Vulnerability Webs: Systemic Risk in Software Networks*

Cornelius Fritz[1], Co-Pierre Georg[2], Angelo Mele[3], and Michael Schweinberger[4]

[1]Trinity College Dublin – School of Computer Science and Statistics

[2]Frankfurt School of Finance and Management

[3]Johns Hopkins University – Carey Business School

[4]The Pennsylvania State University – Department of Statistics

Software development relies on code reuse to minimize costs, creating vulnerability risks through dependencies with substantial economic impact, as seen in the Crowdstrike and HeartBleed incidents. We analyze 52,897 dependencies across 16,102 Python repositories using a strategic network formation model incorporating observable and unobservable heterogeneity. Through variational approximation of conditional distributions, we demonstrate that dependency creation generates negative externalities. Vulnerability propagation, modeled as a contagion process, shows that popular protection heuristics are ineffective. AI-assisted coding, on the other hand, offers an effective alternative by enabling dependency replacement with in-house code.

Keywords: economics of software development, dependency graphs, strategic network formation, exponential random graphs

JEL Classification: D85, L17, C31, L86, O31

*Contact: fritzc@tcd.ie, co.georg@fs.de, angelo.mele@jhu.edu, and mus47@psu.edu. We wish to thank Bryan Graham, Stephane Bonhomme, Louise Laage, Matt Jackson, Eric Auerbach for helpful suggestions. We also thank participants at 2024 Cowles Foundation Conference in Econometrics, ESIF 2024 Conference on ML+AI, Empirical Methods in Game Theory Workshop, MEG 2024, EC24, MGOW 2024, Network Science in Economics 2023, 2023 BSE Summer Forum Networks Workshop, SEA 2023, Econometrics of Peer Effects and Networks, and seminar participants at Georgetown University, University of Washington, Virginia Tech, Boise State University, and the University of Pennsylvania for helpful comments and suggestions. The authors acknowledge support from the German Research Foundation award DFG FR 4768/1-1 (CF), Ripple's University Blockchain Research Initiative (CPG), the U.S. Department of Defense award ARO W911NF-21-1-0335 (MS, CF), and the U.S. National Science Foundation awards NSF DMS-1513644 and NSF DMS-1812119 (MS). Mele acknowledges support from an IDIES Seed Grant from the Institute for Data Intensive Engineering and Science at Johns Hopkins University and partial support from NSF grant SES 1951005.

1 Introduction

On July 19, 2024 a faulty update to the software Crowdstrike—a security software adopted by many companies—created worldwide panic and disruptions, paralyzing domestic and international air traffic for hours, and disrupting operations at banks, hospitals, and hotel chains.[1] All told, the update affected 8.5 million Windows machines worldwide, each of which had to be manually rebooted to patch up the system and restart operations. The faulty update of the Crowdstrike software was not the first instance highlighting systemic risk in software systems, and it will not be the last: e.g., in 2014 the Heartbleed bug—a vulnerability of the OpenSSL authentication library used by many websites—compromised the data of millions of users of websites such as Eventbrite, OkCupid, in addition to disrupting operations at the FBI and hospitals. Such disruptions can generate staggering economic costs: Heartbleed, for example, resulted in at least $500 million in damages. According to the risk management company Parametrix, the disruption caused by Crowdstrike cost Fortune 500 companies $5.4 billion.[2]

The risk of these disruptions is deeply embedded in the structure of software systems. Modern software development is a collaborative effort that produces sophisticated software by making extensive use of already existing code (Schueller et al., 2022). This results in a complex network of dependencies among software packages, best described as dependency graphs.[3] While this re-use of code produces significant efficiency gains for software developers (Lewis et al., 1991; Barros-Justo et al., 2018), it also increases the risk that a vulnerability in one software package renders a large number of other packages equally vulnerable. In line with this view, the number of vulnerabilities listed in the Common Vulnerabilities and Exposure Database has increased almost twelve-fold between 2001 and 2019.[4] An estimate by Krasner (2018) put the cost of losses from software failures at over $1 trillion, up from $59 billion in 2002 estimated by the National Institute of Standards and Technology (2002).[5]

[1] See https://www.nytimes.com/2024/07/19/business/dealbook/tech-outage-crowdstrike-microsoft.html

[2] See Parametrix (2024).

[3] Dependency graphs have been studied for many different programming languages using information from package managers. For an empirical comparison of the most common ones, see Decan et al. (2019).

[4] The Common Vulnerabilities and Exposure database is a public reference for known information-security vulnerabilities which feeds into the U.S. National Vulnerability Database. See `https://cve.mitre.org/`. The reported number of incidents has increased from 3,591 for the three years from 1999 to 2001 to 43,444 for the three-year period from 2017 to 2019.

[5] Industry group Cybersecurity Ventures estimates that the damages incurred by all forms of cyber crime, including the cost of recovery and remediation, totalled USD 6 Trillion in 2021, and could reach USD 10.5 Trillion annually by 2025 (Source).

Because vulnerabilities can potentially spread from one software package to all packages depending on it, coders can create an externality when deciding to re-use code from existing packages rather than implementing the required functionality themselves. Whether this is the case and how large the externality is is ultimately an empirical question.

We model coders' decisions to create a dependency to another software package as an equilibrium game of network formation with observed and unobserved heterogeneity.[6] Developers weigh the costs and benefits of forming links and revise these links at random times when there is a need for an update. Our equilibrium characterization provides the likelihood of observing a particular architecture of the software dependency network in the long run. Using data on open source software projects contained in Python's software package manager Pypi, we obtain 52,897 dependencies among 16,102 repositories. We find evidence that a maintainer's decision to allow a dependency exerts a *negative* externality on other maintainers. In principle, the presence of this externality discourages the formation of dependencies and partially contributes to the sparsity of the network. Our result implies that it is particularly important to ensure that the code in these repositories is free from vulnerabilities that potentially affect the code in a large number of other repositories.

We can use our model to study the effect of interventions that reduce code vulnerabilities that are likely to affect a large subset of the software ecosystem. We assume that vulnerabilities, coding errors, or bugs spread according to an epidemiological contagion model. Our simulations show that strategies to decrease the risk of contagion based on interventions targeting nodes according to measures of centrality or expected spread risk are not very effective in mitigating the average risk of contagion. On the other hand, we observe that the introduction of AI-assisted coding may decrease the incentives to form dependencies through a decrease in the costs of producing code. This process of decreasing the density of the network will mechanically decrease the average risk of contagion. We show that even moderate productivity gains from AI translate into a significant flattening of the risk profile for the software network.

Open-source software is an ideal laboratory to study software dependencies. Since the actual source code of software in a repository can be inspected and modified by others, code reuse is not only possible, it is encouraged. Modern software is not developed in isolation, but within an ecosystem of interdependent packages organized in code repositories. A package manager keeps track of code dependencies and ensures that these are automatically resolved so that

[6] Formally, we aggregate software packages that belong to the same code repository. Each package belongs to a single repository, but each repository can contain multiple packages. Repositories are typically managed by a small team of maintainers who decide which code is added to the repository.

when a user installs one package from a repository, all of the package's dependencies are also installed and automatically satisfied. There are many programming languages that provide software using package managers. We focus on repositories of software packages written in the popular Python programming language and managed by the Pypi package manager.[7] Python is a modern programming language that is very popular in the software development community. We use data from libraries.io, which, for each software repository, includes a full list of all dependencies.

We model the development of software as a process where maintainers reuse existing code, creating a network of dependencies among software packages—organized in repositories of several closely related software packages—in the process. This is typical for today's prevalent object-oriented software development paradigm. Our model describes a system of N individual software repositories, characterized by observable and unobservable types. Each repository is managed by a single maintainer who decides which code to add to the repository and, consequently, which dependencies to form. Maintainers obtain a net benefit of linking to another package which depends on how active this dependency is maintained, how mature, popular, and large it is. We also allow a maintainers' utility to be affected by a local, i.e. type-specific, externality: Since dependent packages have dependencies themselves, they are susceptible to vulnerabilities imported from other packages. So we assume that maintainers care about the direct dependencies of the packages they link to. This specification excludes externalities that are more than two links away, as in the network formation models of DePaula et al. (2018), Mele (2017) and Mele and Zhu (2023). The network of dependencies forms over time and in each period a randomly selected package needs an update, so the maintainers decide whether to form a link or update the software in-house. Before updating the link, the package receives a random match quality shock. We characterize the equilibrium distribution over networks as a mixture of exponential random graphs (Schweinberger and Handcock, 2015; Mele, 2022), which can be decomposed into within- and between-types contribution to the likelihood.

Model estimation is challenging because the likelihood depends on a normalizing constant that is infeasible to compute in large networks. Moreover, the model's unobserved heterogeneity has to be integrated out in the likelihood, making direct computations infeasible even for moderately sized networks. To alleviate these problems, we resort to a novel approximate two-step approach. In the first step, we estimate the discrete unobservable types of the nodes by approximating their conditional distribution via a fast variational mean-field approximation first proposed for stochastic blockmodels (Vu et al., 2013). In the second step, we estimate the structural

[7] PyPI stands for Python Package Index and is Python's official third-party software repository. See here for the official documentation.

payoff parameters using a fast Maximum Pseudo-Likelihood Estimator (MPLE), conditioning on the estimated types (Babkin et al., 2020; Dahbura et al., 2021; Martínez Dahbura et al., 2023). We note that these methods were first developed for undirected graphs, and their application to directed networks necessitates non-trivial extensions of the computational machinery, providing an additional contribution of our work.[8]

Our main result is that maintainers exert a *negative* externality on other maintainers when creating a dependency on another repository: Other maintainers are then less likely to create a dependency on that repository. As a result, the network is relatively sparse. The interdependence among libraries implies that there exists a risk that a vulnerability in a single package has large adverse consequences for the entire ecosystem. An example of how a vulnerability in one package affected a significant part of critical web infrastructure is Heartbleed, the infamous bug in the widely used SSL/TLS cryptography library, resulting from incorrect validation of the input variable (Durumeric et al., 2014). When Heartbleed was disclosed in 2014, it made up to 55% of secure webservers vulnerable to data theft, including the Canadian Revenue Agency and Community Health Systems, a large US hospital chain. Our model provides a further argument for ensuring the security of highly interdependent software packages. Not only can a vulnerability in such a package affect a relatively large fraction of the entire ecosystem, because of the externality we identify, it is also likely that this fraction increases as time goes on.

Another important question in the study of network formation processes is whether linked packages are similar or dissimilar with respect to their covariates. We find evidence of similarity among the same type of software packages in terms of their Size, Popularity and Maturity as well as between different-type software packages in terms of their Popularity. The effect of similarity decreases with the (log of) the number of packages of the same unobservable type. We further find evidence of *dis*-similarity between different-type software packages in terms of their maturity and size. In other words, mature, popular, and large software packages of one type are likely to depend on similar packages of the same type, but on less mature and smaller software packages of another type. This is intuitive because mature, popular, and large packages are likely to have a large user base with high expectations of the software, and coders try to satisfy this demand by providing sophisticated functionality with the help of mature, popular, and large software packages of the same type. These packages are often built using a lot of low-level functionality from large but fairly generic libraries, which is why large popular and mature software packages of one type depend on larger, but less popular and less mature software packages of another type. One example of this is the inclusion of a payment gateway in an

[8] Our methods are implemented in the scalable, open-source, and platform-independent `R` package `bigergm`, which is available on CRAN servers (R Core Team, 2024). Implementation details can be found in the appendix.

e-commerce application. The e-commerce application itself can be sophisticated and complex, like ebay is in the provision of their auction mechanism. For such an application, it is particularly valuable to provide users with the ability to use a variety of different payment methods, including credit card, paypal, or buy now pay later solutions like Klarna. Our result also aligns with the recent trend in software development away from large monolithic applications and towards interconnected microservices (Traore et al., 2022).

Lastly, we adapt a simple epidemiological contagion model to examine the spread of vulnerabilities in the Pypi dependency graph. We measure a repository's *k-step systemicness* as the number of downstream packages that are potentially rendered vulnerable by a vulnerability or bug in the upstream repository. We use this model to study the efficacy of targeted interventions to mitigate vulnerability spread, drawing parallels with vaccination strategies in public health. By focusing on securing the most critical nodes–determined according to in-degree, expected fatality, or a combination of betweenness centrality and expected fatality–we assess the potential to significantly reduce the risk of vulnerability contagion.

Our findings suggest that protecting even ten percent of the most critical repositories may reduce the average systemic risk but cannot completely flatten out the risk profile of the ecosystem. On the other hand, we argue that the introduction of AI-assisted coding may have a greater effect on reducing systemic risk. Recent work has shown that AI tools such as GitHub Copilot can increase the productivity of software developers (Peng et al., 2023; Edelman et al., 2023; Noy and Zhang, 2023). In our model, this corresponds to a decrease in the cost of producing software in-house, without relying on creating dependencies or creating fewer dependencies. This can be modeled as an increase in the relative cost of a dependency. By simulating different magnitudes of the productivity improvement from AI-assisted coding, we show that the average systemic risk profile is flattened out by an increase of cost of merely five percent. The mechanism through which this risk reduction is achieved is the brute-force decrease in density for the network implied by increases in costs of dependencies. In general, our results help us to better understand the various driving forces and motifs in software development and how they shape the network of software dependencies.

Our paper relates to several strands of literature in both economics and computer science. First, our paper contributes to a growing literature on open source software, which has been an interest of economic research.[9] In an early contribution, Lerner and Tirole (2002) argue that coders spend time developing open source software—for which they are typically not compensated—

[9]For an overview of the broad literature in the emerging field of digital economics, see Goldfarb and Tucker (2019).

as a signal of their skills for future employers. Likewise, one reason why companies contribute to open source software is to be able to sell complementary services. Open source projects can be large and complex, as Zheng et al. (2008) point out. They study dependencies in the Gentoo Linux distribution, and show that the resulting network is sparse, has a large clustering coefficient, and a fat tail. They argue that existing models of network growth do not capture the Gentoo dependency graph well and propose a preferential attachment model as alternative.[10]

The package manager model is nowadays adopted by most programming languages which makes it feasible to use dependency graphs to study a wide variety of settings. Kikas et al. (2017), for example, study the structure and evolution of the dependency graph of `JavaScript`, `Ruby`, and `Rust`. The authors emphasize that dependency graphs of all three programming languages become increasingly vulnerable to the removal of a single software package. An active literature studies the network structure of dependency graphs (Decan et al., 2019) to assess the vulnerability of a software ecosystem (see, for example, Zimmermann et al., 2019).

These papers show the breadth of literature studying open source ecosystems and dependency graphs. However, the literature considers the network either as stochastic or even as static and given. In contrast, we model the formation of dependencies as coders' *strategic* choice in the presence of various and competing mechanisms that either increase or reduce utility.[11] Perhaps the closest work is Boysel (2023), that uses data from a sample of repositories to estimate a structural model of network formation and maintainers effort. In his model, maintainers sequentially update their dependencies, then play a network game where their efforts best responses are a function of other maintainers efforts. While related, our approaches are different, since Boysel (2023) relies on sample panel data and the analysis focuses on the dynamics of myopic best-response. In our paper we focus on the long-run equilibrium and the cross-section of the dependencies among *all* repositories at a particular point in time.

The theoretical literature on strategic network formation has pointed out the role of externalities and heterogeneity in shaping the equilibrium networks (Bramoullé et al., 2016; Galeotti et al., 2006; Jackson and Wolinsky, 1996). Estimating strategic models of network formation is a challenging econometric task, because the presence of externalities implies strong correlations among links and multiple equilibria (Mele, 2017; Snijders, 2002; DePaula et al., 2018;

[10] In earlier work, LaBelle and Wallingford (2004) study the dependency graph of the Debian Linux distribution and show that it shares features with small-world and scale-free networks. However, LaBelle and Wallingford (2004) do not strictly check how closely the dependency graph conforms with either network growth model.

[11] Blume et al. (2013) study how possibly contagious links affect network formation in a general setting. While we do not study the consequences of the externality we identify for contagion, this is a most worthwhile avenue for future research.

Chandrasekhar, 2016; DePaula, 2017; Boucher and Mourifie, 2017; Graham, 2017, 2020). In this paper, we model network formation as a sequential process and focus on the long-run stationary equilibrium of the model (Mele, 2017, 2022). Because the sequential network formation works as an equilibrium selection mechanism, we are able to alleviate the problems arising from multiple equilibria.

Adding unobserved heterogeneity further complicates identification, estimation and inference (Schweinberger and Handcock, 2015; Graham, 2017; Mele, 2022). Other works have considered conditionally independent links without externalities (Graham, 2017, 2020; DePaula, 2017; Chandrasekhar, 2016), providing a framework for estimation and identification in random and fixed effects approaches. On the other hand, because link externalities are an important feature in this context, we move away from conditionally independent links, and model nodes' unobserved heterogeneity as discrete types, whose realization is independent of observable characteristics and network, in a random effect approach. We can thus adapt methods of community discovery for random graphs to estimate the types, extending the work of Babkin et al. (2020) and in Dahbura et al. (2021); Martínez Dahbura et al. (2023) to accommodate for directed networks. Our two-steps method scales well to large networks, thus improving the computational challenges arising in estimation of these complex models (Boucher and Mourifie, 2017; Vu et al., 2013; Bonhomme et al., 2019).

Our model is able to estimate the magnitude of externalities as well as homophily (Currarini et al., 2010; Jackson, 2008; Chandrasekhar, 2016), the tendency of individuals to form links to similar individuals. Furthermore, our model is able to detect heterophily (or competition) among maintainers. More specifically, we allow homophily to vary by unobservables, while in most models homophily is estimated only for observable characteristics (Currarini et al., 2010; Schweinberger and Handcock, 2015; DePaula et al., 2018; Chandrasekhar, 2016; Graham, 2020).

Lastly, our contagion analysis is broadly related to the body of work that treats security as a strategic game on a given graph. The literature following Goyal and Vigier (2014) and Dziubiński and Goyal (2013), for example, shows how the optimal point of attack of an adversary and the defense strategy of a designer are based on the centrality of the network. Differently from this literature, though, we also study how changing the structure of the network, induced by a changing relative cost of forming dependencies, affects the spread of contagion and find this to be highly effective.

2 A network description of code

2.1 The open source software paradigm

The goal of computer programs is to implement algorithms on a computer. An algorithm is a terminating sequence of operations which takes an input and computes an output using memory to record interim results.[12] We use the term broadly to include algorithms that rely heavily on user inputs and are highly interactive (e.g. websites, spreadsheet and text processing software, servers). Algorithms are implemented on a computer using code. Formally:

Definition 1. *Code is a sequence of operations and arguments that implement an algorithm. A computer program is code that can be executed by a computer system.*

In order to execute a program, a computer provides resources—processing power and memory—and resorts to a compiler or interpreter, which in themselves are computer programs.[13]

Software developers, which we refer to as coders, use programming languages to implement algorithms. Today, the dominant modern software development paradigm is *object-oriented programming* (OOP).[14] Under this paradigm, code is developed primarily in *classes*, which contain data in the form of variables and data structures as well as code in the form of procedures.[15] Classes can communicate with one another via procedures using calls and returns.

Definition 2. *A class is a code template defining variables, procedures, and data structures. An object is an instance of a class that exists in the memory of a computer system.*

Classes can interact in two ways. First, in the traditional *monolithic* software architecture, widely used for enterprise software like the one developed by SAP, for operating systems like

[12] Memory to record interim instructions is sometimes called a “scratch pad”, in line with early definitions of algorithms which pre-date computers. See, for example, Chapter 1 of Knuth (1997).

[13] The term "compiler" was coined by Hopper (1952) for her arithmetic language version 0 (A-0) system developed for the UNIVAC I computer. The A-0 system translated a program into machine code which are instructions that the computer can execute natively. Early compilers were usually written in machine code or assembly. Interpreters do not translate program code into machine code, but rather parse it and execute the instructions in the program code directly. Early interpreters were developed roughly at the same time as early compilers, but the first widespread interpreter was developed in 1958 by Steve Russell for the programming language LISP (see McCarthy (1996).

[14] For a principal discussion of object-oriented programming and some differences to procedural programming, see Abelson et al. (1996). Kay (1993) provides an excellent historical account of the development of early object-oriented programming.

[15] Also called functions, or methods in the OOP paradigm.

Microsoft's Windows, and even in earlier versions of e-commerce platforms like Amazon, individual components cannot be executed independently. In contrast, many modern software projects use a *microservices* software architecture, which is a collection of cohesive, independent processes, interacting via messages. Both architectures result in software where individual pieces of code depend on other pieces, either within a single application or across various microservices. These dependencies form a network which we formalize in the next section.

2.2 Dependency Graphs

Most open source software is organized by package managers like PyPi for the Python programming language that standardize and automate software distribution. To make the installation of complex software projects easier for users, package managers keep track of a package's dependencies and allow users to install these alongside the package they are installing. The existence of package managers and the level of automation they provide is necessary because modern software is frequently updated and different versions of the same package are not always compatible. Packages are logically grouped into repositories, controlled and managed by a maintainer.[16]

Our unit of analysis is the network of dependencies among repositories. Specifically, we describe a software system consisting of N repositories collected in $\mathcal{N} = \{1, \ldots, N\}$, each managed by a different maintainer who decides whether to implement code themselves or to re-use existing code from other repositories. This (re-)use of existing code gives rise to linkages between repositories. The resulting network $\mathcal{G} := (\mathcal{N}, \mathcal{E})$ with $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is called the *dependency graph* of the software system. We can represent the repository dependency graph $\mathcal{G}$ by a $N \times N$ adjacency matrix $g = \{g_{ij}\}$, where $g_{ij} = 1$ if $(i, j) \subseteq \mathcal{E}$ is an existing link from repo i to repo j, indicating that i depends on j. Since we consolidate code on the level of repositories, we exclude self-loops by defining $g_{ii} := 0$.[17]

There are two reasons for constructing the dependency graph on the repository rather than the package level. First, information about the popularity of a piece of software is relevant for

[16] Each package belongs to one repository, but a repository can contain multiple packages. Maintainers can also be organizations. Some repositories are also controlled by a group of maintainers, but for our purposes, this is immaterial.

[17] This is not to say that there are no dependencies between code inside a repository–there will be many. But these are managed within a single repository by a maintainer. We are interested in the structure of the dependency network of repositories managed by different maintainers.

Table 1: Network statistics for the dependency graph of the PyPi ecosystem.

#Nodes	#Edges	#Components	#Nodes (LCC)	#Edges (LCC)
17,095	53,498	422	16,102	52,897

#Nodes and #Edges is the number of nodes and edges for the total network and for the largest weakly connected component, respectively. #Components is the number of connected components. LCC indicates the largest (weakly) connected components.

maintainers linking decisions, but only created on the repository level.[18] And second, repositories provide a logical grouping of closely related software packages. Alternatively, we could construct the dependency graph on the level of individual software packages. However, the breadth of code and functionality between packages can be much smaller than the breadth of code and functionality within a package.[19] Conveniently, the website libraries.io provides information collected from various open source software package managers, including PyPi.[20] The data provided includes information on projects–essentially the same as a PyPi package–as well as repositories and for each repository a list of all dependencies, defined on the project level. Since each project belongs to exactly one repository, we construct dependencies among repositories from the data provided.

On the repository level, this data includes information about the size of the repository in kB (Size), its popularity, measured as the number of stars it has on the website hosting the repository (Popularity), and as an alternative measure of popularity the number of contributors, i.e. the number of individual coders who have added code to a repository, raised or answered an issue, or reviewed code submitted by others.[21]

Table 1 provides a high-level overview of the repository-based dependency graph constructed in this way. While there are 91 weakly connected components, the largest weakly connected component covers almost all (over 99%) nodes and edges. This means that we still cover all relevant dynamics even by only focusing on the largest weakly connected component of the dependency graph.

[18] For example, users can "star" a repository on GitHub if they find it particularly useful.

[19] Yet another alternative is to study *call graphs* arising from procedures within the same software package (see, for example, Grove et al., 1997). But these are state-dependent, i.e., dependencies arise during runtime and depending on how the package is executed and interacted with.

[20] The website libraries.io obtains this data by scraping publicly available repositories hosted on GitHub, GitLab, and Bitbucket.

[21] We restrict ourselves to repositories that have a size above the 5% percentile of the size distribution, and that have more than 2 stars, but no more stars than the 95% percentile of the distribution of stars.

Table 2: In- and out-degree distribution.

	Mean	Std.Dev.	Min.	p5	Median	P95	Max.
In-Degree	3.29	50.49	0	0	0	6	4,210
Out-Degree	3.29	5.0	0	0	2	12	95

Distribution of in- and out-degree for the largest weakly connected component of the repo-based dependency graph of the PyPi ecosystem with $N = 16,102$ nodes.

Furthermore, Table 2 shows that there is a lot of heterogeneity in how interconnected repositories are. The standard deviation of the out-degree, i.e. how many dependencies a given repository uses, is almost twice as large as the mean. Therefore, we observe substantial heterogeneity among repositories, with the median repository using 2 dependencies and the maximum being 95 dependencies. This heterogeneity is even more pronounced for the in-degree, i.e. how often a given repository is a dependency for another repository. The standard deviation is more than ten times as large as the mean; the median is 0, while the maximum is 14,585.

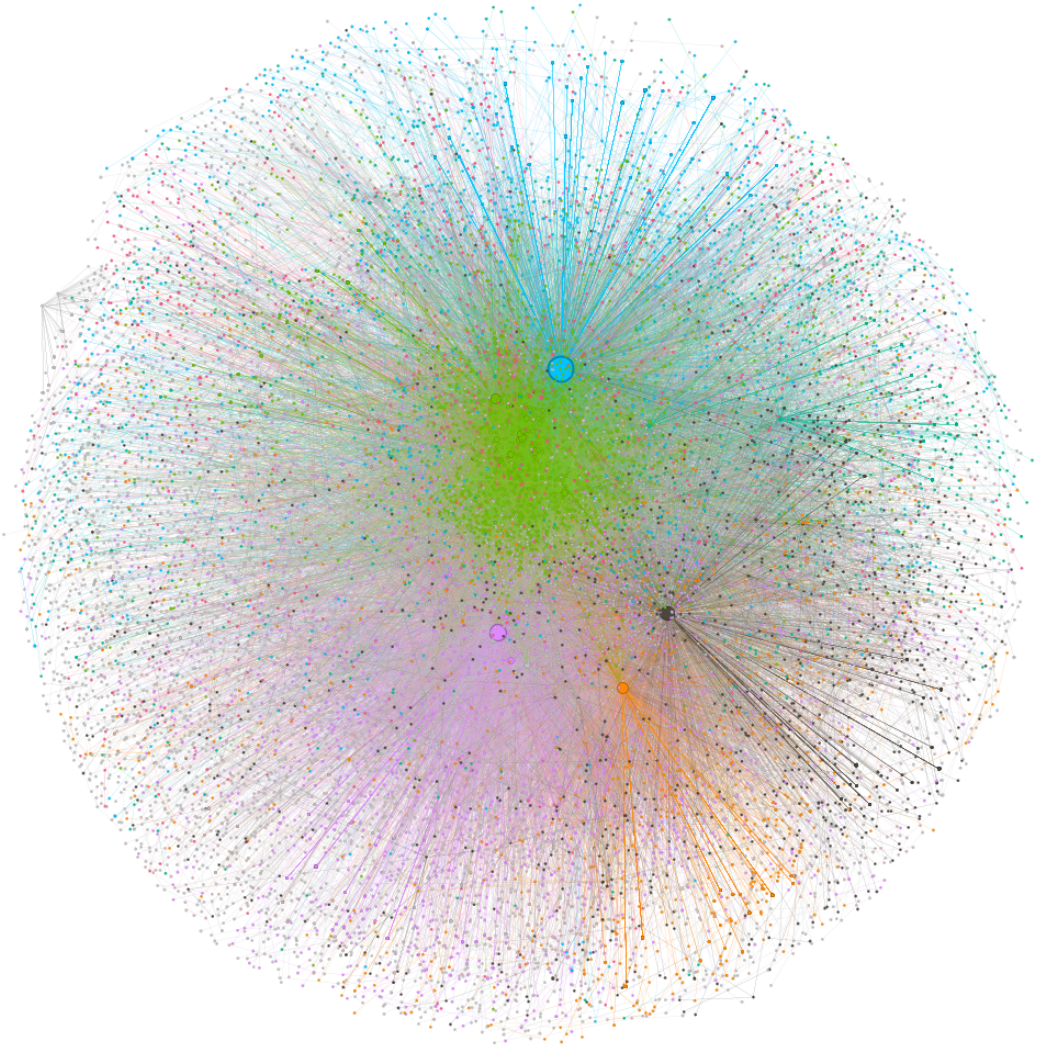
In Table 3 we show the 10 most depended-on repositories (i.e. with the highest in-degree). The repository with highest in-degree is `six`, which facilitates compatibility between python version 2 and 3 (where some major updates have led to serious incompatibility issues). The second-highest in-degree repository is `pytest-cov`, which is used to check a code's test coverage, which is a crucial step in modern software development. Likewise, the third-most commonly linked software repository is `mock`, which is also used for software testing.

The largest weakly connected component of the repo-based dependency graph is shown in Figure 1, where we apply the community detection algorithm of Blondel et al. (2008) to color-code repositories in the same community. The three largest communities contain about 10 – 11% of all repositories each.

2.3 Observables: Covariates

In addition to the software dependency data, we obtain additional data from the `libraries.io` website for each package and repository (e.g. `https://libraries.io/pypi/pandas`), which we show for the ten most depended-on repositories in Table 3. We use three types of covariates, summarized in a vector of observable attributes x_i. First, we use a repository's Size of all code in the repository, measured in Kilobytes. Second, Popularity is measured as the number of stars–an expression of how many people like a repository or find it useful–a repository received on

Figure 1: Dependency graph of the PyPi ecosystem.



Nodes represent repositories in the libraries.io dataset while edges represent dependencies of packages in different repositories. Colors indicate modularity classes according to Blondel et al. (2008).

the website hosting it, e.g., GitHub. And, third, the time in days between the first release of a repository and the reference date of 13 January 2020 is denoted Maturity.

To facilitate the estimation of our model, we create a categorical variable for each of our covariates using quartiles of the distribution. The reason for discretizing variables is computational, as our algorithm is based on stochastic block models with discrete types (Bickel et al., 2013; Vu et al., 2013). If the covariates are discrete, the estimation engine of stochastic block models can

Table 3: List of the ten most depended-on repositories for the PyPi ecosystem.

		Covariates		
Repo name	In-Degree	Popularity	Size	Maturity
benjaminp/six	4,210	579	1,749	1,106
pytest-dev/pytest-cov	2,584	643	777	2,097
testing-cabal/mock	2,002	421	1,306	1,728
pypa/wheel	1,572	156	1,302	902
kjd/idna	1,336	106	507	2,422
pytest-dev/pluggy	952	364	471	1,721
pypa/packaging	782	141	540	2,069
pallets/markupsafe	775	282	175	3,492
PyCQA/mccabe	767	242	82	2,518
coveralls-clients/coveralls-python	761	406	328	2,532

Popularity is measured as the number of stars a repository received on the website hosting it. Size is measured in Kilobytes of all code in the repository. Maturity is the time in days between the first release of a repository and 13 January 2020.

be adapted to estimate the unobserved heterogeneity (Vu et al., 2013; Babkin et al., 2020; Dahbura et al., 2021). Otherwise, when the covariates are continuous, the computational costs are prohibitive for such large networks.

3 Model

We model software development as a dynamic process, assuming that maintainers update their repositories to add new functionality or improve existing functionality. Maintainers can update their repositories by adding functionality to their own code or creating dependencies to other repositories that provide the desired functionality.

3.1 Setup

Each repository i and its maintainer is characterized by a vector of P observable attributes $\boldsymbol{x}_i \in \mathbb{R}^P$, which are stored in a matrix $\boldsymbol{x} \in \mathbb{R}^{N \times P}$. We assume that each of the N repositories belongs to one of $K \geq 2$ types, unknown to researchers but known to other developers. The type of a repository may be its purpose, which may be known to developers but may not be known to

Table 4: Distribution of covariates.

	Mean	Std.dev	Min	p5	Median	p95	Max
Size [kB]	7,729.8	52,557.84	6	23	273	27,627	3,484,077
Popularity	70.49	133.35	3	3	16	364	853
Maturity	1384.6	838.14	4	248	1250	2960	4270

The table reports some descriptives statistics of the covariates for the $N = 16,102$ repositories in the largest connected component of our sample for the PyPi ecosystem. Size is measured in Kilobytes of all code in the repository. Popularity is measured as the number of stars a repository received on the website hosting it. Maturity is the time in days between the first release of a repository and 13 January 2020.

researchers, or may correspond to unobservable qualities of the code and its developer team. To indicate the type k of repository i, we define $z_{ik} \coloneqq 1$ and $z_{il} \coloneqq 0$ for all $l \neq k$ if repository i is of type k, and write $\boldsymbol{z}_i \coloneqq (z_{i1}, \ldots, z_{iK})$. We assume that the unobservables are independent and identically distributed according to a multinomial distribution:

$$\boldsymbol{Z}_i \overset{\text{iid}}{\sim} \text{Multinomial}\big(1; \boldsymbol{\eta}\big), \;\; i = 1, \ldots, N, \tag{1}$$

where $\boldsymbol{\eta} \coloneqq (\eta_1, \ldots, \eta_K)$ is a vector of K probabilities summing to 1. We represent the *software dependency network* $\boldsymbol{g}$ by the adjacency matrix, with elements $g_{ij} = 1$ if repository i depends on repository j and $g_{ij} = 0$ otherwise. In general, software dependency networks are directed: If repository i depends on repository j, then repository j need not depend on repository i.

The utility that developers of repository i receive from the software dependency network $\boldsymbol{g}$, with observables $\boldsymbol{x}$, unobservables $\boldsymbol{z}$, and parameters $\boldsymbol{\theta} \coloneqq (\boldsymbol{\alpha}, \boldsymbol{\beta}, \gamma)$, is of the form

$$U_i(\boldsymbol{g}; \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta}) \coloneqq \sum_{j=1}^{N} g_{ij} u_{ij}(\boldsymbol{\alpha}, \boldsymbol{\beta}) + \sum_{j=1}^{N} \sum_{r \neq i,j}^{N} g_{ij} g_{jr} v_{ijr}(\gamma) + \sum_{j=1}^{N} \sum_{r \neq i,j}^{N} g_{ij} g_{ir} w_{ijr}(\gamma). \tag{2}$$

The function

$$u_{ij}(\boldsymbol{\alpha}, \boldsymbol{\beta}) \coloneqq u(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{z}_i, \boldsymbol{z}_j, \boldsymbol{\alpha}, \boldsymbol{\beta}) \tag{3}$$

is the direct utility of linking repository i to repository j. It is a function of observables $(\boldsymbol{x}_i, \boldsymbol{x}_j)$, unobservables $(\boldsymbol{z}_i, \boldsymbol{z}_j)$, and parameters $(\boldsymbol{\alpha}, \boldsymbol{\beta})$. This represents the benefit of a dependency of repository i on repository j, net of costs: e.g., the developers will have to audit the code of repository j and determine its quality, and have to maintain the dependency on repository j. The cost may include modification and adaptation of the code, because developers must be

able to use the functions and methods available in the repository j. Additional costs may arise from relying on other repositories, because other repositories may contain undetected vulnerabilities. Finally, relying on existing code is convenient but may decrease the incentives of developers to maintain and improve coding skills, which can undermine their future productivity and can hence be viewed as a cost.

Motivated by the application to software dependency networks, we parameterize u as follows:

$$u(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{z}_i, \boldsymbol{z}_j, \boldsymbol{\alpha}, \boldsymbol{\beta}) := \alpha(\boldsymbol{z}_i, \boldsymbol{z}_j) + \sum_{p=1}^{P} \beta_p(\boldsymbol{z}_i, \boldsymbol{z}_j)\, h_{ijp}(\boldsymbol{x}_i, \boldsymbol{x}_j), \tag{4}$$

where

$$\alpha(\boldsymbol{z}_i, \boldsymbol{z}_j) = \begin{cases} \alpha_{w1} + \alpha^{\ell}_{w1} \log(N_k), & \text{if } \boldsymbol{z}_i = k \text{ and } \boldsymbol{z}_j = k \\ \alpha_b, & \text{otherwise} \end{cases} \tag{5}$$

and

$$\beta_p(\boldsymbol{z}_i, \boldsymbol{z}_j) = \begin{cases} \beta_{wp} + \beta^{\ell}_{wp} \log(N_k), & \text{if } \boldsymbol{z}_i = k \text{ and } \boldsymbol{z}_j = k \\ \beta_{bp}, & \text{otherwise} \end{cases} \tag{6}$$

define within- and between-type utilities.

The terms in the within- and between-type utilities can be interpreted as follows: The intercept $\alpha_{w1} \in \mathbb{R}$ represents the fixed cost of forming a dependency; the term $\alpha^{\ell}_{w1} \log(N_k)$ with weight $\alpha^{\ell}_{w1} \in \mathbb{R}$ is the part of the cost that depends on the size N_k of type k;[22] and $h_{ijp}(\boldsymbol{x}_i, \boldsymbol{x}_j) := \mathbf{1}\{x_{ip} = x_{jp}\}$ is an indicator function, which is equal to 1 if the pth covariate of i and j matches and is 0 otherwise. Size-dependent terms of the form $\log(N)$ were suggested by Krivitsky et al. (2011), motivated by invariance considerations (encouraging the expected number of links to be invariant to N), and adapted to observed types by Krivitsky et al. (2023) and to unobserved types by Babkin et al. (2020). The micro-behavioral foundations of size-dependent terms of the form $\log(N)$ were studied by Butts (2019). The vectors $\boldsymbol{\alpha} := (\alpha_{w1}, \alpha^{\ell}_{w1}, \alpha_b) \in \mathbb{R}^3$ and $\boldsymbol{\beta} := (\beta_{w1}, \ldots, \beta_{wP}, \beta_{b1}, \ldots, \beta_{bP}, \beta^{\ell}_{w1}, \ldots, \beta^{\ell}_{wP}) \in \mathbb{R}^{3P}$ are parameters to be estimated.

The second and third term of utility function (2) are externalities generated by linking to repository j, because repository j may be linked to other repositories r as well. This means that the developers of repository i need to check the quality and features of the code in those reposito-

[22] In principle, the specification can be extended to incorporate aggregated terms at the type-level. In this sense, $\log(N_k)$ is one among other examples of terms that can be used to make the model more flexible. For instance, one may include the average of observed characteristics of type k as one of the covariates. We do not pursue this specification here, given the very sparse nature of the software dependency network.

ries r as well, to ensure that those repositories are free of vulnerabilities, and compatible with repository i. On the other hand, any update of repository r may compromise compatibility with repositories i and j, increasing the cost of maintaining repository i. In short, w_{ijr} measures the net benefits of this externality when developers of repository i form a dependency to repository j. In addition, if the developers of repository i create a dependency on repository j, it may compromise compatibility with other repositories r that depend on repository i. This is accounted for by the third term v_{ijr} in utility function (2). We assume that the second and third term in the utility are of the form

$$v_{ijr}(\gamma) = w_{ijr}(\gamma) \coloneqq \begin{cases} \gamma & \text{if } i, j, r \text{ are all of type } k \\ 0 & \text{otherwise,} \end{cases} \tag{7}$$

where $\gamma \in \mathbb{R}$.

The choice of the utility functions U_i guarantees that all externalities are local, in the sense that all externalities are limited to repositories of the same type. Local externalities make sense in large networks, because developers are unlikely to know the functionalities of thousands of other repositories and, more likely than not, will create dependencies to other repositories of the same type, which developers are most familiar with. In addition, local externalities have probabilistic, statistical, and computational advantages. First, local externalities ensure desirable properties of models for large networks (Schweinberger and Handcock, 2015). Second, local externalities facilitate theoretical guarantees for statistical procedures, which we discuss in Section 4. Third, local externalities enable local computing on subnetworks, which facilitates parallel computing and hence large-scale computing (Babkin et al., 2020).

3.2 Equilibrium

Software development is a dynamic process. We assume that time is discrete and that at each time point $t \in \{1, 2, \ldots\}$ a single repository is updated. In other words, maintainers do not update software repositories continuously, but certain events – unobservable to researchers – trigger the need for an update. For example, if repository i depends on repository j, then an update of repository j may require an update of repository i, or a request by users of repository i may trigger an update of repository i. The update can be produced in-house or by forming a dependency to some other repository j containing the same functionality. Since researchers may not be able to observe the timing of such updates, we model updates as a discrete-time Markov process (Norris, 1997).

The first ingredient of the discrete-time Markov process is the update process that determines which repository i requires an update and which dependency j is considered to implement the update. We assume that the probability that repository i requires an update and the developers of repository i consider creating, maintaining, or removing a dependence to repository j can be expressed as a function $\rho_{i,j}$. A simple example is an update process that selects an ordered pair of repositories i and j at random. In other words, a repository i is selected with probability $1/n$ and requires an update. Conditional on the event that repository i requires an update, the developers of repository i consider updating the possible dependency of repository i on repository j with probability $1/(n-1)$. As a result, the probability that repository i requires an update and the developers of repository i consider updating a (possible) dependency of repository i on repository j is

$$\rho_{i,j}\left(\boldsymbol{g}^{t-1}, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{z}_i, \boldsymbol{z}_j\right) \;=\; \frac{1}{n\,(n-1)},$$

where $\boldsymbol{g}^{t-1}$ is the network at time $t-1$. A second example is an update process that allows the conditional probability that an update of the (possible) dependency of repository i on repository j is proposed to depend on the distance $d(\boldsymbol{x}_i, \boldsymbol{x}_j)$ between the repositories i and j. The distance $d(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is any distance function satisfying reflexity, symmetry, and the triangle inequality: e.g., $d(\boldsymbol{x}_i, \boldsymbol{x}_j)$ may be the Euclidean distance between observable characteristics $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ of repositories i and j, that is, $d(\boldsymbol{x}_i, \boldsymbol{x}_j) \coloneqq \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2$. In other words, a repository i is selected with probability $1/n$ and, conditional on the event that repository i requires an update, the developers of repository i consider updating the possible dependency of repository i on repository j with probability $e^{-d(\boldsymbol{x}_i, \boldsymbol{x}_j)} / \sum_{k=1}^{n} e^{-d(\boldsymbol{x}_i, \boldsymbol{x}_k)}$, which implies that

$$\rho_{i,j}\left(\boldsymbol{g}^{t-1}, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{z}_i, \boldsymbol{z}_j\right) \;=\; \frac{1}{n}\,\frac{e^{-d(\boldsymbol{x}_i, \boldsymbol{x}_j)}}{\sum_{k=1}^{n} e^{-d(\boldsymbol{x}_i, \boldsymbol{x}_k)}}.$$

To characterize the stationary distribution of the discrete-time Markov process, the update process needs to satisfy two mild conditions (Mele, 2017). First, the update probabilities $\rho_{i,j}$ can depend on the entire software dependency network excluding (i, j):

$$\rho_{i,j}\Big(\boldsymbol{g}^{t-1}, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{z}_i, \boldsymbol{z}_j\Big) \;=\; \rho_{i,j}\Big(\boldsymbol{g}^{t-1}_{-(i,j)}, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{z}_i, \boldsymbol{z}_j\Big).$$

Second, the update probabilities $\rho_{i,j}$ are strictly positive for all ordered pairs of repositories (i, j):

$$\rho_{i,j}\Big(\boldsymbol{g}^{t-1}_{-(i,j)}, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{z}_i, \boldsymbol{z}_j\Big) \;>\; 0.$$

This implies that each possible dependency can be proposed with a positive probability, no matter how small. These assumptions help ensure that in the long-run equilibrium these probabilities do not affect the distribution of networks (Mele, 2017).

While the update process is stochastic, the decision to create, maintain, or remove a possible dependency of repository i on repository j is a strategic decision. Conditional on the opportunity to update the dependency of repository i on repository j, the developers of repository i make a decision about whether to create, maintain, or remove a dependency on repository j based on utility function U_i. Before the developers of repository i make a decision about the (possible) dependency on repository j, the developers receive random shocks $\varepsilon_{ij0} \in \mathbb{R}$ and $\varepsilon_{ij1} \in \mathbb{R}$ to the perceived utility of a dependency of repository i on repository j ($g_{ij}^t = 1$) and the perceived utility of writing code in-house ($g_{ij}^t = 0$). After receiving random shocks $\varepsilon_{ij0} \in \mathbb{R}$ and $\varepsilon_{ij1} \in \mathbb{R}$, the developers of repository i create a dependency on repository j if the perceived utility of the dependency of repository i on repository j ($g_{ij}^t = 1$) exceeds the utility of writing code in-house ($g_{ij}^t = 0$):

$$U_i\left(g_{ij}^t = 1, \boldsymbol{g}_{-(i,j)}^{t-1}; \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta}\right) + \varepsilon_{ij1}^t \geq U_i\left(g_{ij}^t = 0, \boldsymbol{g}_{-(i,j)}^{t-1}; \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta}\right) + \varepsilon_{ij0}^t, \tag{8}$$

where $\boldsymbol{g}_{-(i,j)}^{t-1}$ refers to network $\boldsymbol{g}^{t-1}$ excluding the possible dependency g_{ij} of repository i on repository j. The random shocks to payoffs $\varepsilon_{ij0}^t \in \mathbb{R}$ and $\varepsilon_{ij1}^t \in \mathbb{R}$ capture the fact that unexpected events can occur when developing code in-house (ε_{ij0}) and when creating a dependency (ε_{ij1}), e.g., changes in the composition of the developer team or additional information about repository j. We assume that the random shocks to payoffs ε_{ij0} and ε_{ij1} do not depend on time, are independent across pairs of repositories (i, j), and follow a logistic distribution, which is a standard assumption in the literature on discrete choice models (Mele, 2017; Graham and dePaula, 2020; Boucher and Mourifie, 2017). As a consequence, when the opportunity to update the dependency of repository i on repository j arises, the conditional choice probability of a dependency of repository i on repository j is

$$\mathbb{P}\left(g_{ij}^t = 1 \mid \boldsymbol{g}_{-(i,j)}^{t-1}; \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta}\right) = \frac{e^{U_i(g_{ij}^t=1, \boldsymbol{g}_{-(i,j)}^{t-1}; \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta}) - U_i(g_{ij}^t=0, \boldsymbol{g}_{-(i,j)}^{t-1}; \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta})}}{1 + e^{U_i(g_{ij}^t=1, \boldsymbol{g}_{-(i,j)}^{t-1}; \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta}) - U_i(g_{ij}^t=0, \boldsymbol{g}_{-(i,j)}^{t-1}; \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta})}}. \tag{9}$$

As shown in Mele (2022) (with unobservables) and Butts (2009) and Mele (2017) (without unobservables), the sequence of networks generated by the process described above is a discrete-time Markov process (Norris, 1997), conditional on the unobservable types $\boldsymbol{z}$.

The discrete-time Markov chain described above is an irreducible and aperiodic discrete-time

Markov chain with finite state space and hence admits a unique limiting distribution π, which is the stationary distribution of the Markov chain (Norris, 1997). The stationary distribution π can be represented in closed form as an exponential-family distribution (Mele, 2017), that is, an exponential-family random graph model (Lusher et al., 2013) with local dependence (Schweinberger and Handcock, 2015), of the form

$$\pi(\boldsymbol{g};\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta}) \;\coloneqq\; \left(\prod_{k=1}^{K}\frac{e^{Q_{kk}(\boldsymbol{g}_{kk};\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta})}}{c_{kk}(\boldsymbol{x},\boldsymbol{z};\boldsymbol{\theta})}\right)\left(\prod_{l\neq k}^{K}\prod_{i=1}^{N_k}\prod_{j=1}^{N_l}\frac{e^{u_{ij}(\boldsymbol{\alpha}_b,\boldsymbol{\beta}_b)}}{1+e^{u_{ij}(\boldsymbol{\alpha}_b,\boldsymbol{\beta}_b)}}\right). \tag{10}$$

The potential function Q_{kk} of the within-type-k distribution is defined as

$$\begin{aligned} Q_{kk}\left(\boldsymbol{g}_{kk};\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta}\right) \;\coloneqq\; & \sum_{i=1}^{N}\sum_{j\neq i}^{N} u_{ij}\left(\boldsymbol{\alpha}_w,\boldsymbol{\beta}_w,\boldsymbol{x}_i,\boldsymbol{x}_j,\boldsymbol{z}_i,\boldsymbol{z}_j\right) g_{ij}z_{ik}z_{jk} \\ + \;& \gamma\sum_{i=1}^{N}\sum_{j\neq i}^{N}\sum_{r\neq i,j}^{N} g_{ij}g_{jr}z_{ik}z_{jk}z_{rk}, \end{aligned} \tag{11}$$

while the normalizing constant c_{kk} of the within-type-k distribution is

$$c_{kk}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta}) \;\coloneqq\; \sum_{\boldsymbol{\omega}\in\mathcal{G}_{kk}} e^{Q_{kk}(\boldsymbol{\omega};\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta})}, \tag{12}$$

where the sum is over all possible networks $g'_{kk}\in\mathcal{G}_{kk}$ of type k.

The limiting and stationary distribution π of the Markov chain represents the long-run distribution of the software development process, conditional on the types $\boldsymbol{z}$ of repositories. The first term in (10) represents the likelihood of dependencies among repositories of the same type, whereas the second term represents the likelihood of dependencies between repositories of different types. The stationary distribution π implies that the incentives of developers are captured by a potential function of the form

$$Q(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta}) \;=\; \sum_{k=1}^{K} Q_{kk}\left(\boldsymbol{g}_{kk};\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta}\right) + \sum_{k=1}^{K}\sum_{l\neq k} Q_{kl}\left(\boldsymbol{g}_{kl};\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta}\right), \tag{13}$$

where Q_{kk} is defined in (11) while Q_{kl} is defined by

$$Q_{kl}\left(\boldsymbol{g}_{kl};\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta}\right) \;\coloneqq\; \sum_{i=1}^{N}\sum_{j=1}^{N} g_{ij}\,u_b(\boldsymbol{x}_i,\boldsymbol{x}_j,\boldsymbol{z}_i,\boldsymbol{z}_j,\boldsymbol{\alpha}_b,\boldsymbol{\beta}_b)\,z_{ik}\,z_{jl}. \tag{14}$$

In this software development process, each developer team optimally responds to the existing software dependency network. The Nash equilibria of the underlying potential game of the

software development process are the networks that maximize the potential function Q (Mele, 2017, 2022; Monderer and Shapley, 1996; Chandrasekhar, 2016; Blume et al., 2013). These are networks such that no developer team is willing to form an additional dependency or delete an existing dependency, when the need arises to update repository dependencies. Upon inspecting the stationary distribution π, it is evident that the Nash equilibria have the highest probability of being observed, because the Nash equilibria maximize Q_{kl} and so maximize π.

Having said that, Nash networks will be inefficient in the presence of externalities. Consider the welfare function defined by the sum of the individual maintainers utilities

$$W(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta}) \quad := \quad \sum_{i=1}^{N} U_i(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta}) \quad = \quad Q(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z},\boldsymbol{\theta}) + \gamma \sum_{k=1}^{K} \sum_{i=1}^{N} \sum_{j\neq i}^{N} \sum_{r\neq i,j}^{N} g_{ij} g_{ri} z_{ik} z_{jk} z_{rk}. \quad (15)$$

The welfare function is not equal to the potential function Q when $\gamma \neq 0$, so the networks that maximize welfare may not maximize the potential function and are hence not Nash equilibria.

4 Estimation and Empirical Results

We infer the software development process by assuming that the observed software dependency network is a sample from the stationary distribution π of the discrete-time Markov chain described in Section 3.2. We develop a scalable two-step algorithm by decoupling the estimation of the unobserved types from the structural parameter vector $\boldsymbol{\theta}$, which we describe Section 4.1. Theoretical guarantees for type recovery and parameter recovery for each step are available: e.g., theoretical guaranteees for type recovery can be found in Schweinberger et al. (2020), whereas theoretical guarantees for parameter recovery given types include consistency (Schweinberger and Stewart, 2020) and asymptotic normality (Stewart, 2025) of maximum likelihood estimators, and rates of convergence for maximum likelihood and pseudo-likelihood estimators (Stewart and Schweinberger, 2025). Empirical results are presented in Section 4.2.

4.1 Two-Step Algorithm

We describe a scalable two-step algorithm for estimating directed network formation models with observable and unobservable types. Note that scalable methods are available for undirected network formation models (Babkin et al., 2020; Dahbura et al., 2021), but no scalable methods are available for directed network formation models.

The likelihood function of the parameter vectors $\boldsymbol{\theta}$ and $\boldsymbol{\eta}$ of the software development process based on a single observation of a software dependency network $\boldsymbol{g}$ is

$$\ell(\boldsymbol{\theta}, \boldsymbol{\eta}) \quad := \quad \log \sum_{\boldsymbol{z} \in \mathcal{Z}} p\left(\boldsymbol{z}; \boldsymbol{\eta}\right) \pi(\boldsymbol{g}; \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta}), \tag{16}$$

where $\mathcal{Z}$ is the set of all K^N possible configurations of types $\boldsymbol{z}$, p is the multinomial probability of $\boldsymbol{z}$ according to (1), and π is the stationary distribution of $\boldsymbol{g}$ conditional on observables $\boldsymbol{x}$, unobservables $\boldsymbol{z}$, and the structural parameter vector $\boldsymbol{\theta}$. Maximizing ℓ with respect to $\boldsymbol{\theta}$ and $\boldsymbol{\eta}$ is challenging, because it involves a sum over all K^N possible assignments of N repositories to K types, which is exponential in N. Worse, for each of the K^N possible assignments of N repositories to K types, the stationary distribution π is a function of an intractable normalizing constant, which is the sum of between- and within-type sums, each of which is exponential in the number of ordered pairs of repositories $N_k\,(N_k - 1)$ of type k $(k = 1, \ldots, K)$. Therefore, direct computation of the likelihood function is infeasible unless N is small (e.g., $N \leq 10$).

To develop scalable approximations of likelihood-based inference, we follow Babkin et al. (2020) and decouple the estimation of the unobserved types from the estimation of the structural parameter vector $\boldsymbol{\theta}$ using the following two-step algorithm:

> **Step 1:** Estimate the types of repositories.
>
> **Step 2:** Estimate the structural parameter vector $\boldsymbol{\theta}$ conditional on the estimated types.

Step 1 We estimate the unobserved types of repositories as follows.

First, we approximate the loglikelihood function ℓ by assuming that there are no externalities ($\gamma = 0$), which reduces the loglikelihood function of the directed network formation model to the loglikelihood function of a directed stochastic block model and allows us to tap into the large toolbox developed for stochastic block models.

Second, we bound the loglikelihood function ℓ with $\gamma = 0$ from below by using Jensen's inequality. Adapting the arguments of Vu et al. (2013) and Babkin et al. (2020) from undirected to directed network formation models reveals that the resulting lower bound ℓ_{LB} of the likelihood function ℓ is

$$\ell_{LB}(\boldsymbol{\theta}, \boldsymbol{\eta}; \Xi) \quad := \quad \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{K} \sum_{l=1}^{K} \xi_{ik} \xi_{jl} \log \pi_{ij,kl}(g_{ij}, \boldsymbol{x}_{ij}) + \sum_{i=1}^{N} \sum_{k=1}^{K} \xi_{ik} \left(\log \eta_k - \log \xi_{ik}\right), \tag{17}$$

where $\Xi := (\xi_{ik}) \in [0,1]^{N\times K}$ are auxiliary parameters, η_k is the prior probability of being of type k, and $\pi_{ij,kl}$ is the conditional choice probability of a dependency of repository i on repository j, provided i is of type k and j is of type l with covariates $\boldsymbol{x}_{ij} = (\boldsymbol{x}_i, \boldsymbol{x}_j)$. The value ξ_{ik} serves as approximations of the posterior probability of repositories i being of type k.

Third, we maximize the lower bound ℓ_{LB} with respect to Ξ to obtain the best bound on the loglikelihood function ℓ. Variational algorithms that directly maximize ℓ_{LB} with respect to Ξ risk being trapped in local maxima and can be time-consuming. To facilitate maximization of ℓ_{LB}, we construct a surrogate function M, which is more convenient to maximize than ℓ_{LB}. Vu et al. (2013) and Babkin et al. (2020) propose the following surrogate function, which can be obtained using the arithmetic-geometric mean inequality along with concavity properties of logarithms:

$$\begin{aligned} M\left(\Xi; \boldsymbol{\theta}, \boldsymbol{\eta}^{(t)}, \Xi^{(t)}\right) \;:=\; & \sum_{i=1}^{N}\sum_{j\neq i}^{N}\sum_{k=1}^{K}\sum_{l=1}^{K}\left(\xi_{ik}^2\frac{\xi_{jl}^{(t)}}{2\xi_{ik}^{(t)}} + \xi_{jl}^2\frac{\xi_{ik}^{(t)}}{2\xi_{jl}^{(t)}}\right)\log\pi_{ij,kl}^{(t)}(g_{ij}, \boldsymbol{x}_{ij}) \\ & + \sum_{i=1}^{N}\sum_{k=1}^{K}\xi_{ik}\left(\log\eta_k^{(t)} - \log\xi_{ik}^{(t)} - \frac{\xi_{ik}}{\xi_{ik}^{(t)}} + 1\right), \end{aligned} \tag{18}$$

where $\xi_{ij}^{(t)}$ denotes the estimate of ξ_{ij} at iteration t. Maximizing (18) with respect to Ξ forces ℓ_{LB} uphill and can be implemented by quadratic programming using fast sparse matrix operations. The resulting minorization-maximization (MM) algorithm is less prone to being trapped in local maxima than variational algorithm that directly maximize ℓ_{LB} with respect to Ξ.

Finally, we assign repository i to type $\widehat{k} = \text{argmax}_k\, \widehat{\xi}_{i,k}$, where $\widehat{\xi}_{i,k}$ is an approximation of the posterior probability of i being of type k.

Step 2 We estimate the structural parameter vector $\boldsymbol{\theta}$ by maximizing the pseudo-likelihood function of $\boldsymbol{\theta}$, which is the product of conditional choice probabilities of links conditional on the types estimated in Step 1 (Babkin et al., 2020; Martínez Dahbura et al., 2023). Maximum pseudo-likelihood estimators are more scalable than maximum likelihood estimators and are therefore preferred on computational grounds.

The resulting two-step algorithm is scalable, in that it can handle small and large networks, and it has been tested on undirected networks with a quarter of million units (Dahbura et al., 2021). Section 4 provides theoretical guarantees for type and parameter recovery in Steps 1 and 2.

Table 5: Parameter estimates and standard errors.

	Within Estimate (Std. Error)	Between Estimate (Std. Error)
Edges (α_1)	−5.801	-7.979
	(0.276)	(0.006)
$\log(n)\times$ Edges (α_2)	−0.533	
	(0.031)	
Externality (γ)	−0.095	
	(0.02)	
Maturity (β_1)	1.791	-0.222
	(0.305)	(0.011)
$\log(n)\times$ Maturity $\left(\beta_1^\ell\right)$	−0.123	
	(0.035)	
Popularity (β_2)	1.114	-0.07
	(0.31)	(0.011)
$\log(n)\times$Popularity $\left(\beta_2^\ell\right)$	−0.122	
	(0.036)	
Size (β_3)	1.944	0.118
	(0.305)	(0.01)
$\log(n)\times$Size $\left(\beta_3^\ell\right)$	−0.18	
	(0.035)	

Estimates and standard errors are obtained by Maximum Pseudolikelihood (MPLE), conditioning on the estimated types in the first step. The number of unobservable types for the first step is $K = 10$. Standard error for the estimates are in parenthesis.

4.2 Results

We estimate a model with $K = 10$ unobserved types and initialize the types allocation for our algorithm using the Walktrap algorithm.[23] While there is no established method to infer K in this class of models, we have run the estimation with $K = 5$ and $K = 15$ for robustness, but in both cases the model's fit was worse than with $K = 10$.

The results of our estimation are reported in Table 5. The first column shows estimates of the structural parameters for links of the same (unobservable) type, while the remaining column provides the estimates for links of different types.

[23] As suggested by Wainwright and Jordan (2008) and others, we ran the algorithm with multiple initial guesses of the type allocations, but report the results that achieved the highest lower bound.

We estimate the externality (γ) to be negative, providing empirical evidence of the interdependence of links suggested by our strategic model. Essentially, our results provide evidence that when developers form connections, they consider the indirect impacts of their actions, at least partially. This behavior leads to a network architecture that might not be as efficient as one where a central planner aims to maximize the overall utility of all participants. Indeed, the externality encourages developers to establish fewer connections than would be ideal from an aggregate utility-maximization perspective. Maintainers see the number of dependencies of a repository they consider linking to as a cost, thus leading to linking decisions that decrease the density of the aggregate network. The negative externality has further implications for contagion, as we explain in the next section.

The parameters α_1 and α_1^ℓ govern the density of the network and are both estimated negative. The parameter α_1 governing links between types is likewise estimated negative. The magnitudes of the estimated parameters imply that—other things being equal—in equilibrium subnetworks of libraries of the same type tend to be less sparse than between-types. The sparsity increases with the number of libraries belonging to a particular type. One interpretation is that α measures costs of forming links, which is higher when there are more available libraries to choose from, possibly because of the need to monitor and evaluate more code when forming a decision.

By contrast, the parameters $\beta_1, \beta_2, \beta_3$ are all positive within types, indicating that developers are more likely to link to repositories of the same type of similar maturity, popularity and size. On the other hand, this homophily seems attenuated by the size of the type, as shown by the negative coefficients of the parameters $\beta_1^\ell, \beta_2^\ell, \beta_3^\ell$.

When forming dependencies with repositories of a different type, the effect of the observables is negative for maturity and popularity, while positive for size. We interpret this as evidence of complementarities across types.

One interpretation consistent with these results is that developers use popular features of same-type repositories to include in their own code but use infrastructure-style code from different-type repositories, which is not necessarily very popular.

4.3 Model Fit

To assess model fit, we follow the methodology developed in the literature on ERGM and HERGMs and simulate the model under the estimated parameters (Hunter et al., 2008; Babkin et al.,

2020). We compute network statistics on those simulated networks and compare them with the corresponding statistics evaluated on the observed network. If the model fits the data, most of the simulated network statistics should resemble their observed counterparts.

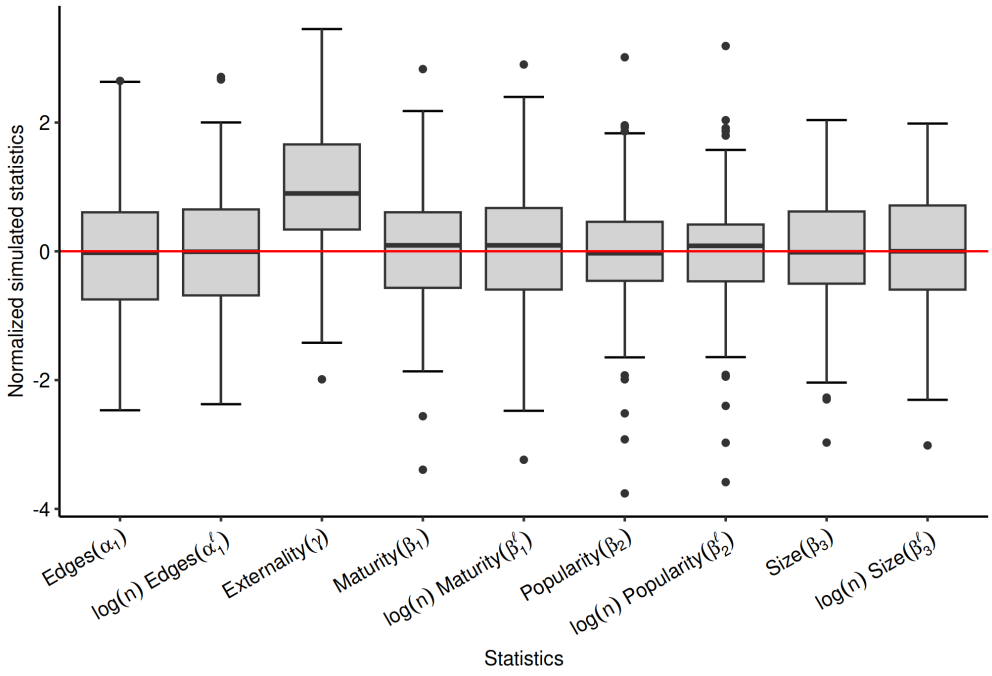
In Figure 2 we show a boxplot with summaries of the networks obtained from 100 simulations of the model. The summaries show in Figure 2 are natural summaries, because they are the sufficient statistics of the model conditional on the types of repositories. [24] These summaries are normalized, so that a value of zero indicates a perfect match. The red line is the observed network, so a perfect fit corresponds to all the network statistics concentrated around the red line. The boxplots show that the model is able to capture most of the aggregate structural properties of the Python dependency network: most of the simulated statistics are centered around the observed network. The only exception is the externality (i.e. the number of two-paths in our model), however the observed network falls in the 95% confidence interval of the simulation output. We conclude that the estimated model does a good job in reproducing important features of the network.

In Figure 3 we report similar boxplots for in-degree and out-degree, showing that the simulated networks replicate the observed one quite closely. These results are particularly striking as it is usually quite hard to fit such a model for a very large network.

In the Online Appendix B, we compare the fit of our model to alternative approaches. An important question is whether the unobserved heterogeneity included in our model is needed. To answer this question we compare our estimated model to a model without unobserved types ($K = 1$). This corresponds to a standard Exponential Random Graph model (ERGM), as studied in Mele (2017). The ERGM is not able to match the aggregate density of the network, while our model's simulations are concentrated around the observed number of links. The second question is whether the externalities are important in explaining the topology of the equilibrium network. Therefore, we compare our model to a Stochastic blockmodel with covariates and $K = 10$ types. This corresponds to our model with $\gamma = 0$. Our simulations show that the our model does a better job matching the externality within types, and should therefore be preferred. Finally, while there is no currently established method to choose the optimal number of types for this class of complex models, we have done a significant effort in setting an appropriate K for estimation. Our practical approach consists of setting the number of types by checking the model's fit for different values of K. In appendix B we show the fit with $K = 5$ and $K = 15$. These alternative models overestimate or underestimate the externality level, so our

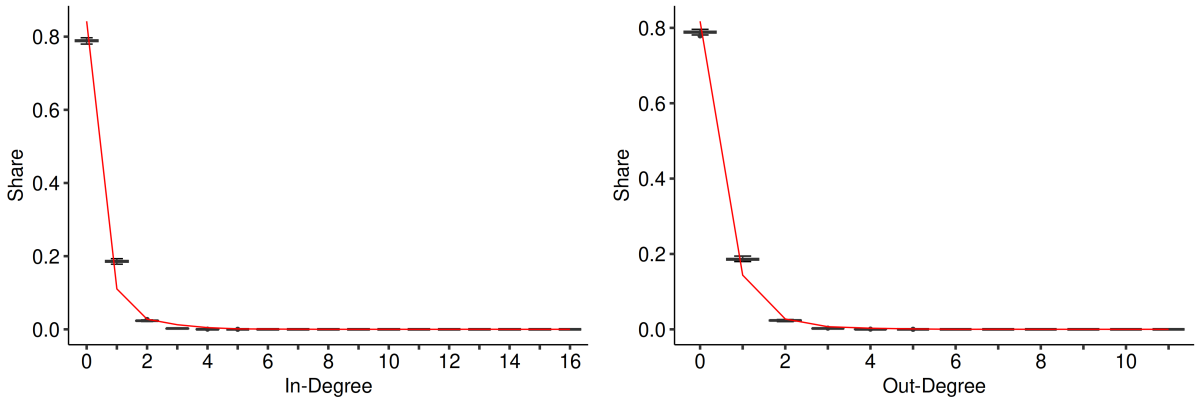
[24] We start the simulations at the observed network, and we use a Metropolis-Hastings sampler to update the network. We first run a burn-in of 500,000 iterations, and then sample a network every 50,000 iterations.

Figure 2: Model fit: Network statistics



Each boxplot is based on 100 simulations of the network generated using the estimated parameters. Simulations are initialized at the observed network and proceed with a burn-in period of 500,000 Metropolis–Hastings steps. After burn-in, one network is retained every 50,000 steps. For each retained network, the corresponding network statistics are computed and then standardized such that a value of zero corresponds to the statistic of the observed network. On the x-axis, one boxplot is displayed for each network statistic associated with a coefficient of the fitted model.

Figure 3: Model Fit: Degree distributions



The two plots represent the in-degree and out-degree distribution of the network. The red solid line is the observed degree distribution, while the (black) boxplots show the results of the simulations from our estimated model.

model with $K = 10$ is preferred.

5 Contagious Vulnerabilities

A growing literature explores contagion on networks, including the statistical literature on infectious diseases (e.g., the contagion of HIV and coronaviruses). Such models (see, e.g., Schweinberger et al., 2022; Groendyke et al., 2012; Britton and O'Neill, 2002) usually first use a network formation model to generate a network of contacts among agents and then study how an infectious disease following a stochastic process spreads within this network.

We adapt these ideas to the contagion of vulnerabilities in software dependency graphs. The prevalence of vulnerabilities in software motivated the creation of the Common Vulnerabilities and Exposure (CVE) program and the National Vulnerability Database (NVD) by the National Institute of Standards and Technology, which is an agency of the United States Department of Commerce (NIST, 2024). A software package depending on a vulnerable package is potentially vulnerable itself. To see how, consider, for example, the Equifax data breach (Federal Trade Commission, 2024) in 2017, caused by a vulnerability in the popular open source web application framework Apache Struts 2.[25] Equifax did not itself develop or maintain Apache Struts 2,

[25] Apache Struts 2 did not properly validate the Content-Type header of incoming http requests when processing file uploads, which allowed attackers to create http requests that include malicious code. Due to the lack of content validation, this allowed attackers to execute arbitrary code on the web server, ultimately granting them full control

but instead used it as part of its own code base. The vulnerability allowed attackers to execute malicious code providing access to the server hosting the web application developed by Equifax using Apache Struts 2 and costing the company more than $1.5 Billion to date.[26]

Package managers are part of a large suite of tools that help system administrators keep their systems up to date and patch any known vulnerabilities. It takes time for vulnerable systems to be updated (Edgescan, 2022),[27] not all vulnerabilities are publicly disclosed (Arora et al., 2008), and it could even be optimal to release vulnerable software packages (Arora et al., 2006). This leaves time for attackers to exploit vulnerabilities in downstream software packages and thereby attack the systems hosting code that depends on the vulnerable packages.

Since it is not ex ante clear which part of the dependency code is vulnerable, we assume that package i uses vulnerable code in dependency j with probability $\lambda_{ij} \in [0,1]$ and, for simplicity, we take $\lambda_{ij} = 1$ and put ourselves in the worst case scenario. where an infected package would transmit the vulnerability with certainty to all downstream packages. This also avoids relying on statistical measures of the extent of contagion.

How contagious a vulnerability is can be measured in this setting by the number of downstream repositories it affects. While Github and other repository hosting services make it easy to see the number of packages directly depending on a given repository, direct dependencies are only imperfect proxies for the total number of vulnerable repositories two or three steps away. Formally, the d-neighborhood $\mathcal{N}_i^d(\boldsymbol{g})$ of node i in network $\mathcal{G}$ is defined via:

$$\mathcal{N}_i^1(\boldsymbol{g}) = N_i(\boldsymbol{g}) \quad \text{and } \mathcal{N}_i^k(\boldsymbol{g}) = \mathcal{N}_i^{k-1}(\boldsymbol{g}) \cup \left(\bigcup_{j \in \mathcal{N}_i^{k-1}(\boldsymbol{g})} N_j(\boldsymbol{g}) \right),$$

where g is the adjacency matrix of $\mathcal{G}$ and $N_i(\boldsymbol{g}) = \{j \in \mathcal{G} : \ g_{ji} = 1\}$ is the (in)-neighborhood of node i. Using this, we define a repository's *k-step systemicness* as $\text{Syst.}k_i(g) \equiv \mathcal{N}_i^k(g)$. We measure the aggregate *systemic risk* of the software dependency network under vulnerability contagion as the *average* k-step systemicness of the network.

Now that we have a measure for how "infectious" a vulnerable software package is, we turn to the question of how to prevent the spread of vulnerabilities. We borrow from Chatterjee and

of the server. For details see the official NIST announcement on CVE-2017-5638 (https://nvd.nist.gov/vuln/detail/CVE-2017-5638).

[26] See https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473. Article accessed 27 January 2024.

[27] It takes organizations between 146 and 292 days to patch cyber security vulnerabilities (Source: Statista. Accessed on 2024-01-31).

Zehmakan (2023), who compare different vaccination strategies in network-based contagion models. Since it is not feasible to find an optimal vaccination strategy–the optimization problem is NP hard–the authors compare various heuristics based on a node's network position and pathogen parameters. They find that a vaccination strategy based on a node's betweenness centrality and a heuristic they call *expected fatality* is most effective in preventing fatalities.

By setting $\lambda_{ij} = 1 \forall i, j \in N$, we assume that all upstream neighbors of a vulnerable node (repository) are also vulnerable. And second, nodes cannot "die" in our setup, so they are not removed from the population. They can, in principle, recover from being exposed to a vulnerable package if the vulnerability is patched and the dependency is updated, but this takes time, as discussed above. Consequently, software systems are vulnerable for a period of time once a vulnerability is discovered. We are interested in contagion processes occurring during this time.

Consequently, we study the effectiveness of a prevention strategy based on securing the most important nodes against vulnerabilities (e.g. by through government funding or regulation). We compare three strategies. First, we rank the repositories based on their in-degree and target the nodes with highest indegree. Second, we follow Chatterjee and Zehmakan (2023) proposal to rank the nodes using an equally-weighted combination of a node's betweenness centrality and expected fatality $ef(i)$ of node i, which in our case is computed as:

$$ef(i) = \sum_{j \in N(i)} \frac{1}{|N(j)|}.$$

Third, we use the expected fatality alone to rank the nodes.

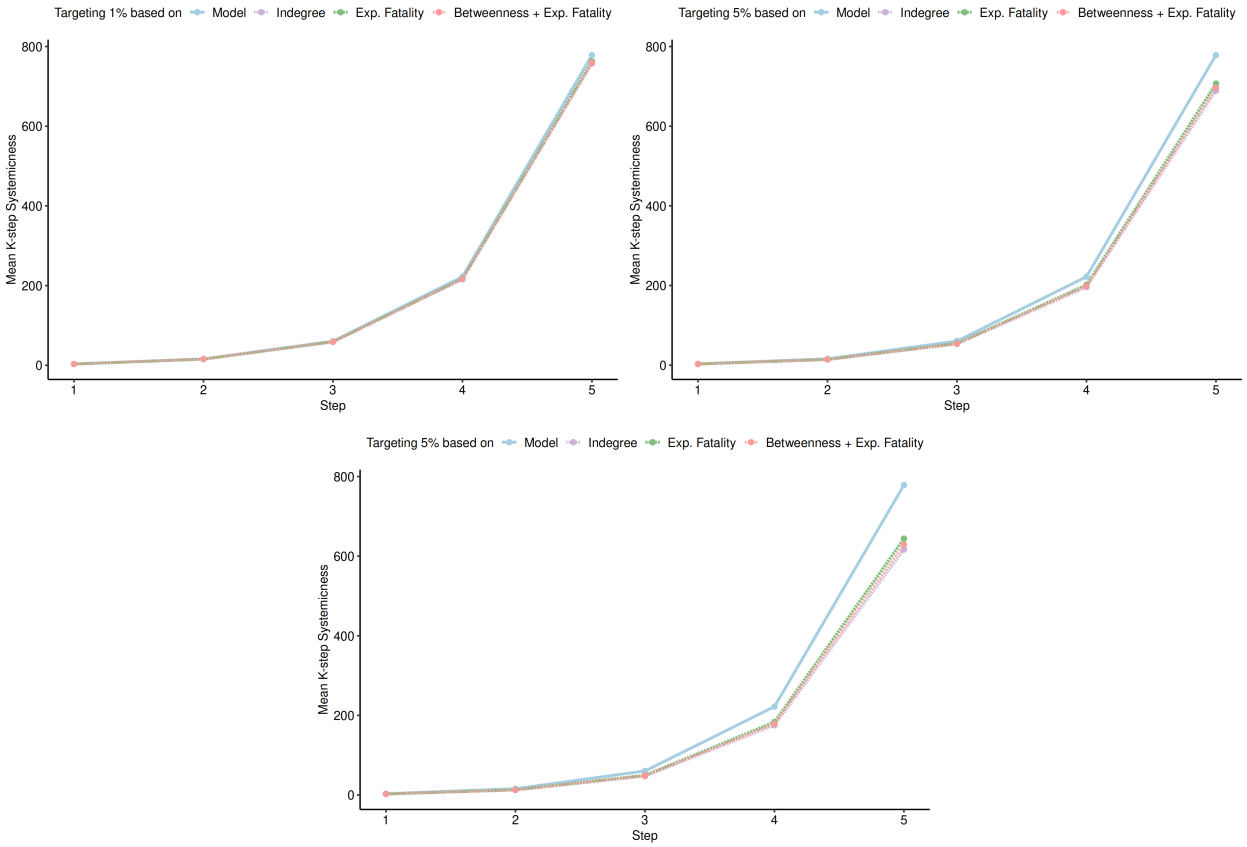
Using the ranking based on the indicators described above, we simulate our model and measure systemic risk. Our policy counterfactuals are obtained by simulating 1000 networks according to the model under the policy and then measuring the systemic risk according to the *k-step systemicness*.[28] For each simulated network we record the mean of k-step systemicness as an aggregate measure of systemic risk.

5.1 Targeted Interventions

In the top left panel of Figure **??** we show the effect of a policy that targets 1% of the nodes and makes them *invulnerable* to contagion. While this seems quite hard to achieve in practice, it

[28] This procedure is essentially the same used for generating the model fit graphs, except that we change some parameters of the simulation to conform to the policies under study.

Figure 4: Mean k-step systemicness of policy targeting 1%, 5% and 10% of the nodes.



Targeting is done according to In-degree, Expected Fatality, and 1/2×Expected Fatality + 1/2× Betweenness. Each line is based on the average of 1000 model simulations. Top left: Targeting based on top 1% of nodes. Top right: Targeting based on top 5% of nodes. Bottom: Targeting based on top 10% of nodes.

is a good benchmark. We provide 3 different policies, targeting the nodes based on the highest in-degree, the expected fatality or a mix of betweenness centrality and expected fatality as explained above. The graph reports the model's average k-step systemicness in 1000 simulations (blue solid line). In equilibrium the average 5-steps systemicness for the Pypi ecosystem is close to 800 libraries. This implies that on average a bug or a vulnerability randomly introduced in the system will affect 800 libraries downstream in 5 steps.

The policy targeting 1 percent of nodes based on the highest in-degrees is shown as purple dashed line; the policy based on the expected fatality is the green long-dash line; and the policy based on the indicator that equally weights betweenness centrality and expected fatality is the red dotted line.

This policy does not seem very effective. The systemic risk, as measured by the average *k-step systemicness* of our simulations does not seem to change in a meaningful way.[29] We repeat the exercise in top right and bottom panel of Figure 4, where we target the highest 5 and 10 percent packages respectively, based on the targeting measure. While the effect of such policies is more pronounced, it does not really flatten out the risk profile of the network.

5.2 The effect of AI-assisted coding

In recent years, the development of AI tools to help software development has increase exponentially. The introduction of LLM-based assistants has the potential to greatly increase productivity. In fact, recent studies about the adoption of such tools in software development support this claim. Peng et al. (2023) provide experimental evidence that software developers using Github Co-Pilot were able to complete a task 55.8% faster, with less experienced developers benefiting the most from the AI-pair programming tool. Edelman et al. (2023), Noy and Zhang (2023) and Yilmaz and Karaoglan Yilmaz (2023) find similar results in slightly different experimental settings.

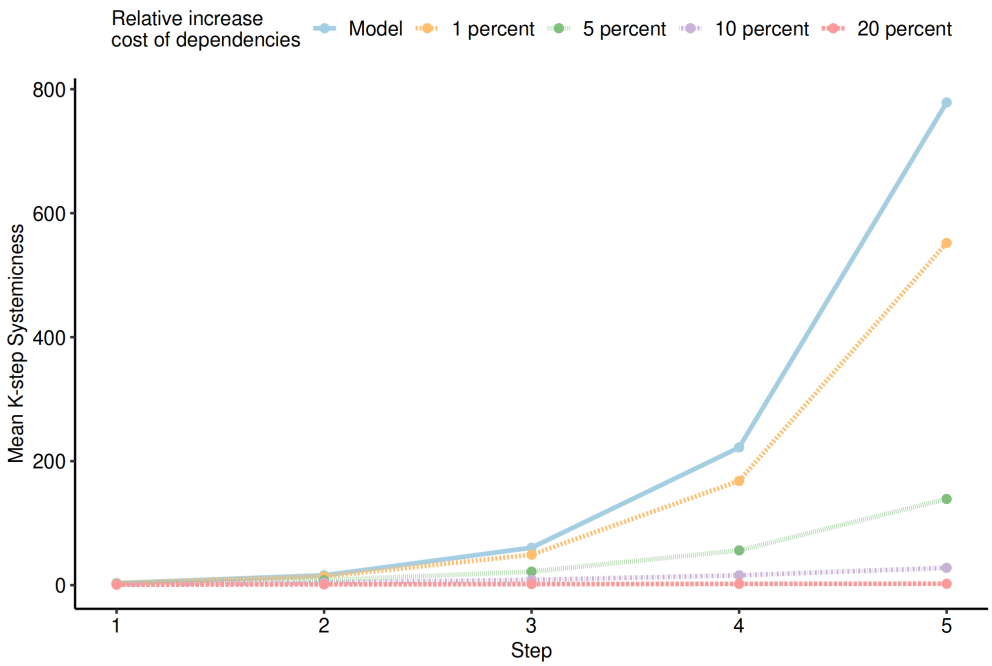
We contend that this development is likely to decrease systemic risk in the long-run. The effect of AI-assisted tools on software programming is as a first order decrease in cost of coding in-house. Each coder becomes more productive, and as a consequence the cost of producing high-quality software would decrease. In our model, this means that the relative cost of forming a dependency will increase, since our model payoffs measure the differential cost between producing in-house vs adapting the code to form a dependency.

In light of this simple observation, we simulate the introduction of AI-assisted programming as an increase in cost of forming dependencies. We simulate different scenarios, where these costs increase by 1, 5, 10, 20 and 30 percent.

These simulations are shown in Figure 5. The model simulations are represented by the blue solid line. We notice that a small change in costs of 1 percent (yellow dashed line) has a pronounced effect on the systemic risk profile. A 5 percent change (green dotted line) dramatically improves the risk profile, thus being extremely effective. A 10 percent change (purple dotted dashed line) in the cost of forming a dependency essentially flattens out the systemic risk profile, as the network becomes relatively sparse and most coders do not form dependencies. This

[29] We have performed t-tests to check whether the policy has significantly altered the average k-step systemicness and our test shows that this is the case, even if the effect is very small in this case.

Figure 5: Mean k-step systemicness arising from introduction of AI-assisted coding.



Each line is based on the average of 1000 model simulations of equilibrium adjustment, following a specific percent decrease in the cost of coding in-house.

effect becomes stronger with a change of 20 percent in cost (represented as a red line).

6 Conclusion

The network of dependencies among software packages is an interesting laboratory to study network formation and how incentives and externalities shape the topology of the network. Indeed, the creation of modern software gains efficiency by re-using existing libraries; on the other hand, dependencies expose new software packages to bugs and vulnerabilities from other libraries. This feature of the complex network of dependencies motivates the interest in understanding the incentives and equilibrium mechanisms driving the formation of such networks.

In this paper, we estimate a directed network formation model to undertake a structural analysis of the motives, costs, benefits and externalities that a maintainer faces when developing a new software package. The empirical model allows us to disentangle observable from unobservable characteristics that affect the decisions to form dependencies to other libraries.

Using data from the Python dependency network, we find evidence that coders create negative externalities for other coders when creating a link. This raises more questions about the formation of dependency graphs. Using our estimated model, we study the vulnerability of the dependency network to spread of a bug. We measure systemic risk as the average number of downstream packages that are affected by a vulnerability, assuming that the contagion follows a SIR model of epidemic. We simulate interventions that target nodes according to their in-degree, expected fatality and betweenness centrality. Such interventions are unlikely to significantly decrease the risk profile of the network. On the other hand, the increase in AI-assisted tools for programmers, may have a beneficial effect by reducing the cost of producing code in-house, without much need for dependencies.

While we have focused on a stationary realization of the network, there are important dynamic considerations in the creation of these dependencies. While the modeling of forward-looking maintainers may be useful to develop intuition about intertemporal strategic incentives and motives, we leave this development to future work, as it involves several complications in the estimation. Finally, we have focused on a single language, but the analysis can be extended to other languages as well, such as `Java`, `R`, `Rust`, `C++` and others.

References

Abelson, H., Sussman, G. J. and Sussman, J. (1996), *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA.

Arora, A., Caulkins, J. P. and Telang, R. (2006), 'Sell first, fix later: Impact of patching on software quality', *Management Science* **52**(3), 465–471.

Arora, A., Telang, R. and Xu, H. (2008), 'Optimal policy for software vulnerability disclosure', *Management Science* **54**(4), 642–656.

Babkin, S., Stewart, J. R., Long, X. and Schweinberger, M. (2020), 'Large-scale estimation of random graph models with local dependence', *Computational Statistics & Data Analysis* **152**, 1–19.

Barros-Justo, J. L., Pinciroli, F., Matalonga, S. and Martínez-Araujo, N. (2018), 'What software reuse benefits have been transferred to the industry? a systematic mapping study', *Information and Software Technology* **103**, 1–21.

Bickel, P., Choi, D., Chang, X. and Zhang, H. (2013), 'Asymptotic normality of maximum likelihood and its variational approximation for stochastic blockmodels', *The Annals of Statistics* **41**(4), 1922 – 1943.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R. and Lefebvre, E. (2008), 'Fast unfolding of communities in large networks', *Journal of Statistical Mechanics: Theory and Experiment* **P1008**.

Blume, L., Easley, D., Kleinberg, J., Kleinberg, R. and Tardo's, E. (2013), 'Network formation in the presence of contagious risk', *ACM Transactions on Economics and Computation* **1**, 6:1–6:20.

Bonhomme, S., Lamadon, T. and Manresa, E. (2019), 'A distributional framework for matched employer employee data', *Econometrica* **87**(3), 699–739.

Boucher, V. and Mourifie, I. (2017), 'My friend far far away: A random field approach to exponential random graph models', *Econometrics Journal* **20**(3), S14–S46.

Boysel, S. (2023), 'No free lunch for programmers: Digital supply chains and the economics of software dependency management'.

Bramoullé, Y., Galeotti, A. and Rogers, B. (2016), *The Oxford Handbook of the Economics of Networks*, Oxford Handbooks, Oxford University Press.

Britton, T. and O'Neill, P. D. (2002), 'Statistical inference for stochastic epidemics in populations with network structure', *Scandinavian Journal of Statistics* **29**, 375–390.

Butts, C. T. (2009), A behavioral micro-foundation for cross-sectional network models, mimeo.

Butts, C. T. (2019), 'A dynamic process interpretation of the sparse ERGM reference model', *Journal of Mathematical Sociology* .

Chandrasekhar, A. G. (2016), Econometrics of network formation, *in* yann bramoulle, andrea galeotti and brian rogers, eds, 'Oxford handbook on the economics of networks.', Oxford Univerisity Press.

Chatterjee, S. and Zehmakan, A. N. (2023), Effective vaccination strategies in network-based SIR model, mimeo.

Currarini, S., Jackson, M. O. and Pin, P. (2010), 'Identifying the roles of race-based choice and chance in high school friendship network formation', *Proceedings of the National Academy of Sciences* **107**(11), 4857ï¿½4861.

Dahbura, J. N. M., Komatsu, S., Nishida, T. and Mele, A. (2021), 'A structural model of business cards exchange networks'.

Decan, A., Mens, T. and Grosjean, P. (2019), 'An empirical comparison of dependency network evolution in seven software packaging ecosystems', *Empirical Software Engineering* **24**, 381–416.

DePaula, A. (2017), Econometrics of network models, *in* B. Honore, A. Pakes, M. Piazzesi and L. Samuelson, eds, 'Advances in Economics and Econometrics: Eleventh World Congress', Cambridge University Press.

DePaula, A., Richards-Shubik, S. and Tamer, E. (2018), 'Identifying preferences in networks with bounded degree', *Econometrica* **86**(1), 263–288.

Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M. and Halderman, J. A. (2014), The matter of heartbleed, *in* 'Proceedings of the 2014 Conference on Internet Measurement Conference', IMC '14, Association for Computing Machinery, New York, NY, USA, pp. 475–488.

Dziubiński, M. and Goyal, S. (2013), 'Network design and defence', *Games and Economic Behavior* **79**, 30–43.

Edelman, B. G., Ngwe, D. and Peng, S. (2023), 'Measuring the impact of ai on information worker productivity'.

Edgescan (2022), 2022 vulnerability statistics report, mimeo. Accessed: 2024-01-31.
URL: *https://www.edgescan.com/resources/vulnerability-stats-report/*

Federal Trade Commission (2024), 'Equifax data breach settlement', Federal Trade Commission - Enforcement.
URL: *https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement*

Galeotti, A., Goyal, S. and Kamphorst, J. (2006), 'Network formation with heterogeneous players', *Games and Economic Behavior* **54**(2), 353–372.

Goldfarb, A. and Tucker, C. (2019), 'Digital economics', *Journal of Economic Literature* **57**(1), 3–43.

Goyal, S. and Vigier, A. (2014), 'Attack, defence, and contagion in networks', *The Review of Economic Studies* **81**(4), 1518–1542.

Graham, B. (2017), 'An empirical model of network formation: with degree heterogeneity', *Econometrica* **85**(4), 1033–1063.

Graham, B. (2020), Network data, *in* S. Durlauf, L. Hansen, J. Heckman and R. Matzkin, eds, 'Handbook of econometrics 7A', Amsterdam: North-Holland,.

Graham, B. and dePaula, A., eds (2020), *The econometric analysis of network data*, Amsterdam: Academic Press, 2020.

Groendyke, C., Welch, D. and Hunter, D. R. (2012), 'A network-based analysis of the 1861 Hagelloch measles data', *Biometrics* **68**, 755–765.

Grove, D., DeFouw, G., Dean, J. and Chambers, C. (1997), Call graph construction in object-oriented languages, *in* 'Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications', OOPSLA '97, Association for Computing Machinery, New York, NY, USA, p. 108–124.

Hopper, G. M. (1952), The education of a computer, *in* 'Proceedings of the Association for Computing Machinery Conference', pp. 243–249.

Hunter, D. R., Goodreau, S. M. and Handcock, M. S. (2008), 'Goodness of fit of social network models', *Journal of the American Statistical Association* **103**, 248–258.

Jackson, M., ed. (2008), *Social and economic networks*, Princeton.

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

Kay, A. C. (1993), The early history of smalltalk, Url: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html.

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

Knuth, D. E. (1997), *The art of computer programming, volume 1 (3rd ed.): Fundamental algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Krasner, H. (2018), The cost of poor quality software in the us: A 2018 report, Report.
URL: *https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf*

Krivitsky, P. N., Coletti, P. and Hens, N. (2023), 'A tale of two datasets: Representativeness and generalisability of inference for samples of networks', *Journal of the American Statistical Association* **118**, 2213–2224.

Krivitsky, P. N., Handcock, M. S. and Morris, M. (2011), 'Adjusting for network size and composition effects in exponential-family random graph models', *Statistical Methodology* **8**, 319–339.

LaBelle, N. and Wallingford, E. (2004), 'Inter-package dependency networks in open-source software'.

Lerner, J. and Tirole, J. (2002), 'Some simple economics of open source', *The Journal of Industrial Economics* **50**(2), 197–234.

Lewis, J. A., Henry, S. M., Kafura, D. G. and Schulman, R. S. (1991), An empirical study of the object-oriented paradigm and software reuse, *in* 'OOSPLA 91', pp. 184–196.

Lusher, D., Koskinen, J. and Robins, G. (2013), *Exponential Random Graph Models for Social Networks*, Cambridge University Press, Cambridge, UK.

Martínez Dahbura, J. N., Komatsu, S., Nishida, T. and Mele, A. (2023), 'Homophily and community structure at scale: An application to a large professional network', *AEA Papers and Proceedings* **113**, 156–60.

McCarthy, J. (1996), The implementation of LISP, Stanford AI Lab: The History of LISP.
URL: *http://www-formal.stanford.edu/jmc/history/lisp/lisp.html*

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

Mele, A. (2022), 'A structural model of homophily and clustering in social networks', *Journal of Business & Economic Statistics* **40**(3), 1377–1389.

Mele, A. and Zhu, L. (2023), 'Approximate variational estimation for a model of network formation', *Review of Economics and Statistics* **105**(1), 113–124.

Monderer, D. and Shapley, L. (1996), 'Potential games', *Games and Economic Behavior* **14**(1), 124–143.

National Institute of Standards and Technology (2002), Software errors cost u.s. economy $59.5 billion annually, Press release.
URL: *https://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm*

NIST (2024), 'The National Vulnerabilities Database', https://nvd.nist.gov/vuln. Accessed: 2024-01-31.

Norris, J. R. (1997), *Markov chains*, Cambridge University Press, Cambridge.

Noy, S. and Zhang, W. (2023), 'Experimental evidence on the productivity effects of generative artificial intelligence', *Science* **381**(6654), 187–192.

Parametrix (2024), Crowdstrike's impact on the fortune 500–an impact analysis, Report.

Peng, S., Kalliamvakou, E., Cihon, P. and Demirer, M. (2023), 'The impact of ai on developer productivity: Evidence from github copilot'.

R Core Team (2024), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria.
URL: *https://www.R-project.org/*

Schueller, W., Wachs, J., Servedio, V., Thurner, S. and Loreto, V. (2022), 'Curated data on the rust ecosystem: Collaboration networks and library dependencies', *Scientific Data* **9**(703).

Schweinberger, M., Bomiriya, R. P. and Babkin, S. (2022), 'A semiparametric Bayesian approach to epidemics, with application to the spread of the coronavirus MERS in South Korea in 2015', *Journal of Nonparametric Statistics* **34**, 628–662.

Schweinberger, M. and Handcock, M. S. (2015), 'Local dependence in random graph models: characterization, properties and statistical inference', *Journal of the Royal Statistical Society–Statistical Methodology Series B* **77**, 647–676.

Schweinberger, M., Krivitsky, P. N., Butts, C. T. and Stewart, J. R. (2020), 'Exponential-family models of random graphs: Inference in finite, super, and infinite population scenarios', *Statistical Science* **35**, 627–662.

Schweinberger, M. and Stewart, J. (2020), 'Concentration and consistency results for canonical and curved exponential-family models of random graphs', *The Annals of Statistics* **48**(1), 374–396.

Snijders, T. A. (2002), 'Markov chain Monte Carlo estimation of exponential random graph models', *Journal of Social Structure* **3**(2).

Stewart, J. R. (2025), 'Rates of convergence and normal approximations for estimators of local dependence random graph models', *Bernoulli* . To appear.

Stewart, J. R. and Schweinberger, M. (2025), 'Pseudo-likelihood-based M-estimators for random graphs with dependent edges and parameter vectors of increasing dimension', *The Annals of Statistics* . To appear.

Traore, S. A. K., Valero, M., Shahriar, H., Zhao, L., Ahamed, S. and Lee, A. (2022), Enabling cyber-analytics using iot clusters and containers, *in* '2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)', pp. 1798–1803.

Vu, D. Q., Hunter, D. R. and Schweinberger, M. (2013), 'Model-based clustering of large networks', *The Annals of Applied Statistics* **7**(2), 1010 – 1039.

Wainwright, M. and Jordan, M. (2008), 'Graphical models, exponential families, and variational inference', *Foundations and Trends@ in Machine Learning* **1**(1-2), 1–305.

Yilmaz, R. and Karaoglan Yilmaz, F. G. (2023), 'The effect of generative artificial intelligence (ai)-based tool use on students' computational thinking skills, programming self-efficacy and motivation', *Computers and Education: Artificial Intelligence* **4**, 100147.

Zheng, X., Zeng, D., Li, H. and Wang, F. (2008), 'Analyzing open-source software systems as complex networks', *Physica A: Statistical Mechanics and its Applications* **387**(24), 6190–6200.

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

ONLINE APPENDIX

A Exploiting sparse matrix operations for updating variational parameters

Next, we detail how the updates of ξ in Step 1 can be carried out in a scalable manner based on sparse matrix operations. Equation (17) can be written as a quadratic programming problem in ξ with the block constraints that $\sum_{k=1}^{K} \xi_{ik} = 1$ needs to hold for all $i = 1, \ldots, N$. We rearrange the first line of (17) as follows:

$$\begin{aligned}
&\sum_{i=1}^{N}\sum_{j\neq i}^{N}\sum_{k=1}^{K}\sum_{l=1}^{K}\left(\xi_{ik}^2\frac{\xi_{jl}^{(t)}}{2\xi_{ik}^{(t)}}+\xi_{jl}^2\frac{\xi_{ik}^{(t)}}{2\xi_{jl}^{(t)}}\right)\log\pi_{kl}^{(t)}(g_{ij},\boldsymbol{x}_{ij})\\
&=\sum_{i=1}^{N}\sum_{k=1}^{K}\sum_{j\neq i}^{N}\sum_{l=1}^{K}\xi_{ik}^2\frac{\xi_{jl}^{(t)}}{2\xi_{ik}^{(t)}}\left(\log\pi_{kl}^{(t)}(g_{ij},\boldsymbol{x}_{ij})+\log\pi_{lk}^{(t)}(g_{ji},\boldsymbol{x}_{ji})\right)\\
&=\sum_{i=1}^{N}\sum_{k=1}^{K}\frac{\Omega_{ik}(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)})}{2\xi_{ik}^{(t)}}\xi_{ik}^2
\end{aligned}$$

with

$$\Omega_{ik}\left(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)}\right) := \sum_{j\neq i}^{N}\sum_{l=1}^{K}\xi_{jl}^{(t)}\left(\log\pi_{kl}^{(t)}(g_{ij},\boldsymbol{x}_{ij})+\log\pi_{lk}^{(t)}(g_{ji},\boldsymbol{x}_{ji})\right). \tag{19}$$

We make the dependence of Equation (19) on $\boldsymbol{g}$ and $\boldsymbol{x}$ specific to ease the notation at later steps of this derivation. This yields for Equation (17)

$$\begin{aligned}
M\left(\Xi;\boldsymbol{\theta},\boldsymbol{\eta}^{(t)},\Xi^{(t)}\right) &= \sum_{i=1}^{N}\sum_{k=1}^{K}\left(\frac{\Omega_{ik}^{(t)}\left(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)}\right)}{2\,\xi_{ik}^{(t)}}-\frac{1}{\xi_{ik}^{(t)}}\right)\xi_{ik}^2+\left(\log\eta_k^{(t)}-\log\xi_{ik}^{(t)}+1\right)\xi_{ik}\\
&= \sum_{i=1}^{N}\sum_{k=1}^{K}A_{ik}\left(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)}\right)\xi_{ik}^2+B_{ik}\left(\boldsymbol{\eta}^{(t)},\Xi^{(t)}\right)\xi_{ik}
\end{aligned}$$

where

$$A_{ik}\left(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)}\right) := \frac{\Omega_{ik}\left(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)}\right)}{2\,\xi_{ik}^{(t)}}-\frac{1}{\xi_{ik}^{(t)}}$$

is the quadratic term and

$$B_{ik}\left(\boldsymbol{\eta}^{(t)}, \Xi^{(t)}\right) := \log \eta_k^{(t)} - \log \xi_{ik}^{(t)} + 1$$

the linear term of the quadratic problem.

To update the estimate of Ξ, we need to evaluate $A_{ik}\left(\boldsymbol{g}, \boldsymbol{x}, \Xi^{(t)}\right)$ and $B_{ik}\left(\boldsymbol{\eta}^{(t)}, \Xi^{(t)}\right)$ for $i = 1, \ldots, N$ and $k = 1, \ldots, K$, which, when done naively, is of complexity $O(N^2 K^2)$. Computing $\Omega_{ik}\left(\boldsymbol{g}, \boldsymbol{x}, \Xi^{(t)}\right)$ for $A_{ik}\left(\boldsymbol{g}, \boldsymbol{x}, \Xi^{(t)}\right)$ is the computational bottleneck driving these complexities. It is thus prohibitively difficult for a large number of population members and communities in the network. Note that in our application, this problem is exasperated as we have more than 35,000 nodes and our theoretical result on the adequacy of the estimation procedure assumes that K grows as a function of N. To avoid this issue, we show how $\Omega_{ik}\left(\boldsymbol{g}, \boldsymbol{x}, \Xi^{(t)}\right)$ can be evaluated through a series of sparse matrix multiplications. Thereby, our implementation is fast and light on memory requirements. The underlying idea is that $\Omega_{ik}\left(\boldsymbol{g}, \boldsymbol{x}, \Xi^{(t)}\right)$ collapses to a simple form if $\mathbf{g}$ is an empty network with all pairwise covariates set to zero, i.e., $g_{ij} = 0$ and $x_q = 0 \,\forall\, i, j = 1, \ldots, N$ and $q = 1, \ldots, p$. Given that, in reality, these are seldom the observed values, we, consecutively, go through all connections where either $g_{ij} = 1$ or $X_{ij,1} = x_1, \ldots, X_{ij,p} = x_p$ for $\sum_{q=1}^{p} x_q \neq 0$ holds and correct for the resulting error. In formulae, this implies the following:

$$\begin{aligned}
\Omega_{ik}(\boldsymbol{g}, \boldsymbol{x}, \Xi^{(t)}) \;:=\; & \sum_{j \neq i}^{N} \sum_{l=1}^{K} \xi_{jl}^{(t)} \left(\log \pi_{kl}^{(t)}(g_{ij}, \boldsymbol{x}_{ij}) + \log \pi_{lk}^{(t)}(g_{ji}, \boldsymbol{x}_{ji}) \right) \\
= & \sum_{j \neq i}^{N} \sum_{l=1}^{K} \xi_{jl}^{(t)} \left(\log \pi_{kl}^{(t)}(0, \mathbf{0}) + \log \pi_{lk}^{(t)}(0, \mathbf{0}) \right) \\
& + \sum_{j \neq i}^{N} \sum_{l=1}^{K} \left(g_{ij} \xi_{jl}^{(t)} \left(\log \frac{\pi_{kl}^{(t)}(1, \boldsymbol{x}_{ij})}{\pi_{kl}^{(t)}(0, \mathbf{0})} \right) + g_{ji} \xi_{jl}^{(t)} \left(\log \frac{\pi_{lk}^{(t)}(1, \boldsymbol{x}_{ji})}{\pi_{lk}^{(t)}(0, \mathbf{0})} \right) \right) \\
& + \sum_{j \neq i}^{N} \sum_{l=1}^{K} \left((1 - g_{ij}) \xi_{jl}^{(t)} \left(\log \frac{\pi_{kl}^{(t)}(0, \boldsymbol{x}_{ij})}{\pi_{kl}^{(t)}(0, \mathbf{0})} \right) + (1 - g_{ji}) \xi_{jl}^{(t)} \left(\log \frac{\pi_{lk}^{(t)}(0, \boldsymbol{x}_{ji})}{\pi_{lk}^{(t)}(0, \mathbf{0})} \right) \right) \\
= & \, \Omega_{ik}\left(\mathbf{0}, \mathbf{0}, \Xi^{(t)}\right) + \Lambda_{ik}\left(\boldsymbol{g}, \boldsymbol{x}, \Xi^{(t)}\right),
\end{aligned} \tag{20}$$

with

$$\begin{aligned}
\Lambda_{ik}\left(\boldsymbol{g}, \boldsymbol{x}, \Xi^{(t)}\right) = & \sum_{j \neq i}^{N} \sum_{l=1}^{K} \left(g_{ij} \xi_{jl}^{(t)} \left(\log \frac{\pi_{kl}^{(t)}(1, \boldsymbol{x}_{ij})}{\pi_{kl}^{(t)}(0, \mathbf{0})} \right) + g_{ji} \xi_{jl}^{(t)} \left(\log \frac{\pi_{lk}^{(t)}(1, \boldsymbol{x}_{ji})}{\pi_{lk}^{(t)}(0, \mathbf{0})} \right) \right) \\
& + \sum_{j \neq i}^{N} \sum_{l=1}^{K} \left((1 - g_{ij}) \xi_{jl}^{(t)} \left(\log \frac{\pi_{kl}^{(t)}(0, \boldsymbol{x}_{ij})}{\pi_{kl}^{(t)}(0, \mathbf{0})} \right) + (1 - g_{ji}) \xi_{jl}^{(t)} \left(\log \frac{\pi_{lk}^{(t)}(0, \boldsymbol{x}_{ji})}{\pi_{lk}^{(t)}(0, \mathbf{0})} \right) \right)
\end{aligned} \tag{21}$$

being the error in $\Omega_{ik}\left(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)}\right)$ arising from assuming an empty network with all categorical covariates set to zero. Next, we show how both terms, $\Omega_{ik}\left(\mathbf{0},\mathbf{0},\Xi^{(t)}\right)$ and $\Lambda_{ik}\left(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)}\right)$, can be computed through matrix operations, completing our scalable algorithmic approach.

We, first, assume an empty network where all categorical covariates set to zero, $g_{ij} = x_q = 0$ for all $i \neq j$ and $q = 1,...,p$ and compute $\Omega_{ik}\left(\mathbf{0},\mathbf{0},\Xi^{(t)}\right)$ via matrix multiplications. Observe that

$$\begin{aligned}
\Omega_{ik}\left(\mathbf{0},\mathbf{0},\Xi^{(t)}\right) &= \sum_{j\neq i}^{N}\sum_{l=1}^{K}\xi_{jl}^{(t)}\left(\log\pi_{kl}^{(t)}(0,0)+\log\pi_{lk}^{(t)}(0,0)\right)\\
&= \sum_{l=1}^{K}\sum_{j\neq i}^{N}\xi_{jl}^{(t)}\left(\log\pi_{kl}^{(t)}(0,0)+\log\pi_{lk}^{(t)}(0,0)\right)\\
&= \sum_{l=1}^{K}\left(\sum_{j=1}^{N}\xi_{jl}^{(t)}-\xi_{il}^{(t)}\right)\left(\log\pi_{kl}^{(t)}(0,0)+\log\pi_{lk}^{(t)}(0,0)\right)\\
&= \sum_{l=1}^{K}\left(\tau_{l}^{(t)}-\xi_{il}^{(t)}\right)\left(\log\pi_{kl}^{(t)}(0,0)+\log\pi_{lk}^{(t)}(0,0)\right)
\end{aligned}$$

holds, where

$$\tau_{l}^{(t)} \coloneqq \sum_{j=1}^{N}\xi_{jl}^{(t)}.$$

With

$$\boldsymbol{A_0}\left(\Xi^{(t)}\right) \coloneqq \begin{bmatrix}
\tau_1^{(t)}-\xi_{11}^{(t)} & \tau_2^{(t)}-\xi_{12}^{(t)} & \dots & \tau_K^{(t)}-\xi_{1K}^{(t)}\\
\tau_1^{(t)}-\xi_{21}^{(t)} & \tau_2^{(t)}-\xi_{22}^{(t)} & \dots & \tau_K^{(t)}-\xi_{2K}^{(t)}\\
\vdots & \vdots & \ddots & \vdots\\
\tau^{(t)}(1)-\xi_{n1}^{(t)} & \tau_2^{(t)}-\xi_{n2}^{(t)} & \dots & \tau_K^{(t)}-\xi_{nK}^{(t)}
\end{bmatrix}$$

and

$$\boldsymbol{\Pi_0}\left(\Xi^{(t)}\right) \coloneqq \begin{bmatrix}
\log\pi_{11}^{(t)}(0,0) & \log\pi_{12}^{(t)}(0,0) & \dots & \log\pi_{1K}^{(t)}(0,0)\\
\log\pi_{21}^{(t)}(0,0) & \log\pi_{22}^{(t)}(0,0) & \dots & \log\pi_{2K}^{(t)}(0,0)\\
\vdots & \vdots & \ddots & \vdots\\
\log\pi_{K1}^{(t)}(0,0) & \log\pi_{K2}^{(t)}(0,0) & \dots & \log\pi_{KK}^{(t)}(0,0)
\end{bmatrix},$$

it follows that

$$\boldsymbol{A_0}\left(\Xi^{(t)}\right)\left(\left(\boldsymbol{\Pi_0}\left(\Xi^{(t)}\right)\right)^{\top}+\boldsymbol{\Pi_0}\left(\Xi^{(t)}\right)\right) = \left(\Omega_{ik}\left(\mathbf{0},\mathbf{0},\Xi^{(t)}\right)\right)_{ik} \tag{22}$$

holds. Put differently, we are able to write $\Omega_{ik}^{(t)}(\mathbf{0},\mathbf{0},\Xi)$ as the (i,k)th entry of the result of (22). To compute $\mathbf{\Pi_0}\left(\Xi^{(t)}\right)$, we first evaluate the matrix $\pi^{(t)}(1,\mathbf{0}) \in [0,1]^{K\times K}$ via sparse matrix operations:

$$\begin{aligned}\boldsymbol{\pi}^{(t+1)}(1,\mathbf{0}) = \Big(\left(\Xi^{(t+1)}\right)^{\top} \boldsymbol{g} \circ (\boldsymbol{J}-\boldsymbol{X_1}) \circ \cdots \circ (\boldsymbol{J}-\boldsymbol{X_p})\ \left(\Xi^{(t+1)}\right)\Big) \oslash \\ \Big(\left(\Xi^{(t+1)}\right)^{\top} (\boldsymbol{J}-\boldsymbol{X_1}) \circ \cdots \circ (\boldsymbol{J}-\boldsymbol{X_p}) \left(\Xi^{(t+1)}\right)\Big),\end{aligned} \tag{23}$$

where $\boldsymbol{A}\circ\boldsymbol{B}$ denotes the Hadamard (i.e., entry-wise) product of the conformable matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, $\boldsymbol{J}$ is a $N\times N$ matrix whose off-diagonal entries are all one and whose diagonals are all zero. We follow the approach Dahbura et al. (2021) to calculate (23) without breaking the sparsity of the covariate matrices and $\boldsymbol{g}$. We can then set

$$\mathbf{\Pi_0}\left(\Xi^{(t)}\right) = \log\left(1-\boldsymbol{\pi}^{(t+1)}(1,\mathbf{0})\right)$$

and calculate $\Omega_{ik}^{(t)}\left(\mathbf{0},\mathbf{0},\Xi^{(t)}\right)$ for $i=1,\ldots,N$ and $k=1,\ldots,K$.

Next, we correct for the error arising from assuming that $\boldsymbol{g}=\mathbf{0}$ and $\boldsymbol{x}=\mathbf{0}$ holds. Since all covariates are assumed to be categorical, the pairwise covariate vector $\boldsymbol{x}_{ij}$ can have at most 2^p possible outcomes, which we collect in the set $\mathscr{X}$. Given this, one may rewrite the sum over $j\neq i$ as a sum over all possible outcomes of the pairwise covariate information:

$$\begin{aligned}\Lambda_{ik}\left(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)}\right) &= \sum_{j\neq i}^{N}\sum_{l=1}^{K}\left(g_{ij}\xi_{jl}^{(t)}\left(\log\frac{\pi_{kl}^{(t)}(1,\boldsymbol{x}_{ij})}{\pi_{kl}^{(t)}(0,\mathbf{0})}\right)+g_{ji}\xi_{jl}^{(t)}\left(\log\frac{\pi_{lk}^{(t)}(1,\boldsymbol{x}_{ji})}{\pi_{lk}^{(t)}(0,\mathbf{0})}\right)\right)\\
&+\sum_{j\neq i}^{N}\sum_{l=1}^{K}\left((1-g_{ij})\xi_{jl}^{(t)}\left(\log\frac{\pi_{kl}^{(t)}(0,\boldsymbol{x}_{ij})}{\pi_{kl}^{(t)}(0,\mathbf{0})}\right)+(1-g_{ji})\xi_{jl}^{(t)}\left(\log\frac{\pi_{lk}^{(t)}(0,\boldsymbol{x}_{ji})}{\pi_{lk}^{(t)}(0,\mathbf{0})}\right)\right)\\
&=\sum_{\boldsymbol{x}\in\mathscr{X}}\sum_{j\neq i}^{N}\sum_{l=1}^{K}g_{ij}\mathbb{I}(\boldsymbol{x}_{ij}=\boldsymbol{x})\xi_{jl}^{(t)}\left(\log\frac{\pi_{kl}^{(t)}(1,\boldsymbol{x})}{\pi_{kl}^{(t)}(0,\mathbf{0})}\right)\\
&+g_{ji}\mathbb{I}(\boldsymbol{x}_{ji}=\boldsymbol{x})\xi_{jl}^{(t)}\left(\log\frac{\pi_{lk}^{(t)}(1,\boldsymbol{x})}{\pi_{lk}^{(t)}(0,\mathbf{0})}\right)+(1-g_{ij})\mathbb{I}(\boldsymbol{x}_{ij}=\boldsymbol{x})\,\xi_{jl}^{(t)}\left(\log\frac{\pi_{kl}^{(t)}(0,\boldsymbol{x})}{\pi_{kl}^{(t)}(0,\mathbf{0})}\right)\\
&+1-g_{ji})\mathbb{I}(\boldsymbol{x}_{ji}=\boldsymbol{x})\,\xi_{jl}^{(t)}\left(\log\frac{\pi_{lk}^{(t)}(0,\boldsymbol{x})}{\pi_{lk}^{(t)}(0,\mathbf{0})}\right)\end{aligned}$$

With

$$\mathbf{\Pi}\left(d,\boldsymbol{x},\Xi^{(t)}\right) \coloneqq \begin{bmatrix} \log\frac{\pi_{11}^{(t)}(d,\boldsymbol{x})}{\pi_{11}^{(t)}(d,\mathbf{0})} & \log\frac{\pi_{12}^{(t)}(d,\boldsymbol{x})}{\pi_{12}^{(t)}(d,\mathbf{0})} & \dots & \log\frac{\pi_{1K}^{(t)}(d,\boldsymbol{x})}{\pi_{1K}^{(t)}(d,\mathbf{0})} \\ \log\frac{\pi_{21}^{(t)}(d,\boldsymbol{x})}{\pi_{21}^{(t)}(d,\mathbf{0})} & \log\frac{\pi_{22}^{(t)}(d,\boldsymbol{x})}{\pi_{22}^{(t)}(d,\mathbf{0})} & \dots & \log\frac{\pi_{2K}^{(t)}(d,\boldsymbol{x})}{\pi_{2K}^{(t)}(d,\mathbf{0})} \\ \vdots & \vdots & \ddots & \vdots \\ \log\frac{\pi_{K1}^{(t)}(d,\boldsymbol{x})}{\pi_{K1}^{(t)}(d,\mathbf{0})} & \log\frac{\pi_{K2}^{(t)}(d,\boldsymbol{x})}{\pi_{K2}^{(t)}(d,\mathbf{0})} & \dots & \log\frac{\pi_{KK}^{(t)}(d,\boldsymbol{x})}{\pi_{KK}^{(t)}(d,\mathbf{0})} \end{bmatrix},$$

$$\mathbf{\Gamma}(d) \coloneqq \begin{cases} \boldsymbol{g} & (d=1) \\ \boldsymbol{J}-\boldsymbol{g} & (d=0) \end{cases},$$

and

$$\mathbf{\Delta}(\boldsymbol{x}) \coloneqq \boldsymbol{X}_{\mathbf{1}}^{x_1} \circ (\boldsymbol{J}-\boldsymbol{X}_{\mathbf{1}})^{1-x_1} \circ \cdots \circ \boldsymbol{X}_{\boldsymbol{p}}^{x_p} \circ (\boldsymbol{J}-\boldsymbol{X}_{\boldsymbol{p}})^{1-x_p},$$

we can write

$$\Lambda_{ik}\left(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)}\right) = \left(\sum_{\boldsymbol{x}\in\mathscr{X}}\sum_{d\in\{0,1\}} \mathbf{\Gamma}(d)\circ\mathbf{\Delta}(\boldsymbol{x})\,\Xi^{(t)}\,\mathbf{\Pi}^{(t)}\left(d,\boldsymbol{x},\Xi^{(t)}\right) + \left(\mathbf{\Gamma}(d)\circ\mathbf{\Delta}(\boldsymbol{x})\right)^{\top}\Xi^{(t)}\,\mathbf{\Pi}^{(t)}\left(d,\boldsymbol{x},\Xi^{(t)}\right)^{\top}\right)_{ik}.$$

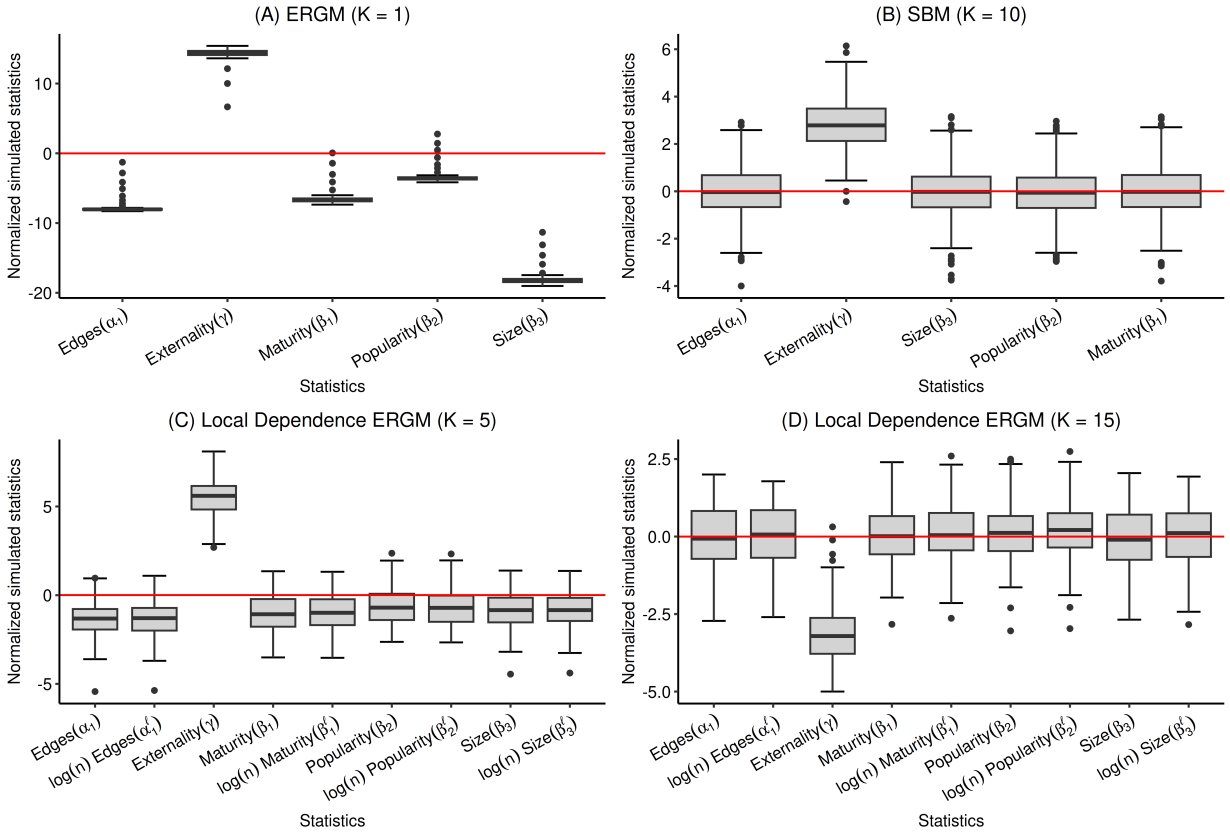
Extending (23) to generic covariate values $\boldsymbol{x}$ yields

$$\begin{aligned}\boldsymbol{\pi}^{(t+1)}(1,\boldsymbol{x}) = &\left(\left(\Xi^{(t+1)}\right)^{\top}\boldsymbol{g}\circ\boldsymbol{X}_{\mathbf{1}}^{x_1}\circ(\boldsymbol{J}-\boldsymbol{X}_{\mathbf{1}})^{1-x_1}\circ\cdots\circ\boldsymbol{X}_{\boldsymbol{p}}^{x_p}\circ(\boldsymbol{J}-\boldsymbol{X}_{\boldsymbol{p}})^{1-x_p}\left(\Xi^{(t+1)}\right)\right)\oslash \\ &\left(\left(\Xi^{(t+1)}\right)^{\top}\boldsymbol{X}_{\mathbf{1}}^{x_1}\circ(\boldsymbol{J}-\boldsymbol{X}_{\mathbf{1}})^{1-x_1}\circ\cdots\circ\boldsymbol{X}_{\boldsymbol{p}}^{x_p}\circ(\boldsymbol{J}-\boldsymbol{X}_{\boldsymbol{p}})^{1-x_p}\left(\Xi^{(t+1)}\right)\right),\end{aligned}$$

enabling the evaluation of $\mathbf{\Pi}\left(d,\boldsymbol{x},\Xi^{(t)}\right)$ for $\boldsymbol{x}\in\mathscr{X}$. Following Dahbura et al. (2021), we can still evaluate this matrix by sparse matrix operations.

In sum, we have shown how to compute the terms $\Omega_{ik}\left(\mathbf{0},\mathbf{0},\Xi^{(t)}\right)$ and $\Lambda_{ik}\left(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)}\right)$ through matrix multiplications. Plugging in these terms into (20) enables a fast computation of $\Omega_{ik}(\boldsymbol{g},\boldsymbol{x},\Xi^{(t)})$, which is the computational bottleneck for evaluating the quadratic term needed to update $\Xi^{(t)}$ to $\Xi^{(t+1)}$.

Figure A1: Comparison of model fits of network statistics of competing models.



Each boxplot is based on 100 simulations of the network generated using the estimated parameters from the corresponding model. Simulations are initialized at the observed network and proceed with a burn-in period of 500,000 Metropolis–Hastings steps. After burn-in, one network is retained every 50,000 steps. For each retained network, the corresponding network statistics are computed and then standardized such that a value of zero corresponds to the statistic of the observed network. On the x-axis, one boxplot is displayed for each network statistic associated with a coefficient the corresponding fitted model.

B Additional figures and estimates: Comparison with other models

In Figure A1 (A), we show the simulated statistics generated by a model with $K = 1$, corresponding to an Exponential Random Graph Model (ERGM). In aggregate, this model performs very poorly, which suggests that the unobserved heterogeneity helps the model fit.

We compare our model with $K = 10$ unobserved types to a stochastic blockmodel with $K = 10$ unobserved types in Figure A1 (B). In essence, the comparison model is the same as our model without the externality. As expected, the stochastic blockmodel (SBM) performs worse in predicting the externality within blocks.

We also performed robustness checks to choose the number of types K. In Figures A1 (C) and (D), we report the boxplot for the fit of models with $K = 5$ and $K = 15$, respectively.

We conclude from the plots that increasing or decreasing the number of types does not seem to help in representing the externality. We report the estimates of all these competing models in Table A1.

Table A1: Parameter estimates and standard errors.

	ERGM (K = 1)		SBM (K = 10)	
	Within Estimate (Std. Error)	Between Estimate (Std. Error)	Within Estimate (Std. Error)	Between Estimate (Std. Error)
Edges (α_1)	−7.961		−5.703	-7.979
	(0.007)		(0.274)	(0.006)
$\log(n)\times$ Edges (α_2)			−0.549	
			(0.031)	
Externality (γ)	−0.178			
	(0.002)			
Maturity (β_1)	−0.131		1.767	-0.222
	(0.01)		(0.303)	(0.011)
$\log(n)\times$ Maturity $\left(\beta_1^{\ell}\right)$			−0.12	
			(0.035)	
Popularity (β_2)	−0.066		1.071	-0.07
	(0.01)		(0.308)	(0.011)
$\log(n)\times$Popularity $\left(\beta_2^{\ell}\right)$			−0.117	
			(0.035)	
Size (β_3)	0.127		1.92	0.118
	(0.01)		(0.303)	(0.01)
$\log(n)\times$Size $\left(\beta_3^{\ell}\right)$			−0.178	
			(0.035)	

Table A1 (ctd.)

	Local Dep. ERGM (K = 5)		Local Dep. ERGM (K = 15)	
	Within Estimate (Std. Error)	Between Estimate (Std. Error)	Within Estimate (Std. Error)	Between Estimate (Std. Error)
Edges (α_1)	−4.893	-7.45	−4.181	-8.28
	(0.263)	(0.006)	(0.305)	(0.006)
$\log(n)\times$ Edges (α_2)	−0.569		−0.765	
	(0.028)		(0.038)	
Externality (γ)	−0.258		0.135	
	(0.013)		(0.013)	
Maturity (β_1)	0.711	-0.267	3.201	-0.198
	(0.302)	(0.012)	(0.321)	(0.011)
$\log(n)\times$ Maturity $\left(\beta_1^{\ell}\right)$	−0.012		−0.307	
	(0.033)		(0.041)	
Popularity (β_2)	1.324	-0.073	0.369	-0.074
	(0.307)	(0.011)	(0.29)	(0.01)
$\log(n)\times$Popularity $\left(\beta_2^{\ell}\right)$	−0.142		−0.048	
	(0.033)		(0.037)	
Size (β_3)	1.316	0.104	1.58	0.117
	(0.31)	(0.011)	(0.287)	(0.01)
$\log(n)\times$Size $\left(\beta_3^{\ell}\right)$	−0.111		−0.15	
	(0.034)		(0.037)	

Estimates and standard errors are obtained by Maximum Pseudolikelihood (MPLE), conditioning on the estimated types in the first step. Standard error for the estimates are in parenthesis.