design decisions were focused on the particular computing capabilities that were available to us at the time and could/should be revisited as the prototype evolves. The entirety of research tasks that WoC provides is not exhausted by the few examples we have investigated and certain tasks may require different solutions. We do, however, think that the micro-services approach allows for simpler addition/extension/replacement of components as needs or opportunities arise than would be possible with a more monolithic architecture.

How to reliably clean, correct, integrate, and augment the collected data so that the resulting analyses accurately reflect the modeled phenomena is a concern. To ensure the performance of the analytics layer certain objects are filtered from it. For example, some of the public repositories are created to test the performance/capabilities of git and contain many millions of files/blobs in a single commit. Such commits are excluded from the analytics layer to speed-up the commit-to-file and commit-to-blob maps. The nature of the data may also create performance problems. For example, the most common blob is an empty file. Mapping such blobs to all commits that create them or to all files does not make sense, since there are millions of commits that have created empty files. These performance-related modifications may affect some arguably superficial analyses, e.g., what are the commits with the largest number of files? We explicitly highlight these modification in the WoC code to minimize potential confusion.

Reproducibility may pose an issue in a constantly updated database. Since git objects are added incrementally and order in which they are stored is preserved, we can reconstruct any past version of the object store. For the analytic layer, which depends on the set of git objects available at the time, we create versions, where each of the maps described above is tagged with a version identifying the state of git object store. Preserving these past versions ensures reproducibility of the results obtained from them.

3.7 Conclusions

We introduce WoC: a prototype of an updatable and expandable infrastructure to support research and tools that rely on version control data from the entirety of open source projects. We discuss how we address some of the data scale and quality challenges related to data discovery, retrieval, and storage. We enable wide data access to collected data source by providing a tool built on top of the infrastructure, which scales well with completion to query in linear time. Furthermore, we implement ways to make this large dataset usable for a number of research tasks by doing targeted data correction and augmentation and by creating data structures derived from the raw data that permit accomplishing these research tasks quickly, despite the vastness of the underlying data. In summary, WoC can provide support for diverse research tasks that would be otherwise out of reach for most researchers. Its focus on global properties of all public source code will enable research that could not be previously done and help to address highly relevant challenges of open source ecosystem sustainability and of risks posed by this global software supply chain. Transforming the WoC prototype into a widely accessible platform is, therefore, our immediate priority.

All source codes can be found in a public repository.[20]

[20] https://github.com/ssc-oscar/Analytics

Chapter 4

New Insights and Analysis from SSC[1]

4.1 Overview

In this chapter, we evaluate SSC by investigating the new insights and analysis derived. More specifically, we conducted two types of studies on top of SSC using WoC infrastructure to verify if our infrastructure is capable of supporting research tasks in SSC, and to see what new insights and analysis can be implemented in SSC.

To achieve this, we implemented several basic research tasks that require the entirety of FLOSS data as a part of the investigation. Furthermore, we recruited three researchers external to our group to either conduct investigations of their own utilizing WoC and SSC, or to provide us with their research problems that can only be solved by using WoC. We found that the combination of SSC and WoC infrastructure can enable and efficiently support various domain specific and cross-ecosystem researches. Furthermore, we plan to move our services to cloud platform to attract more users, and many more promising insights and analysis would be expected in the future.

[1]This chapter, in part, is a reprint of the material as it appears in 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 143-154, titled "*World of code: an infrastructure for mining the universe of open source VCS data*" (2019), and also in 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 458-459, titled "*Constructing supply chains in open source software*" (2018). Authors: **Yuxing Ma** and Audris Mockus. The dissertation author was the primary investigator and author of these two papers. Copyrights of both papers are held by **Yuxing Ma** and Audris Mockus.

Below we report both the experiences and results from these experiments. To providing further assistance to WoC users, we exemplify the usage of WoC by actually implementing a specific analysis through API calls. In the end, we describe a couple of ways to operationalize SSC networks and present a number of network graphs for illustration.

4.2 Use of programming languages

Language popularity may influence developers decisions as it may affect the market for their software as well as their job prospects. For example: What language-specific API should developer provide for their component? What language should the developer use to implement their product?

To plot, for example, Java language use trend we use WoC to identify all files with .java extension. Then, via file-to-commit map, obtain the complete set of commits authoring these files. Commit dates are used to plot the time trends of language-specific commits, authors (property of a commit), projects (via commit to project map) and, if desired, lines of code changed. The entire process is highly parallelizable since each map is separated into 32 instances and can be processed independently. The entire calculation, while not interactive on our hardware, can be performed in tens of minutes. For illustration, we show the ratio of the number of commits over the number of developers (a measure of productivity) each year in Fig. 4.1. The ratio decreases for most languages, perhaps because as a language becomes more popular, the less experienced contributors join and lower the average productivity.

4.3 Correcting Developer Identity Errors

One of the particularly troubling data quality issues with version control systems is developer name disambiguation. Often, names and emails of developers are missing, incomplete, misspelled or duplicate [43, 13]. Performance of any disambiguation algorithm depends on the distribution of the actual misspellings in the underlying data. In order to design and evaluate corrective algorithms, it is important to study a large collection of actual data and unearth patterns of irregularities that compromise data quality. WoC contains a nearly

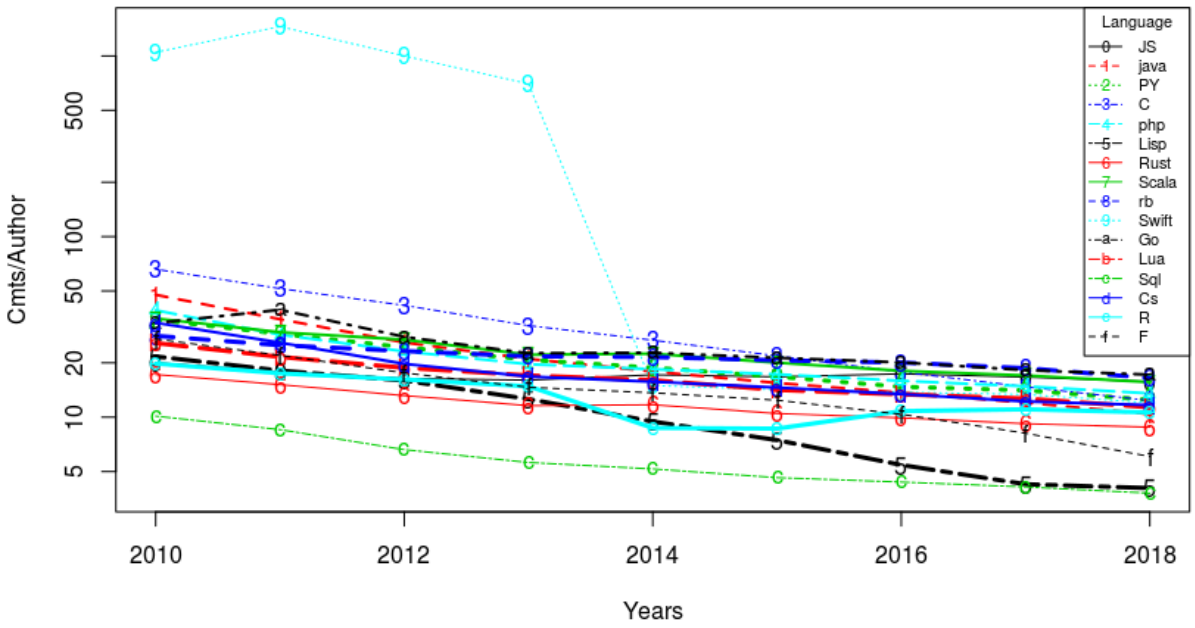

Figure 4.1: Productivity by Language

complete collection of git author ids (name and email combinations) and is, thus, more representative of such irregularities than any specific project.

To obtain author IDs we use author-to-commit map containing roughly 30 million distinct author IDs. Common error patterns include organizational ids and emails (Mozilla, Linux, Google etc), names of tools and projects (OpenStack, Jenkins, Travis CI), roles such as (admin, guest, root etc.) and words that preserve anonymity (student, nobody, anonymous etc) as a part of their credentials. We also found a large number developer IDs to be misspelled.

Traditional identity correction approaches rely on the misspelling patterns of author ID (the full name and email) [13, 117, 118]. With WoC data, we can enhance the traditional string matching with behavioural comparison, by creating similarity measures between author IDs using files modified by developers, time patterns of commits, and writing styles in commit messages. For illustration — two author IDs that modify a similar set of files may suggest that these IDs belong to the same developer. To implement file-based similarity, we used author to commit and commit to file maps to obtain the set of files modified by a single author ID. Then file-to-commit and commit-to-author maps were used to calculate similarity using weighted Jaccard measure. Commit message text was used to fit a Doc2Vec [68] model to associate each author ID with their writing style. Traditional and behavioural similarities were used to train highly accurate machine-learning model [3].

This experiment demonstrates the utility of WoC data for designing tools to solve common and vexing data quality problems when constructing developer networks.

It is also an example of how WoC can be enhanced by incorporating such techniques and providing corrected data to researchers.

4.4 Cross-ecosystem comparison studies

A second research group used the database to gather comparative statistics about different software ecosystems. The purpose was to supplement other comparative data about those ecosystems in support of a study of how ecosystem tools and practices influence development behavior. The ecosystem study involved a survey, interviews, and data mining over 18

ecosystems whose repositories listed more than 1.2M packages. Some questions about ecosystem practices could be mined from metadata available elsewhere; for example detailed information about dependencies, release frequency, and version numbering practices can be easily extracted from libraries.io[2]. However deeper questions about project content would have been out of reach without WoC; independently building the mechanism to collect all of these projects, building a database of blobs, files, projects, and authors, and comparing them using various metrics would have been too much work for too little gain without the availability of this research platform.

4.4.1 File cloning across ecosystems

One such statistic is rate of file cloning. It was theorized that in ecosystems with more flexible support for dependencies and a tolerance for the risk of breaking changes, developers would be more likely to use dependency management tools to make use of functionality from other projects, rather than copying those files in directly; hence in such ecosystems we should find relatively few commits adding a blob that already exists in any other project available through the ecosystem's dependency management system.

Using WoC, this analysis was straightforwardly accomplished by joining blob-to-commit and commit-to-project mappings, filtering for blobs that appeared in multiple projects, and identifying pairs with one commit in the time frame, and at least one older commit. Such blobs were discarded when the files were very small (since these often turned out to be empty or trivial files duplicated by chance or by tools) resulting in a set of duplicates that, on visual inspection of a sample, did appear to represent genuine examples of reuse-by-cloning.

Contrary to our expectations, the ecosystem with the most propensity for cloning was the one with the modern and flexible dependency system: npm. Despite the strengths of npm's dependency management system, there is a strong tradition of copying dependencies like jQuery into projects rather than letting npm retrieve them. Figure 4.2 summarizes the findings for a selection of ecosystems.

[2]https://libraries.io/

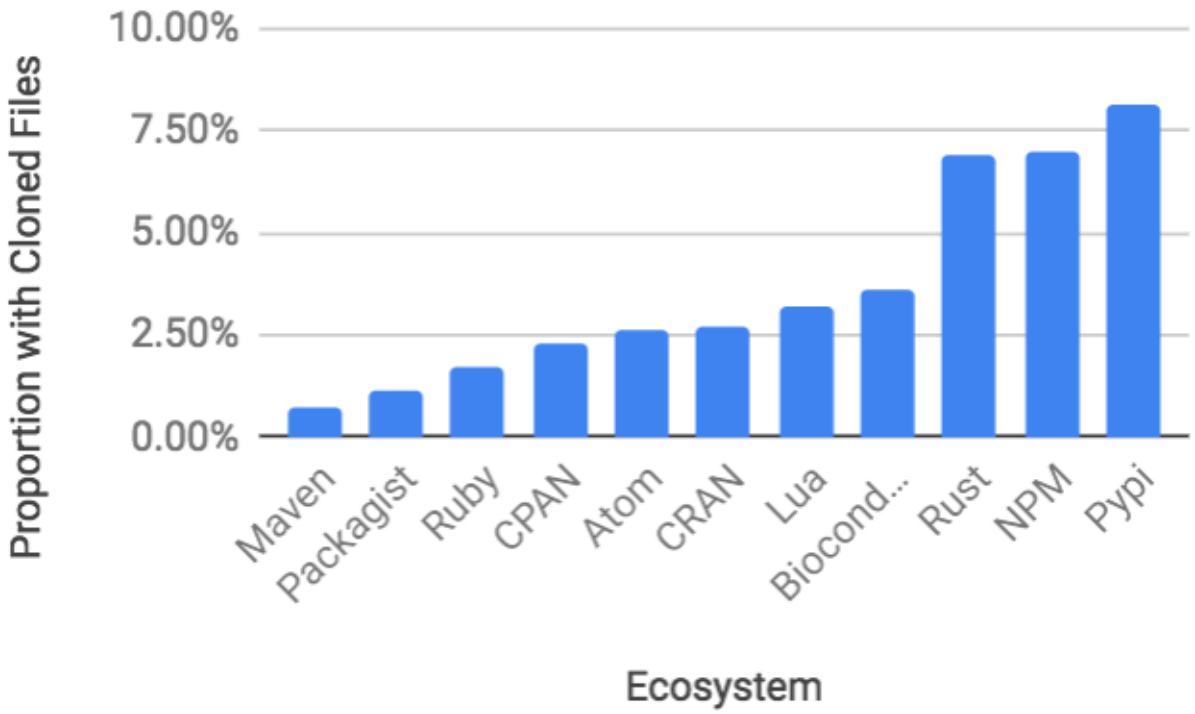

Figure 4.2: Proportion of repository packages that added at least one cloned code file over 1kb in 2016.

4.4.2 Developer migration across ecosystems

Another metric of interest was developer overlap between ecosystems. Our ecosystem comparison had included a survey of values and practices in the 18 ecosystems of interest, and we hypothesized that ecosystems might be similar if many developers were actually working in both ecosystems, or had migrated from one to the other.

This question was answered by joining author-to-commit and commit-to-project data for the 1.2M projects in our study, and relying on the identity matching technique described in Sec 4.3.

Over all pairs of ecosystems, we found a sizable correlation between similarity of average responses on ecosystem *practice* questions (things like frequency of updating, collaboration with other projects, means of finding out about breaking changes), and overlap in committers to those ecosystems (Spearman $\rho = 0.341, p < .00001, n = 16$ ecosystems). Interestingly, perceived *values* of the ecosystem (such as a preference for stability, innovation, or replicability) do *not* seem to align with developer overlap ($\rho = -0.05, p = 0.44$). While more research is needed, we hypothesize that developers may carry practices over from other languages and platforms they have used in the past, in a sometimes cargo-cult-like way, despite recognizing that a new ecosystem is designed to accomplish different ends.

In our very large-scale, wide-ranging study, these questions of developer migration and cloning were of great interest, but would likely have been too expensive to pursue alongside other lower-hanging fruit, absent WoC's prepared set of precomputed maps between files, blobs, authors, projects, and timestamps. The dataset with its analytical maps was not designed with these particular ecosystem comparison in mind, but its design happens to make such ecosystem questions relatively easy to answer.

4.5 Python ecosystem analysis

An external researcher wanted to use WoC to investigate open source sustainability by identifying source code repositories for packages in PyPI ecosystem and to measure package usage directly. While over 90% of npm packages provide repository URLs, less than 65% of Python Package Index (PyPI) packages do.

The researcher obtained all packages from PyPi and calculated blob SHA1s for *setup.py* file of the first PyPI releases of each package. We filter out resulting 101584 blobs to exclude empty or uninformative blobs (blobs that appear in more than one commit using blob-to-commit map). The 54218 informative blobs are then mapped to 54062 unique commits and commits to 51924 unique projects (adjusted for forking as described in Section 3.3.6). Repositories were recovered for 96% of the 54218 original packages in approximately 20 minutes of computation. To ensure that these repositories are, in fact, used to version control corresponding packages, they can be matched via additional blobs for *setup.py* and other files obtained from PyPi for that package.

Another problem being solved by this researcher was identifying which of the seemingly abandoned projects may be "feature complete," i.e. already have the intended scope and do not require further maintenance [21]. Feature complete projects should be widely used in contrast to abandoned projects. Proxies of project usage, e.g., GitHub stars or forks can be used to identify such projects [21]. WoC, however, lets us measure the extent of use directly. As described in Section 4.2, all commits modifying Python files are identified (file-to-commit map) and the resulting commits are mapped to projects (commit-to-project map).

Blobs associated with these commits (commit-to-blob map) are then used to extract imports from these files.

The entire procedure could be completed in approximately four hours using the parallelism of the analytic maps (32 databases) and blob content maps (128 databases).

The reported usage was compared to project development activity, i.e the total number of adoptions versus the total number of commits. In some cases, usage was not accurately reflected in the number of commits. Common examples are packages providing console scripts and CMS-like projects. In the former case, packages are not reused in programmatic code and thus don't get into statistics. In the latter case, website builders often do not publish their code and thus such usage remains unobserved. Therefore, while the number of public reuses provides some extra information about package use, it should be adjusted for package type.

4.6 Repository filtering tool

Millions of repositories on GitHub and other forges also include projects that are completely unrelated to software development. GitHub is widely used for education and other tasks such as backing up text files, images, or other data. Researchers investigating education may need to focus on tutorials, while other researchers may need a sample of actual software development projects. Furthermore, a way to select specific subsets of software development projects in order to conduct, for example, "natural experiments" would also be highly beneficial. WoC can support such project segmentation tasks in a variety of ways. An external education researcher wanted to understand the impact of self-administered programming tutorials. To do that, WoC was used to identify developers who participated in tutorials by searching the set of projects in WoC via keywords related to education: "assignment", "course", "homework", "class", "lesson", "tutorial", "syllabus", "mooc", "udacity". The search yields over 1M projects. While it is only a small fraction of all projects in WoC but it represents a large sample in absolute terms. Further filtering was needed to find developers who also worked on actual software projects to measure the impact of self-administered tutorials. The project-to-commit map identified 605K users of tutorials and, when these users were mapped to all projects they participated in, we determine that only half of them contribute to non-tutorial projects. These 300K individuals are potential subjects of tutorial-impact study. Further information (such as their commit activity and project participation) can be obtained from WoC and combined other data, be used in this research. WoC can be extended with other approaches to segment projects[3]. For example, identification of projects with sound software engineering practices [84] relies on a combination of factors easily obtainable in WoC, such as history, license, and unit tests.

4.7 Other Applications

A number of research publications have utilized the WoC database, including:

[3]Section 4.3 shows how WoC can also be used to improve them

- The relationship between dependencies of NPM packages, collected using the WoC infrastructure, and their popularity was discussed in [29].
- The effort contribution and demand patterns of the contributors to the NPM ecosystem was discussed in [28].

4.8 Archetypical Usage of WoC

To increase the utility of this project to a wider research community, we would like to prioritize easy access to the World of Code to other interested parties. In this section, we provide a brief introduction and an overview of the World of Code and how to use it. Moreover, there are some resources already in place that were designed to assist in this process, which can be found in a public repository[4].

After describing WoC and its applications, in this section we demonstrate how to actually use WoC to implement a specific analysis. A couple of approaches presented here leverage the WoC tool to implement the Java language trend analysis, as described in Section 4.2.

1. Identify Java files based on '.java' extension, collect commits that changed these files, and deduplicate the commits. Now we have all commits where one or more java files were created/modified. The source code of the custom `lsort` command is presented in Appendix C.

```
#start from basemap dump(‘‘file to commit" dump, P represents version),
for i in {0..31}; do zcat /da0_data/basemaps/gz/f2cFullP.$i.s | awk -F ";
    " "/.java;/{print $2 }" done | ~audris/bin/lsort 10G -u | gzip >
    JavaCommits.gz
```

2. For each commit in commit collection, we can use either Python or Perl API to find related author and commit time, and then calculate the number of authors and commits by year – the trend

```
# Using Python
```

[4]https://github.com/ssc-oscar/lookup

```
    import gzip
    from datetime import datetime
    from collections import defaultdict

    year2commit_count = {}
    year2commit_count = defaultdict(lambda: 0, year2commit_count)
    year2author_count = defaultdict(set)
    java_commits = gzip.open("JavaCommits.gz", "r")
    for commit in java_commits:
        time, author = Commit_info(commit).time_author
        year = datetime.fromtimestamp(int(time)).year
        year2commit_count[year] += 1
        year2author_count[year].add(author)
    print(year2commit_count)
    for year, authors in year2author_count.items():
        print("Year: "+ str(year) + "# of authors: " + str(len(authors)))
```

```
    # Using Perl
    # we can run /da3_data/lookup/showCmt.perl on every commit and extract
        author and time info from there
    # A simpler way is to utilize basemap c2taFullP.{0..31}.tch (i.e., the
        basemap from commit to author and commit time) by calling Cmt2ATShow.
        perl (see source code in Appendix B)
    zcat JavaCommits.gz | perl Cmt2ATShow.perl | gzip > JavaYearAuthor.gz
    # count records for each year, we get the number of commits by year. E.g
        ., for year 2014:
    zcat JavaYearAuthor.gz | grep "^2014;" | wc -l
    # after deduplication, count records for each year and we get the number
        of authors by year. E.g., for year 2014:
    zcat JavaYearAuthor.gz | sort -u | grep "^2014;" | wc -l
```

In fact, directly using language maps is more efficient when implementing this analysis, since language specific information have already been extracted from base maps and stored as language maps for use.

```
# Alternatively, we use language map: c2bPtaPkgPjava, which consists of commit, blob, project name, time, author, etc.
zcat c2bPtaPkgPjava.{0..31}.gz | cut -d\; -f3,4 | gzip > JavaYearAuthor.gz
# now follow the similar approach in Perl example shown above to get the final result
```

4.9 Operationalize SSC Networks

To best of our knowledge, no prior research have been conducted on constructing and analyzing SSC, nor approach has been raised to operationalize SSC. Therefore, we propose our approach to operationalize SSC.

In software domain, especially in FLOSS, the basic product is source code. Source code is produced by a group of software developer in repositories and purchased and used by other developers to fulfill a need in their projects freely under particular licenses[94]. Based on granularity size, we divide source code into different categories: the smallest source code is a piece of code snippet that occupies one part of a file; the moderate source code is a file that usually serves as a component in implementing specific functionality; the large source code is a project or package that provides a complete solution to meet a pre-defined need. We refer the above three categories of source code as code snippet, file, project in this thesis.

The second type of entity in SSC, the producer, is software developer. In FLOSS, especially for those hosted on open-source platforms, global software developers join together and each of them devote his/her wisdom and contribute code snippet to a software project. During producing process, software developers may incorporate source code from other projects, e.g., a developer may reuse some of his source code from one of his own projects or others projects, hence, product flows from one producer to another, from one project to another one.

Another type of entity that is hidden inside source code is tacit knowledge[90]. It refers to the kind of knowledge that is difficult to transfer to another person by means of writing it down or verbalizing it. When new software developers join a project, he/she needs to learn at least part of source code in that project before being able to make contribution, which is sometimes referred to as overheads. During this learning process, new developers may gain insights from the source code on design pattern, components structure, interactions of components and so on. These insights require a significant amount of time being spent on source code before being obtained, understood and transferred. We refer to these types of knowledge as tacit knowledge in this paper.

Based on these entities, our next step is to construct SSC networks to show an overarching view of FLOSS ecosystem, to illustrate complicated interactions among same and different types of entities, and to seek solutions for various kinds of problems being faced in software domain. We propose three types of SSC networks: dependency network, code reuse network and knowledge network.

- In dependency network, every node is a project. A link represents the dependency relationship from one project or package to another package. When a project imports or installs a package to leverage some of its functions, a dependency/link is formed between these two nodes.

- In code reuse network, every node is a project. A link is a file that is used by multiple projects. When a file in a project is reused by another project to fulfill similar need, a code reuse link is formed between these two nodes.

- In knowledge flow network, every node is a developer. A link represents the knowledge flowing from one developer to another one. When a developer joins a project, understands its source code, and contributes new piece of code, he or she gains knowledge. The code made by this developer may be read and understood by subsequent developers, along which knowledge flows from that previous developer to subsequent developers.

These SSC networks provide great values in managing software risks, tracing vulnerabilities and detecting developers' relations. For example, when a vulnerability is discovered, a

basic question raised to minimize its potential damage is to determine its effect damage areas, i.e., what software projects are or may be in danger. If the vulnerability lies in a package, we can quickly find all influenced packages and projects by marking all its direct and indirect dependencies (downstreams) through dependency network. If the vulnerability is discovered in a specific file, we can identify affected projects by seeking them in code reuse network where that file is used. Moreover, if taking developer into consideration, a more complete approach is to find authors of this file and find all projects where these authors contributed since these authors may introduce this vulnerability to other projects they participate, and this approach utilizes knowledge flow network.

4.10 Examples of SSC Networks for Illustration

4.10.1 Dependency Network in R CRAN Ecosystem

We start constructing the dependency network by exploring R CRAN ecosystem. R package list is scraped down from R CRAN official website which contains around 11K packages. We used data from METACRAN[5] which provides the latest R CRAN metadata containing the dependency information. There are five types[6] of dependency keywords in R CRAN and we considered 'imports' and 'depends' as dependency, because packages listed in 'imports' must be installed in advance and 'depends'[7] is the old name for 'imports'.

By creating a link from individual package to each dependency in its 'imports' and 'depends', we construct a dependency network for R CRAN in Fig 4.3. Packages with degree less than 20 are removed which ends up with 421 (1.9%) nodes and 3235 (6.6%) edges in Fig 4.3. Node size is proportional to its betweenness centrality value, and the color is based on modularization algorithm[8] of gephi.

[5]METACRAN is a collection of services around the CRAN repository of R packages. https://www.r-pkg.org/about

[6]http://r-pkgs.had.co.nz/description.html

[7]Prior to the rollout of namespaces in R 2.14.0, Depends was the only way to 'depend" on another package. Now, despite the name, you should almost always use Imports, not Depends.

[8]https://github.com/gephi/gephi/wiki/Modularity

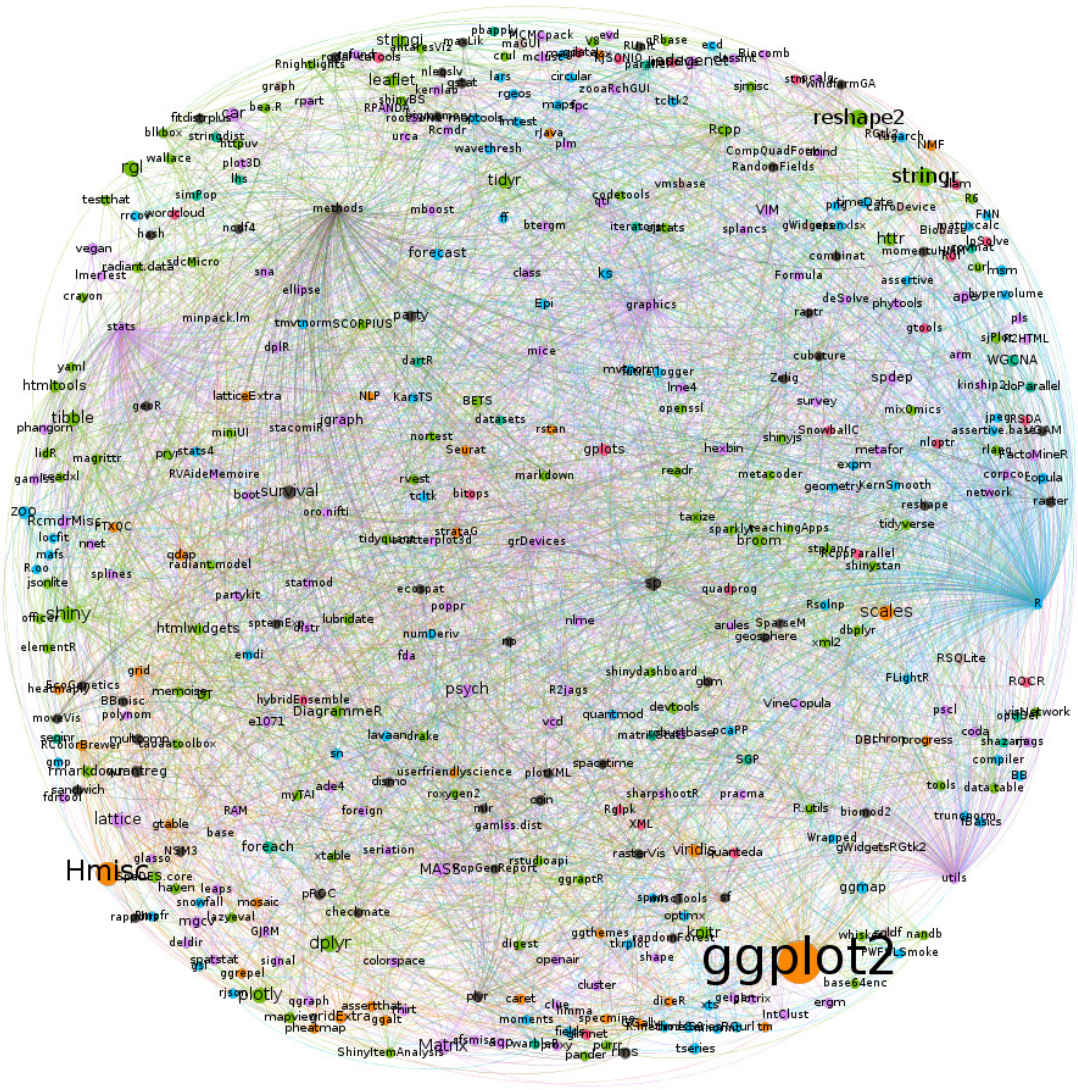

Figure 4.3: R CRAN dependency network.

As we can see (you might need to zoom in) from Fig 4.3, ggplot2, Hmisc, reshape2, stringr, Rcpp and rgl are core packages based on betweenness centrality. These packages provide different functionalities in R:

- ggplot2 and rgl are tools to draw 2D and 3D graphics.
- Hmisc[9] is like a grocery store which contains many functions useful for data analysis, high-level graphics, utility operations, variable clustering, etc.
- reshape2 facilitates data transforming between wide and long formats.
- stringr provides a full set of functions dealing with string/character processing.
- Rcpp[10] offers a seamless integration of R and C++ by enabling maps between R objects and C++ equivalents.

4.10.2 Knowledge Network in Emberjs Ecosystem

We begin constructing the knowledge flow network by investigating the expertise flow in a popular web front side framework – emberjs. Web front side framework has been attractive for a number of years and many open source developers already participated in to contribute their expertise which makes emberjs an ideal software to illustrate how complicated a knowledge flow network could be. In Fig 4.4[11], the productive developers are marked out with a string of author name and email separated by a bar.

4.10.3 Code Reuse Network in Emberjs Ecosystem

We start the construction of the code reuse network by mining code reuse pattern of emberjs. We collected all the blobs(git object preserving the content of a file) for emberjs[12], searched for projects that share blob(source file) with emberjs and investigated blobs that

[9] https://cran.r-project.org/web/packages/Hmisc/index.html
[10] https://cran.r-project.org/web/packages/Rcpp/index.html
[11] Node size and color are set in the same way with Fig 4.3. Note that several labels have been adjusted to fit in page
[12] https://github.com/emberjs/ember.js

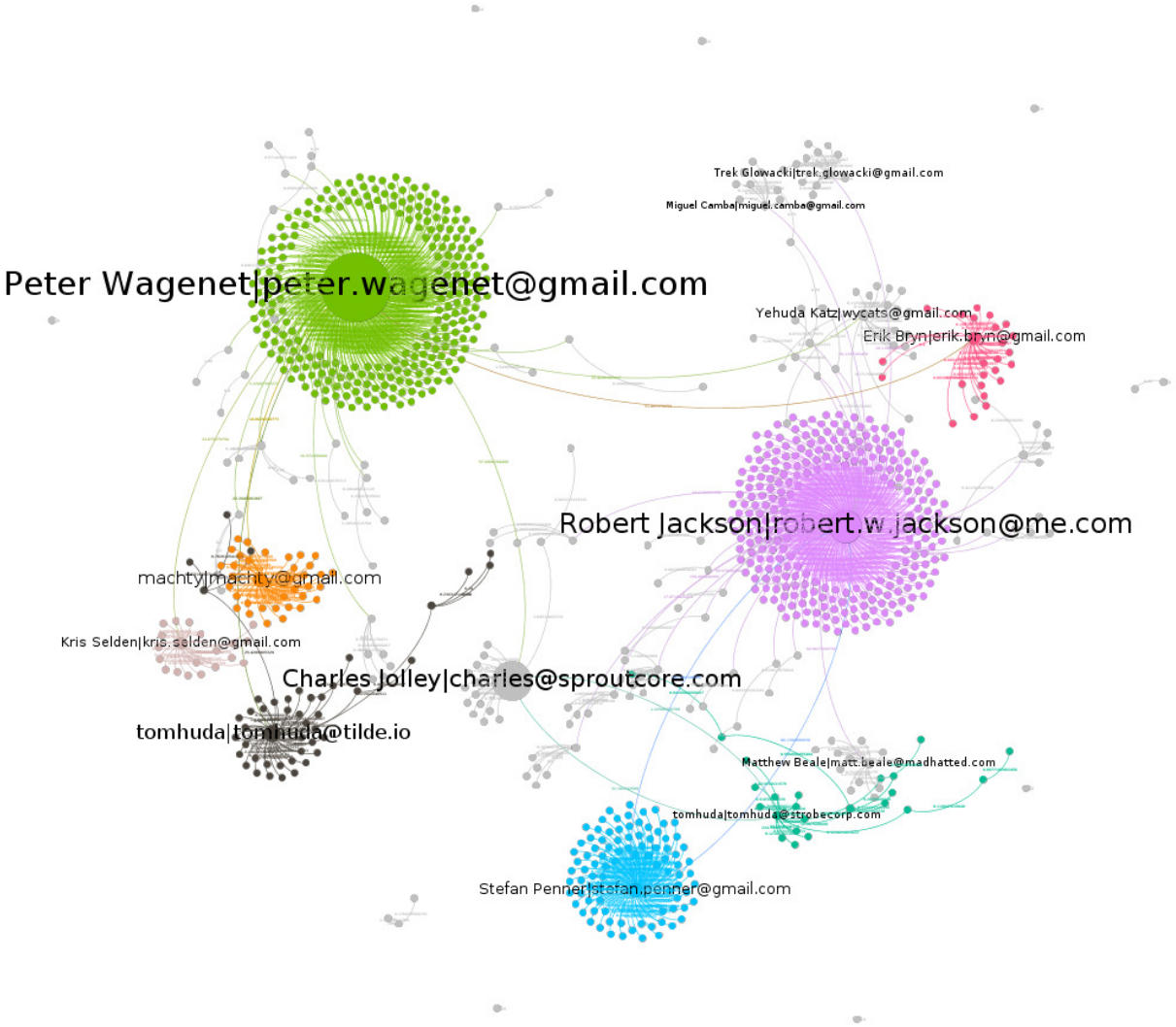

Figure 4.4: Knowledge Flow Network for Emberjs

span multiple projects. These projects were gathered and categorized into different groups indicating different code reuse patterns:

- Build tools: rake – make file for Ruby on Rails.
- Testing: qunit – a testing framework.
- Runtime: jQuery – a JavaScript library.
- Framework: epf – emberjs Persistence Foundation.
- Prior incarnations: SproutCore/Amber.js – early name for the emberjs project.
- Hard forks: innoarch/bricks.ui – a hard fork of emberjs that was then developed as a separate project.
- Tutorials: cookbooks/nodjs: – early code examples.
- Package manager: package.json – a file for NPM package manager.

Chapter 5

Risk Mitigation from SSC[1]

5.1 Overview

Various researches in software domain are enabled and enhanced by leveraging the joining power from SSC concept and WoC infrastructure. We introduced a group of applications in Chapter 4. In this chapter, we evaluate SSC from the perspective of risks, i.e., if SSC is helpful in mitigating risks in OSS community. To exemplify the benefit of SSC on reducing risks, we look into a specific research domain, the analysis of software technologies adoption among developers, and show how the findings can reduce the risk of abandonment from the perspective of a user downstream and the risk of low adoption from the perspective of the producer upstream.

A brief summary of the content in this chapter is presented below:

- Motivation: The question of what combination of attributes drives the adoption of a particular software technology is critical to developers. It determines both those technologies that receive wide support from the community and those which may be abandoned, thus rendering developers' investments worthless.

[1]This chapter, in part, is a reprint of the material as it appears in IEEE Transactions on Software Engineering, titled " *A Methodology for Analyzing Uptake of SoftwareTechnologies Among Developers*" (2020). Authors: **Yuxing Ma**, Audris Mockus, et al. The dissertation author was the primary investigator and author of these two papers. Copyrights of both papers are held by **Yuxing Ma** and Audris Mockus.

- Aim and Context: We model software technology adoption by developers and provide insights on specific technology attributes that are associated with better visibility among alternative technologies. Thus, our findings have practical value for developers seeking to increase the adoption rate of their products.

- Approach: We leverage social contagion theory and statistical modeling to identify, define, and test empirically measures that are likely to affect software adoption. More specifically, we leverage a large collection of open source version control repositories (containing over 4 billion unique versions) to construct a software dependency chain for a specific set of R language source-code files. We formulate logistic regression models, where developers' software library choices are modeled, to investigate the combination of technological attributes that drive adoption among competing data frame (a core concept for a data science languages) implementations in the R language: `tidy` and `data.table`. To describe each technology, we quantify key project attributes that might affect adoption (e.g., response times to raised issues, overall deployments, number of open defects, knowledge base) and also characteristics of developers making the selection (performance needs, scale, and their social network).

- Results: We find that a quick response to raised issues, a larger number of overall deployments, and a larger number of high-score StackExchange questions are associated with higher adoption. Decision makers tend to adopt the technology that is closer to them in the technical dependency network and in author collaborations networks while meeting their performance needs. To gauge the generalizability of the proposed methodology, we investigate the spread of two popular web JavaScript frameworks `Angular` and `React`, and discuss the results.

- Future work: We hope that our methodology encompassing social contagion that captures both rational and irrational preferences and the elucidation of key measures from large collections of version control data provides a general path toward increasing visibility, driving better informed decisions, and producing more sustainable and widely adopted software.

5.2 Introduction

Open source has revolutionized software development by creating and enabling both a culture and practice of reuse, where developers can leverage a massive number of software languages, frameworks, libraries, and tools (we refer to these as software technologies) to implement their ideas. Open source allows developers, by building on the existing work of others, to focus on their own innovation [114, 115, 66, 116], potentially reducing lead times and effort. This approach, however, is not absent of risks. For example, if a particular technology chosen by a developer is later supplanted by another, incompatible technology, the support for the supplanted technology is likely to diminish. Reductions in support for the supplanted technology result in increased effort on the part of the developer to either provide fixes upstream or to create workarounds in their software. Furthermore, the value of the developer's creation to new downstream projects may diminish in favor of the now more popular alternative technology. As a consequence, both the importance of a developer's product and their reputation may suffer. To remedy these two risks, developers must understand how attributes of their software products may be perceived among potential and actual downstream adopters (consumers of the technology), especially in relation to alternative, competing technologies these adopters may have. It is natural, therefore, to adopt the position that open source software development should be investigated from a supply chain perspective, which also pertains to distributed decision and supply networks among different stakeholders. We refer to the collection of developers and groups (software projects) producing updates (patches and new versions) of the source code as a Software Supply Chain (SSC) [56, 41]. The upstream and downstream links from project to project are represented by the source code dependencies, sharing of the source code, and by the contributions via patches, issues, and exchange of information. While the product adoption in supply chains has been well studied [59, 61, 96, 20], little is known or understood about how developers choose what components to use in their own software projects.

As a complex dynamical system, every player in the open source ecosystem may have their specific set of preferences or biases, which can affect the ultimate outcome of wide (or narrow) adoption and/or entire abandonment of formerly popular technologies. These decisions are

not only based on technical merit but the availability and accessibility of relevant information along with the tastes of consumers(adopters). Furthermore, these SSC networks may severely limit developer choices at the particular point in time when they need to make decisions on which components or technologies to use based on what components they are aware of and how much time or inclination they have to investigate the relative merits of the possible choices. This suggest the potentially strong influence of default choice well documented in behavioural economics. Hence, in contrast to common conventions, we should not simply model the preferences of individual developers but must also take into account the complexity of the supply networks and their specific position within them.

We want to address this major gap in knowledge empirically by using a very large data source comprising version control data of millions of software projects. Our methodology involves using this data to construct software supply chain networks, identifying software technology choices, theorizing about factors that characterize the developer and the technologies they chose, and finally fitting and interpreting the models for specific technology choices and, thus, characterizing the implicit primary factors (social, behavioural, and rational) they may use to make their decision.

Despite the practical and theoretical importance of the question how developers make technology choices, the extant literature does not offer theoretical guidance on this subject. We, therefore, leverage social contagion theory, which has been effective, among other things, in clarifying key aspects of organizational adoption of technology [6, 98]. Social contagion theory mimics models of the spread of contagious diseases but apply them in the behavioral/social context instead of the physiological one. The first key concept is exposure or how widespread the infectious agent is in the population. In our case the agent is a specific technology and the population is the entire collection of FLOSS repositories. Exposure is critical in epidemiology because without exposure a disease can not spread. This brings us to

RQ1: Does the exposure to a technology, such as the number of FLOSS repositories in existence, the rate at which new repositories are adopting this technology, or the number of high-score questions on StackExchange affect the decisions of the developers to adopt that technology?

The second key concept is infectiousness: a highly virulent agent is more likely to spread in a population. We deal with technologies (groups of packages), so in our case we would like to establish:

RQ2: Will extremely attractive technology (with few open issues, short response times to issues or pull requests, heavy activity and many authors), be more likely to be adopted?

The final concept is proximity: some infectious agents may not survive the travel through air or physical barriers, thus halting their spread. In our case, the distance from a developer to a technology is not physical, but it may be represented by the technological constraints (lack of compatibility with other technologies the developer already uses), need for certain performance characteristics, or a social distance to collaborators who are working with other developers already exposed to the technology or a related one. Hence:

RQ3: Will proximity of a developer or a project to a technology increase the rate of adoption? More specifically, **RQ3a:** will the proximity of a developer to a related technology used by a developer increase the chances of adoption; **RQ3b:** will the proximity of a developer to collaborators who already use the technology or a related one increase the chances of adoption?; **RQ3c:** will the performance requirements of the project a developer is working on increase the chances of adoption of a technology that has the desired performance attribute?.

To answer RQs, we need to collect data on the actual choices made by developers, operationalize key theory-based measures, and reconstruct the past states (historical states before adoption) of all public software projects that may choose the technology under study. For example, for a project that chooses Technology A in January 2014, we need to establish how many other projects have used A before that date (exposure), what average response time to issues the project had at that time (infectiousness), and what actions the developer making the choice to add the dependence had prior to that point in time, including her social network, technology network, etc.

To exemplify the proposed methodology, we investigate the rapidly growing data-science software ecosystem centered around the `R` language. One of the key technology choices in this area are the data structures used to store data (in the data-science sense). R has two major

competing technologies implemented in packages[2] `data.table` and `tidy`. (a more detailed introduction of these two packages is given in Sec. 5.5). To gauge the generalizability of the proposed methodology, we applied[3] our approach to investigate the spread of two modern popular web JavaScript frameworks Angular and React.

Our research provides several theoretical and practical innovations. From the theoretical standpoint, the novelty of our contribution first lies in introducing social contagion theory that provides first-principles based methods to construct hypotheses and to determine measures that should affect technology adoption. The second novelty is the context in which we investigate technology choices, i.e., a complete SSC [19, 40], not restricted to a set of projects or ecosystems. Third, we use regression models to understand how macro trends at the scale of the entire SSC emerge from actual decisions the individual developers make to select a specific software technology. More specifically, as a result of contextualizing social contagion theory through SSCs, our approach provides novel measures, such as proximity in a dependency network and authorship network, questions and answers with high score in Q&A, performance needs, and total deployments, that strongly affect the spread of technology and that were not used in prior work on library migration.

From the practical standpoint, our contribution consists of proposing a method to explain and predict the spread of technologies, to suggest which technologies are more likely to spread in the future, and suggest steps that developers could take to make the technologies they produce more popular. Developers can, therefore, reduce risks by choosing technology that is likely to be widely adopted. The supporters of open source software could use such information to focus on and properly allocate limited resources on projects that either need help or are likely to become a popular infrastructure. In essence, our approach unveils previously unknown critical aspects of technology spread and, through that, makes developers, organizations, and communities more effective.

In Sec. 5.3 we introduce the diffusion of innovation, social contagion, and the application of choice models. In Sec. 5.4, we describe the dataset and how we operationalize software supply chain. Choice model theory and our candidate technology are introduced in Sec. 5.4.4

[2]We use 'package' in the rest of our paper as a synonym for 'technology', since most software technologies are implemented in package format for use and 'package' is more appropriate to use in analysis.

[3]Source code and result are provided in https://github.com/ssc-oscar/PackageAdoptionAnslysis

and Sec. 5.5.1 respectively. In Sec. 5.5, operationalization of attributes of choice model is illustrated. Sec. 5.6 describes and interprets the result of applying the choice model. Related work is discussed in Sec. 5.8 and major limitations are considered in Sec. 5.7. We summarize our conclusions and contribution in Sec. 5.9.

5.3 Conceptual background

We draw on methodologies from a diverse set disciplines. The phenomena we are investigating is often called adoption [10] or diffusion of innovation [92]. Both theoretical approaches model how products or ideas become popular or get abandoned. We would like to fit such models and, in order to do so, find relevant set of predictors that have theoretical justification. Fichman [42] considered how internal factors such as resources and organization predict innovations in commercial enterprises, and DiMaggio [31] included the factor of environment as well. The adopters of the technology may influence non-adopters over time. Angst *et al.* [6], use the concept of social contagion [16], which consists of observation, information transmission, and learning to study spread of electronic health records. These concepts are familiar to any open source developer. More specifically, in addition to purely social contagion, we also have technical dependencies that act as strong constraints on developer actions. The signaling theory applied for social coding platforms [25, 110] provides some specific guidance as to what may motivate developers to chose one project over another. Many of the actions developers take on GitHub are focused on building or maintaining their reputation, hence they pay a particular attention to measures such as activity, numbers of participants, or "stars"[4].

The basic premise of social contagion theory is that developers may observe the actions and decisions of others, communicate them, and learn to emulate them over time. This premise implies that groups and individuals who are in social and spatial proximity to prior adopters are more susceptible to the influence of prior adopters of technology. This susceptibility (synonymous with potency or infectiousness of influence) is likely to result

[4]placing a star on a GitHub repository allows a developer to keep track of projects they find interesting and to discover similar projects in their news feed.

in an increased likelihood to adopt the same technology [6]. Notice, that the susceptible to influence of prior adopters represents non-rational behaviour. Rational behaviour would require developer to choose the best technology irrespective of social influences. It may also represent cognitive bias of the default choice. The developer may not know about the alternatives if their social or technical networks do not present them with an encounter with alternatives. This would represent the irrational bias toward default choice. These precursors of spread, if measured and calibrated with the actual level of technology spread, would provide the relative importance of each factor in driving the adoption and provide the understanding to help developers choose technologies wisely and provide hints on how to make their own technology more widely adopted. Fortunately, the mathematical adoption models have been developed and refined over time. A variation of multinomial regression models also called choice models[75] can be used to describe the behavior of a decision maker given a set of alternatives. Choice models have been used successfully in the fields of marketing [62, 55, 103, 44] and economics [76, 11, 100] to understand how consumers make choices. Adapting and applying these regression models to technology adoption, we focus on a developer, or more precisely, a software project as a decision maker. The actual decision is operationalized as the first among the alternative technologies that a project in a commit modifying one of the files within a repository. As with the social contagion theory, two types of predictors can be included: properties of the choice (i.e., the technology) and properties of a decision maker (i.e., the project or individual developer).

Equipped with this theoretical and modeling framework, we set out to address RQ1 and RQ2 by empirically characterizing the spread of software technology through analysis of a very large collection of version control data introduced in [73] which is referred to as WOC-DATA in this paper. WOC-DATA is used to construct the SSC [53, 70] by determining dependencies among software projects and developers, then by characterizing these projects according to their technical characteristics and supply chains. The social contagion and signaling theories allow us to select meaningful measures for the decision makers and for their choices.

5.4 Constructing software supply chains

We utilize all Git objects (1.1 billion commits and 4 billion of blobs and trees) from WOC-DATA to construct the relevant supply chain, social, and adoption measures. For our analysis we create mappings among these objects and their attributes, e.g., filename to associated blobs.

5.4.1 Measuring the dependency network

While many types of static dependencies exist, here we focus on explicit specification of the dependency in the source code. For example, 'import' statements in Java or Python, 'use' statements in Perl, 'include' statements in C, or, as is the case for our study, 'library' statements for the R language.

We analyze the entire set of 4 billion blobs existing in the database at the time of the analysis using following steps:

1. Use file to commit map to obtain a list of commits (and files) for all R language files by looking for the filename extension '.[rR]$'

2. Use filename to blob map to obtain the content for all versions of the R-language files obtained in Step 1

3. Analyze the resulting set of blobs to find a statement indicating an install or a use of a package:

   - `install\.packages\(.*"PACKAGE".*\)`
   - `library\(.*[\"']*?PACKAGE[\"']*?.*\)`
   - `require\(.*[\"']*?PACKAGE[\"']*?.*\)`

4. Use blob to commit map to obtain all commits that produced these blobs and then use the commit to determine the date that the blob was created

5. Use commit to project map to gather all projects that installed the relevant set of packages

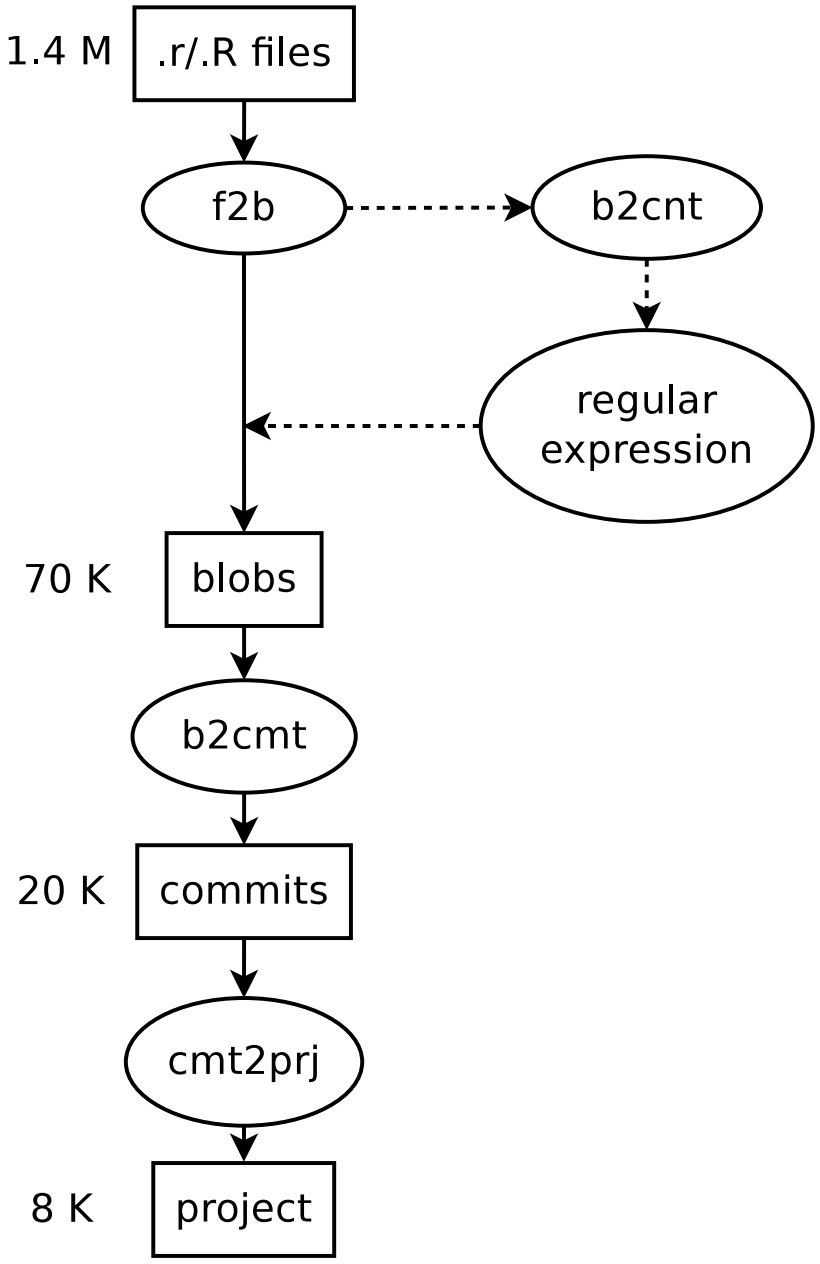

Figure 5.1: Project discovery

These steps are illustrated in a flowchart in Fig. 5.1. In Fig. 5.1, the rectangular boxes represent inputs and outputs, and ovals represent maps or dictionaries we utilized in this study. f2b stands for filename-To-blob map, b2cnt stands for blob-To-content map, b2cmt stands for blob-To-commit map, and cmt2prj for commit-To-project map. The number on the left side represents the unique number of corresponding objects.

A similar approach can be applied to other languages with suitable modification in the dependency extraction procedures, since different package managers or different languages might require alternative approaches to identify dependencies or the instances of use.

In addition to dependencies, we also need to obtain measures that describe various aspects of social relationships among developers because the theories of adoption, such as social contagion theory we employ, need measures of information flows among individuals as an important factor driving the rate of adoption.

5.4.2 Measuring the authorship network

The authorship network can be viewed as the process of developers working with other developers either by implicitly learning skills from other's contribution (source code) or by explicitly communicating through emails or discussion platforms. Here we focus on the former mode of communication since the bulk of direct communication may be private. We consider two types of links among developers. A weak link exists between a pair of developers if they commit in at least one project that is common between them and a strong link exists if they change at least one file in common.

5.4.3 StackExchange

StackExchange is a popular question answer website related to programming. When people search for information there they may notice answers that suggest the use of either `tidy` or `data.table` (discussion about choosing these packages is in Sec. 5.5.1) and, consequently, might be inclined to incorporate one of these packages into their own code. The latest (2017-12-08) StackExchange data dump including 57GB of posts was imported into MongoDB, out of which 6k questions (excluding answers) were found to be related to either `data.table` or

`tidy` by searching for these two terms in the title or the content of the post. We operationalize two measures: one counts the total number of posts while another measure counts only questions that have a score above 20 to gauge the amount of high-score content that is likely to be referred to from search engines.

5.4.4 The Choice Model

The choice set (set of alternatives) needs to exhibit three characteristics to be able to fit a discrete choice model. First, the alternatives need to be *mutually exclusive* from the perspective of decision maker, i.e., choosing one alternative means not choosing any other alternative. Second, the choice set must be *exhaustive* meaning all alternatives need to be included. Third, the number of alternatives must be finite. The last two conditions can be easily met in our case: Our choice set consists of two packages - `data.table` and `tidy`; Decision makers are restricted into the group of projects in our collection where either of those two packages is installed. To ensure the choices are mutually exclusive we model the choice of the first technology selected.

In this paper we applied the mixed logit model to study developers' choice over analogous R packages (`data.table` v.s. `tidy`). While many variations of choice models exist, the mixed logit model has the fewest assumptions on the distribution of the choice. Here we are not trying to solve the classical choice model which, for example, assumes a complete knowledge about the alternatives and produces implicit utility function. Instead, we simply look for factors that strongly affect the decisions developers make, whether these factors may be rational or related to cognitive or social biases.

5.4.5 Issues

It is reasonable to believe that the number of issues and how an issue is solved during the development of a software package may affect a developer's choice. This factor belongs to a set of rational choices. To measure it we collect the issues reported during the development of `data.table` and `tidy` packages. Since both packages are hosted on GitHub, we use

GitHub API to scrape all issues[5] reported for both packages. We collected 2.6k issues for the `data.table` and 1.6k issues for the `tidy`.

5.5 Case study

5.5.1 Selecting candidates for study of adoption

We chose software technologies from the data science ecosystem of projects using the R language because several of the co-authors are knowledgeable and have decades of development experience in R, and we, therefore, do not need to seek external experts to provide interpretations of the findings. As with most language-based ecosystems, the core language provides only basic functionality with most of the external packages being maintained in CRAN and Bioconductor distributions. Each package can be thought as presenting a technology choice. Since the technologies of storing and managing data are crucial in data science, we selected two widely used such technologies: `data.table` and `tidy`.

Apart from the `dataframe` package that is a part of core R language, `data.table` and `tidy*` are the two other most popular packages for data manipulation[6]. More specifically, `tidy*` represents a list of packages that share an underlying design philosophy, grammar, and data structures that are built for data science in R. Hadley Wickham, the Chief Scientist at RStudio and the main developer of `tidy*`, developed a family of packages called `tidyverse` to facilitate the usage of `tidy*` packages by assembling them into one meta package. We extract a set of packages from `tidy*` that share similar functionalities with `data.table` and refer to all of them here as the `tidy` package. This includes `tidyr`, `tibble` and `readr` packages.

`data.table` was written by Matt Dowle in 2008 and is known for its speed and the ability to handle large data sets. It's an extension of base R's data.frame with syntax and feature enhancements for ease of use, convenience and programming speed. It's built to

[5]Notice that GitHub API treats pull requests as issues (https://developer.github.com/v3/issues/), and we dropped the pull requests from all collected issues.

[6]Based on WOC-DATA in https://bitbucket.org/swsc/overview/src/master/deps/README.md

be a comprehensive, efficient, self-contained package, to be fast in data manipulation, and it has a succinct DSL (domain-specific language). Conversely, `tidy` focuses on the beauty of function composition and data layer abstraction which enable users to pull data from different databases using the same syntax.

In addition to the case study in R domain, we selected another case study focusing on the Javascript ecosystem. JavaScript[7] is the most popular programming language and has been commonly used in developing web applications, where one or more web frameworks were involved. Among all successful web frameworks, `Angular` and `React` are two of the most successful and widely supported by big technology companies (Google[8], Facebook[9]). Together, the massive amounts of adoption data, similar functionalities and corporate support from leading companies, make `Angular` and `React` a suitable comparison group for the investigation of technology adoption. The selection of the second case study was based on concerns about the generalizability of the findings from the R domain to other ecosystems. We, therefore, attempted to find a case that is radically different in a number of dimensions. We contrast the domain, scale, and governance principles in the second case study. Specifically, the mundane data management tasks supported by `data.table/tidy` and the need to create an engaging user interface in `Angular/React` contrast not only the domain of application, but also the types of functionality that is being implemented. Regarding scale, the R ecosystem is used by a relatively small number (a few thousand) of data analysts, while the `Angular/React` ecosystems are used by tens of thousands of web developers. Both `tidy` and `data.table` follow the community development model. `Angular` and `React`, in contrast, are primarily funded by corporate sponsors.

5.5.2 Data collection

We leveraged both WOC-DATA and WoC [73] infrastructure for data collection and filtering. According to [73], WOC-DATA approximates the entirety of public version control and includes major forges such as GitHub, BitBucket, GitLab, Bioconductor, SourceForge, the

[7] https://stackify.com/popular-programming-languages-2018/
[8] https://en.wikipedia.org/wiki/Angular˙(web˙framework)
[9] https://en.wikipedia.org/wiki/React˙(web˙framework)

now defunct Googlecode, and many others and contained over 46M projects at the time of analysis. WOC-DATA production involves discovering [74] and cloning the projects, extracting Git objects from each repository, and then storing these objects in a scalable key-value database.

WoC is an open source data mining infrastructure [10] [73], which provides not only APIs for extraction of development data on various levels for open source projects, but also offers intermediate collections and results which are extremely useful for studying domain knowledge. In particular, as illustrated in Fig. 5.1 and Sec. 5.4.1, we used the collection of all R file names and maps of file-to-blob, blob-to-content, blob-to-commit and commit-to-project to discover our targeted projects. Meanwhile, for each project, we found the first commit in which `data.table` or `tidy` was imported by sorting the commit time. By querying the content for this first commit, we learned the author of the commit, the commit message, etc. We set this first commit time as the end point of our analysis for each project, as it is at this time that the developer's choice between `data.table` or `tidy` becomes clear. For every targeted project, we evaluate explanatory behavior, activities, and relationships as computed before this end point, so that the analysis considers all factors as they were at the time the choice was made.

We further refined the list of projects because a large fraction involved forks of other projects. One of the most typical ways to make contributions to the development of a project on GitHub is by creating a fork of the project, making changes to this clone, and then sending a pull request to the original project. As a result, a popular project may have hundreds of forks that share a large portion of the source code and commit history. These forks are not equivalent to the original projects from which these forks were created and, therefore, were removed from consideration. To detect and delete these forks, we classify projects based on common commits, i.e., a pair of projects are linked if they have at least one commit in common. Based on these links, a transitive closure produces disjoint clusters. Each cluster represents a single observation in our study. The date when the first blob containing the focal technology was created is used as the technology adoption date for this cluster.

[10] https://github.com/ssc-oscar/Analytics

The extraction of the supply chain data for these two packages started from 1.4M R files in the entire WOC-DATA collection, with 70K blobs (versions of these files) that contained information pointing to the installation of either package. As a result, fewer than 20K commits were found that produced these 70K blobs. Then by using the commit-to-project map in WoC, we identified around 24K projects (7K for `tidy` and 17K for `data.table`) that installed either `data.table` or `tidy` between June, 2009 and January, 2018. After removing forks, we were left with a total of 8,303 projects (2,660 for `tidy` and 5,643 for `data.table`). Furthermore, we removed 4,961 `data.table` adoptions occurring prior to June 16, 2014, the date when `tidy` was first introduced. Thus, while `data.table` predates the introduction of `tidy`, our analysis focuses on the period of time when both choices are available.

We applied a series of similar procedures (as described above for `data.table` and `tidy`) on `Angular` and `React`, and found the list of projects, which at some point in the past have adopted either `Angular` or `React`. The only difference in the `Angular` and `React` case is that Javascript projects (deployed via NPM) usually record dependency information in a specific file named 'package.json', and we went through all of the versions of the 'package.json' file in WoC to identify projects that adopted either `Angular` or `React`. In summary, we identified 292494 projects (100894 for `Angular`, 191600 for `React`) that adopted either Angular or React.

5.5.3 Operationalizing Attributes for Regression Models

In this section, we define and justify the variables that quantify the key attributes pertaining to the set of software choices available to developers, as well as the characteristics of the developers making the choices. To this end, we propose 11 variables that seek to capture the key factors that may have influenced developers' choice of `data.table` or `tidy`, `Angular` or `React`. To streamline the presentation of the operationalizations of the variables in both studies, we only describe the operationalization for the R domain and note differences, if any. All of these attributes apply to the Javascript domain as well. These variables are listed in Table 5.1 and described in more detail below.

 # commits and authors (Cmts & Aths) are the number of commits and authors, respectively, and aim to capture the size of a project, as project size which may affect package

Table 5.1: Independent variables

Independent variables	Annotation	Category	Property type	Data source
CumNum	the total number of projects that deployed the package	exposure	choice related	WoC
RplGp	the time gap until the first reply to an issue	infectiousness	choice related	GitHub API
Unrslvd	the number of open issues over the number of all issues	infectiousness	choice related	GitHub API
StckExch	the number of questions with score above 20 related to either package	exposure	choice related	StackExchange dump
C	boolean, indicating whether a project contains C file	proximity	decision maker	WoC
Cmts	the number of commits	infectiousness	decision maker	WoC
Aths	the number of authors/developers	infectiousness	decision maker	WoC
Prx2TD	the proximity to tidy through dependency network	proximity	decision maker	WoC
Prx2DT	the proximity to data.table through dependency network	proximity	decision maker	WoC
AthPrx2TD	the proximity to tidy through authorship network	proximity	decision maker	WoC
AthPrx2DT	the proximity to data.table through authorship network	proximity	decision maker	WoC

adoption. Larger projects, for example, may prefer less controversial, more conservative package choices. This is a quality of the choice, so it would most closely fit under the "infectiousness" category according to the social contagion theory. We chose not to use lines of code (LOC) as a measure of size, since it has a less stable distribution than the number of commits, while, at the same time, being highly correlated with it. Operationally, for a particular adopter, we collected all commits prior to the end point (i.e., the first adoption of one of the two targeted packages) by applying the project-to-commit map followed by a time point filtering. We extracted the authors of these commits and counted the number of unique authors.

 # projects of deployments (CumNum) is the overall number of project deployments of `tidy` or `data.table`. A larger **CumNum** should increase the chance that a developer would be aware of, and get exposed to, a particular package and may influence the developer's package adoption decision. This measure falls within the "exposure" category of contagion, because it quantifies the chances that a developer may become a user of the technology. This is characterized as a factor that is not rational, as the project is hypothesized to be biased towards technology that they are more likely to encounter, not necessarily technology that would be optimal for that project. To assess **CumNum** for a given package, we counted the number of projects that adopted `data.table` and `tidy`, respectively, before a decision was made by the developers of the package under evaluation.

 # open issues over all issues (Unrslvd) can be an indicator of package quality. A higher fraction of unresolved issues may indicate that the package has a significant number of problems, which, like a bad review, may undermine people's confidence in it. This is a quality of the choice, so we hypothesize that it relates to the "infectiousness" aspect of the

social contagion paradigm. To measure this quantity, we leveraged GitHub API to collect all issues for `data.table` and `tidy` packages and filtered issues raised before the decision end point for each adopting project. We count the number of unresolved issues (issues that are still open) and normalize it over all issues raised before end point, because in general a project tends to have more issues and unresolved ones as its age grows, and we believe the averaged rate of unresolved issues is more reflective of a package's maintenance and quality.

Association of c code with a project (C) is used as a proxy for the requirement for high performance. Typically, computations that are too slow for the interpreted R language are implemented in C to improve performance. This is a requirement of the decision maker that would most closely fit under the "infectiousness" category because it likely indicates a strong preference for higher performance embodied by the `data.table` choice. This is a good example of a factor that may represent a rational choice for some decision makers. To measure this aspect, we applied commit-to-file map on every commit prior to end point for each project and filtered files with suffix '.[cC]'.

 # related questions on StackExchange with high score (StckExch) is a proxy for the popularity of each package. It counts the number of highly ranked (score > 20) questions related to each package. Developers often search for answers to issues they face and may stumble upon one of these packages presented as a solution to a problem they are facing, thus increasing the chances that they may adopt that technology. From a social contagion perspective this would increase "exposure". We avoid counting the total number of questions because most of the questions tend to be of low score[11] and the search engines may avoid including links to them, thus they do not increase "exposure." According to personal experience of all authors, search engines tend to avoid including links to questions with low score if questions of higher ranks are available. This factor may be interpreted from a rational perspective (leveraging experience of others when lacking other information), but more appropriately, it is a great example of social bias since the developer did not engage in due diligence, instead relying on social cues to make a technical choice. **StckExch** was obtained by selecting all relevant posts in the StackExchange dump (2017-12-08), which have

[11] In total, only 131/1666 data.table related questions and 162/1785 tidy related questions have score larger than 20

`data.table` and `tidy` in the post's title and body (including code snippet if any). Manual inspection found that posts without R-language tag '<r>' tag in the 'Tags' field were not relevant and we excluded them. Furthermore, we only selected posts with score above 20. Again, we counted posts created prior to the end point for each adopter project.

Project proximity to `data.table` and `tidy` in dependency network (Prx2DT/Prx2TD) measure dependency networks and can be understood from the perspective of software supply chain networks. Based on the characteristics of the software domain, especially the open source software community, dependency networks can be viewed as technologies (library/package) spreading from upstream (original package) to downstream (packages where the original package was installed) and, in turn, to further downstream packages.

We consider all downstream packages of `data.table` and `tidy`, e.g. those in the `data.table` and `tidy` clusters respectively. We hypothesize that if a project installed a package within the `data.table` cluster, then the project is more likely to install `data.table` than `tidy`. The rationale of such a hypothesis is that if developers installed a package because of 1)preferences for some of its functionalities or features inherited from an upstream package or 2) the way such a package works, which is sometimes influenced by or derived from an upstream package, then it is more likely that these developers will gravitate toward the upstream package over other alternatives.

Based on the dependencies of R CRAN packages, the clusters of `data.table` and `tidy` are easily constructed. More specifically, we used the METCRAN[12] API and scraped meta data for more than 11K R CRAN packages for which dependency information is available. Table 5.2 summarizes basic information on the networks that were constructed and more detailed information on the methodology follows.

Each downstream package in the `data.table/tidy` dependency network needs to be weighted before calculating proximity of an adopting package to both `data.table` and `tidy`. We suggest that the algorithm used to determine the weights be based on several key principles:

[12]https://www.r-pkg.org/about

Table 5.2: Network characteristics

Characteristics	data.table	tidy
# downstream packages	813	2203
# downstream layers	5	5
# of packages in common	636	
overlap ratio	0.78	0.28

- for each downstream package, only the relative weight to the root package (`data.table/tidy`) matters
- for each downstream package, the sum of its weights to both root packages is a constant
- the closer to a root package, the higher the weight that a downstream package gets relative to that root package

We assume that each package has a weight of 1 in total. Let's denote the packages set in the `data.table` downstream network as S_d, that in `tidy` as S_t, the weight of package a to `data.table` as W_{ad} and that to tidy as W_{at}, the depth of package a in the `data.table` network as D_{ad} and that in the `tidy` network as D_{at}, then based on principles mentioned above, the weights of package a are determined as follows:

- $W_{ad} = 1, W_{at} = 0$ if $a \in S_d$ & $a \notin S_t$
- $W_{ad} = 0, W_{at} = 1$ if $a \in S_t$ & $a \notin S_d$
- otherwise, $W_{ad} = D_{at}/(D_{ad} + D_{at})$, $W_{at} = D_{ad}/(D_{ad} + D_{at})$

The next step is to extract the list of packages installed in each observation/project, after which we can aggregate the weights of these packages to compute the proximity of each project.

As we have mentioned in Sec. 5.4, various maps among Git objects have been created. By utilizing maps of project-To-commit, commit-To-blob, and blob-To-content in sequence and selecting the install statements in blob content via regular expressions similar to those mentioned in Sec. 5.4.1, we get the list of packages installed in each project. From this set, we obtain projects that are either in `data.table` or in `tidy` clusters.

For a project p, denote the list of packages obtained in last step as L_p and denote a package in that list as a. Then the proximity of a project p to `data.table`, denoted as P_{pd}, and to `tidy` as P_{pt}, can be computed:

$$\begin{cases} P_{pd} = \Sigma_a^{L_p} W_{ad} \\ P_{pt} = \Sigma_a^{L_p} W_{at} \end{cases} \tag{5.1}$$

To summarize the process described above, we first measured the weight of each downstream package in either `data.table` or `tidy` by leveraging the R package dependency networks and the formulas above. Secondly, by following a similar flow in Fig. 5.1, we extracted all R packages that were adopted in the commits prior to end point where one of the focal packages was first adopted. Finally, we calculated the proximity to `data.table` and `tidy` by summing up the weights of all downstream packages for each project. Notice that a project's downstream packages that were not in `data.table` or `tidy` downstream set were dropped.

Project proximity to `data.table` and `tidy` in authorship network (AthPrx2DT/AthPrx2TD) represents the proximity of a developer to a focal project as measured through their author network. It can be explained from the perspective of social contagion. Social contagion refers to the propensity for a certain behavior to be copied by others. Consider the fact that developers in GitHub are linked through common projects they are devoted to, where information and ideas are shared and transmitted from one to others, an underlying social network emerges. Organizational actions are deeply influenced by those of other referent entities within a given social system, according to DiMaggio [31]: non-adopters are influenced by adopters over time, and they influence the behavior of other non-adopters after their own adoption [6] if thinking of our case as package adoption. In short, the adoption of `data.table/tidy` is a temporal process of social contagion.

We attempt to look for developers that are exposed to contagious packages — `data.table/tidy`. These developers include not only the authors of each package who are directly exposed inherently, but also developers who cooperate with directly-exposed authors in other projects. Authors of other projects that are directly exposed

to authors of `data.table/tidy`, are identified by applying a project-To-author map to both `data.table/tidy` packages separately and indirectly-exposed authors are obtained by combining the map of author-To-project and the map of project-To-author serially and then applying it on each directly-exposed author.

We classify authors exposed to `data.table` into the `data.table` author cluster and those exposed to `tidy` into the `tidy` author cluster. Projects/observations may have authors who are in either of these two clusters and these authors may influence the choice of data frame technology, i.e., (`data.table` vs. `tidy`). In order to estimate the impact of every author in each cluster, we use the following weights,

- $W_{bd} = 1, W_{bt} = 0$ if $b \in C_d$ & $b \notin C_t$
- $W_{bd} = 0, W_{bt} = 1$ if $b \in C_t$ & $b \notin C_d$
- otherwise, $W_{bd} = D_{bt}/(D_{bd} + D_{bt})$, $W_{bt} = D_{bd}/(D_{bd} + D_{bt})$

where b represents an author in a project; C_d/C_t stands for author cluster of `data.table/tidy`; D_{bd}/D_{bt} refers to the distances from author b to `data.table/tidy`, i.e., author b's depths in the author cluster of `data.table/tidy`, 1 for directly-exposed author and 2 for indirectly-exposed author; W_{bd}/W_{bt} is the proximity of author b to `data.table/tidy`, indicating author b's impact on choosing `data.table/tidy`. Note that these measures are similar to the ones used in calculating *Prx2DT/Prx2TD* and are based on similar principles.

After estimating each exposed author's influence, the overall exposed authors' influence in project p can be measured as follows:

$$\begin{cases} PA_{pd} = \dfrac{\Sigma_b^{A_p} W_{bd}}{N_p} \\ PA_{pt} = \dfrac{\Sigma_b^{A_p} W_{bt}}{N_p} \end{cases} \tag{5.2}$$

where A_p is the set of authors of project p who are in either of `data.table/tidy` author cluster; W_{bd}/W_{bt} is the proximity of author b to `data.table/tidy` calculated in previous step; N_p is the number of authors in project p; PA_{pd}/PA_{pt}, i.e., *AthPrx2DT/AthPrx2TD*, is the overall influence of exposed authors on a project p. Notice that *AthPrx2DT/AthPrx2TD*

is calculated through aggregating the influence of each exposed author and being normalized over the total number of authors in that project. The rationale for normalization is that a project tends to have more exposed authors if it contains more authors, resulting in a higher value for *AthPrx2DT/AthPrx2TD*. By normalization we remove this bias induced by the difference in the number of authors for different projects. This factor falls clearly within a realm of a social bias. It may also be partially explained as cognitive bias if the developer is not aware of alternative choices.

To summarize the computation of proximity through authorship network, we started by measuring the weight of each author who was either a co-author of `data.table/tidy` or had cooperated with at least one of the authors of `data.table/tidy`, which was detailed above. Then we summed up the weight of every author of a project and normalized it over the total number of authors in this project. Again, here we applied end point filter on every step in calculation.

Time gap between the raise of an issue and the first reply (RplGp) measures how fast developers or maintainers of a package respond once an issue has been raised. The timeliness of this response reflects the efficiency of package maintenance and can be attributed to the 'infectiousness' category of social contagion theory and could clearly be of interest for those deciding on which package to adopt.

The calculation of reply gap is worth discussing. We are interested in understanding how long it takes for an issue to get its first reply after being reported. For each individual in the study, we focus on the time period just before the key commit that includes the choice of focal package (`data.table/tidy`). However, several additional obstacles that needed to be addressed in order to measure the reply gap :

1. It is rare that an issue was raised simultaneously with the key commit (inside which either the`data.table/tidy` package is installed).

2. The timeliness of replying to an issue may vary drastically during the development of a package, hence taking the closest issue's reply-time as a representative is not reasonable

3. For some issues, it took a significant amount of time to get a reply and in some cases no reply was ever made to an issue, thus, averaging reply-time to previous issues is problematic due to long right-censored cases.

This is a case where statistical models for survival(time-to-event) are appropriate. In this scenario, an issue can be viewed like a patient under study with the first reply analogous to conclusion of the medical issue or death of the patient. We aim to model the time until reply to the reported issue, i.e., the survival time of the issue, with shorter lifetimes indicating a more interactive development team. Irrespective of package, for each issue, we record the time that it was submitted (timestamp recorded when the issue is raised) and use survival analysis to model the distribution of the issue lifetimes for each package (`data.table/tidy`) using the R package 'survival' [102]. Predictions for the reply time for each project (observation) can be made based on data collected before the key commit. The *RplGp* for a project is simply the median issue lifetime for an issue generated before a key commit. This factor appears to be clearly related to rational choice factors as the delays in response may cause real problems.

In practice, we extracted all issues of `data.table/tidy` from GitHub and measured difference between the time an issue is first raised and the first response time. As described above, we trained a survival model to estimate the distribution of the delay until first response delay. The model was trained using all issues that had been raised before the current package key commit. Those issues that had not been responded to yet were right censored in the model fitting. The reply gap represents the median value of response times.

In summary, we note that for each project that eventually adopts one of the two focal packages (`data.table/tidy`), all of the variables described in this section are calculated dynamically using only data that occurs before the key commit. In addition, for each observation, every predictor with choice property (Table 5.1) needs to be calculated for both packages, e.g., *Unrslvd* needs to be calculated for both `data.table` and `tidy`. These will end up being denoted as *Unrslvd.datatable* and *Unrslvd.tidy*.

5.6 Results

5.6.1 Result of data.table VS. tidy

Table 5.3 summarizes basic statistics for independent variables analyzed in the model. We use the R package 'mlogit'[13] [23] to fit the model using the 11 predictor variables defined above with the response being an indicator of the package chosen.

Very high correlations among predictors (above 0.9) occurred between *Prx2DT* and *Prx2TD*. High correlations may lead to unstable and difficult to interpret models and need to be addressed. Since we do not have any *a priori* theory-derived reasoning for removing one or the other variable, we removed *Prx2DT*. The modeling results remain stable if this approach is reversed. Table 5.4 presents the resulting model fit.

Table 5.3: Summary Statistics for Independent variables (data.table VS. tidy)

Variable	median	mean	std.dev
Cmts	3	46.83	645.68
Aths	1	2.13	8.23
C (boolean)	0	9.79e-03	9.85e-02
Prx2DT	0	0.15	0.95
Prx2TD	0	0.62	2.79
AthPrx2DT	0	6.99e-2	0.17
AthPrx2TD	0	0.11	0.24
CumNum.datatable	2.72e+03	2.66e+03	1.87e+03
CumNum.tidy	305	8.44e+02	9.12e+02
RplGp.datatable	2.09	2.16	0.33
RplGp.tidy	3.03	2.95	0.53
Unrslvd.datatable	0.29	0.28	3.23e-02
Unrslvd.tidy	0.20	0.16	7.7e-02
StckExch.datatable	130	125.76	6.53
StchExch.tidy	158	152.57	10.14

Below we summarize findings for each predictor variable separately.

 # related questions on StackExchange with high score (StckExch): the coefficient is 0.2, indicates that the number of high score questions on StackExchange is associated with the likelihood that a project would adopt the respective technology. The association is positive, holding other factors equal. For illustration, if the number of high score questions increases by 6 questions (1 std. dev.) from a median value of 130 for

[13] https://cran.r-project.org/web/packages/mlogit/vignettes/mlogit.pdf

Table 5.4: The Fitted Coefficients. (data.table VS. tidy)
$McFadden$[75] $R^2 = 0.14$ $n = 7k$

Variable	Estimate	Std. Error	p-val
tidy:(intercept)	-6.07	0.28	2.20e-16
CumNum	1.56e-04	1.44e-05	2.20e-16
Unrslvd	2.45	0.78	1.59e-03
RplGp	-0.38	4.88e-02	5.11e-15
StckExch	0.27	1.26e-02	2.20e-16
tidy:Cmts	-3.95e-04	2.22e-04	7.54e-02
tidy:Aths	-3.02e-04	7.12e-03	0.97
tidy:C	-0.67	0.28	1.82e-02
tidy:Prx2TD	0.17	2.87e-02	6.79e-10
tidy:AthPrx2TD	1.27	0.14	2.20e-16
tidy:AthPrx2DT	-7.06e-02	0.19	0.72

`data.table`, the estimated probability of choosing `data.table` increases from 0.58 to 0.87, while holding all other predictors at their median values.

This result aligns well with the social contagion theory that posits that increased adoption is a consequence of increased exposure. Surprisingly, including an additional predictor that counts the total number of questions (of high and low score), shows no statistical significance. It appears to be counter-intuitive as more exposure should increase adoption. However, when developers want to solve an issue related to the functionality of the R `data.frame`, they often may not search on StackExchange, but use a general search engine and follow links to StackExchange. The total number of posts, therefore, may be not visible to developers, only the set of posts that the search engine deems to be of sufficiently high score. The number of posts (questions), may, therefore, not be a good proxy of exposure. As such, the total number of posts of low-score questions, in fact, appear to discourage developers from using a package.

> ***Finding 1:*** *We found that exposure measured via the total number of questions on StackExchange had no impact on adoption, while the number of high score questions has a strong and positive correlation with increased adoption.*

 # open issues over all issues (Unrslvd): the coefficient is 2.5, indicating that the higher the ratio of unresolved issues a package has, the more likely it would be adopted. While it appears to be counter-intuitive from the perspective that unresolved issues may

indicate a lack of attention from maintainers. However, the causal relationship may go the other way: the increased interest from users when a package becomes popular among developers, may lead to more contributions in the form of issues, and may exceed packages developers' processing capacity, which results in a larger ratio of open(unresolved) issues. In short, the ratio of unresolved issues over all issues might indicate high rates of adoption especially in the early stages of the projects when the total number of issues is low.

> ***Finding 2:*** *We found that infectiousness of a package as measured via the fraction of unresolved issues, is associated with a higher adoption rate for that package.*

Project proximity to `tidy` in authorship network (AthPrx2TD): the coefficient is 1.3, indicating that the closer a project is to authors of the package `tidy` vis-a-vis the author network, the more likely they are to choose `tidy` over `data.table`. If the proximity to `tidy` in the author network increases by one standard deviation of 0.24 from a median value of 0 (e.g., a project that has four authors and one of them cooperates with `tidy`'s developers, but not with any of `data.table`'s developers), the estimated probability of choosing `tidy` increases seven percent from 0.42 to 0.49. This finding supports the basic premise of the social contagion hypothesis that developers' choices are affected by the environment they are in.

> ***Finding 3:*** *Proximity as measured by the fraction of authors who are either developers of the package to be adopted or who work with at least one developer of that package, increase the chances of adoption.*

This may be a consequence of authors who have direct expeience or are familiar through word-of-mouth. However, **Project proximity to `data.table` in authorship network (AthPrx2DT)** is not statistically significant. One reason may be that `data.table` is a more widely deployed package and the deployments may play a larger role than the social connections. Also, each community of users and developers may be different. For example, the `tidy` community may have more social interactions than the `data.table` community. Furthermore, the exposure in the `tidy` community may come from a much larger set of

packages in the `tidyverse`, while `data.table` does not have an equivalent brand that involves a wider variety of tools beyond data handling.

Association of c code with a project (C): the coefficient is -0.6, indicating that a project containing at least one C file is less likely to choose `tidy`. The estimated chances of choosing `data.table` increase by 15 percent from 0.58 to 0.73. The finding is consistent with our hypothesis that if an R project has a need for performance, as evidenced by the use of functionality being developed natively in the C language, then it is more likely to choose the higher performance of `data.table`.

> ***Finding 4:*** *Proximity, as measured by the project's need for performance, is associated with adoption of packages that emphasize high performance*

Time gap between the raise of an issue and the first reply (RplGp): the coefficient is around -0.4, indicating that the more quickly a package's issue gets a response, the more likely that this package will be chosen. If the number of days until first response to an issue increases by 0.21 days (1 std. dev.) from a median value of 1.4 for `data.table`, the estimated chances of a project choosing `data.table` decrease by two percent from 0.58 to 0.56 assuming all other variables remain at their median values. The time until first response is not as readily visible to developers as most other measures that we used, so developers may not be able to observe it when making a choice. However, it appears to be a reasonable proxy for project's reactions to external requests that could be easily gleaned by reading through some of issues on the issue tracker. A well maintained package is more likely to respond to new issues quickly and thoroughly, leaving a good impression and, thus, increasing the likelihood of being adopted. This has implications for designing project dashboards intended to make key project attributes more visible.

> ***Finding 5:*** *Infectiousness of a package as measured by speed of response to issues is associated with a higher adoption rate for that package.*

Project proximity to `tidy` in dependency network (Prx2TD): the coefficient is 0.2, indicating that the closer (through a dependency network) a project is to the package `tidy`, the more likely its authors are to choose `tidy` over `data.table`. If proximity to

`tidy` in dependency network increases by one standard deviation of 2.8 from a median value of 0 (e.g., a project installs/uses three packages that are in first layer downstream from `tidy`), the chances of choosing `tidy` go up by 12 percent from 0.42 to 0.54. It supports our hypothesis that the supply chain influences projects' choices. A project tends to install a specific package if it has already installed other packages that also depend on it, i.e., if a project uses downstream dependencies of a package, it is more likely to use the package itself rather than other alternatives. Being familiar with downstream packages may reduce the overhead or learning curve required for an upstream package, leading to an advantage over other choices.

> ***Finding 6:*** *Proximity to a package as measured via technical dependency networks is associated with a higher adoption rate.*

 # projects of deployments (CumNum): the coefficient is 1.4e-4, indicating that a larger number of deployments of a package in the past will make it more likely to be adopted. If the number of deployments increases by one standard deviation, 1870 projects, from a median value of 2660 projects for `data.table`, the estimated chances of choosing `data.table` go up by seven percent from 0.58 to 0.65 for a project holding all other values at the median. A larger number of overall deployments, on one hand, increases the chance for a package to be known by adopters. On the other hand, from the perspective of adopters, more deployments usually insinuate a stable and mature product (though it is not clear if the number of deployments is visible to a developer), and enhances adopters' confidence in this package. Either of these reasons justifies adoption of the widely deployed package as predicted by the social contagion theory.

> ***Finding 7:*** *Exposure to a package that is widely deployed is associated with a higher adoption rate.*

We also find that the number of authors in the adopting project does not affect the choice of technologies. Social contagion theory does not suggest that this predictor should have an effect, but it could be that project activity (which has a substantial correlation with the number of authors), may already account for the differences in propensity to chose `tidy` over `data.table` making the variation in the number of authors statistically insignificant.

***Finding 8**: We did not find statistically significant association between infectiousness as measured via the number of commits and adoption propensity.*

We achieved a $McFadden$[14] R^2 of 0.14, which is a good fit according to McFadden[75]. (Notice that the R package 'mlogit' use $McFadden$ R^2 instead of R^2 to estimate fitness of the model because logit models do not generate the sums-of-squares needed for standard R^2 calculation.)

Regression models are explanatory, but we can also use them to do prediction. The 10-fold cross-validation done by randomly splitting projects into 10 parts and fitting the model with predictors listed in Table 5.4 on nine parts and predicting on the remaining part yielded a reasonable AUC of 73%. Average accuracy was 70% with balanced Type I and II errors (obtained by choosing predicted probability cutoff of 0.49).

Finally, it is worth noting that out of six predictors that were statistically significant, only **CumNum, RplGp,** and **C** were clearly grouped into predictors that would support rational choice. The remaining three predictors primarily reflect a mixture of social and cognitive biases associated with social preference or default choice when alternatives are not known. If we include the effort needed to obtain the necessary information into the utility function, these social and cognitive biases can, of course, be explained rationally as well.

Based on the findings mentioned above, we derived a list of recommendations for package developers and users: To increase the popularity of a package, package developers should maintain quick response to users' questions and concerns such as raised issues on development platform; package development team should involve active developers, especially authors of existing packages, to have a broad author network; package developers should choose popular packages among alternatives (when there is a need) to use. To choose a popular (right) package among alternatives, users should choose the package with more high score questions on Q&A website such as StackExchange; users should choose the package with better maintenance such as response speed to raised issues and questions on development platform; users should choose the package that is more widely used. Counter-intuitively,

[14]https://stats.stackexchange.com/questions/82105/mcfaddens-pseudo-r2-interpretation

packages with a large number of outstanding issues should also be preferred as they might command a more active or larger contributor community.

5.6.2 Result of Angular VS. React

The basic statistics are shown in Table 5.5. We fitted the same model on this dataset and show the result in Table 5.6. (Notice that we use abbreviation 'AG' for `Angular` and 'RT' for `React`)

Table 5.5: Summary Statistics for Independent variables (Angular VS. React)

Variable	median	mean	std.dev
Cmts	1	13.23	315.07
Aths	1	1.21	3.12
C (boolean)	0	8.81e-04	2.96e-02
Prx2AG	0	7.54e-05	1.22e-02
Prx2RT	0	2.19e-04	1.65e-02
AthPrx2angular	0	0.14	0.25
AthPrx2react	0	0.14	0.25
CumNum.angular	1.0e+05	7.66e+04	3.35e+04
CumNum.react	5.05e+04	6.85e+04	6.27e+04
RplGp.angular	0.47	0.49	0.12
RplGp.react	0.44	0.44	4.83e-02
Unrslvd.angular	5.77e-02	7.42e-02	3.37e-02
Unrslvd.react	7.83e-02	9.19e-02	5.04e-02
StckExch.angular	3272	3.03e+03	4.65e+02
StchExch.react	1213	1.05e+03	2.91e+02

Table 5.6: The Fitted Coefficients. (Angular VS. React)
$McFadden$[75] $R^2 = 0.45$ $n = 292k$

Variable	Estimate	Std. Error	p-val
react:(intercept)	-12.68	0.10	2.20e-16
CumNum	2.02	1.08e-02	2.20e-16
Unrslvd	-27.28	0.39	2.20e-16
RplGp	-0.50	5.92e-02	2.20e-16
StckExch	-7.27e-03	4.95e-05	2.20e-16
react:Cmts	2.05e-04	2.73e-05	6.48e-14
react:Aths	-3.94e-02	3.24e-03	2.20e-16
react:C	-0.24	0.19	0.21
react:Prx2RT	-1.67	0.32	3.04e-07
react:AthPrx2RT	1.37	3.03e-02	2.20e-16
react:AthPrx2AG	0.18	2.83e-02	7.32e-11

Below our findings are briefly summarized.

CumNum (total number of projects using the package), **RplGp** (time it takes to respond to user issues) are both significant and have coefficients pointing in the same direction as the `tidy` and `data.table` study.

Unrslvd (# open issues over all issue) is significant but the coefficient is negative, which indicates that unlike in the case of `tidy` and `data.table`, developers prefer to adopt a technology with the lower fraction of unresolved issues. The reason such difference is exhibited may be due to the fact that both packages have strong commercial backing, so the developers may take this as a signal that a company is not as supportive of the product. In R ecosystem, `data.table` is a completely volunteer-supported operation, hence developers may be more forgiving to the limitations of the maintainers.

StckExch (the number of related questions on StackExchangewith high score) is significant but, unlike in R ecosystem, has a negative coefficient. Developers in Javascript domain may believe that a technology with more posts on StackExchange may be harder to use or more complex. Developers in different domains may have different opinions regarding the 'exposure' on StackExchange and further research on these differences would be needed to better understand this.

From Table 5.6, we can also find that a project with larger numbers of commits and fewer contributors are more likely to adopt `React` than `Angular`. One explanation could be that productive developers may prefer `React` over `Angular`; thus, a more productive project (commits count over contributor count) will be more likely to adopt `React`. These differences may also reflect the different nature of projects adopting each framework that we have not been able to capture using existing variables.

Among the four 'proximity' measures, only the proximity to `React` through author network is statistically significant. Given the complexity of the dependencies in JavaScript domain, the large size and the independent nature of the packages, the dependency network may not be as important when making a choice.

5.6.3 Generalizability between the Javascript and R domains

The two case studies were intended to test the generalizability of social contagion models in two distinct contexts that varied in terms of domain functionality, user base, scale, and

primary governance styles. By comparing Table 5.6 with Table 5.4, we can conclude that despite these radical differences in context there are substantial similarities.

CumNum (total number of projects using the package), and **RplGp** (time it takes to respond to user issues) have the same and expected impact on adoption: more exposure from wider past adoptions and higher infectiousness from better response rates were associated with increased rates of adoption in both case studies.

However, **Unrslvd** (the ratio of unresolved to all issues at the time of adoption) had an expected negative effect in the `data.table/tidy` study, but had a counter-intuitive positive effect in the `Angular/React` study. Similarly, **StckExch** (the number of related questions on StackExchange with high score) switched from having a positive impact to having a negative impact on the chances of adoption.

Our conjecture is that the operationalization of project quality (infectiousness) as expressed by the fraction of unresolved issues and the exposure to StackExchange high-score questions may have limitations. Specifically, `Angular/React` have experienced a spectacular growth over the considered period. We presume that this growth was fueled by a heavy commercial involvement, which, in turn, motivated the adoption and created an inflow of issues that could not be resolved on a timely basis.

Similarly, the rapid adoption was not matched by the corresponding increase in the number of high-score questions on StackExchange, resulting in the negative coefficient.

Furthermore, it may be the case that the developers in the `Angular/React` (Javascript) domain may believe that a technology with more posts on StackExchange may be harder to use or more complex, unlike the developers in the data science domain where users expect a fair amount of esoteric and quirky functionality; thus, good questions on StackExchange are helpful and necessary even when faced with relatively common tasks. This suggests that further investigations are needed on how developers in different domains may react to 'exposure' on StackExchange.

These differences suggest that the social contagion models may need to be adjusted for cases where the adoption is primarily driven by the commercial involvement that may manifest itself in directly funding developers, providing better training, or even by dangling good job opportunities. Specifically, large companies, may create direct signals driving

adoption that do not propagate via social and technical networks and may lead to counter-intuitive values of the model coefficients.

The remaining variables represent the characteristics of the adopter, and cannot be directly compared between the studies. Interpreting them, however, is instructive for potential applications related to understanding the adopter base.

Specifically, while larger projects (measured by the number of commits **Cmts**) prefer `data.table`, presumably due to its stability and ability to handle larger data sets, in the User Interface domain, larger projects tend to use `React`, presumably because it is, according to opinions of knowledgeable developers we have consulted, designed to support large projects and applications. We had a separate predictor **C** indicating the need for performance in the R ecosystem, and, as expected, it had no effect in the Javascript domain where C language is not used to improve performance via native methods as is common for R domain.

From the perspective of the technical network **Prx2TD/RT**, the proximity to `tidy` increases the chances of adoption, presumably due to the rich functionality of the *tidyverse* framework that can be exploited for tasks other than simply doing data management. `data.table`, in contrast, does not have as wide an ecosystem of its own, thus proximity in the technical network does not appear to bring any distinct advantages. The case of `React` vs. `Angular` is unusual as the proximity of projects to each of these frameworks is highly correlated, $\rho = 0.9$. That is, if the proximity in the technical network is high for one of these frameworks it is also high for another. This may reflect the possibility that each of these frameworks are fairly comprehensive for the intended functionality and, therefore, the proximity is based on other JavaScript technologies that may, in fact, be more closely aligned with the `Angular` ecosystem. It is, therefore, important to note that the proximity in the technical network may not always fully reflect the actual interdependencies between the technologies and future work is needed to explore how general this relationship may be. Another possible scenario is that this negative relationship is simply an artifact indicating that `React`'s adoption may have been more strongly influenced by direct marketing signals (such as job opportunities) and the technical networks did not co-evolve at the same rate as the adoption.

From the perspective of social networks **AthPrx2TD/RT**, the proximity to `tidy` increases the chances of adoption, presumably due to the charismatic leadership of the founder and lead developer who also contributes to many other data science projects. The founder of `data.table`, in contrast, does not have as commanding a presence and social following in the community. The proximity to React within a social network increases the chances of adoption while the proximity to `Angular` decreases the adoption of `Angular`. There is some anecdotal evidence that React gets more positive testimonials from its users, see, e.g, "Is React killing Angular?" from Quora[15]. In such a case, knowing someone who actually uses `Angular` and can testify to the potential complexities and effort needed to use it fluently, may discourage adopters from trying it. If the `React` users are much more positive in their testimonial, this would explain the power of the social contagion model to discriminate between such differences.

Further considerations on the generalizability and other limitations of our approach are discussed in the next section.

5.7 Limitations

Empirical studies must be interpreted carefully due to a number of inherent limitations. Here we highlight some of the potential issues and how we tried to address them.

5.7.1 Limitations to Internal Validity

To obtain an unbiased, representative characterization of technology spread, we examined a very large collection of projects. While large, our sample however is not complete, as many projects do not publish their code and our data collection process may have missed even some of the public projects. The sample we have limits the findings of this study to projects that share their version control data on one of the many forges, such as GitHub, BitBucket, GitLab, Bioconductor, SourceForge, etc. However, our project repository may not be representative of the entire universe of projects, especially projects that do not publish their version control data.

[15] https://www.quora.com/Is-React-killing-Angular

We have selected only projects with extension [rR], but some older projects may use extension [sS], indicating the historic name for R language, or some other source code without any (known) extension. Regular expressions that we used to identify instances of package usage or installation can capture most of the install statement in the .r/R file, however, in some cases the install statement may be missed due to a dynamic specification in the installation such as in the case below,

```
ipak <- function(pkg){
    new.pkg <- pkg[!(pkg %in% installed.packages()[, "Package"])]
    if (length(new.pkg)) install.packages(new.pkg, dependencies = TRUE)
    sapply(pkg, require, character.only = TRUE)
}
# usage
packages <- c("ggplot2", "plyr", "reshape2", "RColorBrewer", "scales", "grid")
ipak(packages)
```

Also, multiple packages may be wrapped into a variable before calling the install function:

```
load.lib<-c("EIAdata", "gdata", ...,"stringr","XLConnect",
"xlsReadWrite","zipcode")
install.packages(lib,dependences=TRUE)
```

Moreover, regular expressions may occasionally falsely capture an install statement, e.g., install statements that are commented out may, in rare cases, be captured by regular expressions. Files that are contained in a project but not used may also contain installment statements that are captured by regular expressions. To alleviate this potential issue, we used the R language requirement to have a comment character '#' on each line and ensured that the matched install is never preceded by the comment character.

Another threat to validity is that the import of a package may not always indicate that the package was actually used. In some cases, a project may only contain package import statements without any calls to package API. For example, a developer may have dropped all actual API calls in the code without removing the corresponding package import statement. Alternatively, a developer may only add an import statement as a place holder for future

usage. One approach to mitigate such potential inaccuracies is to identify if any package APIs were called in the project. However, this approach may not cover all possible cases. One such scenario can be that multiple packages may contain one or several common APIs (e.g., the predict API in R exists in multiple packages) and it is generally not possible to automatically identify with precision which package an API belongs to.

We used keyword filtering on both tag field and on the text of the post itself to find the relevant posts from StackExchange dump. We emphasize that a manual inspection was needed as a followup step to ensure the relevance of the posts. While we did a manual inspection of our data, the approach we used may require a lot of manual effort to check the relevance if extremely popular technologies are investigated.

These potential limitations may affect the dependency networks we construct and result in an imprecise count of the number of projects using our two focal packages. Moreover, developer identities may not be consistent across our data sources, which may affect the author network [13]. We have tried to address these and other issues encountered when dealing with operational data from software repositories and big data in accordance with guidelines provided in the literature [82, 80, 46].

It is important to note that the particular operationalizations of the concepts from social contagion theory represent only one possible approach. Measures are not entirely orthogonal, i.e., each measure may capture the aspects of other dimensions beyond the one it is intended to measure. The correlations among predictors may lead to unstable models that are hard to interpret. We address this limitation by carefully considering various interpretations of the measures, conducting exploratory analyses of the obtained measures, selecting a subset that does not pose threats to model stability, and investigating compliance with model assumptions including inspection of outliers, non-homogeneous variance, and performing general model diagnostics. We also model the first choice, but it is also reasonable to model the full set of choices made. In the latter case, we would need to include the third option, i.e., projects choosing both packages: `tidy` and `data.table`. We fitted a variety of alternatives models to ensure that the reported results are not affected by these variations in the approach. We only present here the results for two alternatives due to space considerations, but we have applied our choice model to several other R packages as well.

5.7.2 Limitations to External Validity

We demonstrate how to use social contagion modeling with version control data to evaluate developer behaviour when choosing software packages. The particular results we obtained for R and the two focal packages may not, therefore, generalize beyond this specific context. We evaluated the generalizability of the results in the JavaScript domain in section 5.6 and found some variations to the finding in R ecosystem in JavaScript ecosystem. The framework we provided, however, allows future researches to investigate the nuances of developer behaviour in much greater detail and apply it to other contexts.

5.8 Literature Review

The closest related work involves studies of use and migration of software libraries. A number of metrics and approaches were proposed to mine and explore usage and migration trends. A software library encapsulates certain functionality that is then used by applications (or other libraries). The application may benefit from extra functionality or performance in the new libraries that may be created later, but switching to a new library (library migration) involves some recoding of the application[57, 78, 22, 67, 86]. Most prior work, therefore, focused on costs and benefits of library migration [60, 106, 27, 105, 9, 109, 26, 77]. Similarly to that work we ask why developers chose a new library. In contrast to prior work, we construct new predictors of adoption (e.g., technical and author dependency networks, breadth of deployment, exposure of techniques on StackExchange, quality of support measured through issue number and response times) that are based on sound theoretical foundations and we use choice models to understand how macro trends at the scale of the entire SSC emerge from actual decisions the individual developers make to select a specific software technology.

Approaches to detect library usage include issue report analysis [60]. As in prior work we detect usage by searching for library statements in source files of projects [106]. De la Mora *et al.* [27] introduce an interface to help developers choose among the libraries by displaying their popularity, release frequency, and recency. While building on this research, we add novel network, deployment, and quality measures that would inform developer choice. More importantly, we radically improve the ability of developers' to make informed decisions by

providing a statistical model that explains which of these measures matter and how they affect the choice.

Prior studies that examined technology choices have used a variety of approaches ranging from surveying developer preferences [5] and reasons [119] behind, to mining version control and issue tracking repositories [60, 106, 27]. Similarly, we mine version control data, but at a larger scale of all projects with public version control data that include R language files. This allows us to construct complete software supply chains that depict end-to-end technical and social dependencies.

5.9 Conclusions

Integrating software supply chain concepts and models to operationalize key variables from social contagion theory to investigate software technology adoption appears to have provided a number of potentially useful insights in the present case study of two data manipulation technologies within R language. More specifically, the methodology was able to identify factors that were influential in decision-makers' choices between software technologies and demonstrate the need to account, not only for the properties of the choice, but also of the chooser and of the importance of the supply chain dependencies and information flows. It also validates the measures deemed to be the drivers of technology adoption by the social contagion theory.

This study introduces the concept of two types of software supply chains (based on technical dependencies and on the relationships among developers induced by projects they have worked on) and demonstrates how software supply chains for the entire open source ecosystem can be reconstructed as they have existed at any point in the past from public version control systems. Additionally, by taking a social contagion perspective and employing the logistic regression models, we explicate a parsimonious model that is capable of modeling software technology choices. The findings of this study have wide reaching implications for the software engineering community as well as those who study traditional supply chains. For example, the ability to model and understand which aspects of a network of software supply chain or physical supply chain partners and affiliates influence uptake and spread of a

given artifact (e.g., technology or product) might help contributors adjust their contributions in a way to maximize their reach, while also extending the viability and propagation of a core technology or product. This notion is consistent with our findings that a number of characteristics of a developer and properties of technology are found to be important in the choice between major alternatives. More specifically, technologies with large number of overall adopters, higher responsiveness to new issues, and more high-score stack exchange questions are more likely to be chosen. Furthermore, from the perspective of a project's decision-makers, their technical features and proximity to a technology in both the technical dependency network and author collaboration network increase the probability of adoption. On a more speculative side, we find that half of the significant predictors do not appear to be related to a traditional rational choice, but are likely a reflection of social and cognitive biases or, in plain language, shortcuts people take. Developers, at least in the context of technical decisions regarding which technology to use, do not appear to be immune from these biases.

Source code and data for this study is publicly available [16] to facilitate reproducibility and wider adoption of the proposed methodology.

Acknowledgment

This work was supported by the National Science Foundation NSF Award IIS-1633437.

[16] https://drive.google.com/drive/folders/1YjC31l5NrD5XzI5ZyxtRLF29OM2owb1X?usp=sharing

Chapter 6

Overview, Discussion and Conclusions

6.1 Overview

The objective of this work "Software Supply Chain Development and Application" is to propose a novel approach to understand the development of software, especially OSSs, reveal the complicated interactions and relations among software developers distributed all of the world, capture the entirety of OSS communities, so as to mitigate the risks in software development.

We integrated the concept of supply chains with OSS development and leveraged the existing knowledge on traditional supply chains, big data, and data science that lead to a more comprehensive understanding, the formulation of new research problems, and practical applications. We started by defining SSC and their types for OSS, outline ways to construct valid SSCs, and give examples of new insights, research questions, and applications of this approach.

We proposed a prototype of an updatable and expandable infrastructure, WoC, to support research and tools that rely on version control data from the entirety of open source projects, and discuss how we address some of the data scale and quality challenges related to data discovery, retrieval, and storage. We enable wide data access to collected data source by providing a tool built on top of the infrastructure, which scales well with completion to query in linear time. Furthermore, we implemented ways to make this large dataset usable for a number of research tasks by doing targeted data correction and augmentation and by

creating data structures derived from the raw data that permit accomplishing these research tasks quickly, despite the vastness of the underlying data.

We reported various researches and applications in software domain that are enabled and enhanced by leveraging the joining power from SSC concept and WoC infrastructure. In particular, we looked into the phenomenon of software technologies spread, and investigated what combination of attributes are driving the adoption of a particular software technology, and hope that developers seeking to increase the adoption rate of their products can benefit from our findings.

6.2 Discussion: Primary Findings

In this section we discuss the primary findings of this thesis by answering each research question.

1. **(RQ1) How to define SSC?** After an exploration of existing SSC related works, we found that there was no definition of SSC contextualized for OSS where it may help with distributed decision making, and there was no measurable definition of SSC. We define SSC by making analog of components in traditional SC. SSC has developers and groups (companies), corporate backers supporting these developers or groups ("financing"), relationships among software projects or packages representing the "chain" of the flow, and changes to source code (e.g. files, modules, frameworks, or entire distributions) representing products or information.

2. **(RQ2) How to measure SSC for OSS?** In order to measure SSC, we proposed and created WoC, an infrastructure for mining OSS development data in large from various open source platforms, which loosely follows the microservices architecture [85], where the design and performance of the loosely coupled components can be independently evaluated, each service can utilize a database that is optimal for its needs, and the most computationally-intensive components are extremely portable to ensure they run on any high-performance platform. Our engineering principles are focused on using the simplest possible techniques and components for each specific task ranging from

project discovery to fitting large-scale models. Our current WoC implementation is capable of being updated on a monthly basis and contains over 24B git objects. Various APIs and sample guidance are published to meet users with different backgrounds and preferences.

3. **(RQ3) What can we learn about SSC in OSS?** After various researches and applications being successfully implemented by our team and researchers outside, we conclude that SSC and WoC enable and support both domain focused researches and crossing-ecosystem ones. We also developed various approaches to operationalize SSC networks to represent and reveal the relationships among software developers and projects in OSS community.

4. **(RQ4) How can we reduce the risks through SSC in OSS?** To exemplify the value od SSC in mitigating risks in OSSs, we looked into the phenomenon of software technology adoption among developers and investigated what combination of attributes drives the adoption of a particular software technology. After investigating two competing technologies (data.table Vs.tidy) in R, We found that a quick response to raised issues, a larger number of overall deployments, and a larger number of high-score StackExchange questions are associated with higher adoption. Decision makers tend to adopt the technology that is closer to them in the technical dependency network and in author collaborations networks while meeting their performance needs. We further extended our method on two JavaScript frameworks (React Vs. Angular) and achieved a similar result. Based on these findings, we provide suggestions for developers on how to mitigate risks of abandonment from the perspective of a user downstream and the risk of low adoption from the perspective of the producer upstream.

6.3 Contributions

The concept of software supply chain provides a unique perspective that helps understand the highly interconnected nature of open source software and leads to new research questions and practical applications. The key commodity of the OSS is the actions and effort of developers

that, if applied in proper amounts, to the right projects, by skilled developers, and in a timely manner, would make OSS more predictable and, in turn, attract more innovation. SSCs define the relationships among the projects and developers that are necessary for this vision to succeed. Ways to measure the SSCs would increase the transparency of the activities and qualities of projects and people, and the global coverage and the ability to match producers and consumers would increase the visibility for the open source users on where to source their knowledge or software and for producers it would increase the visibility of users and their needs.

We summarize our contributions in the following:

Firstly, we developed an approach (SSC) to enable systematic analysis of open source community, revealing underlying complicated relations among software projects and developers. As the transparency and visibility increase in SSC, software developers are enabled to have a more comprehensive knowledge and estimation of other software projects, make wise decisions when choosing from other software projects to integrate (as downstream components) through the evaluation of a combination of developer reputation and software project maintenance. Furthermore, risks in software development can be mitigated by an early detection of bug/vulnerability propagation in source code snippets (code reuse network), downstream packages (dependency network), and submissions from inexperienced or malware developers (knowledge flow/authorship network).

Secondly, we built an infrastructure(WoC) to provide broad data access and facilitate SSC network construction and related analysis. We enable wide data access to collected data source by providing a tool built on top of the infrastructure, which scales well with completion to query in linear time. Furthermore, we implement ways to make this large dataset usable for a number of research tasks by implementing targeted data correction and augmentation and by creating data structures derived from the raw data that permit accomplishing these research tasks quickly, despite the vastness of the underlying data. In a nutshell, WoC can provide support for diverse research tasks that would be otherwise out of reach for most researchers. Its focus on global properties of all public source code will enable research that could not be previously done and help to address highly relevant challenges of open source ecosystem sustainability and of risks posed by this global software supply chain.

Lastly, by leveraging SSC networks, we investigated the adoption of technologies in software domain, discovered factors that are influential to decision makers and provide suggestions to software developers on making one's product popular. The contribution consists of proposing a method to explain and predict the spread of technologies, to suggest which technologies are more likely to spread in the future, and suggest steps that developers could take to make the technologies they produce more popular. Developers can, therefore, reduce risks by choosing technology that is likely to be widely adopted. The supporters of open source software could use such information to focus on and properly allocate limited resources on projects that either need help or are likely to become a popular infrastructure. In essence, our approach unveils previously unknown critical aspects of technology spread and, through that, makes developers, organizations, and communities more effective.

Bibliography

[1] Abadi, D. J. (2009). Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, 32(1):3–12.

[2] Agrawal, S., Narasayya, V., and Yang, B. (2004). Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM.

[3] Amreen, S., Mockus, A., Bogart, C., Zhang, Y., and Zaretzki, R. (2019). Alfaa: Active learning fingerprint based anti-aliasing for correcting developer identity errors in version control data. *arXiv preprint arXiv:1901.03363*.

[4] Amreen, S., Mockus, A., Zaretzki, R., Bogart, C., and Zhang, Y. (2020). Alfaa: Active learning fingerprint based anti-aliasing for correcting developer identity errors in version control systems. *Empirical Software Engineering*, pages 1–32.

[5] Anderson, E. E. (1990). Choice models for the evaluation and selection of software packages. *Journal of Management Information Systems*, 6(4):123–138.

[6] Angst, C. M., Agarwal, R., Sambamurthy, V., and Kelley, K. (2010). Social contagion and information technology diffusion: the adoption of electronic medical records in us hospitals. *Management Science*, 56(8):1219–1241.

[7] Bajracharya, S., Ossher, J., and Lopes, C. (2014). Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79:241–259.

[8] Ballou, R. H. (2007). *Business logistics/supply chain management: planning, organizing, and controlling the supply chain*. Pearson Education India.

[9] Bartolomei, T. T., Czarnecki, K., Lämmel, R., and van der Storm, T. (2010). Study of an api migration for two xml apis. In van den Brand, M., Gašević, D., and Gray, J., editors, *Software Language Engineering*, pages 42–61, Berlin, Heidelberg. Springer Berlin Heidelberg.

[10] Bass, F. M. (2004). A new product growth for model consumer durables. *Manage. Sci.*, 50(12 Supplement):1825–1832, doi:10.1287/mnsc.1040.0264.

[11] Berry, S. T. (1994). Estimating discrete-choice models of product differentiation. *The RAND Journal of Economics*, 25(2):242–262.

[12] Bevan, J., Whitehead Jr, E. J., Kim, S., and Godfrey, M. (2005). Facilitating software evolution research with kenyon. *ACM SIGSOFT software engineering notes*, 30(5):177–186.

[13] Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A. (2006). Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 137–143, New York, NY, USA. ACM.

[14] Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., and Devanbu, P. (2009). The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10. IEEE.

[15] Budde, R., Kautz, K., Kuhlenkamp, K., and Züllighoven, H. (1992). Prototyping. In *Prototyping*, pages 33–46. Springer.

[16] Burt, R. S. (1987). Social contagion and innovation: Cohesion versus structural equivalence. *American Journal of Sociology*, 92(6):1287–1335, doi:10.1086/228667.

[17] Capra, E., Francalanci, C., and Merlo, F. (2008). An empirical study on the relationship between software design quality, development effort and governance in open source projects. *IEEE Transactions on Software Engineering*, 34(6):765–782.

[18] Chacon, S. and Straub, B. (2014). *Pro git*. Apress.

[19] Chhajed, A. A. and Xu, S. H. (2005). Software focused supply chains: Challenges and issues. In *Industrial Informatics, 2005. INDIN'05. 2005 3rd IEEE International Conference on*, pages 172–175. IEEE.

[20] Christopher, M. and Lee, H. (2004). Mitigating supply chain risk through improved confidence. *International Journal of Physical Distribution & Logistics Management*, 34(5):388–396, doi:10.1108/09600030410545436.

[21] Coelho, J., Valente, M. T., Silva, L. L., and Shihab, E. (2018). Identifying unmaintained projects in github. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM.

[22] Cossette, B. E. and Walker, R. J. (2012). Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 55:1–55:11, New York, NY, USA. ACM.

[23] Croissant, Y. (2013). *mlogit: multinomial logit model*. R package version 0.2-4.

[24] Czerwonka, J., Nagappan, N., Schulte, W., and Murphy, B. (2013). Codemine: Building a software development data analytics platform at microsoft. *IEEE software*, 30(4):64–71.

[25] Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. (2012). 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. ACM.

[26] Dagenais, B. and Robillard, M. P. (2009). Semdiff: Analysis and recommendation support for api evolution. In *2009 IEEE 31st International Conference on Software Engineering*, pages 599–602.

[27] de la Mora, F. L. and Nadi, S. (2018). Which library should i use?: A metric-based comparison of software libraries. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, pages 37–40, New York, NY, USA. ACM.

[28] Dey, T., Ma, Y., and Mockus, A. (2019). Patterns of effort contribution and demand and user classification based on participation patterns in npm ecosystem. *arXiv preprint arXiv:1907.06538*.

[29] Dey, T. and Mockus, A. (2018). Are software dependency supply chain metrics useful in predicting change of popularity of npm packages? In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 66–69. ACM.

[30] Di Cosmo, R. and Zacchiroli, S. (2017). Software heritage: Whyandhowtopreservesoftwaresourcecode. ipres 2017.

[31] DiMaggio, P. J. and Powell, W. W. (1983). The iron cage revisited: Institutional isomorphism and collective rationality in organizational fields. *American Sociological Review*, 48(2):147–160.

[32] Ducasse, S., Gîrba, T., and Nierstrasz, O. (2005). Moose: an agile reengineering environment. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–102.

[33] Dyer, R. (2013). Task fusion: Improving utilization of multi-user clusters. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, SPLASH SRC, pages 117–118.

[34] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013a). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 422–431.

[35] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2015a). Boa: an enabling language and infrastructure for ultra-large scale msr studies. *The Art and Science of Analyzing Software Data*, pages 593–621.

[36] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2015b). Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34.

[37] Dyer, R., Rajan, H., and Nguyen, T. N. (2013b). Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes. In *Proceedings of the 12th*

International Conference on Generative Programming: Concepts & Experiences, GPCE, pages 23–32.

[38] Eastlake, D. and Jones, P. (2001). Us secure hash algorithm 1 (sha1).

[39] Eghbal, N. (2016). *Roads and bridges: The unseen labor behind our digital infrastructure*. Ford Foundation.

[40] Ellison, R. J. and Woody, C. (2010). Supply-chain risk management: Incorporating security into software development. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–10. IEEE.

[41] Farbey, B. and Finkelstein, A. (1999). Exploiting software supply chain business architecture: a research agenda. In *1st Workshop on Economics-Driven SoftwareEngineering Research (EDSER-1), 21st InternationalConference on Software Engineering*.

[42] Fichman, R. G. (2004). Going beyond the dominant paradigm for information technology innovation research: Emerging concepts and methods. *Journal of the association for information systems*, 5(8):11.

[43] German, D. and Mockus, A. (2003). Automating the measurement of open source projects. In *Proceedings of the 3rd workshop on open source software engineering*, pages 63–67. University College Cork Cork Ireland.

[44] Gilbride, T. J. and Allenby, G. M. (2004). A choice model with conjunctive, disjunctive, and compensatory screening rules. *Marketing Science*, 23(3):391–406, doi:10.1287/mksc.1030.0032.

[45] Gorton, I., Bener, A. B., and Mockus, A. (2016a). Software engineering for big data systems. *IEEE Software*, 33(2):32–35.

[46] Gorton, I., Bener, A. B., and Mockus, A. (2016b). Software engineering for big data systems. *IEEE Softw.*, 33(2):32–35, doi:10.1109/MS.2016.47.

[47] Gousios, G. (2013). The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA. IEEE Press.

[48] Gousios, G., Pinzger, M., and Deursen, A. v. (2014a). An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM.

[49] Gousios, G. and Spinellis, D. (2009). Alitheia core: An extensible software quality monitoring platform. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 579–582. IEEE.

[50] Gousios, G. and Spinellis, D. (2012). Ghtorrent: Github's data from a firehose. In *Mining software repositories (msr), 2012 9th ieee working conference on*, pages 12–21. IEEE.

[51] Gousios, G., Vasilescu, B., Serebrenik, A., and Zaidman, A. (2014b). Lean ghtorrent: Github data on demand. In *Proceedings of the 11th working conference on mining software repositories*, pages 384–387. ACM.

[52] Gousios, G. and Zaidman, A. (2014). A dataset for pull-based development research. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 368–371. ACM.

[53] Greenfield, J. and Short, K. (2003). Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27. ACM.

[54] Hackbarth, R., Mockus, A., Palframan, J., and Weiss, D. (2010). Assessing the state of software in a large enterprise. *Journal of Empirical Software Engineering*, 10(3):219–249.

[55] Hausman, J. A., Leonard, G. K., and McFadden, D. (1995). A utility-consistent, combined discrete choice and count data model assessing recreational use losses

due to natural resource damage. *Journal of Public Economics*, 56(1):1 – 30, doi:https://doi.org/10.1016/0047-2727(93)01415-7.

[56] Holdsworth, J. (1995). *Software Process Design*. McGraw-Hill, Inc.

[57] Hora, A. and Valente, M. T. (2015). Apiwave: Keeping track of api popularity and migration. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, pages 321–323, Washington, DC, USA. IEEE Computer Society.

[58] Howison, J., Conklin, M., and Crowston, K. (2006). Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information Technology and Web Engineering (IJITWE)*, 1(3):17–26.

[59] Huang, S. H., Uppal, M., and Shi, J. (2002). A product driven approach to manufacturing supply chain selection. *Supply Chain Management: An International Journal*, 7(4):189–199, doi:10.1108/13598540210438935.

[60] Kabinna, S., Bezemer, C.-P., Shang, W., and Hassan, A. E. (2016). Logging library migrations: A case study for the apache software foundation projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 154–164, New York, NY, USA. ACM.

[61] Kalish, S. (1985). A new product adoption model with price, advertising, and uncertainty. *Management Science*, 31(12):1569–1585.

[62] Kamakura, W. A. and Russell, G. J. (1989). A probabilistic choice model for market segmentation and elasticity structure. *Journal of Marketing Research*, 26(4):379–390.

[63] Khomh, F., Dhaliwal, T., Zou, Y., and Adams, B. (2012). Do faster releases improve software quality?: an empirical case study of mozilla firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 179–188. IEEE Press.

[64] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. (2005). An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held*

jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pages 187–196.

[65] Kim, S., Zimmermann, T., Kim, M., Hassan, A. E., Mockus, A., Gîrba, T., Pinzger, M., Jr., E. J. W., and Zeller, A. (2006). Ta-re: an exchange language for mining software repositories. In *ICSE'06 Workshop on Mining Software Repositories*, pages 22–25, Shanghai, China.

[66] Kogut, B. and Metiu, A. (2001). Open-source software development and distributed innovation. *Oxford Review of Economic Policy*, 17(2):248–264, doi:10.1093/oxrep/17.2.248.

[67] Lämmel, R., Pek, E., and Starek, J. (2011). Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1317–1324, New York, NY, USA. ACM.

[68] Le, Q. and Mikolov, T. (2014). Distributed representation of sentences and documents. In *Proceedings of the 31 st International Conference on Machine Learning*, volume 32, Beijing,China. JMLR.

[69] Leavitt, N. (2010). Will nosql databases live up to their promise? *Computer*, 43(2).

[70] Levy, E. (2003). Poisoning the software supply chain. *Security & Privacy, IEEE*, 1(3):70–73.

[71] Lichter, H., Schneider-Hufschmidt, M., and Zullighoven, H. (1994). Prototyping in industrial software projects-bridging the gap between theory and practice. *IEEE transactions on software engineering*, 20(11):825–832.

[72] Luhn, H. P. (1958). A business intelligence system. *IBM Journal of research and development*, 2(4):314–319.

[73] Ma, Y., Bogart, C., Amreen, S., Zaretzki, R., and Mockus, A. (2019). World of code: An infrastructure for mining the universe of open source vcs data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 143–154. IEEE.

[74] Ma, Y., Dey, T., Smith, J. M., Wilder, N., and Mockus, A. (2016). Crowdsourcing the discovery of software repositories in an educational environment. *PeerJ Preprints*, 4:e2551v1.

[75] McFadden, D. et al. (1973). Conditional logit analysis of qualitative choice behavior.

[76] McFadden, D. and Train, K. (2000). Mixed mnl models for discrete response. *Journal of Applied Econometrics*, 15(5):447–470, doi:10.1002/1099-1255(200009/10)15:5¡447::AID-JAE570¿3.0.CO;2-1.

[77] Mileva, Y. M., Dallmeier, V., Burger, M., and Zeller, A. (2009). Mining trends of library usage. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 57–62, New York, NY, USA. ACM.

[78] Mileva, Y. M., Dallmeier, V., and Zeller, A. (2010). Mining api popularity. In *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*, TAIC PART'10, pages 173–180, Berlin, Heidelberg. Springer-Verlag.

[79] Mockus, A. (2007a). Large-scale code reuse in open source software. In *ICSE'07 Intl. Workshop on Emerging Trends in FLOSS Research and Development*, Minneapolis, Minnesota.

[80] Mockus, A. (2007b). Software support tools and experimental work. In *Empirical Software Engineering Issues. Critical Assessment and Future Directions*, pages 91–99. Springer.

[81] Mockus, A. (2009). Amassing and indexing a large sample of version control systems: towards the census of public source code history. In *6th IEEE Working Conference on Mining Software Repositories*.

[82] Mockus, A. (2014). Engineering big data solutions. In *ICSE'14 FOSE*.

[83] Moniruzzaman, A. and Hossain, S. A. (2013). Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*.

[84] Munaiah, N., Kroh, S., Cabrey, C., and Nagappan, M. (2017). Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253.

[85] Newman, S. (2015). *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

[86] Nguyen, H. A., Nguyen, T. T., Wilson, Jr., G., Nguyen, A. T., Kim, M., and Nguyen, T. N. (2010). A graph-based approach to api usage adaptation. *SIGPLAN Not.*, 45(10):302–321, doi:10.1145/1932682.1869486.

[87] Oliver, R. K., Webber, M. D., et al. (1982). Supply-chain management: logistics catches up with strategy. *Outlook*, 5(1):42–47.

[88] Ossher, J., Bajracharya, S., Linstead, E., Baldi, P., and Lopes, C. (2009). Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 183–186. IEEE.

[89] Rajan, H., Nguyen, T. N., Dyer, R., and Nguyen, H. A. (2015). Boa website. http://boa.cs.iastate.edu/.

[90] Reber, A. S. (1989). Implicit learning and tacit knowledge. *Journal of experimental psychology: General*, 118(3):219.

[91] Robert B. Handfield, E. L. N. (1999). *Introduction to supply chain management*. New York: Prentice-Hall.

[92] Rogers, E. M. (1995). Innovation in organizations. *Diffusion of innovations*, 4:371–404.

[93] Rosch, E. (1999). Principles of categorization. *Concepts: core readings*, 189.

[94] Rosen, L. (2004). *Open source licensing: Software freedom and intellectual property law*. Prentice Hall PTR.

[95] Rozenberg, D., Beschastnikh, I., Kosmale, F., Poser, V., Becker, H., Palyart, M., and Murphy, G. C. (2016). Comparing repositories visually with repograms. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 109–120. ACM.

[96] Russell, D. M. and Hoag, A. M. (2004). People and information technology in the supply chain: Social and organizational influences on adoption. *International Journal of Physical Distribution & Logistics Management*, 34(2):102–122.

[97] Russom, P. et al. (2011). Big data analytics. *TDWI best practices report, fourth quarter*, 19(4):1–34.

[98] Samadi, M., Nikolaev, A., and Nagi, R. (2016). A subjective evidence model for influence maximization in social networks. *Omega*, 59:263 – 278, doi:https://doi.org/10.1016/j.omega.2015.06.014.

[99] Sayyad Shirabad, J. and Menzies, T. (2005). The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada.

[100] Small, K. and Rosen, H. (1981). Applied welfare economics with discrete choice models. *Econometrica*, 49(1):105–30.

[101] Spinellis, D., Kotti, Z., and Mockus, A. (2020). A dataset for github repository deduplication. *arXiv preprint arXiv:2002.02314*.

[102] T, T. (2015). *survival: A Package for Survival Analysis in S*. R package version 2.38.

[103] Talluri, K. and van Ryzin, G. (2004). Revenue management under a general discrete choice model of consumer behavior. *Manage. Sci.*, 50(1):15–33, doi:10.1287/mnsc.1030.0147.

[104] Tan, K.-C., Kannan, V. R., Handfield, R. B., and Ghosh, S. (1999). Supply chain management: an empirical study of its impact on performance. *International journal of operations & production Management*, 19(10):1034–1052.

[105] Teyton, C., Falleri, J.-R., and Blanc, X. (2012). Mining library migration graphs. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pages 289–298.

[106] Teyton, C., Falleri, J.-R., Palyart, M., and Blanc, X. (2014). A study of library migrations in java. *Journal of Software: Evolution and Process*, 26(11):1030–1052.

[107] Tiwari, N. M., Upadhyaya, G., Nguyen, H. A., and Rajan, H. (2017). Candoia: A platform for building and sharing mining software repositories tools as apps. In *MSR'17: 14th International Conference on Mining Software Repositories*.

[108] Tiwari, N. M., Upadhyaya, G., and Rajan, H. (2016). Candoia: A platform and ecosystem for mining software repositories tools. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 759–764. ACM.

[109] Tonelli, T., Krzysztof, and Ralf (2010). Swing to swt and back: Patterns for api migration by wrapping. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10.

[110] Tsay, J., Dabbish, L., and Herbsleb, J. (2014). 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. ACM.

[111] Tu, Q. et al. (2000). Evolution in open source software: A case study. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 131–142. IEEE.

[112] Upadhyaya, G. and Rajan, H. (2017). On accelerating ultra-large-scale mining. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, pages 39–42. IEEE Press.

[113] Upadhyaya, G. and Rajan, H. (2018). On accelerating source code analysis at massive scale. *IEEE Transactions on Software Engineering*.

[114] Von Hippel, E. (2001). Innovation by user communities: Learning from open-source software. *MIT Sloan management review*, 42(4):82.

[115] von Krogh, G., Spaeth, S., and Lakhani, K. R. (2003). Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217 – 1241, doi:https://doi.org/10.1016/S0048-7333(03)00050-7. Open Source Software Development.

[116] West, J. and Gallagher, S. (2006). Patterns of open innovation in open source software. *Open Innovation: researching a new paradigm*, 235(11).

[117] Winkler, W. E. (1990). String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage.

[118] Winkler, W. E. (2006). Overview of record linkage and current research directions. Technical report, BUREAU OF THE CENSUS.

[119] Xiao, S., Witschey, J., and Murphy-Hill, E. (2014). Social influences on secure development tool adoption: why security tools spread. In *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*, pages 1095–1106. ACM.

Appendices

A List of Publications

This dissertation is composed largely based on the following published works:

1. Yuxing Ma, "Constructing supply chains in open source software", *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion* , pp. 458-459.

2. Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki and Audris Mockus, "World of code: An infrastructure for mining the universe of open source vcs data", *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 143-154.

3. Yuxing Ma, Audris Mockus, Russel Zaretzki, Randy Bradley and Bogdan Bichescu, "A Methodology for Analyzing Uptake of Software Technologies Among Developers", *2020 IEEE Transactions on Software Engineering.*

Other publications:

1. Amreen Sadika, Bogdan Bichescu, Randy Bradley, Tapajit Dey, Yuxing Ma, Audris Mockus, Sara Mousavi, and Russell Zaretzki, "A Methodology for Measuring FLOSS Ecosystems", *in Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability*, pp. 1-29. Springer, Singapore, 2019.

2. Dey, Tapajit, Yuxing Ma, and Audris Mockus. "Patterns of effort contribution and demand and user classification based on participation patterns in npm ecosystem", *in Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 36-45. 2019.

3. Yuxing Ma, Tapajit Dey, and Audris Mockus, 'POSITION PAPER: Modularizing Global Variable In Climate Simulation Software", *2016 IEEE/ACM International Workshop on Software Engineering for Science (SE4Science)*, pp. 8-11.

4. Yuxing Ma, Tapajit Dey, Jarred M. Smith, Nathan Wilder, and Audris Mockus, "Crowdsourcing the discovery of software repositories in an educational environment", PeerJ Preprints 4 (2016): e2551v1.

B Source Code for Cmt2ATShow.perl

```
#!/usr/bin/perl -I /home/audris/lib64/perl5 -I /home/audris/lib/x86_64-linux-gnu/
    perl
use strict;
use warnings;
use Error qw(:try);
use TokyoCabinet;
use Compress::LZF;

sub toHex {
  return unpack "H*", $_[0];
}
sub fromHex {
  return pack "H*", $_[0];
}

my $split = 1;
$split = $ARGV[1] + 0 if defined $ARGV[1];

my %c2at;
for my $sec (0..($split-1)){
  my $fname = "$ARGV[0].$sec.tch";
  $fname = $ARGV[0] if ($split == 1);
  tie %{$c2at{$sec}}, "TokyoCabinet::HDB", "$fname", TokyoCabinet::HDB::OREADER,
        16777213, -1, -1, TokyoCabinet::TDB::TLARGE, 100000
      or die "cant open $fname\n";
}

while (<STDIN>){
  chop ();
  my $c = fromHex($_);
```

```
30   my $ss = pack 'H*', substr ($_, 0, 2);
31   my $sec = (unpack "C", $ss)%$split;
32   if (defined $c2at{$sec}{$c}) {
33     my ($time, $author) = split(/;/, $c2at{$sec}{$c});
34     my @parts = localtime($time);
35     my $year= $parts[5] + 1900;
36     print $year.";".$author."\n";
37   }
38 }
39 for my $sec (0..($split-1)){
40   untie %{$c2at{$sec}};
41 }
```

C Source Code for the custom `lsort` command in tutorial

```
#!/bin/bash
export LC_ALL=C
export LANG=C
sz=${1:-10G}
shift
sort -T. -S $sz --compress-program=gzip $@
```

Vita

Yuxing Ma was born in Hejin, China. Yuxing attended Beijing Institute of Technology to study electronics engineering, and graduated with a B.S. in 2012. He further pursued a study in computer science domain and achieved a M.S. in 2015 from North China Computer System Engineering Institute. In 2015, he moved to the United States and joined the research lab of Dr. Audris mockus to pursue a Ph.D. in Computer Science at the University of Tennessee, Knoxville. At a late period in PhD, he achieved an opportunity to work in IBM as a software engineer intern for data pipeline construction and maintenance.

His research interests include: open source software engineering, software ecosystem evolution, software supply chain.