Research Article

COLLECTIVE INTELLIGENCE

Collective Intelligence
Volume 3:1: 1–16

DOI: 10.1177/26339137241231912
journals.sagepub.com/home/col
Sage

Modeling interconnected social and technical risks in open source software ecosystems

William Schueller[1,2] **and Johannes Wachs**[1,3,4]

[1]Complexity Science Hub Vienna, Vienna, Austria

[2]Veterinary Public Health and Epidemiology, University of Veterinary Medicine, Vienna, Austria

[3]Institute of Data Analytics and Information Systems, Corvinus University of Budapest, Budapest, Hungary

[4]HUN-REN Centre for Economic and Regional Studies, Budapest, Hungary

Abstract

Open source software ecosystems consist of thousands of interdependent libraries, which users can combine to great effect. Recent work has pointed out two kinds of risks in these systems: that technical problems like bugs and vulnerabilities can spread through dependency links, and that relatively few developers are responsible for maintaining even the most widely used libraries. However, a more holistic diagnosis of systemic risk in software ecosystem should consider how these social and technical sources of risk interact and amplify one another. Motivated by the observation that the same individuals maintain several libraries within dependency networks, we present a methodological framework to measure risk in software ecosystems as a function of both dependencies and developers. In our models, a library's chance of failure increases as its developers leave and as its upstream dependencies fail. We apply our method to data from the Rust ecosystem, highlighting several systemically important libraries that are overlooked when only considering technical dependencies. We compare potential interventions, seeking better ways to deploy limited developer resources with a view to improving overall ecosystem health and software supply chain resilience.

Keywords

Open source software, decentralized collaboration, systemic risk, networks, social-technical systems

Introduction

Open source software (OSS) ecosystems are built by decentralized collaborations of thousands of software developers. Developers write specialized libraries by relying on the work of others, growing a complex network of dependencies. The result is a distribution of work and effort that has been shown to create immense value (Blind and Schubert 2023; Greenstein and Nagle 2014). Two key risks threaten the functionality of the system as a whole: the propagation of bugs and vulnerabilities through the ever-growing network of dependencies (Decan et al., 2019), and its reliance on small groups or even individuals who fix such problems in widely used libraries (Avelino et al., 2016; Eghbal 2020; Pfeiffer 2021). Yet risk assessments of OSS ecosystems have thus far largely neglected the fact that these

Corresponding author:
Johannes Wachs, Corvinus University of Budapest, Fovam Ter 8, Budapest 1093, Hungary.
Email: johannes.wachs@uni-corvinus.hu

Data Availability Statement included at the end of the article.

risks are interrelated: key individuals often maintain several libraries across the network of dependencies that make up a software ecosystem. When such individuals leave the ecosystem, multiple libraries may go unmaintained and become unchecked potential sources (Lehman 1980; Valiev et al., 2018) or conduits (Decan et al., 2018; Ohm et al., 2020) for issues in the dependency network. Efforts to quantify systemic risk—the risk that problems in a specific library can impact the functioning of the system as a whole—in such ecosystems should consider this correlation.

Indeed while these two perspectives both highlight important problems, significant systemic risks in OSS ecosystem emerge through the complex *interaction* of their social and technical systems. The risks of ever-expanding dependency networks are amplified when individuals are contributors to several libraries in a dependency chain. In Figure 1, we plot the dependencies among the 100 most downloaded libraries in the Rust ecosystem. Individual developers are the most prolific contributors to multiple libraries and their departures would introduce a correlated shock impacting the future functionality of the system. The aim of our work is to model the systemic importance of *all* developers and libraries that takes these correlations into account.

In particular, we adapt methods used to study the propagation of errors in complex systems to the case of open source software ecosystems. We use a simulation approach to quantify systemic risk and apply it to data from the Rust ecosystem. We simulate the removal of developers from the system, which induces potential failures in libraries that they maintain, which in turn spread with some probability to downstream dependencies. The likelihood that a library fails, governed by a *production function*, increases as its developers leave and as its upstream dependencies fail. We define an iterative equation to calculate the spread of issues resulting from the departure of specific developers from the system. At the system level, we find the significant potential for long cascades of failures when specific individuals leave. We also highlight the libraries that play a major role in many potential cascade paths, representing natural intervention targets.

Our method highlights key libraries that these purely technical dependency-based measures overlook: among the top 1000 Rust libraries by count of their downstream dependencies, our measure of library systemic importance is moderately correlated with its direct (Spearman's $\rho \approx .56$) and transitive dependencies (Spearman's $\rho \approx .54$). Our measure has an even weaker correlation with the number of GitHub stars a library has (Spearman's $\rho \approx .42$), suggesting that social visibility is a poor proxy for systemic importance. Indeed, in an intervention study we show that allocating developers to the most systemically risky libraries improves system robustness more than by adding developers to libraries central in the dependency network or to highly visible libraries. Our measure provides valuable insights for individuals, foundations, and firms who wish to support OSS ecosystem stability (Overney et al., 2020; Spaeth et al., 2015).

Background

We review the measurement of systemic risks and cascades in complex systems. We then turn to the specific case of such risks in software ecosystems, discussing both social and technical factors.

Resilience and vulnerability of complex systems

The field of complexity science has long studied the vulnerability of interconnected systems. Studies of cascading failures in financial networks (Haldane and May 2011; Thurner and Poledna 2013), supply chains (Diem et al., 2022), power distribution networks (Kinney et al., 2005), regional economies (Tóth et al., 2022), and healthcare

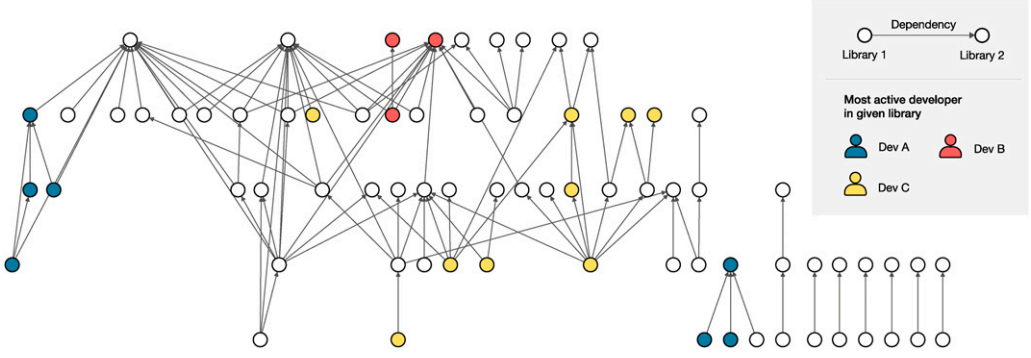

Figure 1. The dependency network among the 100 most downloaded libraries in the Rust ecosystem, observed July 2020. A directed edge between two libraries indicates a dependency. Distinct colors highlight three groups of libraries that have the same developer making the most commits in the previous year. For example, Developer B makes the most commits in each of the three red libraries. Across all 100 libraries, there are 59 unique most active developers, indicating that key individuals often play an important role in multiple interdependent libraries.
Note: 24 disconnected libraries are not shown.

systems (Sardo et al., 2019) all highlight that a few key nodes in a system can play a systemically important role that is not obvious from their local topology or individual size. The spread of errors in coupled networks is even less predictable (Poledna et al., 2015; Schneider et al., 2013). Previous descriptive work in the software engineering research community has pointed out the potential application of complex systems approaches to the study of software systems (Decan et al., 2019; Mens and Grosjean 2015).

Studies of complex systems simulate the spread of failures from specific sources to quantify overall systemic risk and the importance of individual entities in the system (Peters et al., 2008). In these simulations, the functions governing how errors spread are tailored to the specific situation. For instance, recent work on the resilience of supply chains has used production functions such as the Cobb-Douglas and Leontief functions to model the effect of upstream failures on a node's production (Diem et al., 2022). The choice of a specific production function captures whether inputs are complements or substitutes. In the software context, the effort of maintainers and the functionality of upstream dependencies are complementary. The initial condition of the error or failure is also important: contagion in financial networks often begins with the default of a large loan or a bankruptcy. In the case of any specific application, care must be taken to understand the mechanisms of how errors spread.

Vulnerabilities of open source software ecosystems

Technical vulnerabilities. Bugs and vulnerabilities spread through software ecosystems via dependencies (Valiev et al., 2018). A 2018 study estimates that half of libraries in the NPM ecosystem are affected by upstream vulnerabilities (Decan et al., 2018). In 2016, a developer of an auxiliary string formatting program called *left-pad* removed his libraries from NPM, a package manager for JavaScript libraries, and caused a cascade of failures that lead to large scale service interruptions around the web (Hejderup et al., 2018). A significant share of the web's infrastructure depended, often indirectly, on *left-pad's* 11 lines of code. Though *left-pad* itself is a "trivial package"—its removal caused its downstream dependencies, many of which were widely-used, to fail.

Upstream issues can also occur in more substantial pieces of software. Many of these systemically important libraries are overlooked because they have worked well in the background for many years. The Heartbleed bug, introduced into the widely used OpenSSL cryptography library in 2012, made roughly half a million webservers and their user passwords and cookies vulnerable to attack (Durumeric et al., 2014). Though the bug was quickly resolved, servers remained vulnerable until patched.

Software systems are also frequently attacked via loopholes introduced by upstream dependencies (Ohm et al., 2020). In 2017, intruders exploited a vulnerability to access an Equifax database, exposing personal finance data of over one hundred million people. Equifax used an outdated version of Apache Struts 2, a web application framework which had a publicly known (and patched) security vulnerability (Luszcz 2018). Another example of software that is widely relied upon is *log4j*, a "ubiquitous" Java logging library (Newman 2021). In late 2021, a zero-day vulnerability in *log4j* was reported which could be used to take control of software systems remotely. Security researchers have recently demonstrated how dependency managers themselves can be used to introduce malicious lines into the codebases of leading software companies (Birsan 2021). The share of libraries in NPM inheriting vulnerabilities from upstream dependencies is rising over time (Zerouali et al., 2021). Tracking vulnerabilities is also a significant challenge for developers and companies (Pashchenko et al., 2018), and they often persist in ecosystems (Alfadel et al., 2021), for instance when patches are not adopted by downstream dependencies (Decan et al., 2022).

These risks in the "supply chain" of OSS are increasingly recognized and quantified in the empirical software engineering literature (Amreen et al., 2019; Ma 2018). Many upstream dependencies are small libraries in the mold of *left-pad* (Abdalkareem et al., 2017). These so called "trivial" libraries are ironically more likely to occupy critical positions in the dependency network, owing to their widespread use (Chowdhury et al., 2022). This highlights the importance of considering the network as a whole, rather than focusing on important seeming libraries. For instance, having unmaintained upstreams increases the risk that a library will itself be abandoned (Valiev et al., 2018).

Social roots of ecosystem vulnerability. Early advocates for the OSS model of software development argued that small contributions of many developers would lead to high quality software (Raymond 1999). And although the decentralized peer production process has resulted in remarkably successful software (Benkler et al., 2015), the reliance on volunteers and unpaid labor to maintain such widely used software often leads to an *underproduction* of OSS. In this context underproduction, coined by Champion and Mako Hill in their study of the Debian ecosystem, refers to a mismatch between the supply of software development labor and demand of people relying on a particular software library (Champion and Hill 2021).

The extent to which software relies on individual developers has been conceptualized as the *truck factor* or *bus factor* of a library (Torchiano et al., 2011; Williams and Kessler 2003). It has been described as "the number of developers on a team who have to be hit with a truck (i.e., to

go on vacation, to become ill, or to leave the company for another) before the project is in serious trouble" (https://www.agileadvice.com/archives/2005/05/truck). In other words, the truck factor describes the distribution and redundancy of essential knowledge and know-how about a specific software library or project among its developers. A hypothetical library with a truck factor of one relies in some essential way on the contributions, efforts, and knowledge of a single individual. Empirical studies have shown that truck factors of even widely used libraries are often very low (Ferreira et al., 2019; Pfeiffer, 2021). One study of 133 popular projects on GitHub found that nearly two-thirds had a truck factor of two or less (Avelino et al., 2016). In practice, libraries are often abandoned because their original core developers and maintainers lack the time or interest to continue working on them (Coelho and Valente, 2017).

When libraries go under-maintained or become deprecated, issues tend to build up. It is one of Lehman's Laws that software quickly becomes ineffective or nonfunctional without maintenance (Lehman 1980). In practice, the same individuals who write the original code of a program are the ones who maintain it, a task which often requires quick interventions when something goes wrong (Cook 2020). Unmaintained libraries are not adapted to changes of the broader ecosystem. When upstream libraries introduce breaking changes, a deprecated library will cease to function as expected and can pass issues downstream. This is not just a theoretical concern: over half of NPM libraries depend transitively on at least one deprecated library (Cogo et al., 2022).

Previous work diagnosing the health of ecosystems has not directly addressed the phenomenon of developers working on multiple libraries within an ecosystem. Such developers can make highly valuable contributions, for example, because they facilitate coordination and communication between interlinking parts of a larger system (Herbsleb and Grinter, 1999) or because they are uniquely placed to anticipate failures or issues (Cataldo and Herbsleb, 2013). Although their attentions may be divided (Vasilescu et al., 2016), developers involved in multiple part of an ecosystem are in a position to better consider how new developments in one library may affect others. At the same time, these same aspects make such developers essential to the system as a whole. When such a developer leaves the OSS world, whether it is because they find a new job and no longer have the time, or because they retire, or simply because they no longer want to participate, they may leave several key libraries under- or unmaintained at the same time.

Data

We now turn to the data we use to build and test our systemic risk measurement framework. We use data from the Rust ecosystem, utilizing a dynamic database of dependencies and contributions to Rust libraries assembled by Schueller et al. (Schueller et al., 2022). Rust is a relatively young but growing programming language, which was recently adopted as the second official language of the Linux kernel project. We chose Rust for several reasons. First, the Rust dependency manager Cargo stores data on the evolution of dependencies between libraries overtime. It also has data on the number of downloads over time, allowing us to test the impact of hypothetical failures on end users. Second, a large majority of Rust libraries available on Cargo are hosted on GitHub or Gitlab (over 80%, increasing to over 90% for libraries downloaded at least 10,000 times (Schueller et al., 2022)), likely because of the language's youth relative to these platforms and its community's strong OSS orientation, allowing us to download nearly all libraries and their complete development histories. Finally, as a growing ecosystem, Rust allows us to track the evolution of systemic risk across its life-course.

In short, the Rust ecosystem offers, to the best of our knowledge, the most complete data on contributions, dependencies, and outcomes of any OSS ecosystem. We note that while other ecosystems may not have the same quality and scope of data, our framework is modular and can be adapted to different datasets.

Package metadata, repositories, dependency network

The core of the dataset is derived from a database dump from Cargo (available on https://crates.io/data-access) containing extensive metadata about packages. The dataset includes package names, URLs, versions, dependencies, creation date, and daily downloads. URLs can be linked to valid repository (repo) URLs on GitHub, Gitlab, and other platforms. It also provides details on dependencies between packages, enabling the construction of a dependency network at any point in time, which is important as libraries add and remove dependencies on a regular basis. The database discards information about versions by considering only the dependencies of the latest version of a library—recognizing that version conflicts are a major way in which libraries break because of undermaintenance (Decan et al., 2019; Wang et al., 2020).

Developer contributions

We consider commits as the elemental contributions that developers make to projects. Though we acknowledge that other forms of contributions such as issue reporting represent valuable contributions to OSS projects and ecosystems as a whole (Trinkenreich et al., 2020), solving problems created by changes in upstream dependencies, for

example, typically requires committing code. The dataset we employ disambiguates user accounts and removes bot accounts (Golzadeh et al., 2021; Schueller et al., 2022). We associate developers to libraries, weighing their contributions by their share of the total commits made. In the analysis carried out in the rest of the paper we fix the scope of the dataset: we consider the dependency network between Rust repos as observed on January 1, 2022. We consider contributions (commits) made to repos from January 1, 2021 to January 1, 2022. To emphasize this point of flexibility, we refer to *libraries* rather than repos or packages in the subsequent sections.

Methods and analysis

Modeling library functionality via production functions

The key insight our paper brings from the complex systems literature is that when systems have rich interdependencies, small changes in seemingly unimportant parts of a system can have an outsized effect on the functioning of the whole (Peters et al., 2008). To carry out this kind of analysis in the context of spreading failures in the Rust ecosystem, we will now describe how to quantify a library's functionality in terms of social and technical inputs.

A library i requires both functioning upstream dependencies and active contributors to continue to function properly. A developer stopping to contribute, or a dependency missing, compromised or exposed to bugs will expose the library itself to an increased risk. We assume that both sources of risk can be combined into one quantity: a probability of *failure* $0 \leq F(i) \leq 1$. In our context, failure is an abstraction of the issues, bugs, and vulnerabilities that a library can spread to its downstream dependencies.

Risk is minimized when a library has active maintainers and functioning dependencies. Risk is highest when all developers having stopped maintenance work on the library and/or all upstream dependencies have failed. To combine the two sources of risk, we adapt the notion of a *production function* from the economics literature (Brown 1957). Production functions are used in a variety of contexts to describe how inputs are combined to generate outputs. A traditional example is how capital and labor combine to create goods in an economy. The chosen functional form governs how the inputs interact with one another. For instance, two inputs may substitute for or complement one another. In one extreme case, one or more inputs may be essential to production.

In our case, we argue that maintaining developers and functioning upstream dependencies are both required for a library to continue to work. These two inputs to software maintenance can only substitute for each other in a limited way. This perspective aggregates and necessarily simplifies several sources of risks, but provides a flexible framework to consider the impact of both social and technical vulnerabilities. Specifically we use a Cobb-Douglas style production function (Brown 1957) P_i, which considers the product of the shares of functioning its upstream dependencies (d_i) and active contributors (c_i)[1]

$$F_i = 1 - P_i = 1 - \left(c_i^{1/2} \star d_i^{1/2}\right)$$

For example, a library or package with one half of its contributors available, and two-thirds of its upstream dependencies functioning, will have a roughly 43% ($1 - ((1/2)^{1/2} \star (2/3)^{1/2})$) risk exposure, that is, its chance of failing. We selected the Cobb-Douglas production function to model the spread of risk because it suggests that contributors and upstream health are complements and imperfect substitutes, and that libraries can fail if either is missing. Indeed the Cobb-Douglas functional form is often used to model or estimate the relative contributions of labor and capital to output in a firm or industry (Brown 1957). The relative weighting of dependencies and maintainers can be adjusted via the exponents. Other production functions such as the Leontief production function ($1 - \min(c_i, d_i)$), in which inputs cannot be substituted at all, or a linear production function ($1 - c_i + d_i/2$), in which inputs are perfect substitutes, present alternatives. In this sense, our framework is flexible and can diagnose systemic risk with respect to different kinds of issues. We plot the Cobb-Douglas function predicted chance of library failure in Figure 2.

Spread of failures

In order to model the spread of library failures in the ecosystem, we need represent two kinds of relationships. The first kind consists of maintenance activity by developers in specific libraries. We store this information as a matrix* C, in which the entry $C_{i,j}$ counts the number of commits made by developer i to library j. We normalize the columns of this matrix and obtain $\widehat{C}$, in which the entry $\widehat{C}_{i,j}$ can be interpreted as the share of contributions to library j made by developer i.

The second kind of relationships within the ecosystem we consider are the dependencies between libraries. We store this information in a matrix D. D is a square matrix with rows and columns equal to the number of dependencies. Entry $D_{i,j}$ is equal to 1 if library j depends on library i, and is 0 otherwise. Similar to the previous case, we normalize the columns of this matrix to obtain $\widehat{D}$, in which the entry $\widehat{D}_{i,j}$ can be interpreted as the share of dependency of library j on library i.

We now define two vectors that track the state of the system. S^C is a vector corresponding to the contributors in the ecosystem. The i-th entry of S^C is 1 if contributor i is

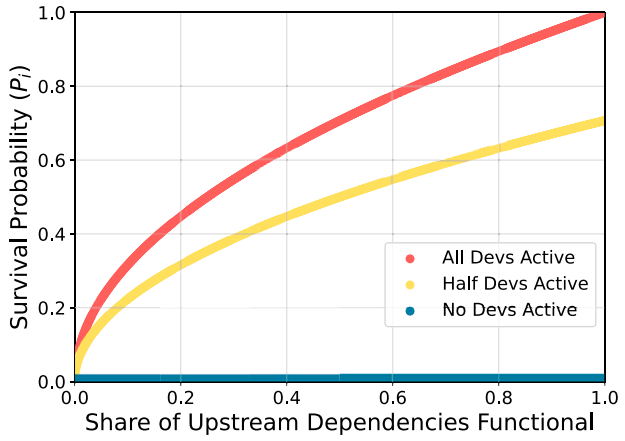

Figure 2. The chance of library failure in terms of the shares of inactive developers and failed upstream dependencies, as quantified using a Cobb-Douglas production function. A library with all of its original developers still active, and 60% of its direct upstream dependencies functional has a roughly 80% chance to survive.

active, otherwise 0. As our analysis deals with the potential consequences of contributors leaving the ecosystem, this vector will be an input to our scenarios.

The second vector S^L tracks the state of each library. That is to say it defines the likelihood that a library will fail, given the status of its upstream dependencies and contributors.

When all libraries are fully functioning, every coordinate of S^L is equal to 1. Those coordinates correspond conceptually to $1 - F_i$ as defined in the previous subsection, and depending on the imposed conditions—for example, some developers missing—can take values between 0 and 1, 0 being the highest possible level of risk exposure.

Our scenarios consider what happens to the libraries after a contributor leaves the ecosystem. The departure of a developer triggers potential issues: either directly on those libraries to which she contributes, or indirectly, on those libraries which depend on libraries she maintains, as suggested by Lehman's Law (Lehman 1980). This information is captured by the following self-referential equation

$$S^L = \left(S^{C\intercal} \star \widehat{C}\right)^{1/2} \odot \left(S^{L\intercal} \star \widehat{D}\right)^{1/2}$$

In this equation $\odot$ denotes element-wise matrix multiplication. Likewise, the exponents are to be taken element-wise. $\intercal$ denotes the transpose of the vectors. In plain terms, the left factor corresponds to the effect of absent contributors on the states of the libraries, while the right factor corresponds to the effect of potential malfunctioning upstream dependencies. These effects are combined by the Cobb-Douglas style production function.

In our application, $\widehat{C}$, $\widehat{D}$, and S^C are fixed and we can represent the equation in the following form $S^L = f(S^L)$, emphasizing that the library state vector is updating. To find the solution of this equation, we iterate from the initial state in which all libraries are functioning, which we denote S_0^L

$$S_0^L = \mathbf{1}$$

$$S_{n+1}^L = f\left(S_n^L\right)$$

This sequence converges towards a stable solution in a finite number of steps, which we denote $S^L = S_{Fin}^L$.

We now describe the specific calculations we make. We proceed by initializing all entries of the library state vector S^L to 1. We then remove a developer from the system, changing a single entry of the contributor state vector S^C from 1 to 0. This allows us to calculate a step of the spread or diffusion of issues. Our practical implementation includes two small corrections to the equation to handle edge cases. When a library has no dependencies, its entry on the right factor is set to one at the end of every step in the iteration. When a library has no contributors the left factor is similarly hardcoded to one. In the next step, we observe that the left factor is unchanged—we do not remove additional developers—and that the right factor is simply the result of the calculation carried out in the previous step. We repeat this calculation several times, until the state vector of S^L is unchanged. As the dependency network has no cycles (i.e., is a directed acyclic graph), the process always converges. In practice, this happens in a relatively small number of steps—approximately 20 in our application—as the depth of the dependency network is small relative to the size of the system as a whole.

We now carry out one step of an example diffusion on a toy ecosystem. We visualize this example in Figure 3. In this ecosystem, there are four libraries maintained by three developers. The normalized contributors' matrix is

$$\widehat{C} = \begin{bmatrix} 0 & .5 & 0 & 0 \\ 1 & .25 & 0 & 0 \\ 0 & .25 & 1 & 1 \end{bmatrix}$$

Column 2 indicates that contributor 1 is responsible for half of the contributions to library 2, while contributors 2 and 3 are responsible for one quarter each. Contributor 2 is the sole contributor to library 1. The normalized dependency matrix, on the other hand is a square 4 × 4 matrix:

$$\widehat{D} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & .5 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & .5 & 0 \end{bmatrix}$$

This matrix indicates that library 1 has no upstream dependencies (column 1), while libraries 2 and 4 depends

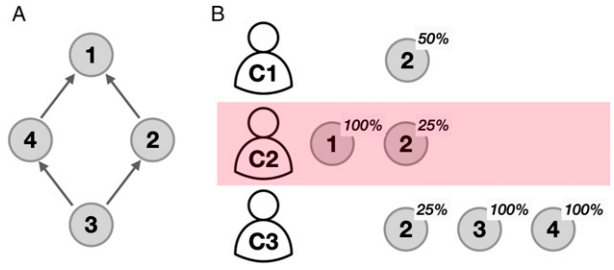

Figure 3. An example ecosystem. (a) The network of dependencies between four libraries. (b) The three contributors to the libraries. Percentages denote what share of contributions they make to a specific library. For instance, contributor 1 makes 50% of all contributions to library 2. In our example calculation, we simulate the consequences of the departure of contributor 2 from the system.

solely on library 1. Library 3 (column 3) depends on both libraries 2 and 4. We calculate what happens according to our method if contributor 2 is removed from the system, as indicated in Figure 3. We obtain the following equation:

$$S_1^L = \left(\begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \star \widehat{C} \right)^{1/2} \odot \left(\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \star \widehat{D} \right)^{1/2}$$

Carrying out the matrix multiplications and the exponents (element-wise) of the two factors yields

$$S_1^L = \begin{bmatrix} 0 \\ \sqrt{\frac{3}{4}} \\ 1 \\ 1 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \sqrt{\frac{3}{4}} \\ 1 \\ 1 \end{bmatrix}$$

In the next step of the calculation, the left factor would be unchanged, but the right factor would be changed, reflecting the spread of the probability of failure from libraries 1 and 2 (recall that the removed contributor 2 contributed to both). As the reader can verify, the removal of contributor 2 leads to a chain reaction in which the functionality of all libraries in the ecosystem is affected. Specifically the next step of the iteration yields

$$S_2^L = \begin{bmatrix} 0 \\ 0 \\ \sqrt{\frac{\sqrt{3}+2}{4}} \\ 0 \end{bmatrix}$$

The next iteration after that results in a library state vector of all zeros: $S_3^L = S_{Fin}^L = \overrightarrow{0}$. In other words, the removal of the single developer has lead to a cascade of failures impacting all libraries in the toy system.

Ranking contributors, libraries, and the ecosystem as a whole

With this method to model the spread of failures resulting from the removal of individual contributors, we proceed to test the robustness of the whole Rust ecosystem, aiming to rank contributors and libraries for their systemic importance. To do so, we simply repeat the iterated calculation above for every contributor in the ecosystem. That is, we remove each contributor, alone, a single time. Note that more complex removals are possible—the diffusion equation is flexible and can accommodate any valid input contributor state vector. For example, one could remove all contributors supported by a specific corporation or foundation.

For each contributor we remove from the system, our calculations yield a final library state vector S_{Fin}^L. These vectors represent the functionality of the libraries following the cascade induced by a specific contributor's departure. We define the risk R_i to library i as the difference

$$R_i = 1 - S_{\text{Fin}}^L(i)$$

As not all libraries are equally important to the ecosystem as a whole, we weigh these entries by the share of downloads the specific library has of all total downloads. Specifically the download-weighted risk of a simulated removal to library i is defined as

$$R_i^{dl} = \left(1 - S_{\text{Fin}}^L(i)\right) \cdot \frac{dl_i}{\sum_j dl_j}$$

where the denominator in the right factor of the product denotes the sum of downloads of all libraries. Recall that downloads (like commits) are counted only in the year January 1, 2021 to January 1, 2022 for the analysis and results presented in this paper.

Summing the resulting download-weighted risk score over all libraries in the ecosystem yields our final risk score for a specific scenario (defined originally by the input contributor state vector S^C. It also carries a straightforward interpretation: it describes the average risk exposure of a random individual download of a library in the ecosystem for the given scenario. With this framework in hand, we can now define how we rank contributors and libraries, and quantify overall ecosystem's risk.

Ranking contributors

Ranking contributors in terms of their systemic importance in this context is straightforward. Given the removal of contributor j, implemented by setting the j-th entry of the input contributor state vector S^C to 0, the overall **Contributor Impact** I_j is simply the sum of all library download-weighted risk scores

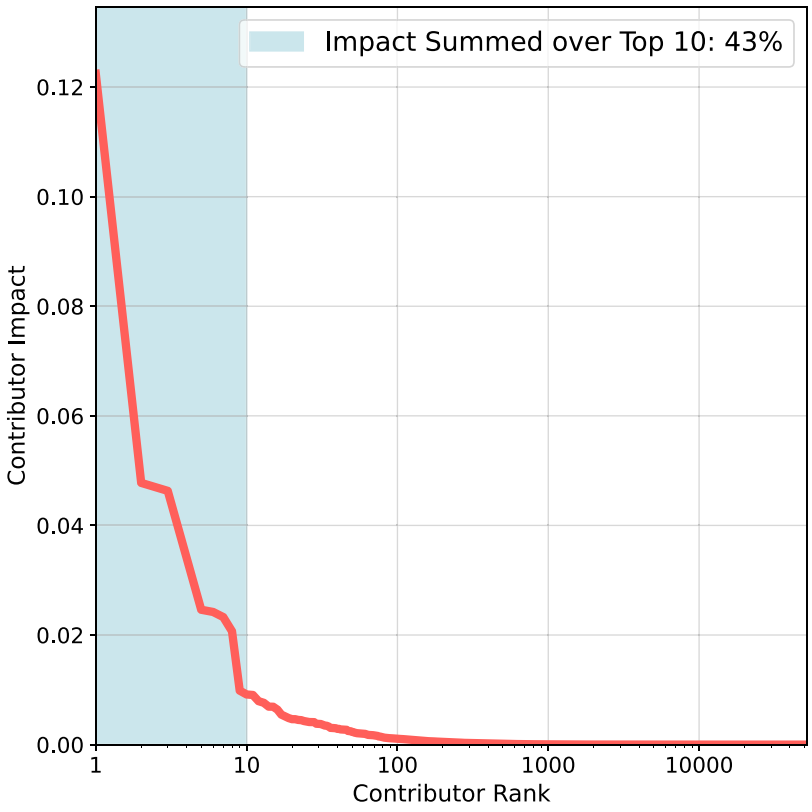

Figure 4. The rank-ordered contributions of Rust contributors to systemic risk, on a logarithmic scale. The top 10 developers account for 43% of systemic risk.

$$I_j = \sum_i R_i^{dl}$$

In other words, this measures the average impact that the removal a contributor from the system would have on a random observed download of a library in the Rust ecosystem.

Overall ecosystem risk

We derive global measure G of the systemic risk of the ecosystem by summing up all contributor impact scores

$$G = \sum_j I_j$$

This quantity can be used to compare the change in overall risk due to some intervention, as we will implement later in the paper. It also provides a baseline which we use to define library importance.

Ranking libraries

Finally, we also derive a measure ranking the systemic importance of libraries. This is perhaps the most important ranking, as interventions can most easily be made at the level of libraries. Specifically, we consider how often a library serves as a conduit of spreading failure to downstream dependencies. We do this by rerunning the full set of developer removals and failure propagation calculations with each library "immunized" to failure, one by one. In terms of our equation, the immunized library's entry in the library state vector S^L is hard-coded to 1. This counterfactual allows us to quantify how a library amplifies or *transmits* issues through the software dependency network. The change in the overall ecosystem risk score G when a library is protected in this way serves as a quantification of its contribution to overall risk. We define this measure, which we call the **Risk Transmission Score** of a library i, as follows

$$RTS_i = G - G_i$$

where G_i denotes the ecosystem risk score calculated when library i is immune to failure.

Results

We plot the rank-ordered distribution of Contributor Impact in Figure 4. We observe a remarkable

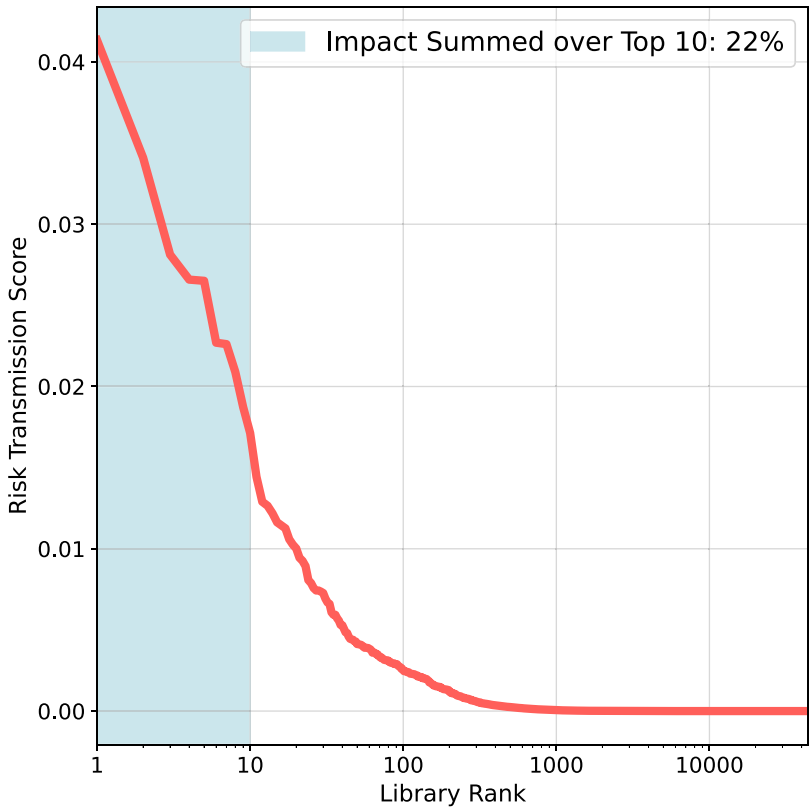

Figure 5. The risk transmission rank (RTR) importance of Rust libraries, on a logarithmic scale. We quantify a library's importance by rerunning our failure cascade model with that library immunized against failure, and then comparing the overall outcome against the general case when the library can fail. The top 10 libraries account for 22% of the total observed differences across all libraries.

concentration of systemic risk: with the top 10 contributors accounting for over a 40% of the risk observed in our analyses. We observe a similar, though slightly less concentrated distribution when we consider the importance of different libraries in terms of their Risk Transmission Score, see Figure 5. This recalls our motivating example from earlier in the paper: individual developers are playing an important role in multiple important libraries.

To compare our method of ranking important libraries with alternative measures, for example, by the count of their transitive dependencies, we zoom in on the very top ranked libraries. We plot libraries ranked among the top 20 according to Risk Transmission Rank *or* by the count of their transitive dependencies in Figure 6. Our method suggests that libraries above the diagonal are more important than a count of their downstream dependencies suggests. Our method highlights several libraries ("rand" and "syn," among others) that are among the top 10 libraries according to Risk Transmission Rank, but do not break into the top 100 libraries by number of transitive dependencies. These libraries have a prominent position in the network topology that amplify their contribution to systemic risk.

Interventions

While it is valuable to highlight vulnerabilities in a system, our methodology can be used to suggest how to intervene in the system to improve its resilience by allocating scarce development resources. In particular, our ranking of libraries in the Rust ecosystem can be used to allocate support. For instance, a foundation or firm may have funds to sponsor development or maintenance on a specific library. Though in reality, developer resources are not fungible and cannot be allocated to any library in arbitrary amounts (Mockus 2009), a prioritization in terms of risk remains useful. Thus in this section, we describe an intervention in terms of development time contributed to a fixed number of libraries. We compare the impact on systemic risk of various allocation strategies based on rankings of libraries, including our Risk Transmission Rank score. We first describe alternative rankings, then describe how we implement the interventions, and finally report results.

Rankings

As our aim is to compare reasonable strategies to allocate development resources across libraries, we consider several

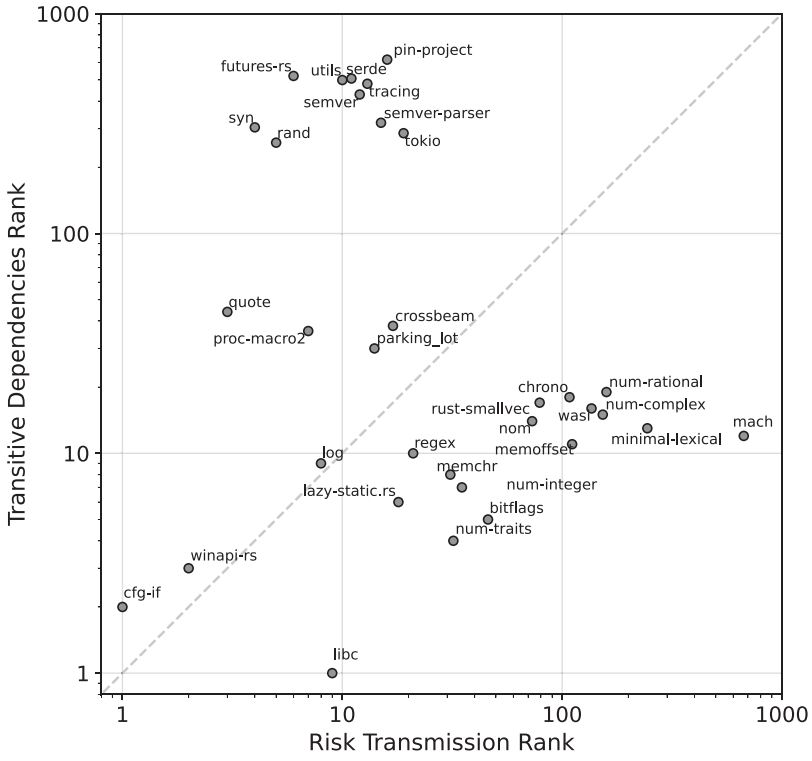

Figure 6. Libraries of the Rust ecosystem in the top 20 according to either their risk transmission rank (RTR) or the count of (downstream) transitive dependencies. We observe several libraries in the top 10 of the RTR, hence likely of systemic importance, that are not even in the top 100 according to the number of transitive dependencies they have.

perspectives on what makes a library important or well-known. We consider a library's place in the dependency network, its overall use, its age, its popularity, and its systemic importance as quantified by our Risk Transmission Rank. We also test random allocations as a benchmark. We summarize these rankings in Table 1, and describe them below.

In a technical sense, a library is important in an ecosystem if many other libraries depend on it. These dependencies can be direct or indirect (sometimes called transitive). We therefore use the count of *transitive dependencies* as one ranking of libraries (Decan et al., 2019). A simpler way to rank libraries is by their age, arguing that older (active) libraries are more likely to be important.

In practice, libraries are often ranked by their use and popularity in the broader community. We measure use via the count of *downloads* of a Rust library, as tracked by Cargo. Popularity or social visibility is measured by counting the number of *stars* a library has received on GitHub (Borges and Tulio Valente 2018). Both factors are thought to play an important role in the success of open source software. A large active user base of a specific library provides a kind of defense against errors as suggested by Raymond's notion that "with many eyes all bugs are shallow" (Raymond 1999). The launch of GitHub Sponsors and growing adoption of crowdfunding to support OSS maintainers suggests that highly visible libraries will be even more likely receive resources (Overney et al., 2020). However, we know from examples that not all systemically importance packages are highly visible. Anecdotally, OpenSSL was taken for granted before the discovery of the Heartbleed vulnerability.

Intervention design and quantifying impact

While software developers and software development time is not a fungible resource, we make the simplifying assumption that a donor can contribute to a library by adding a single developer. Developers added this way make a uniform weekly contribution of commits, which we fix at 5/7 times the number of days across which we analyze the system (in our case 365). We denote this contribution factor (5/7 times 365) by e, roughly one contribution per weekday. We add a single developer to the top K libraries according to each ranking for a range of values between 1 and 1000.

After allocation, we rerun our framework—one by one we remove each developer in the system and calculate the resulting cascades. We calculate G, the overall systemic risk of the ecosystem in each of these scenarios, comparing it the original estimate derived in the previous section. As more contributors are added, the overall systemic risk falls. We

Table 1. Ways to rank libraries in the rust ecosystem to allocate developer resources in an intervention.

Name	Description
Transitive dependencies	# of upstream dependencies
Downloads	# of downloads on Cargo
Age	Time since appearance
Stars	Number of stars on GitHub
Random	Random allocation (baseline)
Risk transmission rank	**Spreading-based measure**

compare the effectiveness of allocations according to the different rankings.

We need to make one modification to the methodology in this case. Specifically, if we are adding contributions to a library as part of an intervention, we should not simulate what happens if these new resources are withdrawn. Rather, we represent these extra contributions as a kind of *overproduction* (c.f. (Champion and Hill 2021)) that can be used to absorb shocks. Specifically, we add a term $\mathbf{X}$ to our self-referencing equation for the library state vector

$$S^L = \left(S^{C\top} \star \widehat{C}\right)^{1/2} \odot \left(S^{L\top} \star \widehat{D} + \mathbf{X}\right)^{1/2}$$

Here $\mathbf{X}$ is a vector corresponding to the libraries of the ecosystem. An entry i is equal to e/N, where N is the total number of commits to library i, if library i is allocated a developer, 0 otherwise. Recall that e captures the contributions of these allocated developers, estimated at a rate of one commit per week day. This correction insures that additional resources allocated by the intervention can only help a library. In each round of the calculation, we cap entries of the library state vector S^L at 1 to insure convergence.

Intervention results

We visualize the improvement in systemic risk as a function of developers added according to the different ranking heuristics in Figures 7 and 8. We observe a sharp decrease in overall risk using the Risk Transmission Rank, the number of downloads, or the number of transitive dependencies to rank libraries. Using GitHub stars, library age, or a random ranking to allocate development resources are significantly less effective strategies. Given that GitHub stars are a major source of visibility in the OSS community (Borges and Tulio Valente 2018), we suggest that equating systemic importance with stars may be leading to systemic misallocation of attention and help.

To quantify the relative performance of these rankings, we calculate the area below the horizontal line at the baseline systemic risk level and above the curve for each heuristic up to different numbers of developers added, and normalize by the total area under the baseline. We report these values in Table 2. These results verify the patterns we observe in the figures: allocating developers by RTR, downloads, and transitive dependencies are better strategies than allocating them by age or visibility (i.e., stars on GitHub). Though RTR performs best across the entire range of the intervention, it is still interesting to note that ranking libraries by transitive dependencies and downloads are also relatively strong heuristics. As these require less data and calculation, they may be good strategies in other ecosystems with less data.

Discussion

OSS ecosystems tend to evolve, like many complex systems, towards efficiency. If there is a library that does something well, it can quickly become widely used. However, this drive towards efficiency may increase systemic risks. Studies of risk in ecosystems that focus on the structure of technical dependencies overlook the potential synchronized risks coming from departing developers active in multiple libraries. In this work, we present a framework to quantify these risks.

Our method highlights individual libraries which are worthy of more attention and support. As the most central and important developers in OSS ecosystems are under increasing pressure and stress (Eghbal 2020), measures of ecosystem health need to consider the interaction of these social aspects with their technical structure (Constantinou and Mens 2017). In other words, OSS ecosystem sustainability can be thought about via the networks of dependencies between its libraries and how their maintainers span them. Within individual libraries or projects, the importance of socio-technical congruence—that is the coordination between people working on interdependent modules—is well known (Cataldo et al., 2008); our paper suggests how such congruence at the ecosystem level could provide warnings and help maintainers of downstream libraries anticipate upstream issues. More generally, our work contributes to a growing literature on the sustainability and resilience of key software supply chains (Lamb and Zacchiroli 2021; Ohm et al., 2020; Zimmermann et al., 2019).

The primary shortcoming of our work is the evaluation of our ranking of developer and libraries importance using ground-truth empirical data. Two distinct challenges make this difficult and merit future work. First, we need to detect the consequential departure of developers from roles as maintainers, either in specific libraries or the ecosystem as a whole. It is difficult to distinguish unplanned departures from more gentle farewells, in which other developers are ready to take over or a library is clearly labeled as deprecated and its downstream dependencies are informed. Second, we need to quantify the impact of such a departure,

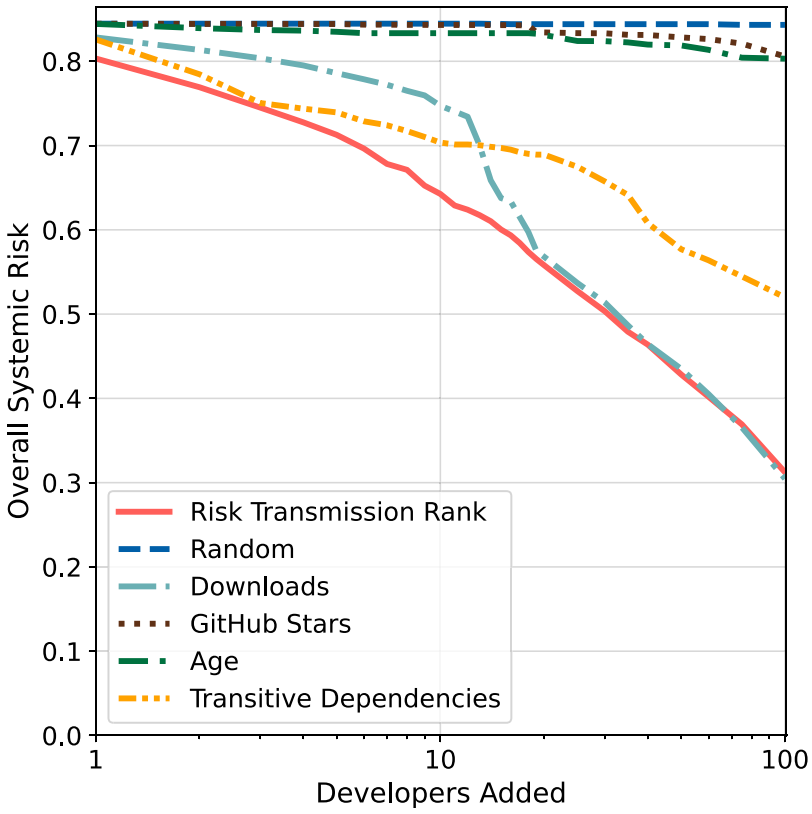

Figure 7. The change in overall systemic risk as a function of developers added by various library ranking heuristics, ranging from adding one contributor to 100 (log scale).

which may manifest in many ways. One possibility is to consider the growth in issues reported in downstream libraries and their rate of resolution over time. Another is to consider the spread of security vulnerabilities around un- or undermaintained libraries (Zheng et al., 2023).

Otherwise, our framework also has several potential empirical extensions. One could expand data to other ecosystems (Decan et al., 2019) or define dependencies at more granular levels (Blincoe et al., 2015; Hejderup et al., 2022). One could also expand the scope of contributions to include issue reporting and community management (Trinkenreich et al., 2020), or consider heterogeneity in commit sizes (Gote et al., 2021). Issue and pull request response times and rates (Dey and Mockus 2020), could be deployed as metrics for system health.

Another simplifying assumption of our work is that the libraries we observe are functioning and properly maintained, no matter how many developers they have. This is often false, as shown by Champion and Mako Hill in their recent work on the Debian package ecosystem (Champion and Hill 2021). They introduce the notion of underproduction and quantify it by tracking issue survival rates. One could extend our model by adapting this measure or by adopting different kinds of production functions.

Our framework can also be adapted to analyze the consequences of multiple developers leaving an ecosystem at the same time. This is not a hypothetical scenario—the increased reliance on centralized sponsors of libraries and even ecosystems presents another kind of risk. For example, many of the core developers of the Rust ecosystem were employed by Mozilla. In a round of layoffs in the summer of 2020, many of these developers lost their jobs at the same time. While Rust seems to have weathered this storm, it demonstrates that often multiple developers leave a system at around the same time. External geopolitical shocks like wars and economic sanctions between nations can also have a significant impact on software developers and the ecosystems they operate in (Wachs 2023). Our framework can also be used to study how different kinds of resources are allocated to support OSS. While our analyses considered the case of individual developers being added to specific projects, it can also be adapted to study the potential impact of, for example, collective efforts to fix issues in multiple libraries in a specific system[2].

In general, people leave OSS projects for reasons that may be correlated within an ecosystem, such as the end of funding of a university project or changing workplaces (Miller et al., 2019). Luis Villa of Tidelift, a firm that helps

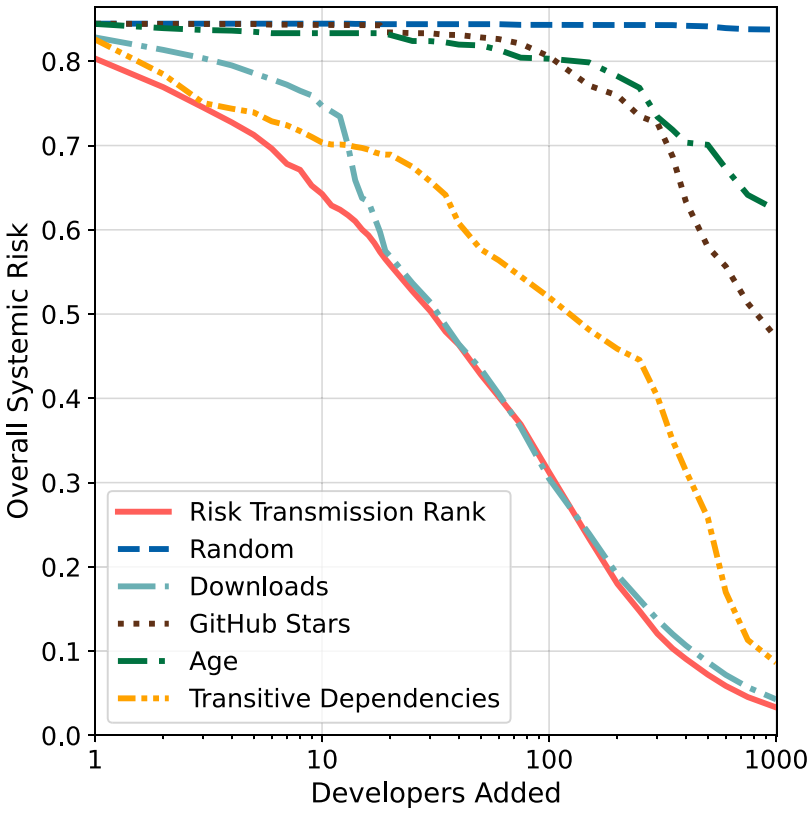

Figure 8. Extending the previous figure to scenarios adding between 100 and 1000 developers to the ecosystem.

Table 2. Cumulative reduction of systemic risk when adding developers according to different intervention strategies, relative to the baseline systemic risk. whether adding just a few or many developers, adding them to high RTR ranked libraries yields the greatest decrease in systemic risk.

Developers added	1(%)	2(%)	5(%)	10(%)	20(%)	50(%)	100(%)	250(%)	500(%)	1000(%)
Intervention										
Risk transmission rank	2.4	4.7	9.5	14.8	21.9	34.3	45.3	62.9	75.4	84.9
Downloads	1.0	1.9	4.0	6.5	14.5	30.8	43.7	61.8	74.0	83.4
Transitive dependencies	1.1	2.9	7.7	11.2	14.4	20.7	28.0	37.5	48.6	66.3
GitHub stars	0.0	0.0	0.0	0.1	0.2	1.0	2.0	6.3	14.0	26.2
Age	0.0	0.1	0.5	0.8	1.0	2.1	3.6	5.7	9.4	15.7
Random	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.5

users and firms support OSS with financial contributions, suggests that we should perhaps rather talk of a "boss factor" than truck factor.[3] Indeed, many of the most widely used and influential OSS projects are maintained by companies and paid individuals (Germonprez et al., 2019). In this way, OSS ecosystems can be thought of as a co-production of volunteers and companies (O'Neil et al., 2021). Policymakers seeking to promote the use of OSS should consider these aspects of sustainability (Blind et al., 2021).

Our work provides additional motivation for getting more people involved in OSS. Mentorship of new contributors has been shown to be a key determinant of people becoming active participants in ecosystems (Steinmacher et al., 2021). At the same time, our work shows indirectly how social barriers to participation (Steinmacher et al., 2015) make software ecosystems more brittle in the long run. More work is needed to understand how disparities in participation in open source, for example, owing to gender (Terrell et al., 2017; Vasilescu et al., 2014) or geography (Braesemann et al., 2019; Takhteyev 2012; Wachs et al., 2022), block us from realizing more stable systems.

Acknowledgements

The authors thank Christian Diem, Tobias Reisch, Hannah Schuster, Balint Daroczy, Aleksandra Urman, Rositsa Ivanova, and participants of seminars at the Complexity Science Hub Vienna and WU Wien for valuable feedback. Johannes Wachs acknowledges support from the Center for Collective Learning (101086712-LearnData-HORIZON-WIDERA-2022-TALENTS-01 financed by European Research Executive Agency (REA)) and the Hungarian National Scientific Fund (OTKA FK 145960).

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the European Research Executive Agency; 101086712-LearnData-HORIZON-WIDERA-2022-TALENTS-01 and Hungarian National Scientific Fund (OTKA FK-145960).

ORCID iD

Johannes Wachs https://orcid.org/0000-0002-9044-2018

Data Availability Statement

An anonymized dataset is available on Figshare (https://figshare.com/s/93158d03416765444650). The underlying code for both data collection and processing is released as an open-source Python library called RepoDepo: https://github.com/wschuell/repodepo. Code to reproduce our analyses is available at https://github.com/wschuell/misteriosse.

Notes

1. We refer to developers and contributors interchangeably, preferring to use the notation C to distinguish their role from the role of technical dependencies.
2. An example of such an effort in the Rust ecosystem is the Rust Lib Blitz project, see: https://blog.rust-lang.org/2017/05/05/libz-blitz.html
3. See: https://blog.tidelift.com/bus-factor-boss-factor-and-the-economics-of-disappearing-maintainers

References

Abdalkareem R, Nourry O, Wehaibi S, et al. (2017) Why do developers use trivial packages? an empirical case study on npm. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineeringm, Paderborn, Germany, 4–8 Sep 2017, pp. 385–395.

Alfadel M, Costa DE and Shihab E (2021) Empirical analysis of security vulnerabilities in python packages. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, 12 March 2021, pp. 446–457. IEEE.

Amreen S, Bichescu B, Bradley R, et al. (2019) A methodology for measuring floss ecosystems. In: *Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability*. Salmon Tower Building, New York City: Springer, 1–29.

Avelino G, Passos L, Hora A, et al. (2016) A novel approach for estimating truck factors. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), Austin, TX, 17 May 2017, pp. 1–10. IEEE.

Benkler Y, Shaw A and Hill BM (2015) Peer production: a form of collective intelligence. *Handbook of Collective Intelligence 175*. Cambridge, MA: MIT Press.

Birsan A (2021) Dependency confusion: how i hacked into apple, microsoft and dozens of other companies. *Medium*.

Blincoe K, Harrison F and Damian D (2015) Ecosystems in GitHub and a method for ecosystem identification using reference coupling. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–17 May 2015, pp. 202–211. IEEE.

Blind K and Schubert T (2023) Estimating the gdp effect of open source software and its complementarities with r&d and patents: evidence and policy implications. *The Journal of Technology Transfer* 1: 1–26.

Blind K, Böhm M, Grzegorzewska P, et al. (2021) *The Impact of Open Source Software and Hardware on Technological Independence, Competitiveness and Innovation in the EU Economy*. Brussels: European Commission.

Borges H and Tulio Valente M (2018) What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146: 112–129.

Braesemann F, Stoehr N and Graham M (2019) Global networks in collaborative programming. *Regional Studies, Regional Science* 6(1): 371–373.

Brown EHP (1957) The meaning of the fitted cobb-douglas function. *Quarterly Journal of Economics* 71(4): 546–560.

Cataldo M and Herbsleb JD (2013) Coordination breakdowns and their impact on development productivity and software failures. *IEEE Transactions on Software Engineering* 39(3): 343–360.

Cataldo M, Herbsleb JD and Carley KM (2008) Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Kaiserslautern, Germany, 9–10 October, 2008, pp. 2–11.

Champion K and Hill BM (2021) Underproduction: an approach for measuring risk in open source software. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, 9–12 March 2021, pp. 388–399. IEEE.

Chowdhury MAR, Abdalkareem R, Shihab E, et al. (2022) On the untriviality of trivial packages: an empirical study of npm javascript packages. *IEEE Transactions on Software Engineering* 48: 2695–2708.

Coelho J and Valente MT (2017) Why modern open source projects fail Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017, pp. 186–196.

Cogo FR, Oliva GA and Hassan AE (2022) Deprecation of packages and releases in software ecosystems: a case study on npm. *IEEE Transactions on Software Engineering* 48: 2208–2223.

Constantinou E and Mens T (2017) Socio-technical evolution of the ruby ecosystem in GitHub. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 20–24 Feb 2017, pp. 34–44. IEEE.

Cook RI (2020) Above the line, below the line. *Communications of the ACM* 63(3): 43–46.

Decan A, Mens T and Constantinou E (2018) On the impact of security vulnerabilities in the npm package dependency network. In: Proceedings of the 15th International Conference on Mining Software Repositories, Gothenburg, Sweden, 28–29 May 2018, pp. 181–191.

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

Decan A, Mens T, Zerouali A, et al. (2022) Back to the past–analysing backporting practices in package dependency networks. *IEEE Transactions on Software Engineering* 48: 4087–4099.

Dey T and Mockus A (2020) Effect of technical and social factors on pull request quality for the npm ecosystem. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Bari, Italy, 5 – 9 October, 2020, pp. 1–11.

Diem C, Borsos A, Reisch T, et al (2022) Quantifying firm-level economic systemic risk from nation-wide supply networks. *Scientific Reports* 12: 7791, 1-13.

Durumeric Z, Li F, Kasten J, et al. (2014) The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference, Vancouver, Canada, 5–7 November 2014, pp. 475–488.

Eghbal N (2020) *Working in Public: The Making and Maintenance of Open Source Software*. San Francisco and Dublin: Stripe Press.

Ferreira M, Mombach T, Valente MT, et al. (2019) Algorithms for estimating truck factors: a comparative study. *Software Quality Journal* 27(4): 1583–1617.

Germonprez M, Lipps J and Goggins S (2019) *The Rising Tide: Open Source's Steady Transformation*. Greenville, South Carolina: First Monday.

Golzadeh M, Decan A, Legay D, et al. (2021) A ground-truth dataset and classification model for detecting bots in GitHub issue and PR comments. *Journal of Systems and Software* 175: 110911.

Gote C, Scholtes I and Schweitzer F (2021) Analysing time-stamped co-editing networks in software development teams using git2net. *Empirical Software Engineering* 26(4): 1–41.

Greenstein S and Nagle F (2014) Digital dark matter and the economic contribution of Apache. *Research Policy* 43(4): 623–631.

Haldane AG and May RM (2011) Systemic risk in banking ecosystems. *Nature* 469(7330): 351–355.

Hejderup J, van Deursen A and Gousios G (2018) Software ecosystem call graph for dependency management. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER), Gothenburg, Sweden, 27 May 2018, pp. 101–104. IEEE.

Hejderup J, Beller M, Triantafyllou K, et al. (2022) Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering* 27(5): 102–142.

Herbsleb JD and Grinter RE (1999) Splitting the organization and integrating the code: conway's law revisited. In: Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA, 22 May 1999, pp. 85–95.

Kinney R, Crucitti P, Albert R, et al. (2005) Modeling cascading failures in the north american power grid. *The European Physical Journal B* 46(1): 101–107.

Lamb C and Zacchiroli S (2021) *Reproducible Builds: Increasing the Integrity of Software Supply Chains*. New York City, NY: IEEE Software.

Lehman MM (1980) Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE* 68(9): 1060–1076.

Luszcz J (2018) Apache struts 2: how technical and development gaps caused the equifax breach. *Network Security* 2018(1): 5–8.

Ma Y (2018) Constructing supply chains in open source software. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), Gothenburg, Sweden, 27 May 2018, pp. 458–459. IEEE.

Mens T and Grosjean P (2015) The ecology of software ecosystems. *Computer* 48(10): 85–87.

Miller C, Widder DG, Kästner C, et al. (2019) Why do people give up flossing? a study of contributor disengagement in open source. In: IFIP International Conference on Open Source Systems, Montreal, QC, Canada, 26–27 May 2019, pp. 116–129. Springer.

Mockus A (2009) Succession: measuring transfer of code and developer productivity. In: 2009 IEEE 31st International Conference on Software Engineering, Washington, DC, 16–24 May 2009, pp. 67–77. IEEE.

Newman LH (2021) The internet is on fire. *Wire Magazine* 1: 1–15.

Ohm M, Plate H, Sykosch A, et al. (2020) Backstabber's knife collection: a review of open source software supply chain attacks. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Lisbon, Portugal, 24-26 June 2009, pp. 23–43. Springer.

Overney C, Meinicke J, Kästner C, et al. (2020) How to not get rich: an empirical study of donations in open source. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Korea, 5–11 Oct 2020, pp. 1209–1221.

O'Neil M, Muselli L, Raissi M, et al. (2021) 'Open source has won and lost the war': legitimising commercial–communal

hybridisation in a FOSS project. *New Media & Society* 23(5): 1157–1180.

Pashchenko I, Plate H, Ponta SE, et al. (2018) Vulnerable open source dependencies: counting those that matter. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, New York, NY, 11–12 Oct 2018, pp. 1–10.

Peters K, Buzna L and Helbing D (2008) Modelling of cascading effects and efficient response to disaster spreading in complex networks. *International Journal of Critical Infrastructures* 4(1-2): 46–62.

Pfeiffer RH (2021) Identifying critical projects via pagerank and truck factor. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 22–30 May 2021, pp. 41–45. IEEE.

Poledna S, Molina-Borboa JL, Martínez-Jaramillo S, et al. (2015) The multi-layer network nature of systemic risk and its implications for the costs of financial crises. *Journal of Financial Stability* 20: 70–81.

Raymond E (1999) The cathedral and the bazaar. *Knowledge, Technology & Policy* 12(3): 23–49.

Lo Sardo DR, Thurner S, Sorger J, et al. (2019) Quantification of the resilience of primary care networks by stress testing the health care system. *Proceedings of the National Academy of Sciences of the United States of America* 116(48): 23930–23935.

Schneider CM, Yazdani N, Araújo NA, et al. (2013) Towards designing robust coupled networks. *Scientific Reports* 3(1): 1–7.

Schueller W, Wachs J, Servedio VD, et al. (2022) Evolving collaboration, dependencies, and use in the rust open source software ecosystem. *Scientific Data* 9(1): 703.

Spaeth S, von Krogh G and He F (2015) Research note—perceived firm attributes and intrinsic motivation in sponsored open source software projects. *Information Systems Research* 26(1): 224–237.

Steinmacher I, Conte T, Gerosa MA, et al. (2015) Social barriers faced by newcomers placing their first contribution in open source software projects. In: Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing, Vancouver, BC, Canada, 14–18 March 2015, pp. 1379–1392.

Steinmacher I, Balali S, Trinkenreich B, et al. (2021) Being a mentor in open source projects. *Journal of Internet Services and Applications* 12(1): 7–33.

Takhteyev Y (2012) *Coding Places: Software Practice in a South American City*. Cambridge, Massachusetts: MIT Press.

Terrell J, Kofink A, Middleton J, et al. (2017) Gender differences and bias in open source: pull request acceptance of women versus men. *PeerJ Computer Science* 3: e111.

Thurner S and Poledna S (2013) Debtrank-transparency: controlling systemic risk in financial networks. *Scientific Reports* 3(1): 1–7.

Torchiano M, Ricca F and Marchetto A (2011) Is my project's truck factor low? theoretical and empirical considerations about the truck factor threshold. In: Proceedings of the 2Nd International Workshop on Emerging Trends in Software Metrics, Honolulu, HI, May 24, 2011, pp. 12–18.

Tóth G, Elekes Z, Whittle A, et al. (2022) *Technology Network Structure Conditions the Economic Resilience of Regions*. Vancouver, BC, Canada: Economic Geography.

Trinkenreich B, Guizani M, Wiese I, et al. (2020) Hidden figures: roles and pathways of successful OSS contributors. *Proceedings of the ACM on Human-Computer Interaction* 4(CSCW2): 1–22.

Valiev M, Vasilescu B and Herbsleb J (2018) Ecosystem-level determinants of sustained activity in open-source projects: a case study of the pypi ecosystem. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista FL, 4–9 Nov 2018, pp. 644–655.

Vasilescu B, Capiluppi A and Serebrenik A (2014) Gender, representation and online participation: a quantitative study. *Interacting with Computers* 26(5): 488–511.

Vasilescu B, Blincoe K, Xuan Q, et al. (2016) The sky is not the limit: multitasking across GitHub projects. In: Proceedings of the 38th International Conference on Software Engineering, Austin Texas, 14 May 2016, pp. 994–1005.

Wachs J (2023) Digital traces of brain drain: developers during the Russian invasion of Ukraine. *EPJ Data Science* 12(1): 14.

Wachs J, Nitecki M, Schueller W, et al. (2022) The geography of open source software: evidence from github. *Technological Forecasting and Social Change* 176: 121478.

Wang Y, Wen M, Liu Y, et al. (2020) Watchman: monitoring dependency conflicts for python library ecosystem. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June 2020, pp. 125–135.

Williams L and Kessler RR (2003) *Pair Programming Illuminated*. Glenview, IL: Addison-Wesley Professional.

Zerouali A, Mens T, Decan A, et al (2021) On the Impact of Security Vulnerabilities in the NPM and Rubygems Dependency Networks. *arXiv preprint arXiv:2106.06747*.

Zheng X, Wan Z, Zhang Y, et al. (2023) A closer look at the security risks in the rust ecosystem. *ACM Transactions on Software Engineering and Methodology* 33: 1–30.

Zimmermann M, Staicu CA, Tenny C, et al. (2019) Small world with high risks: a study of security threats in the npm ecosystem. In: 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, 28 May 2019, pp. 995–1010.