No Free Lunch For Programmers: Digital Supply Chains and the Economics of Software Dependency Management

Sam Boysel[*]

March 2, 2023

Abstract

Developers of software projects can leverage the functionality of existing open source projects. This practice can potentially lower the cost of development albeit at the inherent risk of relying on external components. A "downstream" project maintainer can choose to "import" elements of an "upstream" project to outsource functionality, but is uncertain how future changes in this dependency project may expose her own project to software faults or vulnerabilities. Software dependency networks therefore represent a "digital supply chain", an ecosystem of interdependent public goods that confer an intricate set of both positive and negative externalities for project maintainers and end users. Focusing on microeconomic fundamentals of the dependency management problem faced by the risk averse project maintainer, we use both reduced form and structural approaches to study how dependency networks create value, what forces shape their formation, and how individual behavior can influence the robustness of equilibrium network structure. We use a sample of open source software projects from the Node.js JavaScript packaging ecosystem for which contribution and dependency formation decisions are observed in real-time. Finally, we consider several policy interventions that can improve equilibrium welfare. In particular, we find that removing less that 1% of core projects can reduce aggregate project quality by more than 5% for the remaining peers.

[*]boysel@usc.edu

Contents

List of Figures

List of Tables

1 Introduction

Modern software typically borrows 70 to 90% of functionality from free and open source software (FOSS) projects (Nagle et al., 2022). The use of external software components can significantly lower development costs, reduces the need to "reinvent the wheel", and allows specialized code to be organized into modular packages.[1] Relationships between projects within this ecosystem are known as *software dependency networks*, structures akin to "digital supply chains" in which any number of downstream dependents can costlessly share functionality served from an upstream dependency (Eghbal, 2016).[2] Given the nature in which modern software services are produced and deployed, it is important to note that dependent software components can be affected contemporaneously by the dependencies they borrow from.[3] For example, the maintainer of an upstream dependency project may introduce a change that is backwards incompatible for downstream dependents, forcing the downstream maintainer to expend development effort to maintain functionality. In more serious cases, upstream changes may even introduce faults or exploits that can affect the operation and security of downstream dependents (Ohm et al., 2020). Therefore, a central economic question in this setting is how the maintainer of a given software project must balance the benefits of development expedience offered by using existing codebases against the risk introduced by relying on a network of potentially problematic external dependencies.[4]

While attractive for efficiently building complex projects, the hidden costs of using software dependencies can range from mild maintenance costs[5] to catastrophic risk for downstream applications and end users. In practice, software developers are said to spend roughly as much time managing their code and dependencies as they do writing new features (Grams, 2019). Digital supply chain risk is not just a problem for the maintainers of software projects. Some famous cases demonstrate

[1]To paraphrase the Unix philosophy espoused by Ken Thompson, "Make each program do one thing well." (McIlroy et al., 1978). See Lerner and Tirole (2002) and Baldwin and Clark (2006) for discussions of the economics of software modularity.

[2]For example, a visual representation of the dependency network for a sample of Node.js JavaScript projects can be seen at `https://graphcommons.com/graphs/a7ec343d-2a0c-47bb-9658-bb8315e8a096`.

[3]An overview of how services in modern software ecosystems are deployed and maintained can be found in Boldi and Gousios (2020). Importantly, increased uptake of OSS components by users makes them attractive targets for malicious actors (Ladisa et al., 2022).

[4]To paraphrase DeVault (2021), there is "no free lunch" for the maintainers of software.

[5]A quote from an anonymous developer of the Eclipse integrated development environment (IDE) for Java: "I only depend on things that are really worthwhile. Because basically everything that you depend on is going to give you pain every so often. And that's inevitable." (Decan et al., 2019)

the intricate and pervasive nature of open software components and how faults or changes can have widespread and costly impacts across dependent userbases.[6] In March 2016, the maintainer of Node.js package `left-pad` abruptly removed the package from the `npm` package registry[7], making the software unavailable to thousands of downstream dependents (Schlueter, 2016). This led to over 2% of all `npm` packages failing to operate properly until maintainers could replace the missing functionality. While the package itself was only 17 single lines of code and was replaced immediately, the aftermath of this abrupt removal begins to highlight the extent to which developers have come to rely on the availability of open code. In April 2014, a Google engineer reported an exploit that became known as the Heartbleed[8] bug in the source code of the OpenSSL library used for encryption, potentially exposing sensitive user information across an estimated 17% of public web servers (Mutton, 2014). Similarly, a fix was issued the same day the exploit was reported but hundreds of thousands of unpatched servers remained vulnerable as late as 2017, five years after the vulnerability had been introduced into the codebase (Carey, 2017).[9][10] In September 2017, Equifax publicly announced a vulnerability stemming from their use of the Apache Struts website framework beginning in May 2017, exposing private records of over 147 million users (US CFPB, 2022). The company agreed to a settlement with the Federal Trade Commission and the Consumer Financial Protection Bureau that entitled compromised users of the service up to $425 million USD in restitution (US FTC, 2022). In general, the average cost of a data breach in 2021 was estimated to be $4.24 million USD (IBM, 2021). Together, these case studies illustrate the scope of vulnerability under which technological services reliant upon OSS ecosystems operate.

Software dependency networks share features with other networked settings commonly studied in the economic literature: joint research and development efforts between firms (Goyal and Moraga-Gonzalez, 2001), innovation and patents (Jaffe et al., 1993; Hall et al., 2005; Acemoglu et al., 2016),

[6]By "userbase", we mean the incredibly broad set of stakeholders than have come to rely on the functionality and security of a given software component: developers who use software as intermediate inputs, individual end users, private firms, public institutions, etc.

[7]`npm` is the Node Package Manager is the *de facto* standard for developing and distributing Node.js packages. See `https://www.npmjs.com/`.

[8]See `https://heartbleed.com/`.

[9]It is thought that the severity of an exploit is amplified if malicious actors are aware that valuable targets remain exposed to the vulnerability even after its disclosure.

[10]Yet another recent example of the wide-ranging impact of software faults is the Log4Shell vulnerability, introduced in 2013 and disclosed in December 2021, which allows an attacker to leak sensitive information passing between network connected devices (WIRED, 2021). It is estimated that the exploit exposed hundreds of millions vulnerable devices or 93% of enterprise cloud environment (Wiz, 2021).

academic publications (Hsieh et al., 2018), linkages between financial institutions (Elliott et al., 2014; Acemoglu et al., 2015), risk sharing (Fafchamps and Gubert, 2007), and inter-firm trade (Elliott et al., 2022). OSS projects are collaboratively developed, interdependent network public goods that generate value as both intermediate and final goods.[11] The setting embeds both positive and negative network externalities in complex ways. Prudent or risk averse maintainers create value for downstream dependents by freely sharing software functionality with minimal fluctuations in dependency project quality. Linkages can also directly or indirectly transmit contagion between projects in the form of lapses in functionality, technical debt[12], and even software faults and vulnerabilities.

With these features in mind, we seek to study the evolution of these networks and the implications of equilibrium structure by focusing on the microeconomic behavior of software project maintainers. Specifically, we develop a framework in which each maintainer will make decisions over (1) a level of development resources to invest in their own project and (2) which external projects to use as dependencies in an effort to minimize development costs and maintain a preferred level of expected project quality.[13] In doing so, we can learn which factors influence both (1) the level of overall welfare induced by the dependency ecosystem and (2) the robustness of equilibrium dependency structure to cascading failures. After developing some intuition for these mechanics, we can then consider a set of potential policy interventions that can improve equilibrium welfare, allowing maintainers to make project development decisions more efficiently.[14]

The chapter is organized as follows. In Section 2 we survey the literature. In Section 3, we introduce a framework to ground our study of the sociotechnical software dependency ecosystem, which gives focus to the behavior of cost-minimizing yet risk averse maintainers making development decisions for interrelated projects under uncertainty. We illustrate the key features of this setting with simple examples. In Section 4, we introduce the data used in both the reduced form and structural em-

[11]For example, a software engineer seeking to develop a project for consumers may opt to use an external dependency as a production input.

[12]In engineering and software development, technical or design debt occurs when a short term solution incurs larger costs over the long run (Techopedia, 2017). Proponents of efficient software design patterns argue that excessive dependency reliance contributes to technical debt (Jackson, 2019).

[13]In this sense, transaction costs driven by information asymmetry confront maintainers with a "make-versus-buy" decision when developing the functionality of their project (Coase, 1937; Williamson, 1975, 1985). Under these conditions, some maintainers may prefer to invest more development effort in order to avoid external dependencies and provide a greater degree of "vertical integration" within their project (Grossman and Hart, 1986).

[14]In other words, either at lower cost or lower uncertainty over project quality, or both.

pirical analyses while illustrating several descriptive patterns that guide our methodologies. With the empirical setting in place, we build intuition over equilibrium outcomes with a reduced form methodology in Section 5. Finally, in Section 6, we develop a complete structural model of software dependency network formation, assess its equilibrium properties, and discuss the specifics of estimation. In Section 7 use the estimates of structural parameters to conduct counterfactual analysis of potential policy interventions. We conclude with final remarks in Section 8.

2 Literature

We begin our study by reviewing relevant strands of literature to illustrate relevant empirical patterns in software dependency management, place the current study into context, and identify existing methodological approaches that can inform our analysis.

The empirical software engineering literature has established several salient stylized facts to characterize the state of software dependency networks in the wild. Kikas et al. (2017) and Decan et al. (2019) find substantial indirect dependency between projects in software networks also characterized by limited direct dependency. Hence, empirical evidence suggests the observed behavior of project maintainers results in fragile dependency networks, vulnerable to contagion.[15] Common types of vulnerabilities can "break" functionality or expose sensitive user information for a package and its dependents (Prana et al., 2021). Decan et al. (2018b) find that it takes on average 24 months to find 50% of all vulnerabilities[16], vulnerabilities are prevalent across releases, and downstream dependents often remain unpatched even after vulnerability is fixed upstream. Vulnerabilities can be further exacerbated by the reuse of software code, both in the form of reused code within packages and by reusing outdated dependencies (Pham et al., 2010). In some cases, up to 40% of errors in packages can be traced to changes in upstream projects (Decan et al., 2016). Up to 80% of maintainers do not keep their dependencies up to date while almost 70% are simply unaware of upstream version changes (Kula et al., 2017). Vigilant maintainers must not only manage development decisions within their own codebase, but also track changes upstream

[15]As put by Zimmermann et al. (2019), such fragile networks can be described as "small world, high risk".

[16]The delay between the identification of a vulnerability and the distribution of a patch fixing it is known as "technical lag" (Decan et al., 2018a; Zerouali et al., 2018).

A significant body of research has sought to better understand the mechanics driving these observed forces in software networks as well as their implications. One strand of literature has explored the costs of software vulnerabilities, including general inquiries into dependency risk (Schueller and Wachs, 2022), the link between vulnerability disclosure and firm valuation (Acquisti et al., 2006; Telang and Wattal, 2007; Anwar et al., 2018), the efficacy of vulnerability rewards programs (Finifter et al., 2013; Zhao et al., 2015; Roumani et al., 2016), end economic theory behind optimal patch management (Cavusoglu et al., 2006; Finifter et al., 2013). Several ambitious empirical studies have even endeavored to estimate the value of the entire OSS digital supply chain itself (Keller et al., 2018; Robbins et al., 2018). Despite these efforts, the present study fills a gap in the literature by studying empirically the decision of the individual maintainer to outsource functionality for their project and how this behavior influences the resulting equilibrium.

Our preferred modelling approach attempts to explain the formation of software dependency networks based on the micro-founded behavior of individual maintainers and therefore draws from several distinct efforts within the economic literature. A starting point for the complementarities in production and the formation of fragile supply chains under risk was outlined by Kremer (1993).[17] More general theoretical treatments have developed theory for the micro-foundations of network formation under risk aversion (Kovářík and Van der Leij, 2009; Blume et al., 2013; Kovářík and Van der Leij, 2014).[18] The percolation of supply chain disruptions downstream has also been studied empirically through the use of natural experiments (Bernard et al., 2019; Carvalho et al., 2021). Motivated by linkages between financial institutions, a considerable number of studies pay particular attention to conditions under which network structure is susceptible to contagion (Elliott et al., 2014; Acemoglu et al., 2015; Erol and Vohra, 2018; Marbukh, 2018).

The current study is most closely related to recent work by Elliott et al. (2022), who consider the formation and robustness of supply chains in the presence of risk. The authors model complex production networks as the multi-sourcing strategies of individual firms, each subject to idiosyn-

[17]For overviews on production networks and input-output shock propagation, see Carvalho (2014) and Carvalho et al. (2021).

[18]Yet another adjacent strand of research has focused on incentives for agents to form networks insure against risk (De Weerdt, 2002; Fafchamps and Lund, 2003; De Weerdt and Dercon, 2006; Fafchamps and Gubert, 2007; Bramoullé and Kranton, 2007; Ambrus et al., 2014)

cratic disruptions which therefore exposes the entire network to contagious risk.[19] They find that even when firms can hedge against supply chain risk through multi-sourcing strategies, aggregate production remains quite sensitive to shocks in equilibrium. As anecdotal evidence suggests similar patterns may also be present within software dependency networks, the present study represents an empirical application continuing this strand of research in the domain of open source software and public good production.

For the purposes of structural estimation, we also look to a more general literature on strategic network formation (Bloch and Jackson, 2006; Galeotti and Goyal, 2010; Choi et al., 2019; Christakis et al., 2020) as our objective in Section 6 is simultaneously model the coevolution of agent choice of actions and link formation by following the work of Hsieh et al. (2022). In the spirit of Ballester et al. (2006), the authors also integrate a counterfactual analysis that considers the welfare impact of removing critical agents or "key players" from the network. In Section 7, we adapt this methodology by simulating dependency formation under the absence of "key dependency projects". We also draw from work defining graph theoretic measures of to characterize properties of social networks, such as node centrality (Bloch et al., 2019; Everett and Schoch, 2022) and network fragility (Doyle et al., 2005; Wan et al., 2021).

3 Framework

To build intuition for our setting and methodology, we next sketch out a framework for the evolution of software dependency networks, centering our attention on the problem of an individual project maintainer who seeks to efficiently develop a level of software quality under uncertainty. We illustrate, in turn, the general setting of software dependency networks, how indirect risk accumulates across interdependent projects, the maintainer's choice over dependencies, and factors that influence overall network robustness to perturbations. The discussion in this section is merely meant to fix ideas and serves as a primer to Section 6, in which we develop a complete micro-founded structural model to more formally characterize this behavior.

[19] Elliott et al. (2022)'s consideration of endogenously chosen "relationship strengths" in inter-firm trade is analogous to our focus on the maintainer's choice between different dependency projects.

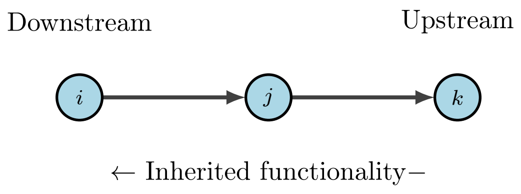


Figure 1: Project i depends directly on project j and project j depends directly on project k. Hence $G_{ij} = G_{jk} = 1$ are the only non-zero elements of the adjacency matrix G. We say project k is an *indirect dependency* of project i. Additional terminology: Project k is *upstream* of both projects i and j. Project i is *downstream* of both projects j and k. It is important to reiterate that $G_{ij} = 1$ implies that i inherits some functionality from j. In other words, the dependence relationship runs in the opposite direction of the flow of inherited functionality.

3.1 Setting

Consider a set of software projects $i \in \mathcal{N} = \{1, \dots, N\}$. Each project can be indexed by a measure of its *quality* $y = (y_i)_{i \in \mathcal{N}}$. Software projects can *depend* on one another, in which case the dependent (downstream) project borrows a subset of functionality from the dependency (upstream) project at a nominal price of zero[20], assuming the upstream project is publicly available and released under a permissive license. These unilateral dependency relationships between projects can be collected into a directed graph $G = [G_{ij}]_{i,j \in \mathcal{N}}$, which in turn can be represented by the $N \times N$ adjacency matrix $G = [G_{ij}]_{i,j \in \mathcal{N}}$ with elements $G_{ij} \in \{0, 1\}$ for all $i, j \in \mathcal{N}$.[21] If $G_{ij} = 1$, then project i imports some functionality from project j and therefore depends the quality of project j to some extent:

$$G_{ij} = 1\{\text{project } i \text{ depends on project } j\}.$$

Otherwise, $G_{ij} = 0$. We say package j is a *direct* dependency of package i if $G_{ij} = 1$.[22] Package k is an *indirect*[23] dependency of package i if there exists a *directed path* from package i to package k.[24] In the parlance of software dependencies, we can also say that packages i is *downstream* of package j and package k is *upstream* of packages i and j. We illustrate the basics of this networked setting

[20]Note the "nominal" aspect of free and open source software. The very spirit of this chapter is to highlight sources of hidden costs associated with relying on public software infrastructure.

[21]Following convention, $G_{ij} = 0$ when $i = j$.

[22]Alternatively, the graph G can be represented as a tuple $G = (\mathcal{N}, \mathcal{E})$ where $\mathcal{E} = \{(i, j) \mid G_{ij} = 1\}$.

[23]In the software development community, indirect dependencies are sometimes known as transitive dependencies. As "transitive relationship" has a different meaning in social networks literatures, we opt to use the term "indirect dependency".

[24]That is, there exists a directed sequence of distinct dependency relationships $\{G_{ij} \mid i, j \in \mathcal{S} \subset \mathcal{N}\}$ such that $G_{ij} = 1$ for each $i, j \in \mathcal{S}$.

in Figure 1.

Each project $i \in \mathcal{N}$ has a corresponding decision-making agent whom we will refer to as the project maintainer.[25][26] The objective of each maintainer is to efficiently develop their project while maintaining its expected quality above a given threshold.[27] The action space for each maintainer i consists of choices over (1) the level of costly development effort to their project, $x_i > 0$, and (2) the subset of projects to import as dependencies, $\{G_{ij}\}_{j \neq i}$. We assume that the cost of maintainer i's development effort is given by $c_i(x_i, G)$ but that importing software dependencies has an upfront cost of zero:[28][29]

$$c_i(x, G) = \frac{1}{2}x_i^2 - \left(a_i + \alpha \sum_{j \neq i} G_{ij} x_j \right) x_i. \tag{1}$$

We further assume that project quality y_i is a linear function of the dependency network G, the quality of external projects $y_{-i} \equiv (y_j)_{j \neq i}$, and aggregate effort $x \equiv (x_j)_{j \in \mathcal{N}}$:

$$y_i(x, y_{-i}, G) = b_i x_i + \beta \sum_{j \neq i} G_{ij} y_j + \xi_i. \tag{2}$$

To introduce risk and uncertainty in this framework, we assume that some share of project quality ξ_i is stochastic, unobservable, and known only in distribution by maintainers. Finally, we assume that maintainers are heterogenous in their relative level of risk aversion and define maintainer i's preferences over the quality of their project as $u_i(x, y, G) = \mathbb{E}\left[v_i(y_i)\right]$ where $v_i(\cdot)$ is a Bernoulli function. For example,

$$u_i(x, y, G) = \mathbb{E}\left[-e^{-r_i y_i}\right]. \tag{3}$$

[25]We will use the terms *project manager* and *maintainer* interchangeably.

[26]In reality, contribution and design decisions in large OSS projects are often shaped by the consensus of many distinct developers. We simplify our modeling framework by assuming that any potentially collective decisions made in equilibrium are ultimately made by a single "project maintainer".

[27]In Section 6, we will treat this threshold as exogenously given, unique to each project maintainer, and unobservable to the econometrician.

[28]Despite this assumption, using external software likely entails some fixed cost, as the downstream developer needs to understand and integrate the upstream package into their project. When a rational developer imports a dependency, we infer that the expected benefit of using the external package outweighs both fixed and marginal costs of working with the dependencies as well as the perceived risk of the dependency. In our modeling approach, dependency fixed costs are subsumed into the maintainer's choice across risky alternatives.

[29]The cost function in Equation 1 requires some explanation. Without parameter restrictions, aggregate costs may behave bizarrely as contribution effort increases. If $a_i > 0$, the marginal cost of effort increases with effort up to a point and then begins to fall. This may capture a scenario in which a developer "learns by doing" and becomes more efficient as she authors more code. A more problematic scenario may arise if $a_i > 0$ is such that costs are actually negative. We discuss how these complications influence our estimation in Appendices C and D.

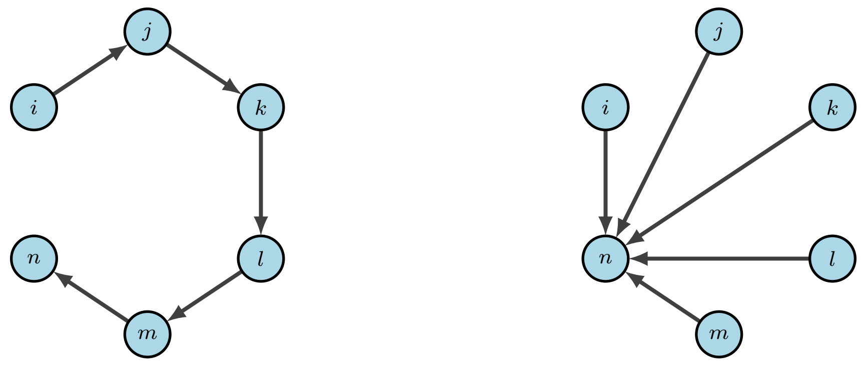


Figure 2: The network in Panel 2a is subject to more indirect risk than the network in Panel 2b. When considering the extent of both direct and indirect dependence, project n is the most critical project in both networks.

Here $r_i > 0$ captures maintainer i's relative risk tolerance: as r_i increases, i becomes more risk averse and enjoys less utility under network G where the quality of her project is subject to greater fluctuations in quality. Putting all these elements together, we assume that each maintainer chooses $(x_i^\star, \{G_{ij}^\star\}_{j\neq i})$ to (1) minimize development costs while (2) keeping expected utility over project quality above a threshold $\underline{u}_i$:[30]

$$(x_i^\star, \{G_{ij}^\star\}_{j\neq i}) = \underset{x_i>0, \{G_{ij}\}_{j\neq i}}{\arg\min} \quad c_i(x, G) \quad \text{s.t.} \quad u_i(x, y, G) \geq \underline{u}_i. \tag{4}$$

In the remaining subsections we discuss how dependency network structure embeds risk for individual projects, the decision of a project maintainer to import on external projects, and how network structure can be prone to fragility.

3.2 Risk Embedded in Dependency Network Structure

We use Example (3.1) and Figure 2 to show how network structure exposes projects to risk in the form of quality shocks to direct and indirect dependencies.

[30]We should acknowledge that given the way in which this modeling framework is written, project maintainer choose dependency relationships on a rather ambiguous basis. A more realistic approach would account that certain projects require very specific inputs and place little to no value on dependencies that are not relevant to them. For example, a text-based application would have little need for dependencies providing graphical processing functionality. This oversimplification stems from both modeling convenience and the lack of available data classifying OSS projects by functionality. Future work in this space would do well to measure the match quality between various software projects.

Example 3.1 (Risk Embedded in Dependency Network Structure)**.** *Consider the example networks in Figure 2. The set of projects is $\mathcal{N} = \{i, j, k, l, m, n\}$. In Panel 2a, $G_{ij} = G_{jk} = G_{kl} = G_{lm} = G_{nm} = 1$. In Panel 2b, $G_{in} = G_{jn} = G_{kn} = G_{ln} = G_{mn} = 1$. Assume that project quality for each project is the form given by Equation 2. Notice that in both networks, the removal of project n has the greatest effect on downstream projects. When considering the extent of both direct and indirect dependence, project n is the most critical project in both networks.*

In Panel 2a, the profile of project quality can be represented by the following system:

$$
\begin{aligned}
y_i &= b_i x_i + \beta y_j + \xi_i \\
y_j &= b_j x_j + \beta y_k + \xi_j \\
y_k &= b_k x_k + \beta y_l + \xi_k \\
y_l &= b_l x_l + \beta y_m + \xi_l \\
y_m &= b_m x_m + \beta y_n + \xi_m \\
y_n &= b_n x_n + \xi_n
\end{aligned}
$$

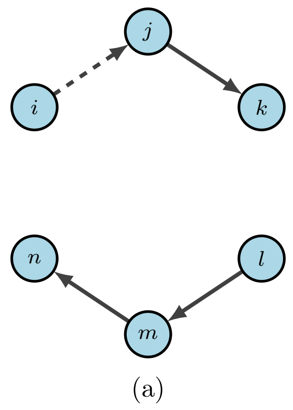
Recursive substitution shows that while project n is subject only to fluctuations in ξ_n, the remaining projects inherit some risk from indirect upstream dependents. The unobservable portions of quality in projects m, l, k, j, and i are $\beta\xi_n + \xi_m$, $\beta^2\xi_n + \beta\xi_m + \xi_l$, $\beta^3\xi_n + \beta^2\xi_m + \beta\xi_l + \xi_k$, ... and so on.

In Panel 2b, consider the same set of projects under a different (hub) dependency network. Projects i, j, k, l, and m each depend on project n, which itself has no dependencies. Therefore, relative to the network in Panel 2a, projects i and j are subject to less indirect risk.

As we have seen in real world examples in Section 1, faults and vulnerabilities in upstream applications can have consequences in downstream dependents. Maintainers understand this risk and make development decisions conditional on the current state of the dependency network.

3.3 A Maintainer's Choice Between Risky Alternatives

Using external software in a project can lower development costs and improve quality but also entails risk for maintainers. We seek to model dependency formation as a choice conditional on a



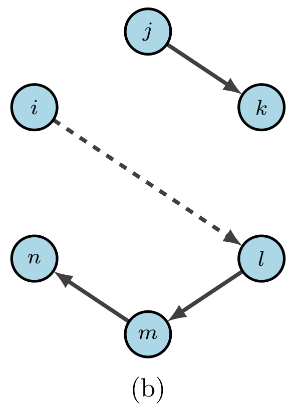


Figure 3: In Panel 3a, maintainer i prefers depending on project k over project l and avoids a greater level of indirect risk embedded in project l. In Panel 3b, maintainer i prefers l to j despite a greater level of indirect risk embedded by project l.

maintainer's private level of risk aversion: a maintainer ought to only use an upstream project as a dependency if they find it beneficial to their project's quality net of any risk the dependency introduces. Therefore, conditional on project quality and effort choices (y, x), the dependency selection elements of the maintainer's decision in Equation (4) can roughly be summarized as follows:[31]

$$\text{Maintainer } i \text{ uses project } j \text{ as a dependency} \iff y_i(x, y_{-i}, G + ij) \succsim_i y_i(x, y_{-i}, G - ij).$$

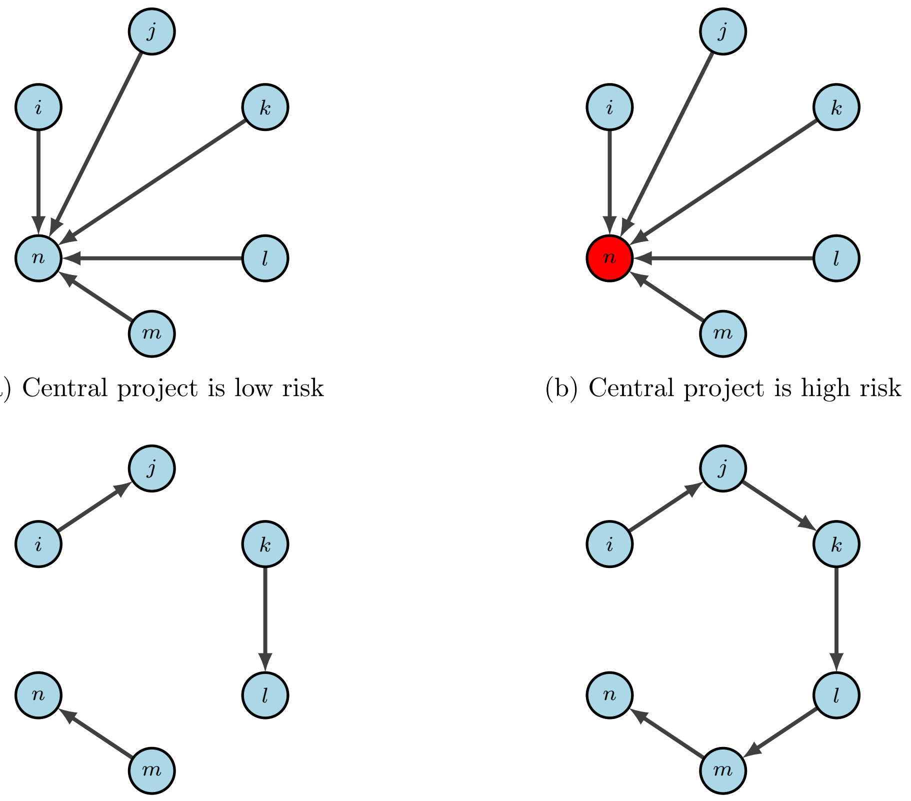
We formalize this choice by specifying a utility function u_i for the preference relation $\succsim_i$ that reflects maintainer i's individual level of risk aversion in Section 6. Since some portion of project quality is stochastic and unobservable (ξ_i), preferences can be represented in terms of expected utility à la von Neumann-Morgenstern.[32] Maintainer preference heterogeneity is a result of in variation across $v_i(\cdot; r_i)$, a Bernoulli value function parameterized by a measure of risk aversion $r_i > 0$.[33] We illustrate the maintainer's choice amongst dependencies with a simple example.

Example 3.2 (Maintainer Risk Aversion and Dependency Choice). *Consider maintainer i's choice between two candidate dependency projects, j and l, represented in Figure 3. In both panels, project j depends on project k while project l depends on project m and project n. Conditional on maintainer i's preferences for risk, she will choose to depend on a particular project if it improves the expected*

[31]Some notation for modifying a single relationship in a given dependency graph G: Let $G + ij$ denote the dependency graph that differs from G only in that $G_{ij} = 1$ and therefore project manager i imports functionality from project j. Similarly, let $G - ij$ denote a dependency network where the only difference from G is that $G_{ij} = 0$.

[32]In other words, $\succsim_i$ and v_i are such that $y_i(x, y_{-i}, G+ij) \succsim_i y_i(x, y_{-i}, G-ij) \iff u_i(x, y, G+ij) \geq u_i(x, y, G-ij)$

[33]Therefore, preferences are represented by $u_i(x, y, G) = \mathbb{E}\left[v_i(x, y, G; r_i)\right]$.



(a) Central project is low risk

(b) Central project is high risk

(c) Structure isolates risk

(d) Structure amplifies risk

Figure 4: In Panels 4a and 4b, different project characteristics can influence system-wide fragility for networks with identical structure. In Panels 4c and 4d, different network structures can influence fragility when projects characteristics are held constant.

quality of her own project. In Panel 3a, maintainer i imports project j as a dependency over packages l, indicating that she prefers the quality improvement and lower level of indirect risk introduced by relying on project j over that offered by project l. In Panel 3b, maintainer i instead prefers to use project l as a dependency. This indicates that although project l embeds more indirect risk than project j, maintainer i finds the benefits of using l outweigh the costs.

Maintainer preferences for dependency stability is a driving force that determines equilibrium structure of the network. As we discuss in the following section, this behavior has implications on the relative robustness or fragility of the entire ecosystem.

3.4 Fragile Dependency Networks

Both individual characteristics and the structure of the dependency network combine to expose individual projects to varying levels of risk, with implications for the overall value or health of the software ecosystem. We present two examples to illustrate these different channels.

Example 3.3 (Fragile Dependency Networks)**.** *Consider two alternative dependency networks in Panel 4a and Panel 4b of Figure 4. Assume that the only difference between these networks is that the variability in quality for project n is greater in Panel 4b than it is in Panel 4a. Notice that given the structure of the dependency networks in both settings, all projects are exposed to disturbances stemming from project n. In this sense, the network in Panel 4b is relatively more fragile than the network in Panel 4a since the central or hub project n is riskier.*

Next, consider the networks in Panel 4c and Panel 4d. In this case, difference in network structure can lead to increased fragility. The removal of project n in Panel 4c impacts only project m since the network is the union of three disconnected components. In Panel 4d, project n is a dependency, either direct or indirect, for all of its peers. Hence, we can say that the network structure in Panel 4d is relatively more fragile than in Panel 4c, since the removal of the same project is more disruptive to overall project quality.

It is useful to consider measures with the potential to characterize a given network G in terms of fragility. One approach is to consider measures of network centrality. Specifically, consider Katz-Bonacich centrality for the nodes of the graph G. Roughly speaking, a node has greater Katz-Bonacich centrality when it is a hub for many other high in-degree nodes.[34] In the world of software networks, central hub dependencies lie at the core of the dependency network and serve, both directly and indirectly, as the foundation for many other projects. Using the logic outlined in the beginning of this subsection, software networks characterized by highly centralized projects may efficiently serve functionality to many dependents, but do so at the cost of increased network fragility.

[34]Formally, denote the Katz-Bonacich centrality for project node i in graph G as k_i. Then for a decay factor $\rho \in (0,1)$, Katz-Bonacich centrality is defined as $k_i(\rho, G) = \sum_\ell \rho^\ell \sum_j G^\ell_{ij}$ where ℓ is the length of a walk between nodes i and j (Bloch et al., 2019). In matrix notation, this becomes $(I - \rho G)^{-1}\rho G\mathbf{1}$.

4 Data

The data used in both our reduced form and structural approaches seeks to characterize (1) the features and outcomes within software projects and (1) the dependency relationships between them. How do upstream dependencies influence project contribution and quality? What social or technical characteristics of projects are associated with many upstream or downstream dependency relationships? Can the equilibrium structure of software dependency networks result from or contribute to these dynamics? What is the economic significance of these outcomes?

To address these questions empirically, we develop a dataset of sociotechnical measures for a sample of interrelated OSS projects. We choose to focus on projects from the Node.js JavaScript ecosystem.[35] The dependency relationships between widely used open source Node.js projects are tracked over time by the `npm` (Node.js package manager) registry. Most critically, the `npm` registry records (1) timestamps for when specific versions of packages are published and (2) the set of external dependencies, along with their respective versions, declared by the parent package. Hence, by knowing what external components a package relies on at a given point in time, we can observe the evolution of a software ecosystem and its dependency graph as a network panel. A notable advantage of this data is that it captures more information about the exact timing of package publication dates and dependency formation compared to single-network observations prevalent in the literature.[36] In addition to simplifying structural estimation, this data enables our structural approach to consider the interrelated decisions of a project manager over internal project development and dependency formation with external projects.[37]

Another attractive property of this empirical setting is that it is possible to observe how sociotechnical features for each individual project evolve over time. The `npm` registry documents the repository URL for the package's source code. Furthermore, a project's source code is typically managed using a version control system (VCS), such as `git`, which has the benefit of chronicling development of

[35]The rationale for this choice is discussed in sections below. Simply put, the `npm` is the largest open source package ecosystem in terms of number of packages (2.61 million packages as of January 2020 (Katz, 2020)). Moreover, we focus on a single programming language ecosystem to make more appropriate comparisons between packages.

[36]Previous authors have developed estimation strategies to exploit repeated observations of networks (Snijders et al., 2010).

[37]Critically, we can capture the initial conditions of the sample network to overcome any bias that might arise characterizing the data generating process in our structural approach.

the project at high granularity: one can use the version control log to know which developer contributed which lines of code to the project at specific moments in time. We use both the dependency relationships between packages[38] and the technical features of the project recorded in the VCS log.[39]

In the following sections, we discuss the procedure we use to develop our empirical sample, the measurement of software quality, and illustrate the dataset with a selection of descriptive statistics.

4.1 Sampling Procedure

Software dependency networks can grow incredibly large and can be observed at a high temporal granularity. We must therefore resort to sampling a set of representative projects with the potential to capture the essence of dependency management dynamics. We focus on a single packaging ecosystem[40] to minimize irregularities that may arise from cross-language comparisons. Motivated by the case studies of major disruptions caused by widely used software projects mentioned in Section 1, we choose to focus on the largest packaging ecosystem tracked by the Libraries.io service: `npm` JavaScript packages.

We will begin by describing how we obtain a set of OSS projects and record their dependency relationships over time. Our sampling procedure can be summarized by the following steps. In Step 1, we sample the top ten most widely depended upon Node.js packages in the `npm` registry as of September 2022. In Step 2, for each of these packages, we record a sequence of timestamps associated with minor version releases.[41] In Step 3, for each package at a specific timestamped version, we record the set of upstream runtime[42] dependencies the package depends upon at that point in time. We add this set of dependencies to the running list of sampled projects and return

[38]Dependency relationships are tracked by the `npm` registry, a publishing platform for Node.js packages. Once published on the registry, users and developers can install these packages using the `npm` (Node.js Package Manager) tool.

[39]Technical project features can be observed in the source code of the repository. Some technical details: the granularity of the VCS log allows us to download the source code of a project and "rewind" it to its state at a specific point in time.

[40]In other words, a set of OSS projects written in a common language.

[41]Best practices in software development encourage the use of semantic versioning, a labelling system for published releases of software to indicate the degree to which the project has changed. Among other reasons, this is done to improve downstream compatibility, as managers of dependent projects can use the semantic version to determine if their dependency is likely to have any breaking or backwards-incompatible changes. See https://semver.org/ for more details.

[42]Meaning the dependency is required for the dependent for basic functionality. Maintainers can also declare dependencies needed only for project development or extended functionality.

to Step 2. To reduce the size of the resulting sample, we restrict the set of timestamps sampled to minor package versions and limit the depth of upstream dependency projects sampled to 5th degree neighbors of the initial set of 10 seed projects. We refer to the set of versioned timestamps at which a sampled package and its dependencies are observed as our set of *sample moments*, points in time at which the dependency network potentially changes.[43] Throughout this analysis, we will refer to this specific recursive network sampling procedure as an *upstream sample* that captures the most central projects of the Node.js ecosystem. While this choice of sampling procedure naturally biases the selection of projects towards core libraries used in the development of larger, user-oriented projects, it is deliberate. In addition to keeping the size of the sample within reason[44], any dynamics affecting this set of core packages will have widespread influence on downstream packages outside the sample. Hence, any welfare effects estimated within this core sample can be viewed as a lower-bound estimate for the `npm` ecosystem at large. The resulting dependency subnetwork contains 1,263 Node.js projects observed at 40,440 distinct sample moments from October 2010 through September 2022. A snapshot of the network sample as it was observed in September 2022 is depicted graphically in Figure 5. Evolution of the sample dependency graph over time can be seen in Figure 6.

Once we have obtained a panel of dependency relationships, we next use the source code of each project to derive measures of sociotechnical outcomes. For each package, we observe these outcomes for the set of project-specific moments, defined as when either (1) a minor version of the package is published or (2) a minor version of a package is declared as a dependency of another project.[45] We use the repository URL of the project to download a copy of its source code and version control history.[46] For each project moment, we use the VCS log to observe social features such as the cumulative level of commits to the project (i.e., contribution), the cumulative number of

[43]It's important to note the use of the term "potentially". A new version release of a software package may likely contain the exact same set of dependencies as the previous version.

[44]If we had conversely sampled downstream from the top ten most widely depended upon packages, the resulting sample may include hundreds of thousands of packages.

[45]Note that to keep the number of observations in the empirical dataset manageable for the purposes of structural estimation, we do not observe every single package for each moment, only the packages specified in a version's changeset. We can get a sense of this technicality from the reduced form estimates in Tables 2 and 3, where the number of observations ranges from 206,598 to 196,894, depending on the availability of each covariate measure.

[46]A forensic analysis of software source code revision history for sociotechnical measures falls under a branch of research in the computer science literature known as mining software repositories (MSR). Notable tools in this space include `reaper` (Munaiah et al., 2017), `pydriller` (Spadini et al., 2018), `augur` (CHAOSS, 2017), and `grimoirelab` (Dueñas et al., 2021).

contributors, and the number of core contributors to the project and its associated "bus factor"[47], the project's age, and an estimate of the number of hours spent[48] on project development. We also use the source code of the project itself to measure[49] technical features of the codebase such as the number of single lines of code (SLOC), the cumulative size of the codebase in megabytes, the number of files, the number of distinct languages used, and the number of lines in the codebase that are considered documentation, and other derived measures such as "modularity", defined as the number of lines per file in the codebase, and "churn", the ratio of cumulative commits to SLOC in the codebase. Projects in which the same sections of code are constantly under revision will, all else equal, have larger values for the churn measure. Finally, most importantly for our measure of software project quality, we can derive a measure of sophistication for the codebase known as cyclomatic complexity.[50]

4.2 Measuring Software Quality

The notion of a software's quality is a nebulous concept.[51] In the simplest sense, software code is a collection of instructions for a machine to perform a specific task. Developers and users of a particular project may derive value from it in different ways. For example, a user may consider a software of high quality if it can perform its stated purpose successfully, perform efficiently, and do so with minimal errors. Developers, on the other hand, may understandably place more emphasis on the "maintainability" of the software's codebase.[52]

[47]We define the number of core contributors as the smallest number of contributors who together have contributed at least 80% of aggregate commits to the project. This measure is related to the so-called "bus factor" commonly discussed in the literature, which is used as an estimate of how susceptible a project is to the loss of key contributors. In our study, we define the bus factor for a project as the count of total cumulative contributors divided by the count of cumulative core contributors: the greater the bus factor (i.e., closer to 1), the more the project relies on a smaller set of core contributors. See https://chaoss.community/metric-bus-factor/ for more details.

[48]We use an algorithm developed by Brunfeldt (2014). See https://github.com/kimmobrunfeldt/git-hours. The algorithm takes the revision history of the project (i.e. the `git` log) and identifies distinct "coding sessions", defined by sequences of commits made less than 2 hours apart from each other. For each coding session, time allocation is estimated by the duration as measured by the time between the first commit timestamp of the session and the last. Finally, the sum of all session durations, from the initial commit at t_0 and the final commit before observation at time t, is the estimated time allocation for the entire codebase observed at t.

[49]To generate these technical metrics, we use the static code analysis tool Succinct Code Counter, `scc` (Boyter, 2018). See https://github.com/boyter/scc for more information.

[50]Cyclomatic complexity measures the number of linearly independent paths through the control flow of a software's functionality. Simply put, smaller and simpler software projects will likely have lower measures of cyclomatic complexity. See https://www.ibm.com/docs/en/raa/6.1?topic=metrics-cyclomatic-complexity

[51]See Spinellis et al. (2009) for an overview in methods for evaluating the quality of OSS.

[52]In economic terms, maintenance costs.

Even after settling on a particular definition of software quality, how can it be measured? The use quality of a project may be proxied by the extent of popular uptake. How many "followers" have indicated interest in the project on software development platforms like GitHub? How frequently is the software discussed by users in external communities?[53] Perhaps most pertinent to the present study, how many external projects depend upon a particular software package? The technical quality of the project can be measured in yet other ways. For example, static code analysis tools and "linters" can scan the project's codebases for potential vulnerabilities, poorly written or documented code, or other bad software development practices. It is important, however, to acknowledge that each of these measures have relative strengths and weaknesses. The exact definition of software quality is likely best defined contingent on the context of its application.

For the purposes of the reduced form and structural analyses, we opt for a rudimentary measure of software quality designed to reflect two distinct notions of a codebase's overall value. We say a project is of high quality if it is (1) complex and (2) attracts numerous contributors.[54] This measure captures both developer interest in contributing to the project along with a rough proxy for the level of engineering sophistication it entails.[55] In some reduced form specifications in Section 5, we will also argue that the number of downstream dependents a project serves can also be used a measure of the value or quality of the software project.

4.3 Descriptive Statistics

The sample gives insight over the (1) actions, (2) outcomes, and (3) structure that characterize a software dependency network. We briefly provide some descriptive statistics for this empirical sample.

Figure 5 presents a snapshot of the sample dependency network as it is observed in September 2022. At first glance, this snapshot reveals a tendency towards a hub structure for our empirical sample: a relatively small group of central nodes support their remaining peers both directly and

[53]For example, we can measure this using the relative frequency of the project's name in search engine trends or in software-specific Q&A forums such as StackOverflow.

[54]Specifically, we will define quality as the sum of log cyclomatic complexity and the log of the number of cumulative contributors to the project. We then scale the resulting sum to reside within the interval $[0, 1]$.

[55]Similar measures of OSS codebase quality are used in Libraries.io's SourceRank metric (Katz, 2020). For additional information on the SourceRank measure, see https://docs.libraries.io/overview.html#sourcerank.

indirectly. Figure 6 shows the growth in the network over time, revealing that as new packages enter the ecosystem, the dependency network becomes less dense.[56] Lower density networks may involve additional software development expenditures but at the same time can satisfy a wider arrange of computing application needs and can also serve to isolate dependency risk. Another way to characterize the extent to which certain packages are relied upon in the dependency network is through measures of the package's centrality. For the purposes of illustration, we observe the network in several annual snapshots and calculate each node's (1) Katz-Bonacich centrality and (2) betweenness centrality.[57] We present a bivariate scatter plot of each node's centrality measures in Figure 7. Naturally, we can see that smaller networks feature nodes with greater centrality. However, we can also see that as the network grows larger in later periods, small groups of outlying nodes have exceptionally greater measures of relative centrality. In a sense, the larger dependency networks diversify some risk away with the introduction of new packages but few dependency "hubs" serve a larger number of dependents. Despite these insights, the overall effect of such network structures on maintainer welfare remains unclear.

Table 1 in Appendix B contains summary statistics, notation, and brief descriptions of the key sociotechnical project-level measures used in both the reduced form and structural analysis. We highlight the key insight from these features. Most importantly, the vast majority of these project-level measures convey a common pattern of (right) skewness across projects that ought to have bearing on the interpretation of any sample-wide estimates. For example, the median project in the sample is a terminal dependency with no dependencies of its own, and hence dependency quality and contribution are both absent (i.e., zero).[58] Another important feature of this sample is that it is skewed towards upstream dependencies: the average (median) package has 2 (1) upstream dependencies but 5 (2) downstream dependents. Moreover, the average (median) package in the sample consists of 2,703 (195) cumulative commits and features dependencies with 1,451 (0) cumu-

[56]We acknowledge this phenomenon may simply be an artifact of our sampling methods. However, Decan et al. (2018a) document growth in dependency networks by observing the entire population of packages for several ecosystems. In particular, the authors find that package growth in the Node.js ecosystem is exponential over the observation period, roughly similar to the finding for our empirical sample in Figure 6.

[57]Greater levels of either centrality metric opens up the network to additional risk, all else being held equal. See Bloch et al. (2019) and Everett and Schoch (2022) for deeper discussions of network centrality metrics and their implications for social networks.

[58]The prevalence of skewness is a pattern reminiscent to the contribution behavior observed in Chapter ??.

lative commits.[59] Finally, the average (median) package observation consists of 233 (20) cumulative contributors and 20 (1) core contributors, highlighting the skewed distribution of work in these ecosystems. A significant share of core dependencies rely on maintenance efforts by a small group of dedicated individuals. Guided by the framework discussed in Section 3, we give deeper consideration to both (1) the relationships between these various features and (2) network structure itself throughout the reduced form analysis in Section 5.

5 Reduced Form

Before developing a fully structural model of software dependency management, we begin our analysis with a reduced form approach to build intuition over empirical patterns. Our objectives in this section are two-fold. First, we begin to explore the extent to which upstream dependencies influence downstream dependent projects, by lowering contribution costs or improving project quality. Second, we estimate several linear specifications in which (1) the number of upstream projects a maintainer depends upon and (2) the number of downstream dependents a project supports are regressed in turn on a set of observables such as features of the project itself and the current state of the dependency network as a whole.

We assume that panel data $\mathcal{D}_t \equiv (y_t, x_t, G_t, W_t)$, is observable by both maintainers and the econometrician where $t \in \mathcal{T}$ represent a sequence of observations.[60] To be consistent with the notation of our framework outlined in Section 3, here the vector $x_t \equiv (x_{it})_{i \in \mathcal{N}}$ contains all project contribution levels measured in number of commits, the vector $y_t \equiv (y_{it})_{i \in \mathcal{N}}$ contains measures of project quality, G_t captures the dependency network structure[61], and $W_t \equiv (W_{it})_{i \in \mathcal{N}}$ collects node (project) characteristics at the sample moment $t \in \mathcal{T}$. We assume that for the equilibrium captured in these observables, behavior for maintainer i in each period t is a function of peer actions $j \neq i$, the state of the world $\mathcal{D}_{t-1}$, and stochastic shocks. Our discussion outlines a set of econometric specifications, provides economic intuition for estimated parameters, and addresses issues pertaining to identification.[62]

[59] This is also likely an artifact of sampling, as many of our observations consist of early period core dependencies with no observed dependents.

[60] We will assume that time is discrete and therefore without loss of generality, let $\mathcal{T} \subseteq \mathbb{N} = \{1, 2, \ldots\}$.

[61] In other words, the adjacency matrix for the empirical sample network at moment t.

[62] Chandrasekhar (2016), Bramoullé et al. (2020), De Paula (2020), Graham (2020), and Graham and De Paula

5.1 Contribution Levels

Suppose we are first interested in the relationship between upstream and downstream contribution. Our preferred econometric specification, given in Equation (5), mirrors the first order necessary condition from the maintainer's choice over contribution effort:[63]

$$x_{it} = a_i + \alpha \sum_{j \neq i} G_{ijt} x_{jt} + \delta' W_{it} + \epsilon_{it} \tag{5}$$

. In this specification, the fixed effect a_i captures a time invariant propensity for contribution to project i. The term $\sum_{j \neq i} G_{ijt} x_{jt}$ in Equation (5) is the sum of contribution activity in project i's dependencies. Therefore the parameter of interest in this specification, $\alpha \in \mathbb{R}$, measures the relative influence of upstream contribution on (downstream) contribution to project i. If $\alpha < 0$, then increased upstream contribution is associated with less downstream contribution on average. Conversely, $\alpha > 0$ implies that downstream contribution increases with the level of upstream dependency development. The direction of this net effect therefore maps into substitution: if $\alpha > 0$, upstream and downstream contribution are gross complements. We can interpret this in two ways. First, the level of upstream development lowers the marginal cost of downstream contribution, generating a positive productivity effect. Second, large dependency trees require the maintainer of the downstream dependent to exert considerable effort to integrate and maintain. If $\alpha < 0$, upstream and downstream contribution are gross substitutes. This implies that larger dependencies allow downstream dependents to exert less development effort. Any one of these mechanisms seems plausible and none can be ruled out *ex ante*. Moreover while the specification assumes a common net pattern of substitution across projects and time, heterogeneous effects are likely more realistic.

The vector of controls W_{it} includes other observable characteristics for project i that might conceivably influence contribution levels.[64] Specifically, we include controls such as a measure of project quality y_{it}, a quadratic term in project age, the total number of contributors as well as the number of core contributors, technical characteristics of the project such as single lines of code and cyclo-

(2020) provide excellent surveys of empirical methods in social network analysis.

[63]Recall the maintainer's problem given in System (4). We will fully specify the maintainer's optimization problem in Section 6.

[64]Therefore δ is a vector of coefficients corresponding to these covariate controls.

matic complexity, and temporal lags of both contribution to project i and upstream contribution[65]. The term ϵ_{it} represents project contribution influences that are unobserved by the econometrician, independent and identically distributed[66], and mean zero in expectation. In a setting in which (1) dependencies are formed unilaterally, (2) project managers are distinct across projects, and (3) the dependency network is acyclical, the terms on the right-hand side of Equation (5) are plausibly exogenous.[67][68] Therefore, in lieu of more rigorous argumentation, we are reasonably comfortable interpreting α as the causal effect of upstream contribution on downstream contribution in our fixed effects specifications.

We summarize coefficient estimates for the specification in Equation (5) in Table 2. To reiterate, the interpretation for the coefficient estimates for α is the effect of increased dependency contribution on the level of contribution in downstream projects.[69] The main takeaway from these results is that while in some specifications it would appear that there is a small positive productivity effect from upstream dependencies ($\hat{\alpha} = 0.034$ in Model 1 of Table 2), this effect seems to diminish or vanish completely on average after controlling for project-specific fixed effects ($\hat{\alpha} = 0.003$ in Model 3) and/or covariate controls ($\hat{\alpha} = 0.000$ in Model 2). Therefore we cannot say that on average a downstream project with many large dependencies is necessarily larger in terms of commits once individual project characteristics are accounted for.

There are several ways to interpret this pooled estimate. First, we must acknowledge that this particular reduced form model captures only intensive margin productivity effects. One may argue the sheer fact that the downstream dependent exists at all is simply because the upstream dependency sufficiently lowers some fixed cost of development. Second, a contemporaneous link between the project development level across dependencies simply may not exist. This would arise if a developer imports a dependency once and does not change her own contribution patterns in light of upstream

[65]That is to say, we include multiple lags of both the left-hand side endogenous and key right-hand side exogenous variables.

[66]$\mathbb{E}[\epsilon_{it}\epsilon_{js}]$ for each $j \neq i \in \mathcal{N}$ and $t \neq s \in \mathcal{T}$.

[67]That is, $\mathbb{E}[\epsilon_{it} \sum_{j \neq i} G_{ijt} x_{jt}] = \mathbb{E}[\epsilon_{it} W_{it}^k] = 0$ for $i \in \mathcal{N}$, $t \in \mathcal{T}$, all covariates W_{it}^k in the vector W_{it}.

[68]Consider the case in which the same set of developers contribute to a project i and its dependency j. In this case, the potential for simultaneity or reverse causality threatens naive estimates of α with endogeneity bias. A similar form of endogeneity may arise whenever a set of package dependencies form a cycle. For example, if for the set of projects i, j, and k, $G_{ij} = G_{jk} = G_{ki} = 1$. We assume away any pervasive threats from the former case and argue that software engineering best practices mitigate the latter.

[69]On average, *ceteris paribus*.

changes. This certainly can be the case if the dependency is small and not undergoing significant development. Finally, the observed equilibrium may describe a situation where large, general-purpose dependencies enable the efficient development of smaller, more specialized dependent projects.[70]

5.2 Project Quality

In a similar fashion, we can next turn our attention to the relationship between the quality of a project and the quality of its dependencies. We use the specification in Equation (6):

$$y_{it} = b_{0i} + b_{1i}x_i + \beta \sum_{j \neq i} G_{ijt} y_{jt} + \delta' W_{it} + \epsilon_{it} \tag{6}$$

. The project quality fixed effect b_{0i} captures an intrinsic level of quality independent of upstream dependencies, controls, or temporal fluctuations. The term b_{1i} captures the marginal product of contribution in terms of improving the quality of project i. The aggregate quality of upstream dependencies at time t is $\sum_{j \neq i} G_{ijt} y_{ijt}$ and therefore the parameter of interest $\beta \in \mathbb{R}$ represents an attenuation factor with respect to quality influences transmitted through the dependency network. Similar to Equation (5), a vector of controls W_{it} includes other observables that can potentially influence quality: cumulative contribution, project age, the size of the contributor base, maintainer characteristics, and lags of project quality, contribution, and upstream quality. As in Equation (5), we make similar assumptions for the unobserved component ϵ_{it} in Equation (6).[71]

We summarize coefficient estimates for the specification in Equation (6) in Table 3. Similar to our analysis of contribution productivity in the previous section, the effect that upstream dependency projects have on downstream quality is small and largely determined by individual project characteristics. Moreover, it is interesting to note that the number of commits in a project has little effect on its quality (i.e. $\hat{b_{1i}} \approx 0$ in all specifications). We cannot say, given our chosen project quality metric and conditional on individual project features, that upstream dependencies significantly improve the quality of downstream dependents.

[70] An apt analogy in this case may be a parallel between basic (i.e., generic dependencies) versus applied (i.e. dependents) research studied in the innovation literature.

[71] Note that the residuals of the regression from estimating the specification in Equation (6) can be used to proxy for volatility or uncertainty in project quality. More details can be found in our discussion of structural estimation in Section 6.3.

5.3 Dependency Formation

Up until now, our reduced form analysis on the influence of upstream dependencies has focused exclusively on intensive margin effects from upstream dependencies. We have not addressed factors that influence the likelihood of dependency formation and therefore know very little about the extent to which project characteristics and maintainer preferences can drive equilibrium dependency structure. In this section, we investigate features of OSS projects that either (1) form many upstream dependencies or (2) serve many downstream dependents. We operationalize this by regressing both the number of upstream dependencies or the number of downstream dependents a package has on covariate controls.[72]

Let $d_{it}^{\text{out}} \equiv \sum_{j \neq i} G_{ijt}$ denote the number of external projects that package i has declared as (upstream) dependencies at time t.[73] We study factors that drive a package to form many upstream dependency relationship using the specification described in Equation (7):

$$d_{it}^{\text{out}} = \delta' W_{it} + \epsilon_{it} \tag{7}$$

. where W_{it} is a vector of observables for project i at time t drawn from Table 1. Similarly, let $d_{it}^{\text{in}} \equiv \sum_{j \neq i} G_{jit}$ denote the number of external (downstream) projects that declare package i as a dependency at t.[74] Factors that are associated with the attractiveness of package i as a dependency can be studied using the specification in Equation (8):

$$d_{it}^{\text{in}} = \delta' W_{it} + \epsilon_{it} \tag{8}$$

. As the number of downstream dependents is one way to measure a project's importance or quality, the specification in Equation (8) is an alternative to the specification in Equation (6) to reveal which

[72]We acknowledge that there are alternatives to count regression to study characteristics of dependency formation. For example, we could assess the effect of observables on dependency formation using dyadic regression (Helmers et al., 2017; Bramoullé et al., 2020):

$$G_{ijt} = 1\{r_i + \gamma_j + \delta' W_{ijt} + \epsilon_{ijt} \geq 0\}$$

. Without sub-sampling, to estimate such a specification on the entire sample entails an onerous computation burden and hence we opt for the simpler and arguably more interpretable approach of count regression.

[73]In graph terminology, the out-degree of node i at time t for the graph G_t.

[74]In other words, package i's in-degree.

observables features contribute to package quality.

We present coefficient estimates for the specifications in Equations (7) and (8) in Table 4. Several patterns emerge. First, models (1) through (4) of Equation (7) suggest that higher quality packages declare a larger number of upstream dependencies. This pattern is notably stronger than the intensive margin quality effects collected in Table 3 and underscores the notion that popular, complex projects likely outsource much of their functionality to external packages. Second, as expected from somewhat of a mechanical correlation, packages with more dependencies have higher dependency quality. However on the other hand, packages with many downstream dependents feature fewer upstream dependencies and therefore enjoy less quality effects from their own dependencies.[75] Third, hub dependencies tend to be well documented while packages with many dependencies are not. It is likely that well documented software is easier to work with and therefore more attractive to use. Finally, a project's lines of code, the number of contributors, and age are all not strong predictors of either upstream or downstream dependency.[76]

5.4 Robustness

Pooled estimates of the effect of dependencies on downstream contribution and quality may mask effects present in various sub-samples. To this end, we also estimate the dependency effects α of Equation (5) and β of Equation (6) at both (1) the project-level and (2) over time, and (3) for the sub-sample of projects with at least one dependency.

In Figure 8, we can see the project-level estimates for the impact of dependencies on downstream contribution α are somewhat symmetrically centered around zero. The same is true for upstream quality effects β at the project level. In Figure 9, we estimate α by annual sub-sample. Interestingly enough, we can see that the effect of dependencies on downstream productivity is much greater in earlier sample years when both projects and the dependency network itself were much smaller. This would suggest earlier periods of the sample dependency network featured a stronger degree of complementarity between upstream and downstream contribution for core Node.js packages. On the

[75] One potential explanation for this pattern is modularity: larger and more complex packages import more dependencies and are more likely located downstream in the network.

[76] We acknowledge that our measure of quality correlates somewhat strongly with SLOC and therefore may simply not add much predictive power with respect to dependency formation.

other hand, upstream quality effects are not markedly different in earlier sample periods compared to later years or the fully pooled sample.

Finally, we estimate these specifications for the sub-sample of projects with at least one dependency declared. These estimates ought to reflect dependency influences on the projects that actually rely on external software for some functionality. However, we find that these estimates are actually quite similar to those for the pooled sample, especially after controlling for project-specific fixed effects.

5.5 Summary

Overall, our reduced form methodology finds a limited impact of upstream contribution and quality on downstream project contribution or quality, respectively. The notable exception is that the impact of dependency contribution seems to have had a stronger impact on downstream contribution productivity in earlier periods of the sample (Figure 9). An obvious potential explanation for this effect is that a considerable level of software functionality was absent in earlier periods of the sample and therefore increased project development effort was required on average. As the space of available functionality grows with the arrival of new dependencies, less "glue code" was required to integrate various functional components.

These insights guide our structural approach. First, reduced form analysis emphasizes the complexity of dynamics within the empirical setting of software dependency management. Without a well-specified structural model, it's unclear to what extent any of these estimated effects impact equilibrium welfare. Second, project-level fixed effects (i.e. a_i, b_{0i}, b_{1i}) seem to matter much more than average, intensive margin effects (e.g. α, β) when considering the influence of upstream dependencies on downstream outcomes. This result further motivates a structural approach that permits counterfactual analysis in which key central projects are removed. Third, the reduced form approach does not attempt to address factors such as project development costs, uncertainty over dependency quality, or maintainer risk aversion. We place these considerations at the forefront of our structural model. Finally, sample evidence suggests that (1) higher quality packages have import dependencies and (2) well documented packages are more likely to serve as dependencies.

6 Structural Approach

Reduced form analysis serves as a starting point for characterizing key empirical patterns that begin to illustrate the framework outlined in Section 3. We next seek to formalize the microeconomic behavior of software project maintainers in an effort to explain how dependency networks evolve over time and deliver benefits to users. Using the network formation model suggested by Hsieh et al. (2022) as a basis, our structural approach models the coevolution of both individual software projects and the dependency network. The structural model allows us to conduct two distinct types of counterfactual policy analysis and assess changes to equilibrium welfare, which we measure as the aggregate time cost for software developers. First, we can perturb structural parameters such as the distribution of maintainer risk aversion or variation in project quality. Second, we can simulate the removal of "key projects" (Ballester et al., 2006).

6.1 Setup

The setup of the structural model follows the framework from Section 3. We specify a project quality relation, contribution costs, preferences, and information available to each maintainer.[77]

6.1.1 Project Quality

Project quality y_i is a function of (1) contribution effort $x_i > 0$ and (2) dependency relationships summarized by the directed network:

$$y_i(x_i, y_{-i}, G) = b_i x_i + \beta \sum_{j \neq i} G_{ij} y_j + \xi_i \quad \forall i \in \mathcal{N}. \tag{9}$$

Here, b_i is the marginal product of manager i's contribution and β captures the attenuation factor over quality derived from project i's upstream dependencies.[78] The term ξ_i are unobservable influences that partially determine project quality.

[77]While the model structure captures a sequence of static equilibria over a number of periods, we will suppress the time subscript throughout most of Sections 6.1 and 6.2 for the sake of streamlined notation. This should not affect any implications of the model. Assumption 3 in Section 6.2.1 discusses the specific sequence which these static equilibria follow.

[78]During estimation, we include a constant term in Equation (9), similar to the reduced form analog in Equation (6), which we omit here to simplify notation.

We opt for a linear quality specification in Equation (9) to simplify the mathematics of strategic network formation under conventional methods (Mele, 2017; De Paula, 2020; Badev, 2021; Hsieh et al., 2022). This assumption is not without perils. For example, this functional form implies that dependencies linearly and continuously influence dependent quality as a function of their size. In reality, the addition or removal of a key dependency may make or break a package, suggesting that quality effects are non-linear. In its current form, the best we can do is adapt our data such that the specification in Equation (9) is log-linear. Either innovations in strategic network formation modeling or a completely different methodological approach are required to account for more arbitrary kinds of non-linearity.

6.1.2 Contribution costs

Contribution costs are assumed to be a convex function of effort level $x_i > 0$:

$$c_i(x, G) = \frac{1}{2}x_i^2 - \left(a_i + \alpha \sum_{j \neq i} G_{ij} x_j\right) x_i. \tag{10}$$

Notice that c_i is decreasing in a_i and α, which capture manager i's own productivity and any productivity spillovers from contribution in upstream dependencies, respectively.[79] Compared with parameters (b_i, β) in (9) which capture the marginal productivity of contribution effort in terms of project quality, parameters (a_i, α) in (10) allow us to distinctly characterize contribution productivity in terms of marginal costs. As discussed in Section 5, however, if upstream and downstream contribution are gross complements, then $\alpha < 0$ and dependency usage imposes net costs on the maintainer.

[79] While contribution costs in Equation (10) are expressed in rather arbitrary terms, we can derive a mapping between contribution costs implied by the structural model and time allocation (hours) to project development, ω_i, observed in the empirical sample. Given estimates for a_i, α and data x, G, we can estimate parameters γ_0, γ_1 from a simple linear specification:

$$c_i(x, G) = \gamma_0 + \gamma_1 \omega_i + \epsilon_i$$

6.1.3 Preferences

Project managers derive utility from their expected private valuation of their project:

$$u_i(x, y, G) = \mathbb{E}\left[v_i\left(y_i\right)\right]. \tag{11}$$

By allowing variation in the Bernoulli function $v_i(\cdot)$, we capture the idea that maintainers will differ with respect to how much dependency risk they are willing to take on. In particular, we will assume some level of concavity in the function $v_i(\cdot)$:

Assumption 1 (Exponential Utility). *$v_i(z; r_i)$ is an exponential or constant absolute risk aversion (CARA) utility function*

$$v_i(z; r_i) = -e^{-r_i z} \tag{12}$$

where the absolute risk aversion parameter, $r_i > 0$, varies across maintainers $i \in \mathcal{N}$.

Under Assumption (1), the parameter $r_i \in \mathbb{R}$ captures project manager i's relative level of risk aversion: maintainer i is said to be more risk averse as $r_i \to \infty$. Since a portion of project quality in Equation (9) is uncertain and unobservable, the expected quality preferences under Equation (11) and Assumption 1 together imply that as a project manager becomes more risk averse, she suffers greater disutility with increased volatility of both her own package and the inherited volatility of upstream dependencies. We make an assumption over the specific form of this uncertainty in the following section.

6.1.4 Information Sets

To introduce uncertainty over project quality, we assume that stochastic quality disturbances ξ_i are unobservable to maintainers *ex ante*.

Assumption 2 (Uncertainty in Package Quality). *Assume the following*

1. *$\xi \overset{iid}{\sim} N(0, \Sigma)$ where $\Sigma = I\sigma^2$ and $\sigma^2 = (\sigma_i^2)_{i \in \mathcal{N}}$ are known only in distribution by project maintainers.*

2. *ξ is independent and identically distributed across time periods.*

3. *Observables (x, y, G) and parameters $\theta = (a, \alpha, b, \beta, r, \Sigma)$ are public information to all maintainers $i \in \mathcal{N}$.*

Therefore, maintainers are uncertain about the quality of all projects, including their own. The risk averse maintainer will enjoy greater utility when she takes actions to minimize exposure of her project to any sources of quality risk. Since the distribution ξ is common knowledge, the setting is characterized by a shared level of uncertainty rather than information asymmetries between agents (e.g., Akerlof (1978)).

6.2 Equilibrium

With the basic elements of the structural model now established, we now discuss how maintainers are expected to behave in equilibrium. Assume that each project maintainer $i \in \mathcal{N}$ chooses a tuple $(x_i^\star, \{G_{ij}^\star\}_{j \neq i})$ to minimize development costs $c_i(x, G)$ while keeping their private utility of expected project quality[80] $\mathbb{E}\left[v_i\left(y_i\right)\right]$ above a threshold $\underline{u}_i$. We can express the maintainer's static cost minimization problem as

$$\begin{aligned} \min_{x_i \geq 0, \{G_{ij}\}_{j \neq i}} \quad & c_i(x, G) \\ \text{s.t.} \quad & u_i(x, y, G) \geq \underline{u}_i, \end{aligned} \tag{13}$$

where c_i, y_i, and u_i are defined in Equation (10), Equation (9), and Assumption 1, respectively. We analyze the equilibrium of this system is several distinct phases.[81] First, we must make an assumption over the sequence of project development choices for each individual maintainer. Second, we characterize the equilibrium choices of the continuous quantities $(y^\star, x^\star)$. Third, we derive an expression for the project maintainer's expected utility over the quality of their project in equilibrium, $\mathbb{E}\left[v_i(y_i^\star)\right]$. Fourth, we characterize factors influencing the formation and dissolution of software dependencies and derive an expression for $\Pr(G_{ij} = 1)$, the probability that maintainer i imports project j. Finally, we conclude with some comparative statics for equilibrium quantities.

[80] Or put more precisely, maintainer i's private valuation of expected project quality.

[81] It should be noted that the maintainer's optimal choice over contribution levels and dependencies can be represented with alternative formulations. We present some of these alternatives in Appendix C.1 and discuss how we use them to map between the exposition here in Section 6.2 and structural estimation covered in Section 6.3.

6.2.1 Timing

Maintainers myopically best respond to solve the development cost minimization problem in Equation (13), conditional on both the state of the system at the beginning of the period, $\mathcal{D}_{t-1}$ and the optimal strategies of other maintainers. Here the state of the dependency ecosystem $\mathcal{D}_t \equiv \{y_t, x_t, G_t, W_t\}$ is defined as it was in Section 5.[82]

Assumption 3 (Timing)**.** *At the beginning of each period t, a single maintainer $i \in \mathcal{N}$ is presented with the opportunity to change the dependency relationships of their software project. Next, all agents $i, j \in \mathcal{N}$ adjust their contribution levels under the new network. These developments unfold according to the following sequence:*

(S1) Maintainer i chooses a set of optimal dependencies $\{G^{\star}_{ijt}\}_{j\neq i}$, conditional on the state of the ecosystem in the last period, $\mathcal{D}_{t-1}$. This updates the network to $G_{t-1} \mapsto G^{\star}_t$.

(S2) In accordance with the response functions derived in Equation (13), all agents $i, j \in \mathcal{N}$ determine their best response contribution levels $x^{\star}_{it}$ under the new network $G^{\star}_t$. This updates the remaining observables in the ecosystem: $x_t \mapsto x^{\star}_t$, $y_{t-1} \mapsto y^{\star}_t$, and $W_{t-1} \mapsto W_t$.

Therefore, by the end of (S2), $\mathcal{D}_{t-1} \mapsto \mathcal{D}_t$.

The purpose of Assumption 3, beyond providing some structure to the game, is to connect the data generating process of the observed data to an estimation strategy that we outline in Section 6.3.[83] Observing the disaggregated evolution software dependency networks over time allows us to model network formation as a sequence choices made by individual agents in each period, greatly simplifying the estimation procedure compared situations often found in the literature[84] in which only a single network observation is observed. In the following two sections, we derive a characterization of the equilibrium data generating process via backwards induction of the maintainer's game.

[82]Note that $\mathcal{D}_t$ tracks other observable features of projects W_t even though it has no bearing on the structural model as specified.

[83]In the empirical sample, a software project and its dependencies when a new version is registered with the `npm` registry. Hence the timing of our model associates each sample moment $t \in \mathcal{T}$ with a single agent i who then makes $j \neq i$ linking decisions, $\{G_{ijt}\}_{j\neq i}$. This updates the network $G_{t-1} \mapsto G_t$. Then we allow all agents to optimally adjust their contribution levels conditional on the new network G_t so that $(x_{t-1}, y_{t-1}, W_{t-1}) \mapsto (x_t, y_t, W_t)$.

[84]For example, Leung (2015), Mele (2017), Christakis et al. (2020), and Ridder and Sheng (2020), to name a few.

6.2.2 Optimal contribution decision (x_i)

We first derive equilibrium contribution effort $x^\star$ and project quality $y^\star$, taking the dependency network G as given. The first order necessary conditions for maintainer i's optimal choice of $x_i > 0$ imply[85]

$$x_i^\star = a_i + \alpha \sum_{j \neq i} G_{ij} x_j^\star. \tag{14}$$

In matrix form, the system described in Equation (14) becomes $x^\star = Aa$ where $A \equiv (I - \alpha G)^{-1}$ and $a = (a_i)_{i \in \mathcal{N}}$. For A to exist, it must be that $|\alpha| < 1$. Therefore, $x_i^\star = \sum_j A_{ij} a_j$. In equilibrium, this implies project quality will be given by

$$y_i^\star = b_i \left(a_i + \alpha \sum_{j \neq i} G_{ij} x_j^\star \right) + \beta \sum_{j \neq i} G_{ij} y_j^\star + \xi_i. \tag{15}$$

Similarly, $y^\star = B\left(b \circ x^\star + \xi\right) = B\left(b \circ (Aa) + \xi\right)$ where $B \equiv (I - \beta G)^{-1}$, $|\beta| < 1$, and $b \circ x^\star$ denotes the Hadamard product of the vectors b and $x^\star$: $(b \circ x^\star)_i = b_i x_i^\star$ for $i \in \mathcal{N}$.

Remark (Leontief Inverse for Software Dependency Networks)**.** *The matrices A and B would be known as the "Leontief inverse" in the literature on input-output modelling and capture the extent to which the effects, such as increased contribution or fluctuations in quality, percolate through the dependency network G to affect the welfare of dependents. This is the source of network externalities in this framework. Notice that if the spectral radius of αG is less than 1, then $A = I + \sum_{k=1}^{\infty} \alpha^k G^k$.*[86] *Roughly speaking, if maintainer j increases her contribution effort by 1%, then maintainer i will be induced to increase her contribution effort by $A_{ij} a_j$%. This is the source of productivity and quality externalities: the use of dependency j influences the marginal cost of contribution for maintainer i in her own project. For project quality, the influence of dependencies can operate in different directions since the relative influence of dependency j on project i, $B_{ij} = b_j x_j + \xi_j$, is the sum of*

[85] Technically, since the first-order necessary conditions for the maintainer's problem in System (13) imply $\frac{\partial c_i(x^\star, G)}{\partial x_i} = \lambda_i \frac{\partial u_i(x^\star, y^\star, G)}{\partial x_i}$ for $x_i^\star > 0$ and a Lagrange multiplier $\lambda_i \geq 0$, the parameter a_i in equilibrium condition Equation (14) does not exactly equal the parameter a_i from the maintainer's cost function in Equation (10). We take a simplifying approach to represent equilibrium effort choice in Equation (14) to simplify both exposition and estimation. We discuss these details in Appendix C.1, providing alternative characterizations for the maintainer's problem in (13) and provide conditions under which the equilibrium allocations $(x^\star, y^\star)$ across these characterizations coincide. See Proposition 1 in Appendix C.1.

[86] This concept of attenuating indirect influence from dependencies further and further upstream is akin to Katz-Bonacich centrality from the perspective of the dependent package. We represent the intuition of this influence in Example 3.1.

an upstream contribution effect $b_j x_j$ and unobservable fluctuations or uncertainty ξ_j.

Finally, these simplifications together imply that equilibrium project quality for project i in Equation (15) can also be expressed as follows:

$$y_i^\star = \sum_j B_{ij} \left(b_j \left(\sum_k A_{jk} a_k \right) + \xi_j \right).$$

Notice that equilibrium project quality is therefore simply a function of equilibrium contribution and fluctuations ξ. The advantage of this characterization of project quality implied by Equations (14) and (15) is that a maintainer's utility over the expected quality of their project, $u_i(x, y, G)$, can now be expressed as simply a function of the network G and parameters. We derive an expression for maintainer utility in the following section.

6.2.3 A Maintainer's Utility over Expected Project Quality

Before discussing optimal dependency formation behavior, we first seek to simplify the project manager's expected utility of their project $u_i(x, y, G) = \mathbb{E}[v_i(y_i)]$ under the dependency graph G and equilibrium contribution $x^\star$. We will use the derived expression in the subsequent section to evaluate maintainer i's expected incremental utility from forming a dependency relationship with project j.

Conditional on optimal choices of contribution effort, $x^\star$, the resulting project quality $y^\star$, and the dependency graph G, by the normality of ξ_i stipulated by Assumption 2, $u_i(x^\star, y^\star, G)$ becomes

$$\begin{aligned} u_i(x^\star, y^\star, G) &= -\exp\left(-r_i \sum_j B_{ij} \left(b_j x_j^\star - \frac{r_i}{2} B_{ij} \sigma_j^2 \right) \right) \\ &= -\exp\left(-r_i \sum_j B_{ij} \left(b_j \left(\sum_k A_{jk} a_k \right) - \frac{r_i}{2} B_{ij} \sigma_j^2 \right) \right). \end{aligned} \tag{16}$$

Details for this simplification can be found in Appendix C.2. Equation (16) makes clear the notion that maintainer preferences are shaped by both (1) their relative degree of risk tolerance r_i and (2) the extent to which upstream dependents vary with respect to quality, σ_j^2 for $j \in \mathcal{N}$.

6.2.4 Optimal dependency formation decision (G_{ij})

In the previous section, we saw how preferences reflecting risk aversion influences a manager's expected utility of their project in equilibrium. In this section we explore expected incremental utility changes under the formation of new links. Intuitively, manager i ought only to form a dependency with project j if, conditional on the realization of linking disturbances, their expected utility $u_{ij}^{+} \equiv u_i(x^\star, y^\star, G + ij)$ is greater than the utility they expect without using project j, $u_{ij}^{-} \equiv u_i(x^\star, y^\star, G - ij)$.

First, in order to simplify our modeling of the maintainer's dependency management decision, we make a rather strong assumption that there is no upfront cost to creating or removing dependency relationships. Assumption 4 will change our interpretation of the risk aversion parameter r_i such that it now implicitly subsumes both (1) a level of maintainer risk aversion and (2) the average net benefit project maintainer i derives from importing dependencies. To introduce this assumption, we temporarily introduce the notion of time to this exposition.

Assumption 4 (Costless dependency formation)**.** *Project maintainers incur a cost of zero to either import a dependency, $G^\star_{ijt} = 0 \mapsto G^\star_{ijt+1} = 1$, or remove a dependency, $G^\star_{ijt} = 1 \mapsto G^\star_{ijt+1} = 0$*

With Assumption 4, we can approach the maintainer's dependency management using a conventional methodology (Jackson and Wolinsky, 1996). However, we must also address an additional complication arising from our specification of maintainer preferences. The non-linearity of $u_i(\cdot)$ is to designed reflect the fact maintainers may vary with respect to the amount of risk they are willing to introduce into their own project by importing dependencies. This is a notable departure from much of the literature on strategic network formation, which typically utilize linear utility specifications to simplify the calculation of incremental changes to utility under different links (De Paula, 2020).

To address this complication, we first make an assumption on the way in which stochastic and unobserved utility shocks, realized when either forming or not forming the dependency, enter into this decision. This assumption will then allow us to adopt a change-of-variable technique suggested by Fosgerau and Bierlaire (2009) in an effort to facilitate easier estimation of parameters within the maintainer's discrete choice problem, exploiting the fact that by Assumption 1, $u_i(\cdot) = \mathbb{E}\left[v_i(\cdot)\right] < 0$

over its domain. Assume that ε_{ij}^{+} and ε_{ij}^{-} are stochastic and unobserved utility shocks realized by maintainer i from either forming (ε_{ij}^{+}) or not forming (ε_{ij}^{-}) the dependency with j.

Assumption 5 (Link Formation with Multiplicative Disturbances)**.** *Assume $\varepsilon_{ij}^{+}, \varepsilon_{ij}^{-} \in (0, +\infty)$ are independent and identically distributed across project pairs and enter the dependency formation problem of the maintainer multiplicatively. Further, assume that Assumption 4 holds. Upon learning a realization for $(\varepsilon_{ij}^{+}, \varepsilon_{ij}^{-})$, project manager i uses project j as a dependency according to the following rule:*

$$G_{ij} = 1 \iff u_{ij}^{+}\varepsilon_{ij}^{+} \geq u_{ij}^{-}\varepsilon_{ij}^{-} \tag{17}$$

and $G_{ij} = 0$ otherwise.

Following the linearization transformation of Fosgerau and Bierlaire (2009), we can show that under Assumption 5, the equilibrium probability that maintainer i imports project j as a dependency becomes

$$\Pr(G_{ij} = 1) = F_{\epsilon}\left(r_i \underbrace{\left(\sum_j \Delta B_{ij} b_j \left(\sum_k \Delta A_{jk} a_k\right)\right)}_{Z_{0ij}} - \frac{1}{2} r_i^2 \underbrace{\left(\sum_j \Delta B_{ij}^2 \sigma_j^2\right)}_{Z_{1ij}} ; \theta_{\epsilon} \right), \tag{18}$$

where θ_{ϵ} is a vector of parameters for the random variable $\epsilon_{ij} \equiv \epsilon_{ij}^{+} - \epsilon_{ij}^{-} \sim F_{\epsilon}$. Complete details for this derivation can be found in Appendix C.3.[87]

In Equation (18), we define $\Delta B_{ij} \equiv B_{ij}^{+} - B_{ij}^{-}$ and $\Delta A_{jk} = A_{jk}^{+} - A_{jk}^{-}$ as the difference between elements of the Leontief inverse matrices for project quality and contribution effects under $G + ij$ and $G - ij$.[88] Intuitively, ΔB_{ij} and ΔA_{ij} will reflect the change in exposure to net quality and contribution cost influences that arise when maintainer i imports project j as a dependency.

[87]Briefly, this simplification works as follows. First, define $\overline{u}_{ij}^{+} \equiv -\lambda \ln(-u_{ij}^{+})$ and $\overline{u}_{ij}^{-} \equiv -\lambda \ln(-u_{ij}^{-})$. Second, define $\epsilon_{ij}^{+} \equiv -\lambda \ln(\varepsilon_{ij}^{+})$, $\epsilon_{ij}^{-} \equiv -\lambda \ln(\varepsilon_{ij}^{-})$. Third, define $\epsilon_{ij} \equiv \epsilon_{ij}^{-} - \epsilon_{ij}^{+} \overset{\text{iid}}{\sim} F_{\epsilon}(z; \theta_{\epsilon})$ where $\lambda > 0$ and $u_{ij} \equiv \overline{u}_{ij}^{+} - \overline{u}_{ij}^{-}$, $\epsilon_{ij} \equiv \epsilon_{ij}^{-} - \epsilon_{ij}^{+}$. Ultimately, $\Pr(G_{ij} = 1) = \Pr(u_{ij} \geq \epsilon_{ij}) = F_{\epsilon}(u_{ij}; \theta_{\epsilon}/\lambda)$ connects equilibrium linking behavior in Assumption (5) with the empirical likelihood of observing a dependency relationship in Equation (18).

[88]Let $\Delta B_{ij} \equiv B_{ij}^{+} - B_{ij}^{-}$ where $B + ij = [B_{ij}^{+}]_{i,j \in \mathcal{N}} = (I - \beta(G + ij))^{-1}$ and $B - ij = [B_{ij}^{-}]_{i,j \in \mathcal{N}} = (I - \beta(G - ij))^{-1}$. Equivalently, let $\Delta A_{jk} = A_{jk}^{+} - A_{jk}^{-}$ where $A + ij = [A_{jk}^{+}]_{j,k \in \mathcal{N}} = (I - \alpha(G + ij))^{-1}$ and $A - ij = [A_{jk}^{-}]_{j,k \in \mathcal{N}} = (I - \alpha(G - ij))^{-1}$.

For the purposes of exposition, we rearrange u_{ij} into a quadratic function of r_i and label the coefficients Z_{0ij} and Z_{1ij} in the last equality of Equation (18). These coefficients are functions of the network G and parameters $(\alpha, a, \beta, b, \Sigma)$. In Section 6.3 and Appendix D, we show that since $(\alpha, a, \beta, b, \Sigma)$ can be estimated using moment conditions contained in Equations (9) and (14) via generalized method of moments, the result in Equation (18) will form the basis for a likelihood function for the remaining unknown parameters, r and θ_ϵ. Hence, under the assumptions outlined in this section, r and θ_ϵ can be estimated via maximum likelihood.

Finally, a distributional assumption over ϵ_{ij} to refine Assumption 5 helps inform comparative statics in Section 6.2.5 and the estimation procedure described in Section 6.3.

Assumption 6. *ϵ_{ij} is a logistic random variable, independent and identically distributed across potential links and time.*

Note that Assumption 6 can be supported if conditional on Assumption 5, ϵ_{ij}^+ and ϵ_{ij}^- are assumed to be mutually (i.e., across both project pairs and time) independent Gumbel random variables.

6.2.5 Comparative Statics

How do various parameters influence the dependency formation in equilibrium? Consider the effect of slight perturbations to parameters on the likelihood of dependency formation (Equation (18)):

1. **Risk Aversion (r_i):** Using the likelihood that maintainer i imports project j, one can show that

$$\frac{\partial \Pr(G_{ij} = 1)}{\partial r_i} \begin{cases} < 0 & \text{if } Z_{ij} < r_i \\ \geq 0 & \text{if } Z_{ij} \geq r_i \end{cases}$$

since $\frac{\partial F_\epsilon}{\partial z} > 0$ and $r_i > 0$ and $Z_{ij} \equiv \frac{Z_{0ij}}{Z_{1ij}}$. Therefore, maintainer i becomes less likely to import dependency j as her risk aversion r_i increases beyond a threshold, Z_{ij}. We can interpret Z_{ij} as a net benefit threshold for maintainer i. If $Z_{ij} < r_i$, then maintainer i's is risk averse to the point that she considers the additional risk of depending upon project j to outweigh the net benefits in terms of (dis)utility.

2. **Project Quality Variance (σ_k^2):**

$$\frac{\partial \Pr(G_{ij}=1)}{\partial \sigma_k^2} \leq 0$$

Notice also that, since we have ruled out risk seeking preferences by restricting $r_i > 0$, increased project volatility will deter all maintainers to some extent. This effect will be stronger for more risk averse maintainers: $\frac{\partial^2 \Pr(G_{ij}=1)}{\partial r_i \partial \sigma_k^2} \leq 0$.

In other words, conditional on our set of assumptions and chosen functional forms, more risk averse maintainers are less likely to rely on dependencies, *ceteris paribus*, once they are beyond a certain threshold.[89] Similarly, increased volatility in project quality reduces the likelihood of dependency formation. Overall, while intuitive, these comparative statics reveal a level of sophistication embedded in the dependency management choice that confronts the project maintainer. Predicting dependency formation is a function of a variety of different influences. These complexities ought to guide the design and interpretation of simulated counterfactuals

6.3 Estimation

In general, estimating strategic network formation models is complicated. Two broad classes of approaches deal with estimating network formation when only a single network observation is available: (1) non-iterative estimation using link formation strategy is assumed to take place under incomplete information (Leung, 2015; Ridder and Sheng, 2020) and (2) iterative strategic network formation where opportunities to form or dissolve links arrive according to some specified sequence (Mele, 2017; Christakis et al., 2020; Badev, 2021; Hsieh et al., 2022). While our model falls into the latter class, we can further exploit the fact that the dependency network is effectively observed in continuous time. In this case, the complete sequence of linking decisions is known to the econometrician, an advantage not present in empirical settings where only the final equilibrium network is observed. By Assumption 3, we leverage this feature to model the data generating process as a Markov chain: in each period, a single maintainer makes new or revises existing dependency decisions based on the current state of the network. After this dependency revision is made, all agents can subsequently

[89] In other words, as $r_i \to \infty$.

adjust their contribution levels under the new network, and the process repeats with another maintainer's link decision. Our estimation framework is therefore quite similar to the approach described in Snijders et al. (2010), as the empirical setting bears a closer resemblance to a network panel and the model falls into a broad class of "stochastic actor-oriented models". Consequently, our model of the coevolution of both contribution actions and dependency formation decisions is actually much simpler than similar approaches in which only a single network observation is available (Badev, 2021; Hsieh et al., 2022). In these cases, a high-dimensionality state space of potential networks and action profiles must be tracked[90] in order to explain observed equilibria.

We relegate full details and discussion of our structural estimation strategy (i.e., the moment conditions and likelihood function) to Appendix D. In broad strokes, the procedure can be described as follows. The observed data is $\mathcal{D} = (x_t, y_t, G_t, W_t)_{t \in \mathcal{T}}$. Structural parameters to be estimated are $\theta = (a, \alpha, b, \beta, \Sigma, r, \theta_\epsilon)$.[91] We break estimation into two phases. In the first phase described in Steps 1 through 3, we use moment conditions and identities from the structural model to recover estimates for $(a, \alpha, b, \beta, \Sigma)$ using the generalized method of moments (GMM).[92] In the second phase described in Step 4, we combine observed data with the parameter estimates in the first phase to estimate (r, θ_ϵ), using equilibrium dependency formation characterized in Equation (18) in maximum likelihood estimation (MLE).

6.3.1 Dimensionality Reduction

As discussed previously in the description of the empirical sample, we take steps to reduce the dimensionality to facilitate and simplify structural estimation. First, we limit the size of the empirical sample by beginning with a "seed" of just 10 core projects, sample upstream[93], and restrict the set of observed sample moments to only minor versions of package releases. Second, we restrict the range of the risk aversion parameter r to the half-interval $(0, 1]$ as it is somewhat of a nuisance parame-

[90] Badev (2021) and (Hsieh et al., 2022) use an approach in which behavior is determined by an exact potential game, the equilibrium of which can be characterized by a Gibbs measure. Both the presence of externalities and the non-linear nature of our structural model would make the use of a similar method quite complicated.

[91] While not a part of the discussion of the structural model in the main body of Section 6, we introduce the parameter γ in Appendix C.

[92] We also describe how to use simpler methods such as ordinary least squares (OLS) to recover these parameters in sequence.

[93] As opposed to sampling downstream. The top 10 most depended upon packages in the NPM ecosystem features tens of thousands of downstream dependent packages each.

ter which is really only of interest in distribution to characterize the DGP. From a computational perspective, we discuss additional simplifications to ease structural estimation in Appendix D.1.

7 Counterfactual Analysis

Developing a structural model allows us to completely characterize the data generating process for the decision-making process of project maintainers and the evolution of the software dependency network. Importantly, it allows us to explore the effect of counterfactual interventions on the resulting equilibrium of the system. Specifically, we can either (1) perturb parameters or (2) remove certain key projects, re-simulate the data generating process[94], and analyze the impact of the counterfactual on contribution, project quality, contribution costs, and project maintainer welfare (i.e., utility).

To evaluate the impact of each counterfactual, we first must define a social welfare function for the software dependency graph G, conditional on the set of project $\mathcal{N}$ and parameters θ:

$$u_{\mathcal{N}}(G;\theta) = \sum_{i\in\mathcal{N}} u_i(G;\theta)^{1/\lambda_i}. \tag{19}$$

Notice several notational simplifications.[95] Importantly, we rescale each maintainer's utility by a factor λ_i. For the purposes of our counterfactual analysis, we set $\lambda_i = r_i$ for all $i \in \mathcal{N}$, effectively normalizing welfare by the risk aversion profile. This is done so that differences in welfare under a given network is not driven by variation in maintainer risk aversion. We also consider simpler aggregate functions for contribution levels, project quality, and contribution costs. For each counterfactual, we specify a change to parameters or the set of projects and then simulate the data generating process under this new system, beginning at the beginning of the sample period.[96] Using the aggregate welfare functions, we compare the counterfactual equilibrium for the final sample period[97] with a baseline based on the observed data.

[94] That is, we can use the arrival sequence of projects observed in the sample moments and use the equilibrium conditions of the structural model to simulate contribution decisions, project quality evolution, and dependency formation decisions from for the sample period from start to finish.

[95] Since each y_i is ultimately just a function of x, we can write $u_i(x, G;\theta)$ and $u_{\mathcal{N}}(x, G)$. Furthermore, in equilibrium $x^\star$ is really just a function of the network G and parameters θ, we could even go one step further to write $u_{\mathcal{N}}(G;\theta)$.

[96] We take the average of 10 counterfactual simulations.

[97] September 2022

7.1 Reducing Fluctuations in Project Quality

Our discussion of structural model comparative statics in Section 6 establishes that risk averse project maintainers are less likely to use packages highly volatile in quality as dependencies. Moreover, both conventional wisdom and empirical evidence suggest developers are reluctant to import dependencies that are either immature or subject to frequent, backwards-incompatible changes (Zerouali et al., 2018). Innovations in software best practices can promote stability in the quality of OSS projects, such as including testing frameworks to ensure intended functionality (Ellims et al., 2006), using automation and continuous integration to efficiently and safely integrate contributions from the wider community (Vasilescu et al., 2015; Hilton et al., 2016), keeping the design scope of the project focused and succinct[98], and using systems like semantic versioning to release software often but in a manner respectful of downstream dependents (Raemaekers et al., 2017; Decan and Mens, 2019). For the purposes of the counterfactual analysis, we specifically consider alternative levels of project quality volatility $\Sigma \mapsto \Sigma'$ for $\Sigma' \in \{0.5\Sigma, 2\Sigma, 4\Sigma\}$.

7.2 Increasing Developer Risk Aversion

While some theoretical (Walsh and Schneider, 2002) and experimental (Kina et al., 2016) analyses of risk aversion for software developers exist, there is comparatively little empirical evidence of how risk tolerance influences project maintainer decision-making and the resulting equilibrium. In our structural model, the likelihood to import dependencies increases with a maintainer's level of risk tolerance, which has the potential to improve package quality and reduce development costs. On the other hand, increased risk aversion may prevent risky dependency relationships from being formed, albeit at increased contribution costs. In our counterfactuals, we modify the profile of maintainer risk aversion $r \mapsto r'$ for $r' \in \{r + \sigma_r, \mathbf{1}, \min(r)\}$.[99]

[98]Recall the UNIX philosophy: "Make each program do one thing well."

[99]That is, when risk aversion for each project maintainer is increased by one standard deviation, equal to 1, and equal to the minimum value of the estimated from the sample.

7.3 Key Projects

In the spirit of the "key player analysis" described by Ballester et al. (2006), Lee et al. (2021), and Hsieh et al. (2022), we define a key software project $i^\star$ such that

$$i^\star = \arg\max_{i \in \mathcal{N}} u_{\mathcal{N}}(x, G; \theta) - u_{\mathcal{N} \setminus i}(x, G; \theta), \tag{20}$$

conditional on a set of parameters θ. We could determine the key project $i^\star$ by iteratively simulating aggregate welfare by Equation (20). However, simulating 1,263 counterfactuals entails a significant computational burden. We therefore opt for a simpler approach of removing the package with the most downstream dependents in the final period observed in the sample. The package with the most downstream dependents in our sample is `babel`, a "tool that helps you write in the latest version of JavaScript" (Babel, 2014).[100] Additionally, we estimate a counterfactual equilibrium after removing the top ten packages ranked by number of downstream dependents in the final sample period. For the key player analysis, we compare the welfare of the remaining packages under the baseline with their outcomes in a world in which the critical set of packages is removed. In this way, we are estimating the value of externalities the key packages generate for the software network as a whole.

7.4 Summary

We present the results of the counterfactual analysis in Table 5. Overall, the overall impact of our counterfactuals is comparatively small in percentage terms. This result is largely driven by the fact that none of our counterfactuals significantly alters the network formation path. We argue this pattern is driven by the fact that individual project features largely drive project management decisions, which are in turn robust to marginal perturbations to project quality volatility, risk aversion, and the removal of core packages.[101]

In particular, contribution is virtually unchanged under different levels of risk aversion. On the other hand, the results in Table 5 seem to indicate that significantly increasing maintainer risk aversion

[100] As of October 2022, the `babel` package has over 15,450 commits, 1,033 distinct contributors, 5,500 forks, and 41,000 stars on GitHub.

[101] We should also be forthcoming that this lack of influence on the data generating process may arise because our model largely treats packages as perfect substitutes. We could enrich further analysis with better information on each package's particular functionality.

($r_i' = 1$) can increase aggregate package quality (2.11%) enough to effectively offset the direct welfare effects[102] (-0.04%). In other words, upstream maintainers can create value for downstream dependents by exercising more discipline when choosing what software to rely on that in turn offsets increased disutility from quality uncertainty in aggregate.[103]

The effect of increasing package volatility is slightly more puzzling. We can see that reducing volatility actually has a small increase on aggregate project quality (0.19%). Increasing volatility has a larger positive influence on aggregate quality. There is virtually no change to contribution patterns or welfare. We argue this is a result of strong package fixed characteristics: maintainers seem less concerned with uncertainty over package unobservables compared with their immediately appreciable benefits.

Finally, our counterfactuals that remove key packages reveal the most interesting results. Core packages with the highest Katz-Bonacich centrality measures create significant value for the dependency network: removing the top 10 packages, which number less than 0.8% of the sample, reduces aggregate package quality by -5.73% for their remaining peers. Moreover, aggregate contribution falls by -1.3%, suggesting that maintainers find contribution in their own packages complementary with these upstream core packages.

8 Discussion

In an effort to understand the dynamics and value created by software dependency networks, we have studied micro-founded decision-making from the perspective of the maintainer of an OSS project. We have developed both reduced form and structural methodologies and brought them to bear on an empirical sample of 1,263 Node.js packages observed over time. Overall, we find that individual project features are largely responsible for driving maintainer decisions. In our reduced form approach, we find that upstream projects have relatively limited effect on downstream quality and contribution levels on average. Complementarity between upstream and downstream contribution was greatest in the earlier periods of the dependency network. Our structural approach, to the

[102] $\frac{\partial u_i}{\partial r_i} < 0$ for $r_i > 0$.

[103] One could also interpret this as shifting the burden of risk from dependents to upstream maintainers.

best of our knowledge, is the first attempt to micro-found the cost minimization decision of a risk averse software maintainer within a strategic network formation model. In doing so we can characterize aggregate network evolution as the aggregation of individual decision-making over time. Our counterfactuals reveal that while the network formation process is relatively robust to perturbing individual parameters like maintainer risk aversion and project quality volatility, removing highly critical core dependencies can have outsized influences on package quality downstream.

A better understanding of software dependency formation and its impact on downstream users concerns a broad population of stakeholders and has even garnered attention at the public policy level (Executive Order 14028, 2021). Our study develops a framework that would clearly benefit from extension and further inquiry. In particular, while we have endeavored to cast welfare effects of dependency networks in terms of production costs, estimates of the consumption value of OSS remains an ongoing challenge. Additional research is needed to connect the implications of software dependency management to value created with respect to labor markets, firm profitability, and innovation.

Finally, while we have attempted to characterize the salient features of this setting in our framework, our results seem to indicate that individual project features, as well as features of their maintainers, are critically important to understand welfare effects in detail. A major innovation in this line of inquiry would be to integrate individual characteristics of maintainers themselves. These characteristics have historically been difficult to observe in aggregate, but recent efforts are underway to collect more refined information about OSS collaboration communities (CHAOSS, 2017; Dueñas et al., 2021).

References

Acemoglu, D., U. Akcigit, and W. R. Kerr (2016). Innovation network. *Proceedings of the National Academy of Sciences 113*(41), 11483–11488.

Acemoglu, D., A. Ozdaglar, and A. Tahbaz-Salehi (2015). Systemic risk and stability in financial networks. *American Economic Review 105*(2), 564–608.

Acquisti, A., A. Friedman, and R. Telang (2006). Is there a cost to privacy breaches? an event study. *ICIS 2006 Proceedings*, 94.

Akerlof, G. A. (1978). The market for "lemons": Quality uncertainty and the market mechanism. In *Uncertainty in economics*, pp. 235–251. Elsevier.

Ambrus, A., M. Mobius, and A. Szeidl (2014). Consumption risk-sharing in social networks. *American Economic Review 104*(1), 149–82.

Anwar, A., A. Khormali, D. Nyang, and A. Mohaisen (2018). Understanding the hidden cost of software vulnerabilities: Measurements and predictions. In *International Conference on Security and Privacy in Communication Systems*, pp. 377–395. Springer.

Babel (2014). babel.

Badev, A. (2021). Nash equilibria on (un) stable networks. *Econometrica 89*(3), 1179–1206.

Baldwin, C. Y. and K. B. Clark (2006). The architecture of participation: Does code architecture mitigate free riding in the open source development model? *Management science 52*(7), 1116–1127.

Ballester, C., A. Calvó-Armengol, and Y. Zenou (2006). Who's who in networks. wanted: The key player. *Econometrica 74*(5), 1403–1417.

Bernard, A. B., A. Moxnes, and Y. U. Saito (2019). Production networks, geography, and firm performance. *Journal of Political Economy 127*(2), 639–688.

Bloch, F. and M. O. Jackson (2006). Definitions of equilibrium in network formation games. *International Journal of Game Theory 34*(3), 305–318.

Bloch, F., M. O. Jackson, and P. Tebaldi (2019). Centrality measures in networks. *Available at SSRN 2749124*.

Blume, L., D. Easley, J. Kleinberg, R. Kleinberg, and É. Tardos (2013). Network formation in the presence of contagious risk. *ACM Transactions on Economics and Computation (TEAC) 1*(2), 1–20.

Boldi, P. and G. Gousios (2020). Fine-grained network analysis for modern software ecosystems. *ACM Transactions on Internet Technology (TOIT) 21*(1), 1–14.

Boyter, B. (2018). scc.

Bramoullé, Y., H. Djebbari, and B. Fortin (2020). Peer effects in networks: A survey. *Annual Review of Economics 12*, 603–629.

Bramoullé, Y. and R. Kranton (2007). Risk-sharing networks. *Journal of Economic Behavior & Organization 64*(3-4), 275–294.

Brunfeldt, K. (2014). git-hours.

Carey, P. (2017, 7). Heartbleed's Heartburn: Why a 5 Year Old Vulnerability Continues to Bite. *The Security Ledger*. Accessed: 2022–06–01.

Carvalho, V. M. (2014). From micro to macro via production networks. *Journal of Economic Perspectives 28*(4), 23–48.

Carvalho, V. M., M. Nirei, Y. U. Saito, and A. Tahbaz-Salehi (2021). Supply chain disruptions: Evidence from the great east japan earthquake. *The Quarterly Journal of Economics 136*(2), 1255–1321.

Cavusoglu, H., H. Cavusoglu, and J. Zhang (2006). Economics of security patch management. In *WEIS*. Citeseer.

Chandrasekhar, A. (2016). Econometrics of network formation. *The Oxford handbook of the economics of networks*, 303–357.

CHAOSS (2017). augur.

Choi, S., S. Goyal, and F. Moisan (2019). Network formation in large groups. Technical report.

Christakis, N., J. Fowler, G. W. Imbens, and K. Kalyanaraman (2020). An empirical model for strategic network formation. In *The Econometric Analysis of Network Data*, pp. 123–148. Elsevier.

Coase, R. H. (1937). The nature of the firm. *economica 4*(16), 386–405.

De Paula, Á. (2020). Econometric models of network formation. *Annual Review of Economics 12*, 775–799.

De Weerdt, J. (2002). *Risk-sharing and endogenous network formation.* Number 2002/57. WIDER Discussion Paper.

De Weerdt, J. and S. Dercon (2006). Risk-sharing networks and insurance against illness. *Journal of development Economics 81*(2), 337–356.

Decan, A. and T. Mens (2019). What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering 47*(6), 1226–1240.

Decan, A., T. Mens, M. Claes, and P. Grosjean (2016). When github meets cran: An analysis of inter-repository package dependency problems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Volume 1, pp. 493–504. IEEE.

Decan, A., T. Mens, and E. Constantinou (2018a). On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 404–414. IEEE.

Decan, A., T. Mens, and E. Constantinou (2018b). On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*, pp. 181–191.

Decan, A., T. Mens, and P. Grosjean (2019). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering 24*(1), 381–416.

DeVault, D. (2021, 11). I will pay you cash to delete your npm module. Accessed: 2022–06–01.

Doyle, J. C., D. L. Alderson, L. Li, S. Low, M. Roughan, S. Shalunov, R. Tanaka, and W. Willinger (2005). The "robust yet fragile" nature of the internet. *Proceedings of the National Academy of Sciences 102*(41), 14497–14502.

Dueñas, S., V. Cosentino, J. M. Gonzalez-Barahona, A. del Castillo San Felix, D. Izquierdo-Cortazar, L. Cañas-Díaz, and A. Pérez García-Plaza (2021). Grimoirelab: A toolset for software development analytics. *7*(e601).

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

Ellims, M., J. Bridges, and D. C. Ince (2006). The economics of unit testing. *Empirical Software Engineering 11*(1), 5–31.

Elliott, M., B. Golub, and M. O. Jackson (2014). Financial networks and contagion. *American Economic Review 104*(10), 3115–53.

Elliott, M., B. Golub, and M. V. Leduc (2022). Supply network formation and fragility. *Available at SSRN 3525459*.

Erol, S. and R. Vohra (2018). Network formation and systemic risk. *Available at SSRN 2546310*.

Everett, M. and D. Schoch (2022). An extended family of measures for directed networks. *Social Networks 70*, 334–340.

Executive Order 14028 (2021, 5). Improving the nation's cybersecurity.

Fafchamps, M. and F. Gubert (2007). The formation of risk sharing networks. *Journal of development Economics 83*(2), 326–350.

Fafchamps, M. and S. Lund (2003). Risk-sharing networks in rural philippines. *Journal of development Economics 71*(2), 261–287.

Finifter, M., D. Akhawe, and D. Wagner (2013). An empirical study of vulnerability rewards programs. In *22nd USENIX Security Symposium (USENIX Security 13)*, pp. 273–288.

Fosgerau, M. and M. Bierlaire (2009). Discrete choice models with multiplicative error terms. *Transportation Research Part B: Methodological 43*(5), 494–505.

Galeotti, A. and S. Goyal (2010). The law of the few. *American Economic Review 100*(4), 1468–92.

Goyal, S. and J. L. Moraga-Gonzalez (2001). R&d networks. *Rand Journal of Economics*, 686–707.

Graham, B. and A. De Paula (2020). *The Econometric Analysis of Network Data.* Academic Press.

Graham, B. S. (2020). Network data. In *Handbook of Econometrics*, Volume 7, pp. 111–218. Elsevier.

Grams, C. (2019, 10). How much time do developers spend actually writing code? Acessed: 2022–06–01.

Grossman, S. J. and O. D. Hart (1986). The costs and benefits of ownership: A theory of vertical and lateral integration. *Journal of political economy 94*(4), 691–719.

Hall, B. H., A. Jaffe, and M. Trajtenberg (2005). Market value and patent citations. *RAND Journal of economics*, 16–38.

Helmers, C., M. Patnam, and P. R. Rau (2017). Do board interlocks increase innovation? evidence from a corporate governance reform in india. *Journal of Banking & Finance 80*, 51–70.

Hilton, M., T. Tunnell, K. Huang, D. Marinov, and D. Dig (2016). Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 426–437. IEEE.

Hsieh, C.-S., M. D. König, and X. Liu (2022). A structural model for the coevolution of networks and behavior. *Review of Economics and Statistics 104*(2), 355–367.

Hsieh, C.-S., M. D. Konig, X. Liu, and C. Zimmermann (2018). Superstar economists: Coauthorship networks and research output. *Available at SSRN 3266432*.

IBM (2021, 9). Cost of a Data Breach Report 2021. Accessed: 2022–06–01.

Jackson, J. (2019, 2). To Reduce Tech Debt, Eliminate Dependencies (and Refactoring). Accessed: 2022–06–01.

Jackson, M. O. and A. Wolinsky (1996). A strategic model of social and economic networks. *Journal of Economic Theory 71*, 44–74.

Jaffe, A. B., M. Trajtenberg, and R. Henderson (1993). Geographic localization of knowledge spillovers as evidenced by patent citations. *the Quarterly journal of Economics 108*(3), 577–598.

Katz, J. (2020, 1). Libraries.io Open Source Repository and Dependency Metadata. Version 1.6.0.

Keller, S., G. Korkmaz, C. Robbins, and S. Shipp (2018). Opportunities to observe and measure intangible inputs to innovation: Definitions, operationalization, and examples. *Proceedings of the National Academy of Sciences 115*(50), 12638–12645.

Kikas, R., G. Gousios, M. Dumas, and D. Pfahl (2017). Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 102–112. IEEE.

Kina, K., M. Tsunoda, H. Hata, H. Tamada, and H. Igaki (2016). Analyzing the decision criteria of software developers based on prospect theory. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Volume 1, pp. 644–648. IEEE.

Kovářík, J. and M. J. Van der Leij (2009). Risk aversion and networks: Microfoundations for network formation.

Kovářík, J. and M. J. Van der Leij (2014). Risk aversion and social networks. *Review of Network Economics 13*(2), 121–155.

Kremer, M. (1993). The o-ring theory of economic development. *The Quarterly Journal of Economics 108*(3), 551–575.

Kula, R. G., D. M. German, A. Ouni, T. Ishio, and K. Inoue (2017, may). Do developers update their library dependencies? *Empirical Software Engineering 23*(1), 384–417.

Ladisa, P., H. Plate, M. Martinez, and O. Barais (2022). Taxonomy of attacks on open-source software supply chains.

Lee, L.-F., X. Liu, E. Patacchini, and Y. Zenou (2021). Who is the key player? a network analysis of juvenile delinquency. *Journal of Business & Economic Statistics 39*(3), 849–857.

Lerner, J. and J. Tirole (2002). Some simple economics of open source. *The journal of industrial economics 50*(2), 197–234.

Leung, M. P. (2015). Two-step estimation of network-formation models with incomplete information. *Journal of Econometrics 188*(1), 182–195.

Marbukh, V. (2018). Network formation by contagion averse agents: modeling bounded rationality with logit learning. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 1232–1233. IEEE.

McIlroy, M., E. Pinson, and B. Tague (1978). Unix time-sharing system. *The Bell system technical journal 57*(6), 1899–1904.

Mele, A. (2017). A structural model of dense network formation. *Econometrica 85*(3), 825–850.

Munaiah, N., S. Kroh, C. Cabrey, and M. Nagappan (2017). Curating github for engineered software projects.

Mutton, P. (2014, April). Half a million widely trusted websites vulnerable to Heartbleed bug.

Nagle, F., J. Dana, J. Hoffman, Steven Randazzo, and Y. Zhou (2022, March). Census II of Free and Open Source Software — Application Libraries. Technical report, The Linux Foundation and The Laboratory for Innovation Science at Harvard.

Ohm, M., H. Plate, A. Sykosch, and M. Meier (2020). Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 23–43. Springer.

Pham, N. H., T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen (2010). Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 447–456.

Prana, G. A. A., A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo (2021). Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Software Engineering 26*(4), 1–34.

Raemaekers, S., A. van Deursen, and J. Visser (2017). Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software 129*, 140–158.

Ridder, G. and S. Sheng (2020). Estimation of large network formation games. *arXiv preprint arXiv:2001.03838*.

Robbins, C. A., G. Korkmaz, J. B. S. Calderón, D. Chen, C. Kelling, S. Shipp, and S. Keller (2018). Open source software as intangible capital: measuring the cost and impact of free digital tools. In *Paper from 6th IMF Statistical Forum on Measuring Economic Welfare in the Digital Age: What and How*, pp. 19–20.

Roumani, Y., J. K. Nwankpa, and Y. F. Roumani (2016). Examining the relationship between firm's financial records and security vulnerabilities. *International Journal of Information Management 36*(6), 987–994.

Schlueter, I. Z. (2016, 3). kik, left-pad, and npm. Accessed: 2022–06–01.

Schueller, W. and J. Wachs (2022). Modeling interconnected social and technical risks in open source software ecosystems. *arXiv preprint arXiv:2205.04268*.

Snijders, T. A., J. Koskinen, and M. Schweinberger (2010). Maximum likelihood estimation for social network dynamics. *The annals of applied statistics 4*(2), 567.

Spadini, D., M. Aniche, and A. Bacchelli (2018). PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, New York, New York, USA, pp. 908–911. ACM Press.

Spinellis, D., G. Gousios, V. Karakoidas, P. Louridas, P. J. Adams, I. Samoladas, and I. Stamelos (2009). Evaluating the quality of open source software. *Electronic Notes in Theoretical Computer Science 233*, 5–28.

Techopedia (2017). What is Technical Debt? - Definition from Techopedia. Accessed: 2022–06–01.

Telang, R. and S. Wattal (2007). An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Transactions on Software engineering 33*(8), 544–557.

US CFPB (2022). Equifax Data breach settlement.

US FTC (2022, February). Equifax Data Breach Settlement.

Vasilescu, B., Y. Yu, H. Wang, P. Devanbu, and V. Filkov (2015). Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pp. 805–816.

Walsh, K. R. and H. Schneider (2002). The role of motivation and risk behaviour in software development success. *Information research 7*(3), 7–3.

Wan, Z., Y. Mahajan, B. W. Kang, T. J. Moore, and J.-H. Cho (2021). A survey on centrality metrics and their network resilience analysis. *IEEE Access 9*, 104773–104819.

Williamson, O. E. (1975). Markets and hierarchies: analysis and antitrust implications: a study in the economics of internal organization. *University of Illinois at Urbana-Champaign's Academy for Entrepreneurial Leadership Historical Research Reference in Entrepreneurship*.

Williamson, O. E. (1985). The economic institutions of capitalism: Firms, markets, relational contracting. *University of Illinois at Urbana-Champaign's Academy for Entrepreneurial Leadership Historical Research Reference in Entrepreneurship*.

WIRED (2021, 12). A Log4J Vulnerability Has Set the Internet 'On Fire'. Accessed: 2022–06–01.

Wiz (2021, 12). Log4Shell 10 days later: Enterprises halfway through patching. Accessed: 2022–06–01.

Zerouali, A., E. Constantinou, T. Mens, G. Robles, and J. González-Barahona (2018). An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*, pp. 95–110. Springer.

Zhao, M., J. Grossklags, and P. Liu (2015). An empirical study of web vulnerability discovery ecosystems. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1105–1117.

Zimmermann, M., C.-A. Staicu, C. Tenny, and M. Pradel (2019). Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 995–1010.

A Figures

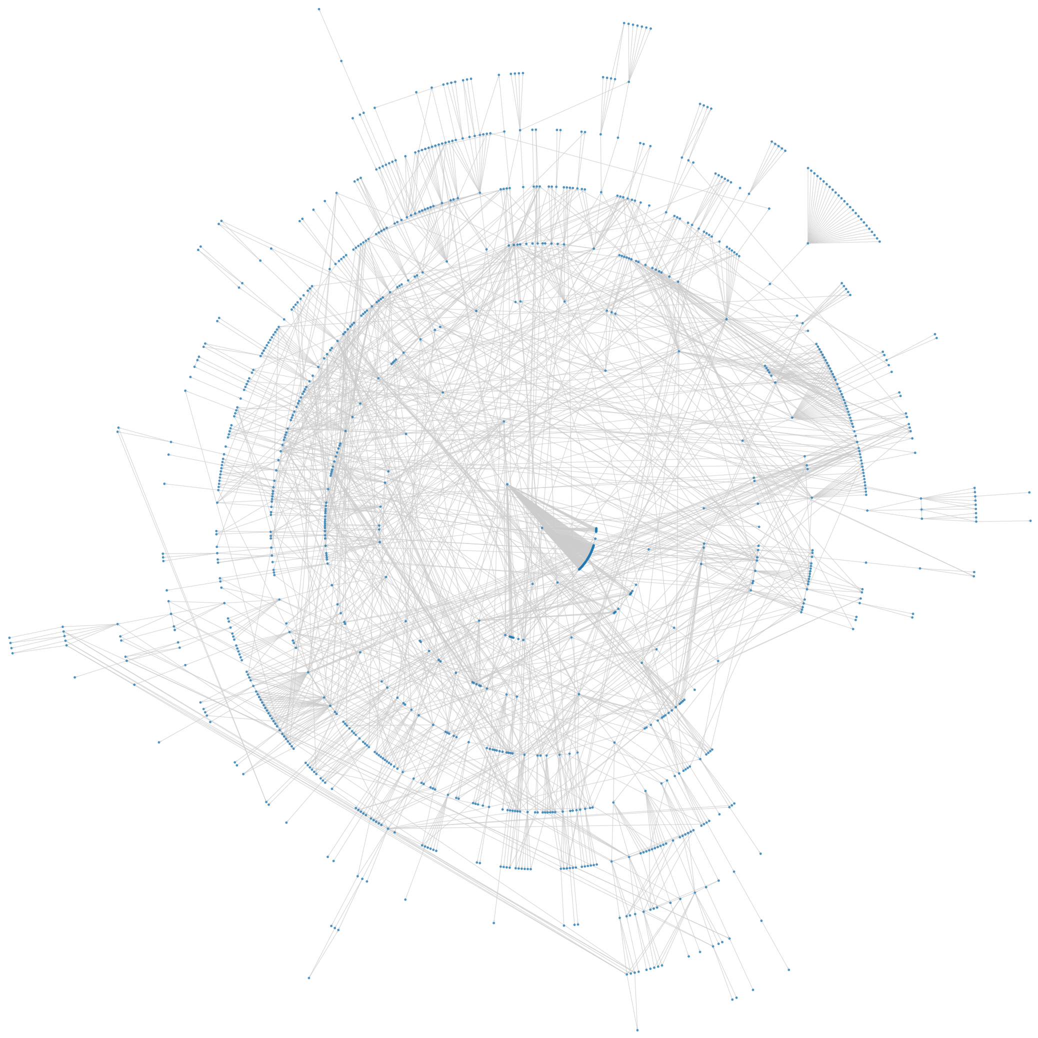


Figure 5: Empirical Node.js Dependency Network Sample (September 2022 snapshot)

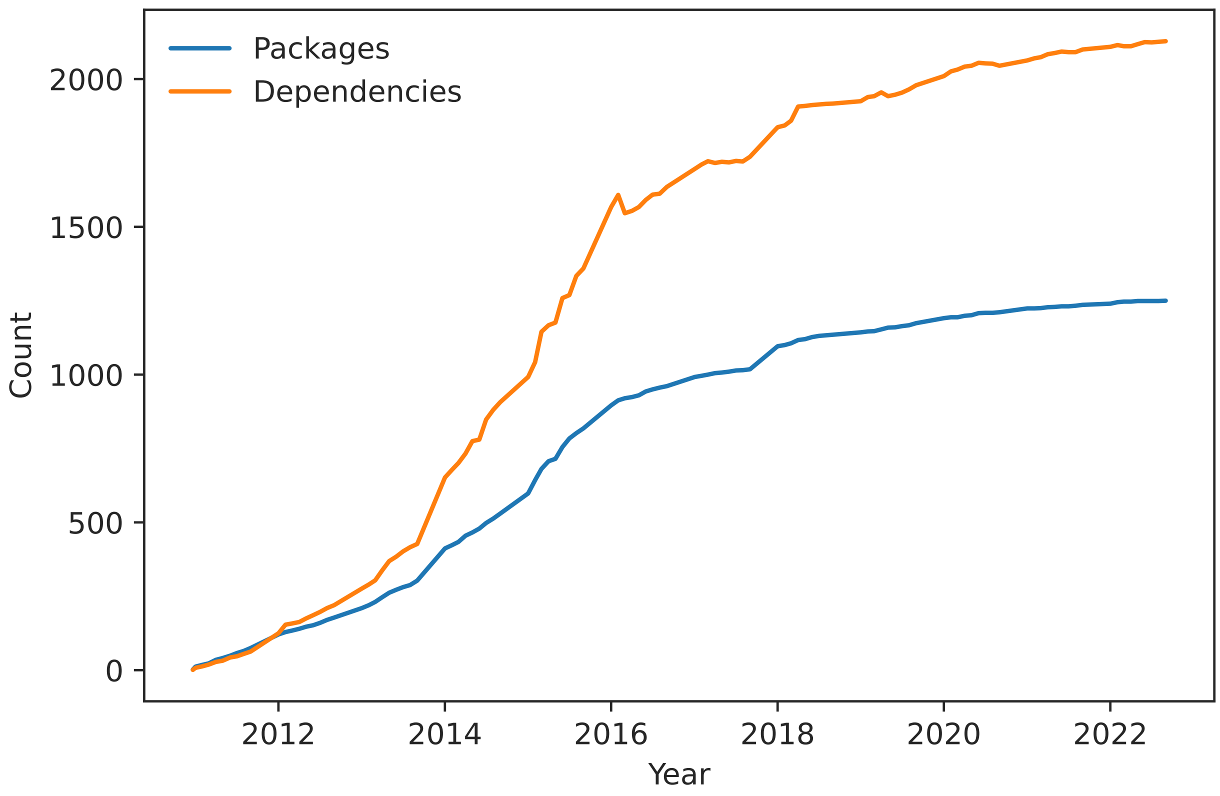


Figure 6: Empirical Dependency Network Sample (growth over time)

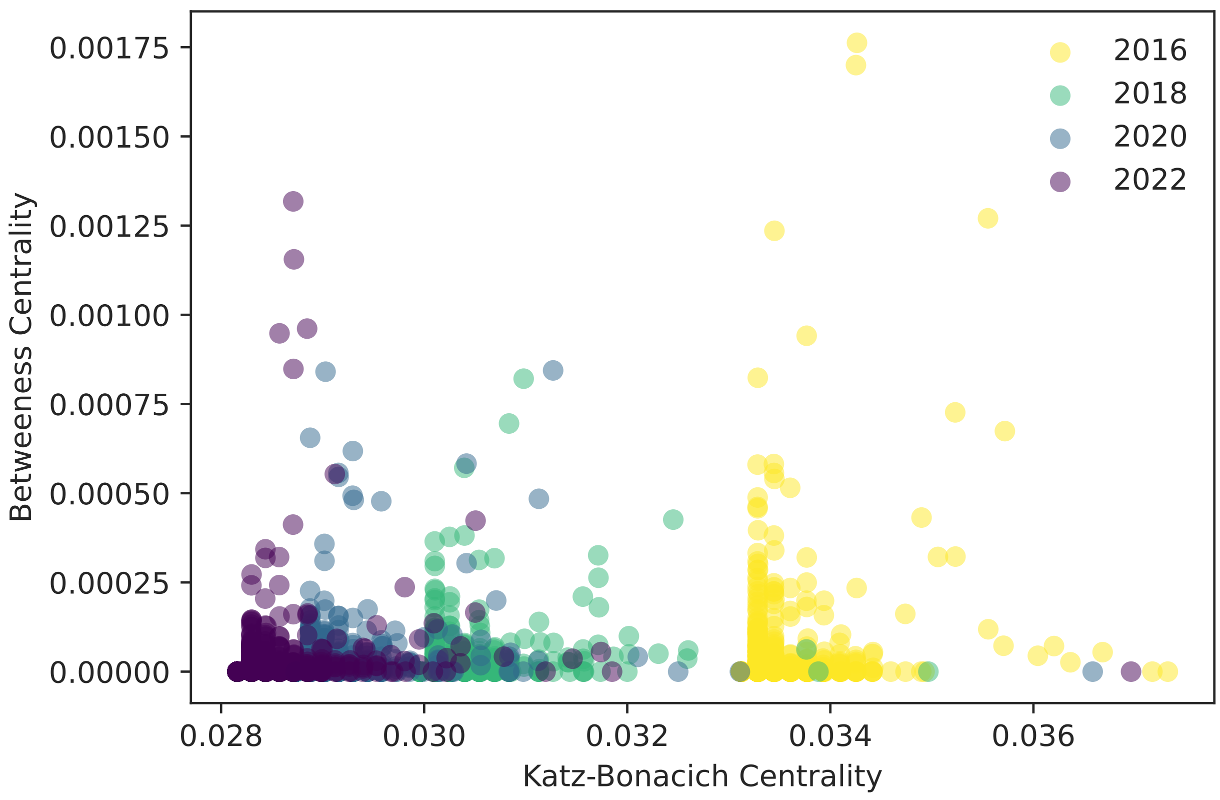


Figure 7: Empirical Dependency Network Sample (relationship between Katz-Bonacich and Betweenness centrality at the project level)

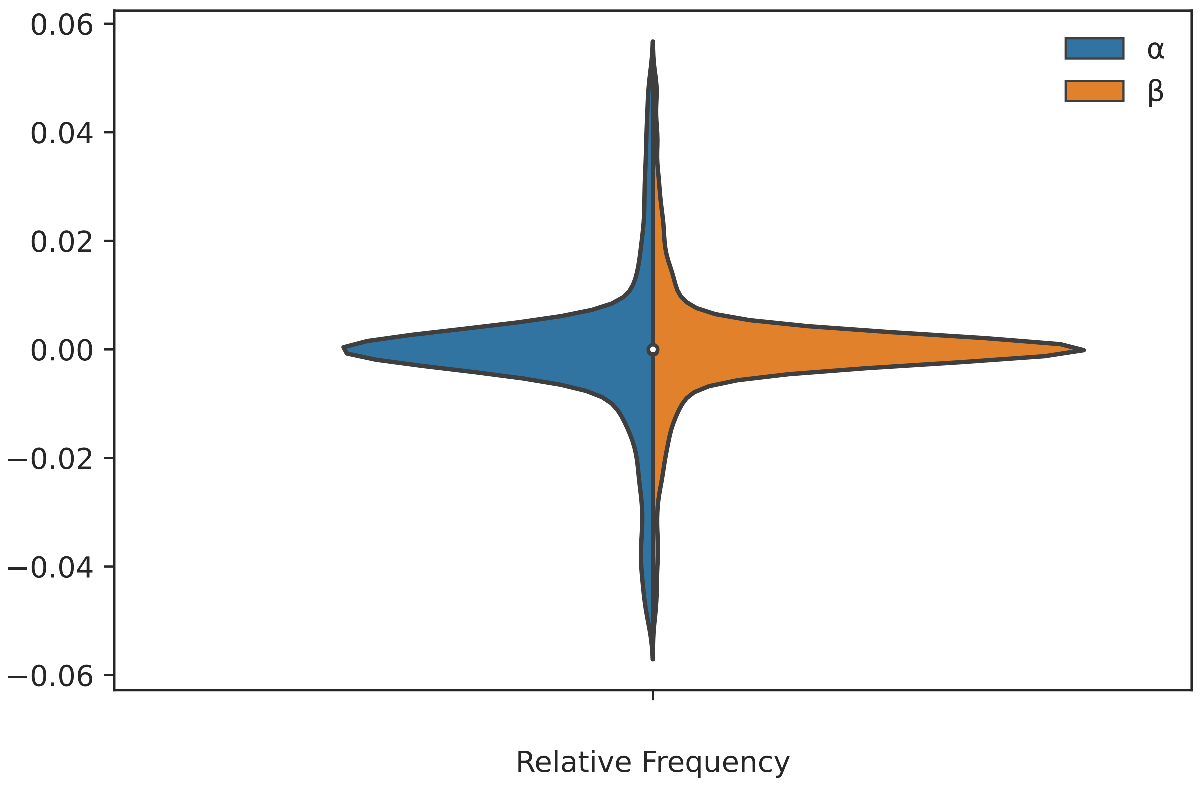


Figure 8: Project-level Heterogeneity in Reduced Form Estimates for Equation (5) and Equation (6).

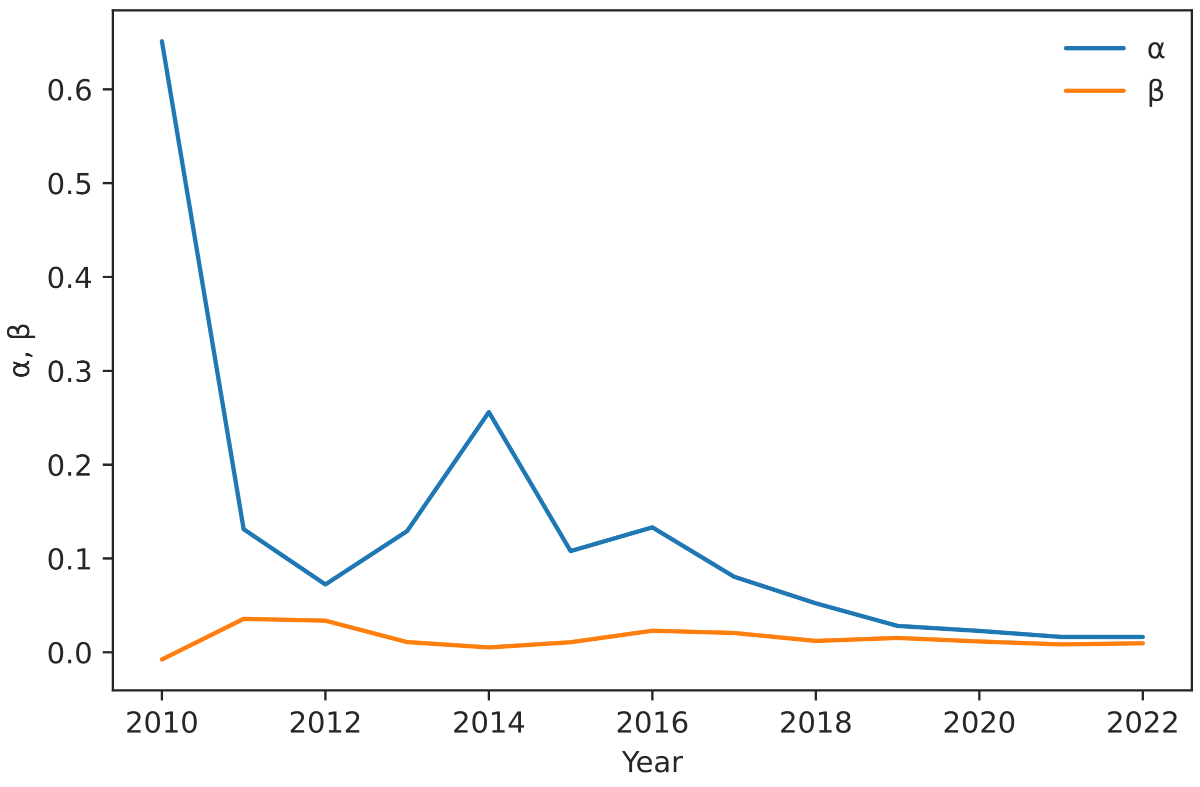


Figure 9: Temporal Heterogeneity in Reduced Form Estimates for Equation (5) and Equation (6).

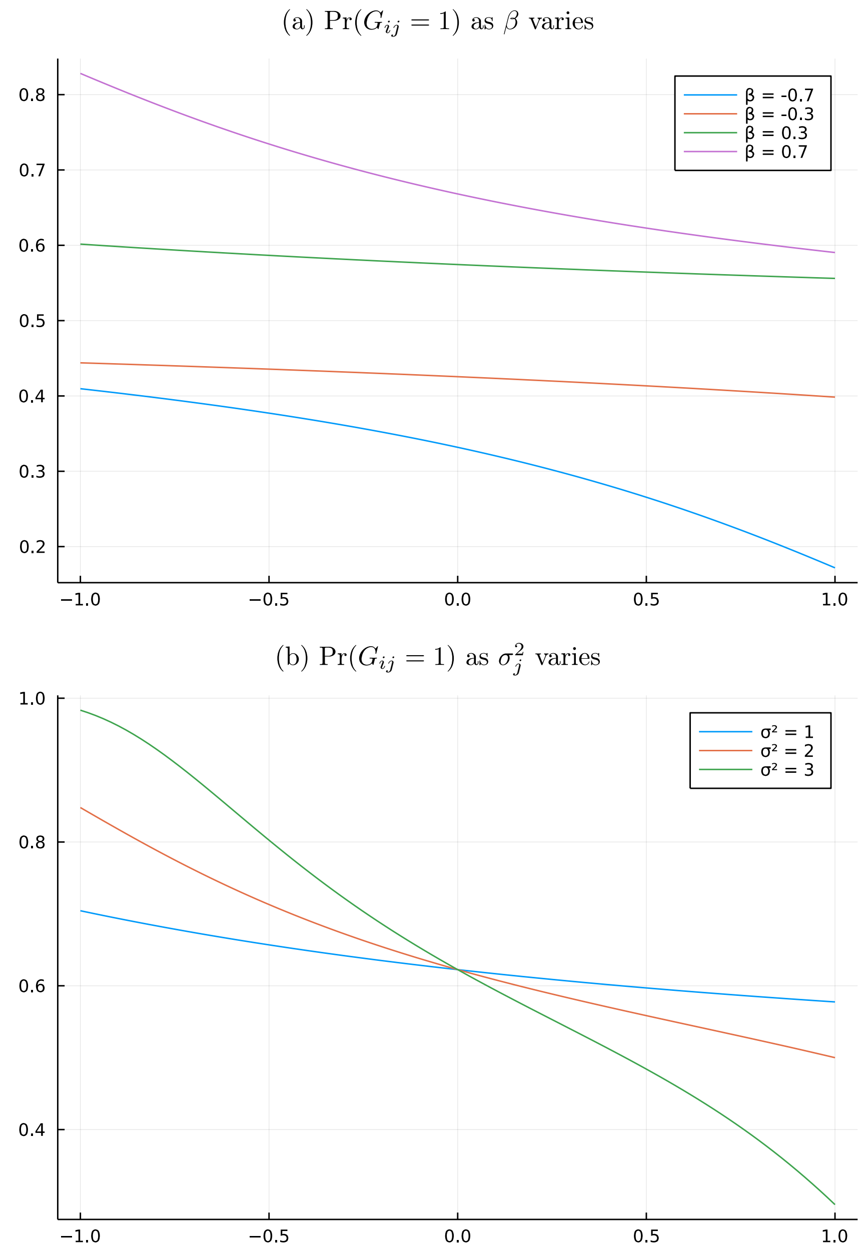


Figure 10: Comparative Statics – Probability of dependency formation, $\Pr(G_{ij} = 1)$, as risk aversion, r_i, varies.

B Tables

Table 1: Empirical Node.js Dependency Network Sample Descriptive Statistics

Notation	Measure	Obs.	Mean	SD	Min	Median	Max
x_{it}	**Project Contribution:** Cumulative total of commits to project i at time t.	206,598	2,703	6,573	1	195	81,641
$\sum_{j\neq i} G_{ijt}x_{jt}$	**Dependency Contribution:** Cumulative total of commits to project i's dependencies.	206,598	1,451	19,427	0	0	1,098,604
y_{it}	**Project Quality:** Sum of (log) complexity and cumulative contributors (log), scaled to $[0, 1]$.	206,598	0.363	0.199	0	0.334	1
$\sum_{j\neq i} G_{ijt}y_{jt}$	**Dependency Quality:** Sum of quality for project i's dependencies.	206,598	0.225	1.236	0	0	55.2
ω_{it}	**Time Allocation**: Cumulative project labor hours.	206,598	2,909	7,571	2	202	105,504
W_{it}	**Contributors**: Cumulative number of contributors.	206,598	233	983	1	20	17,195
W_{it}	**Core Contributors**: Smallest number of cumulative contributors with $\geq$80% of total contribution.	206,598	20	233	1	1	4,421
W_{it}	**Bus Factor**: Ratio of core contributors to total contributors.	206,598	0.209	0.260	0.003	0.121	1
W_{it}	**Files**: Number of files in codebase.	206,598	2,692	7,109	1	26	65,724
W_{it}	**SLOC**: Single lines of code in codebase.	206,598	129,556	326,412	2	2,717	2,974,829
W_{it}	**Modularity**: Ratio of file count to SLOC.	206,598	222	2,528	1.87	50.5	74,349
W_{it}	**Documentation**: Ratio of commented lines to SLOC.	206,598	0.110	0.172	0	0.046	3.18
W_{it}	**Number of languages**: Count of distinct programming languages.	206,598	7.87	3.34	1	7	29
W_{it}	**Project Age**: Days since first commit.	206,598	1,240	987	0	1,019	4,278
$d^{out}_{it} \equiv \sum_{j\neq i} G_{ijt}$	**Upstream Dependencies:** Number of external project dependencies project i declares.	161,211	2	3.76	0	1	74
$d^{in}_{it} \equiv \sum_{j\neq i} G_{jit}$	**Downstream Dependents:** Number of external projects that depend on project i.	161,211	5	7.94	0	2	66

Table 2: Reduced Form – Effect of Upstream Dependencies on Project Contribution

	Project Commits							
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Constant	2,653***	-68.19***						
	(14.51)	(9.744)						
Dependency Commits	0.034***	0.000***	0.003***	0.007***	0.002	0.000*	0.000	-0.000
	(0.002)	(0.000)	(0.001)	(0.001)	(0.001)	(0.000)	(0.000)	(0.000)
Project Quality		156.2***				959.5	126.9***	790.1***
		(22.26)				(518.5)	(19.57)	(322.0)
# Contributors		0.101***				0.546*	0.211***	1.542*
		(0.017)				(0.276)	(0.034)	(0.742)
Bus Factor		18.96***				68.50*	7.401*	6.286
		(3.851)				(32.56)	(2.928)	(21.80)
SLOC		0.000***				0.000	0.000***	0.000
		(0.000)				(0.000)	(0.000)	(0.000)
Documentation		-6.203				108.5	9.413***	136.5*
		(3.201)				(67.99)	(3.578)	(61.60)
Modularity		-0.001***				-0.001	-0.001***	-0.002
		(0.000)				(0.001)	(0.000)	(0.001)
# Languages		6.423***				7.306	5.836***	2.513
		(0.996)				(11.41)	(0.973)	(11.57)
Age		-0.012***				-0.002	-0.017***	-3.617
		(0.002)				(0.026)	(0.004)	(2.415)
Controls		✓				✓	✓	✓
Project FE			✓		✓	✓		✓
Time FE				✓	✓		✓	✓
R^2	0.010	0.999	0.964	0.674	0.988	0.999	0.999	0.999
Observations	206,598	202,905	206,575	201,336	201,308	202,869	196,930	196,894

Note: This table contains coefficient estimates from ordinary least squares (OLS) and fixed effect (FE) estimates for the reduced form relationship between project contribution and upstream dependency contribution in Equation (5). Specification variations are reported in columns. Standard errors, heteroskedasticity-robust for OLS and clustered by project for FE models, are reported in parentheses below each coefficient. Additional covariate controls not reported in the table include 3 lags each of project commits and dependency commits and the square of project age. All terms rounded to four significant figures. Statistical significance indicators: *** $\Rightarrow p \leq 0.001$, *** $\Rightarrow p \leq 0.01$, and * $\Rightarrow p \leq 0.1$ where p is the p-value for the coefficient estimate.

Table 3: Reduced Form Equation (5) – Effect of Upstream Dependencies on Project Quality

	Project Quality							
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Constant	0.302***	0.002***						
	(0.001)	(0.049)						
Dependency Quality	0.017***	0.000	0.004***	0.009***	0.002	0.001	0.000**	0.000
	(0.001)	(0.000)	(0.002)	(0.001)	(0.001)	(0.001)	(0.000)	(0.001)
Project Commits	0.000***	0.000	0.000*	0.000***	0.000***	0.000	0.000	0.000
	(0.000)	(0.000)	(0.000)	(0.000)	(0.000)	(0.063)	(0.047)	(0.052)
# Contributors		0.000				-0.000	0.000	0.000
		(0.000)				(0.000)	(0.000)	(0.002)
Bus Factor		-0.003***				-0.007***	-0.003***	-0.007***
		(0.000)				(0.001)	(0.000)	(0.001)
SLOC		-0.000				0.000	-0.000	-0.000
		(0.000)				(0.000)	(0.000)	(0.001)
Documentation		0.001***				0.002***	0.001***	0.003
		(0.000)				(0.000)	(0.000)	(0.003)
Modularity		0.000				0.000	0.000	0.000
		(0.000)				(0.000)	(0.000)	(0.000)
# Languages		0.000***				0.002***	0.000***	0.002
		(0.000)				(0.000)	(0.000)	(0.002)
Age		0.000				0.000	0.000	0.000
		(0.000)				(0.000)	(0.000)	(0.000)
Controls		✓				✓	✓	✓
Project FE			✓		✓	✓		✓
Time FE				✓	✓		✓	✓
R^2	0.514	0.999	0.966	0.764	0.987	0.999	0.999	0.999
Observations	206,598	202,905	206,575	201,336	201,308	202,869	196,930	196,894

Note: This table contains coefficient estimates from ordinary least squares (OLS) and fixed effect (FE) estimates for the reduced form relationship between project quality and upstream dependency quality in Equation (6). Specification variations are reported in columns. Standard errors, heteroskedasticity-robust for OLS and clustered by project for FE models, are reported in parentheses below each coefficient. Additional covariate controls not reported in the table include 3 lags each of project quality and dependency quality and the square of project age. All terms rounded to four significant figures. Statistical significance indicators: *** $\Rightarrow p \leq 0.001$, *** $\Rightarrow p \leq 0.01$, and * $\Rightarrow p \leq 0.1$ where p is the p-value for the coefficient estimate.

Table 4: Reduced Form Equations (7) and (8) – Effect of Project Features on Dependency Formation

	# of Upstream Dependencies				# of Downstream Dependents			
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Constant	0.720***				4.720***			
	(0.121)				(0.147)			
Project Commits	-0.000	-0.000	0.000	-0.000	0.001	0.000	0.001	0.001
	(0.027)	(2.493)	(0.126)	(0.000)	(0.038)	(0.001)	(0.210)	(0.001)
Upstream Commits	-0.000	-0.000	-0.000	-0.000	0.000	0.000**	0.000	0.000**
	(0.001)	(0.205)	(0.028)	(0.000)	(0.017)	(0.000)	(0.038)	(0.000)
Project Quality	3.073***	4.898***	5.180***	4.306*	3.935***	7.444	5.850***	5.350
	(0.029)	(0.255)	(0.002)	(1.986)	(0.098)	(4.481)	(0.044)	(3.444)
Upstream Quality	3.000***	1.227***	2.818***	1.176***	-1.683***	-0.313***	-1.585***	-0.327***
	(0.005)	(0.037)	(0.030)	(0.222)	(0.012)	(0.089)	(0.035)	(0.080)
# of Contributors	-0.000	0.000	-0.001	0.002	-0.009***	-0.002	-0.008	-0.004
	(0.001)	(0.019)	(0.000)	(0.001)	(0.003)	(0.003)	(0.005)	(0.004)
Bus Factor	-0.181***	0.078***	-0.227***	0.215	-1.098***	0.595	-0.787***	1.040*
	(0.010)	(0.003)	(0.000)	(0.243)	(0.018)	(0.667)	(0.001)	(0.526)
SLOC	-0.000	-0.000	-0.000	0.000	-0.000	-0.000	-0.000	-0.000*
	(0.004)	(0.037)	(0.005)	(0.000)	(0.016)	(0.000)	(0.001)	(0.000)
Documentation	-0.941***	-0.979***	-1.170***	-0.873	4.325***	2.607*	4.640***	2.513***
	(0.058)	(0.042)	(0.001)	(0.544)	(0.124)	(1.047)	(0.005)	(0.697)
Modularity	-0.000	-0.000	-0.000	-0.000	0.000	0.000	-0.000	0.000
	(0.005)	(0.010)	(0.002)	(0.000)	(0.012)	(0.000)	(0.009)	(0.000)
# of Languages	-0.055***	-0.036	-0.061	0.002	-0.575***	-0.272	-0.427**	-0.280
	(0.002)	(0.693)	(0.063)	(0.001)	(0.005)	(0.217)	(0.156)	(0.166)
Age	-0.000	-0.000	-0.000	-0.000	0.002	0.002***	0.004	-0.056
	(0.002)	(0.004)	(0.001)	(0.000)	(0.004)	(0.000)	(0.003)	(0.067)
Project FE		✓		✓		✓		✓
Time FE			✓	✓			✓	✓
R^2	0.535	0.875	0.622	0.899	0.143	0.946	0.384	0.965
Observations	161,211	161,183	161,211	161,175	161,211	161,183	161,211	161,275

Note: Columns (1) through (4) report coefficient estimates for the specification in Equation (7), a regression of the number of upstream dependencies for a given project (i.e., out-degree) on observable project characteristics. Columns (5) through (8) report coefficient estimates for the specification in Equation (8), a regression of the number of downstream dependents for a given project (i.e., in-degree) on observable project characteristics.

Table 5: Counterfactual Analysis

Counterfactual	Details	Welfare (Δ%)	Project Quality (Δ%)	Contribution (Δ%)	Costs (Δ%)
	Minimize Risk Aversion ($r'_i = \min(r_i) \, \forall i \in \mathcal{N}$)	0.22	0.40	0.00	0.00
$r \to r'$	Increase risk aversion ($r'_i = r_i + \sigma_r \, \forall i \in \mathcal{N}$)	-0.07	-0.09	0.00	0.00
	Maximize Risk Aversion ($r'_i = 1 \, \forall i \in \mathcal{N}$)	-0.04	2.11	0.00	0.00
	Reduce quality volatility ($\Sigma' = 0.5\Sigma$)	0.00	0.19	0.00	0.00
$\Sigma \to \Sigma'$	Increase quality volatility ($\Sigma' = 2\Sigma$)	0.00	1.51	0.00	0.00
	Increase quality volatility ($\Sigma' = 4\Sigma$)	0.05	1.14	0.00	0.00
	Remove top package	0.00	0.03	0.00	0.00
Key Package Analysis	Remove top 10 packages	-0.07	-5.73	-1.30	-0.87

C Mathematical Details

C.1 Alternative Representations for the Maintainer's Problem

We note that the maintainer's cost minimization problem presented in Equation 13 can have alternative representations that under certain conditions, can deliver an equivalent set of equilibria. Alternative representations can be helpful during estimation. First, the cost minimization problem of Equation 13 is presented here in Equation P1:

$$\begin{aligned} \min_{x_i \geq 0, y_i, \{G_{ij}\}_{j \neq i}} \quad & c_i(x, G) \\ \text{s.t.} \quad & u_i(x, y, G) \geq \underline{u}_i \end{aligned} \tag{P1}$$

Alternatively, a project maintainer could equivalently be modelled as maximizing expected project quality subject to a cost constraint.

$$\begin{aligned} \max_{x_i \geq 0, y_i, \{G_{ij}\}_{j \neq i}} \quad & u_i(x, y, G) \\ \text{s.t.} \quad & c_i(x, G) \leq \omega_i \end{aligned} \tag{P2}$$

Notice that by assumptions over u_i and c_i, both P1 and P2 are convex problems in x_i. Finally, the decision problem can be reframed to make it more amenable to established network formation estimation procedures (Hsieh et al., 2022). Consider P3, in which contribution cost is embedded into the maintainer's utility function:

$$\max_{x_i \geq 0, y_i, \{G_{ij}\}_{j \neq i}} \quad \mathbb{E}\left[v_i\left(\gamma y_i(x, G) - c_i(x, G)\right)\right] \tag{P3}$$

Note that the parameter γ can be interpreted as converting project quality into the value the maintainer places on the project in terms of time saved for some computing task. Hence, $\gamma y_i - c_i$ is the net value of project i in measured in hours.

The following proposition establishes an equivalence between the solutions of P1, P2, and P3.

Proposition 1 (Equivalence between P1, P2, and P3)**.** *Assume $x_i^\star > 0$.*

1. *The solutions to P1 and P3 coincide if the minimum project quality threshold binds: $u_i(x, y, G) = \underline{u}_i$.*
2. *The solutions to P2 and P3 coincide if the contribution cost constraint binds: $c_i(x, G) = \omega_i$.*

Proof. The Lagrangian for P1

$$\mathcal{L}_i(x_i, G; \lambda_i) = c_i(x, G) + \lambda_i \left(\underline{u}_i - u_i(x, G)\right)$$

First Order Necessary Conditions (FONCS) for P1:

$$\begin{aligned}\frac{\partial c_i}{\partial x_i} - \lambda_i \frac{\partial u_i}{\partial x_i} &\leq 0 \\ x_i \left(\frac{\partial c_i}{\partial x_i} - \lambda_i \frac{\partial u_i}{\partial x_i} \right) &= 0 \\ x_i &\geq 0 \\ u_i(x, G) &\geq \underline{u}_i \\ \lambda_i(\underline{u}_i - u_i(x, G)) &= 0 \\ \lambda_i &\geq 0\end{aligned} \tag{21}$$

The Lagrangian for P2

$$\mathcal{L}_i(x_i, G; \mu_i) = -u_i(x_i, G) + \mu_i \left(c_i(x_i, G) - \omega_i \right)$$

First Order Necessary Conditions (FONCS) for P2:

$$\begin{aligned}-\frac{\partial u_i}{\partial x_i} + \mu_i \frac{\partial c_i}{\partial x_i} &\leq 0 \\ x_i \left(-\frac{\partial u_i}{\partial x_i} + \mu_i \frac{\partial c_i}{\partial x_i} \right) &= 0 \\ x_i &\geq 0 \\ c_i(x_i, G) &\leq \omega_i \\ \mu_i(c_i(x_i, G) - \omega_i) &= 0 \\ \mu_i &\geq 0\end{aligned} \tag{22}$$

Finally, the FONCs for P3:

$$\begin{aligned}-\gamma \frac{\partial y_i}{\partial x_i} + \frac{\partial c_i}{\partial x_i} &\leq 0 \\ x_i \left(-\gamma \frac{\partial y_i}{\partial x_i} + \frac{\partial c_i}{\partial x_i} \right) &= 0 \\ x_i &\geq 0\end{aligned} \tag{23}$$

If $x_i > 0$, then $\frac{\partial c_i}{\partial x_i} = \lambda_i \frac{\partial u_i}{\partial x_i}$, $\mu_i \frac{\partial c_i}{\partial x_i} = \frac{\partial u_i}{\partial x_i}$, and $\frac{\partial c_i}{\partial x_i} = \gamma \frac{\partial y_i}{\partial x_i}$ for P1, P2, and P3 respectively.

Case 1: Suppose the minimum utility threshold constraint from P1 binds. Then $\lambda_i > 0$ and since $\frac{\partial u_i}{\partial x_i} > 0$ under the exponential utility Assumption 1, then $\frac{\partial c_i}{\partial x_i}$. Then further assuming that $\gamma \frac{\partial y_i}{\partial x_i} > 0$, then the equilibrium solutions to P1 and P3 coincide if and only if for each $x_i^\star$

$$\frac{\partial c_i}{\partial x_i} = \lambda_i \frac{\partial u_i}{\partial x_i} = \gamma \frac{\partial y_i}{\partial x_i} > 0$$

Case 2: Suppose the contribution cost constraint in P2. Then $\mu_i > 0$. Similar to the argument in Case 1, the equilibria of P1 and P3 coincide if and only if for each $i \in \mathcal{N}$

$$\frac{\partial c_i}{\partial x_i} = \frac{1}{\mu_i} \frac{\partial u_i}{\partial x_i} = \gamma \frac{\partial y_i}{\partial x_i} > 0$$

■

Proposition 1 implies that in the edge case where both the cost and minimum quality constraints bind, all representations of the maintainer's problem result in the same solution and how the Lagrange multipliers of P1 and P2 correspond to the parameter γ in P3. We use the result of this proposition along with its assumptions to simplify estimation.

Remark (Equilibrium Contribution in Equation 14)**.** *The exposition in Proposition 1, although tedious and perhaps a bit excessive, makes clear the relationship between observed equilibria (e.g., $x_i > 0$ or $c_i(x, G) = \omega_i$) and the equilibrium conditions implied by each of the FONCs of P1, P2, and P3. The parameter a_i, representing maintainer i's intrinsic marginal cost of contribution, one would obtain from estimating Equation 14 does not exactly, equal a_i from Equation 10. Instead, for the purposes of estimation in Section 6.3, we assume that the cost constraint binds. Then by Proposition 1, we replace a_i in Equation 14 with $\tilde{a}_i = a_i + \gamma b_i$ for the purposes of estimation. This should have no effect on the exposition in Section 6.*

C.2 Expected Project Quality

Under Assumption (2), maintainer utility becomes

$$\begin{aligned} u_i(x^\star, y^\star, G) &= \mathbb{E}\left[v_i\left(y_i^\star\right)\right] \\ &= \mathbb{E}\left[-\exp\left(-r_i\left(\sum_j B_{ij}(b_j x_j^\star + \xi_j)\right)\right)\right] \\ &= -\exp\left(-r_i \sum_j B_{ij} b_j x_j^\star\right) \mathbb{E}\left[\exp\left(-r_i \sum_j B_{ij}\xi_j\right)\right] \\ &= -\exp\left(-r_i \sum_j B_{ij} b_j x_j^\star\right) \exp\left(\frac{r_i^2}{2} \sum_j B_{ij}^2 \sigma_j^2\right) \\ &= -\exp\left(-r_i \sum_j B_{ij}\left(b_j x_j^\star - \frac{r_i}{2} B_{ij}\sigma_j^2\right)\right) \end{aligned} \tag{24}$$

The third line of (24) follows from Assumption 2 on maintainer information sets. By the normality of ξ established by Assumption (2), the fourth line of Equation (24) uses the moment generating function for a linear combination of the normally distributed random vector ξ.

C.3 Optimal Dependency Formation

Following Fosgerau and Bierlaire (2009), let $\epsilon_{ij}^{+} = -\lambda \ln(\varepsilon_{ij}^{+})$ and $\epsilon_{ij}^{-} = -\lambda \ln(\varepsilon_{ij}^{-})$ where $\lambda > 0$. Under Assumption 5, we can show

$$\begin{aligned}
\Pr(G_{ij} = 1) &= \Pr\left(u_{ij}^{+}\varepsilon_{ij}^{+} \geq u_{ij}^{-}\varepsilon_{ij}^{-}\right) \\
&= \Pr\left(-\ln(-u_{ij}^{+}) - \ln(\varepsilon_{ij}^{+}) \geq -\ln(-u_{ij}^{-}) - \ln(\varepsilon_{ij}^{-})\right) \\
&= \Pr\left(\overline{u}_{ij}^{+} + \epsilon_{ij}^{+} \geq \overline{u}_{ij}^{-} + \epsilon_{ij}^{-}\right) \\
&= \Pr\left(u_{ij} \geq \epsilon_{ij}\right)
\end{aligned} \tag{25}$$

where $(\overline{u}_{ij}^{+}, \overline{u}_{ij}^{-}) \equiv (-\lambda \ln(-u_{ij}^{+}), -\lambda \ln(-u_{ij}^{-}))$, $u_{ij} \equiv \overline{u}_{ij}^{+} - \overline{u}_{ij}^{-}$, and $\epsilon_{ij} \equiv \epsilon_{ij}^{-} - \epsilon_{ij}^{+}$. The advantage of this approach is that now the equilibrium link formation decision can be represented as a random utility with additive disturbances and is linear-quadratic in the parameter of interest, r_i. If we substitute the expression for expected project quality under both $G + ij$ and $G - ij$ from Equation (24) to form u_{ij}^{+} and u_{ij}^{-}, the likelihood that maintainer i imports project j in Equation (25) becomes

$$\begin{aligned}
\Pr(G_{ij} = 1) &= \Pr\left(-\ln\left(\mathbb{E}\left[\exp\left(-r_i y_{ij}^{+}\right)\right]\right) + \ln\left(\mathbb{E}\left[\exp\left(-r_i y_{ij}^{-}\right)\right]\right) \geq \epsilon_{ij}/\lambda\right) \\
&= \Pr\left(r_i\left(\sum_j \Delta B_{ij}\left(b_j\left(\sum_k \Delta A_{jk} a_k\right) - \frac{r_i}{2}\Delta B_{ij}\sigma_j^2\right)\right) \geq \epsilon_{ij}/\lambda\right) \\
&= F_\epsilon\left(r_i \underbrace{\left(\sum_j \Delta B_{ij} b_j \left(\sum_k \Delta A_{jk} a_k\right)\right)}_{Z_{0ij}} - \frac{1}{2} r_i^2 \underbrace{\left(\sum_j \Delta B_{ij}^2 \sigma_j^2\right)}_{Z_{1ij}}; \theta_\epsilon\right)
\end{aligned} \tag{26}$$

where $u_{ij} \equiv \overline{u}_{ij}^{+} - \overline{u}_{ij}^{-}$, $\epsilon_{ij} \equiv \epsilon_{ij}^{-} - \epsilon_{ij}^{+}$, $y_{ij}^{+} \equiv y_i(x, y_{-i}, G + ij)$ and $y_{ij}^{-} \equiv y_i(x, y_{-i}, G - ij)$. Finally, we define $\Delta B_{ij} \equiv B_{ij}^{+} - B_{ij}^{-}$ as the difference between elements of the Leontief inverse matrices for project quality under $G + ij$ and $G - ij$: $B + ij = [B_{ij}^{+}]_{i,j \in \mathcal{N}} = (I - \beta(G + ij))^{-1}$ and $B - ij = [B_{ij}^{-}]_{i,j \in \mathcal{N}} = (I - \beta(G - ij))^{-1}$. Equivalently, we define $\Delta A_{jk} = A_{jk}^{+} - A_{jk}^{-}$ for the Leontief inverse matrices for project contribution: $A + ij = [A_{jk}^{+}]_{j,k \in \mathcal{N}} = (I - \alpha(G + ij))^{-1}$ and $A - ij = [A_{jk}^{-}]_{j,k \in \mathcal{N}} = (I - \alpha(G - ij))^{-1}$.

For notational convenience, in the last equality of Equation (26), we rearrange u_{ij} into a quadratic function of r_i and label the coefficients Z_{0ij} and Z_{1ij}. These coefficients are functions of the network G and parameters $(\alpha, a, \beta, \Sigma)$. In Section 6.3 and Appendix D, we show that since $(\alpha, a, \beta, \Sigma)$ can be estimated using moment conditions (9) and (14) via GMM, the result in Equation (26) will form the basis for a likelihood function for the remaining unknown parameters, r and θ_ϵ. Hence, under the assumptions outlined in the body of the paper, r and θ_ϵ can be estimated via maximum likelihood. We will assume that ϵ_{ij} is a logistic random variable, independent and identically distributed across potential links. This can arise if ϵ_{ij}^{+} and ϵ_{ij}^{-} are independent Gumbel random variables. Therefore, $F_\epsilon(\cdot)$ is the logistic function which has the convenient property that $F'_\epsilon(\cdot) = F_\epsilon(\cdot)(1 - F_\epsilon(\cdot))$. Furthermore, θ_ϵ is a vector of two parameters for a logistic distribution (i.e., location and scale). We make no specific assumption on the value of the free parameter λ other than $\lambda > 0$.

Therefore, the value of λ will simply influence the estimated scale parameter for the distribution of ϵ_{ij}.

D Estimation Details

Data is $\mathcal{D} = (x_t, y_t, G_t, W_t)_{t \in \mathcal{T}}$. Parameters are $\theta = (a, \alpha, b, \beta, \Sigma, r, \gamma, \theta_\epsilon)$.

1. Estimate b, β given $\mathcal{D}$ using the project quality specification Equation (9) in separate OLS regressions for each project $i \in \mathcal{N}$.

$$y_{it} = b_i x_{it} + \beta \sum_{j \neq i} G_{ijt} y_{jt} + \tilde{\xi}_{it}$$

where $\tilde{\xi}_{it} = \delta' W_{ijt} + \xi_{it}$ is an are other influences partitioned in observable $\delta' W_{ijt}$ and unobservable x_{it} components. Furthermore, we can use the residuals of each OLS regression to estimate Σ. Therefore the structural estimation of b, β is equivalent to the reduced form estimation of project quality influences in Equation 6. Furthermore, our consideration of Heterogeneity beyond the framework of the model matches the structural estimation approach outlined by Hsieh et al. (2022). In our setup this may allow us to control for technical aspects of projects that at least partially determine quality or fixed costs of project contribution that are absent from our structural discussion in Section 6.

2. Estimate a, α, γ given $\mathcal{D}$ and estimates for b using Proposition 1 and a modified version of the project contribution specification in Equation 14 using OLS regressions for each project

$$\begin{aligned} x_{it} &= \tilde{a}_i + \alpha \sum_{j \neq i} G_{ijt} x_{jt} + \tilde{\nu}_{it} \\ &= a_i + \gamma b_i + \alpha \sum_{j \neq i} G_{ijt} x_{jt} + \tilde{\nu}_{it} \end{aligned}$$

where, as before, $\tilde{\nu}_{it} = \delta' W_{ijt} + \nu_{it}$ and ν_{it} is independent and identically distributed and mean zero in expectation.

3. (Optional) If we assume that contribution costs $c_{it}(x, G)$ exactly equal, conditional on some noise or measurement error d_{it}, estimates of time allocation ω_{it}, we can estimate fixed costs of contribution for each project $i \in \mathcal{N}$ using the following specifications to back out a residual d_{it}:

$$d_{it} = \omega_{it} - \frac{1}{2} x_{it}^2 + a_i x_{it} + \alpha \sum_{j \neq i} G_{ijt} x_{jt}$$

where ω, x, G are observed in $\mathcal{D}$ and a, α were recovered in previous steps. Taking the average of for each project gives an estimate of the fixed costs of contribution: $\overline{d}_{it} = \frac{1}{|T_i|} \sum d_{it}$ where T_i is defined as the number of time periods in which project i appears in the empirical sample. These estimates will help refine the estimation of welfare effects under counterfactual analysis.

While we suggest that the parameters in Steps 1–3 above can be estimated with simple OLS, it might be more prudent to organize analogs of Equations (15), (14), and (10) described above into a set of moment conditions ans subsequently estimate $(a, \alpha, b, \beta, \Sigma, \gamma, d)$ using the generalized method of moments (GMM) with constraints: $\alpha, \beta \in (-1, 1)$, $\sigma_i > 0$

4. Estimate r, θ_ϵ by means of MLE, maximizing a likelihood function based on equilibrium link

formation described in Equation (18).

$$\begin{aligned} L(\theta \mid \mathcal{D}) = \prod_{t \in \mathcal{T}} \Pr(\mathcal{D}_t | \mathcal{D}_{t-1}, \theta) &= \prod_{t} \prod_{j \neq i} \Pr(G_{ij} = 1 \mid \mathcal{D}_{t-1}, \theta) \\ &= \prod_{t} \prod_{j \neq i} F_\epsilon(u_{ijt})^{G_{ijt}} \left(1 - F_\epsilon(u_{ijt})\right)^{1-G_{ijt}} \end{aligned} \tag{27}$$

As mentioned previously, this likelihood function forms a Markov chain of likelihoods for the observed sequence of linking decisions. MLE estimates for r and θ_ϵ minimize $-\ln L(\theta \mid \mathcal{D})$.

D.1 Additional simplifications to reduce computational burden

Given the size of our empirical sample, the estimation procedure as specified remains a time intensive task on available hardware. In conjunction with the dimensionality reduction we discuss in Section 6.3.1, we take a few computation shortcuts to calculate the coefficients Z_{0ijt} and Z_{ijt} for each sample moment $t \in \mathcal{T}$, the project $i \in \mathcal{N}_t$ associated with that particular moment t and each potential dependency $j \neq i \in \mathcal{N}_t$. First, we approximate the true matrix inverse using $A = I + \sum_{k=1}^{K} \alpha^k G^k$ where $K = 5$ provides a decent approximation. Second, following the suggestion of Hsieh et al. (2022), we use the Sherman-Morrison formula to efficiently calculate a new proposal Leontief inverse A or B to calculate ΔA_{ij} and ΔB_{ij}. Third, instead of considering proposal links for all $j \neq i$ in the current network, we consider a set of randomly selected potential dependencies from $\mathcal{N}_t$ that is (1) equal in size to the set of current dependencies for project i and (2) contains potential dependencies not in the current set of dependencies for project i.