Rede de Dependências em Projetos e Software

Bruno Rodrigues Lopes de Souza
rodrigues.lopes@aluno.ufabc.edu.br RA: 11202321911

Raquel da Silva Rodrigues
raquel.r@aluno.ufabc.edu.br
RA: 11202131327

Emilly Palhares Melo
emilly.palhares@aluno.ufabc.edu.br
RA: 11202320270

Samuel Oliveira Costa
samuel.costa@aluno.ufabc.edu.br
RA: 11202510049

”Universidade Federal do ABC, Santo André, São Paulo, Brasil”

Resumo — A reutilização de pacotes é um dos pilares fundamentais do desenvolvimento de software moderno. Ecossistemas como o PyPI (Python Package Index) reúnem milhões de bibliotecas interligadas por relações de dependência, formando estruturas complexas. Este projeto modela essas dependências como um grafo direcionado, com base em dados reais extraídos do Libraries.io. Por meio da análise de métricas como grau de entrada, centralidade de intermediação e PageRank, identificamos padrões estruturais e bibliotecas críticas. Os resultados revelam riscos potenciais de propagação de falhas e destacam ecossistemas com diferentes níveis de coesão e modularidade.

Palavras-chave: Redes complexas, dependência de software, bibliotecas críticas, NetworkX, open source.

I. INTRODUÇÃO

Nos últimos anos, o desenvolvimento de software tem evoluído para modelos altamente modulares e interdependentes. Em ambientes open source, como PyPI, NPM ou RubyGems, desenvolvedores frequentemente constroem novos projetos a partir de bibliotecas existentes, confiando que essas dependências continuarão disponíveis, atualizadas e seguras.

Entretanto, essa estrutura de dependência — embora eficiente e colaborativa — também introduz fragilidades: um erro, falha ou remoção em um único pacote amplamente utilizado pode causar interrupções em milhares de projetos que dependem direta ou indiretamente dele. Isso caracteriza o que chamamos de riscos sistêmicos em redes de software.

Este projeto se propõe a investigar a arquitetura subjacente dessas redes, buscando responder: *quais pacotes exercem papel central no funcionamento dos ecossistemas de software?* A partir da construção de grafos direcionados com dados reais, analisamos a topologia da rede de dependências e propomos interpretações sobre sua estrutura, vulnerabilidades e comportamento.

II. OBJETIVOS

Objetivo geral: Investigar e modelar a rede de dependências entre pacotes de software open source, com foco em identificar os elementos mais centrais e estruturalmente críticos.

A. Objetivos específicos:

- Extrair e tratar dados reais de dependência entre pacotes em múltiplos ecossistemas;
- Representar as dependências como grafos direcionados usando ferramentas computacionais;
- Calcular métricas de análise de redes, como grau de entrada, centralidade de intermediação e PageRank;
- Comparar a estrutura de diferentes plataformas (ex: PyPI, RubyGems, Cargo);
- Discutir a resiliência e os riscos associados à falha de pacotes centrais.

III. FERRAMENTAS E FONTE DE DADOS

A. Libraries.io

O projeto utilizou o dataset completo do Libraries.io, um indexador de pacotes open source que cobre mais de 30 gerenciadores de pacotes diferentes. O arquivo base contém registros com o

nome do pacote, a plataforma (ex: npm, pypi, rubygems) e a dependência correspondente.

B. Zenodo

O dataset foi obtido por meio do repositório oficial no Zenodo (DOI: 10.5281/zenodo.2536573), no formato CSV. Após download, foi convertido para o formato Parquet, visando maior eficiência no carregamento com Pandas.

C. Ferramentas de Processamento

- Python 3.11
- Pandas – para manipulação e filtragem de dados
- NetworkX – para modelagem dos grafos e cálculo das métricas
- Jupyter Notebook – ambiente de desenvolvimento
- Markdown e GitHub – documentação e versionamento do projeto: https://github.com/samuelc254/cr-ufabc

IV. METODOLOGIA

A. Estruturação dos Dados

O primeiro passo consistiu na filtragem do dataset, mantendo apenas:

- Dependências mais recentes por pacote;
- Linhas com informações completas de origem e destino;
- Pacotes identificados com nome válido.

Cada linha da base foi convertida para uma aresta do tipo: Pacote A → Pacote B.

B. Construção do Grafo

A rede foi modelada como um grafo direcionado (DiGraph) com a biblioteca NetworkX. O grafo completo contém:

- 1.758.429 nós (pacotes únicos)
- 14.078.099 arestas (relações de dependência).

Devido à escala, a análise foi dividida em duas frentes:

- Análise geral da rede unificada e
- Análises individuais por plataforma (ex: PyPI, Maven, NPM).

C. Métricas Calculadas

As principais métricas de rede aplicadas foram:

- Grau de entrada (in-degree)
- Grau de saída (out-degree)
- PageRank
- Centralidade de Intermediação (Betweenness)
- Componentes fracamente conectados (WCC)
- Densidade
- Clusterização e modularidade (Louvain)

V. RESULTADOS

A. Análise Comparativa por Plataforma

O estudo individual das redes de dependências das plataformas analisadas permitiu identificar tanto características estruturais comuns quanto comportamentos atípicos, revelando o grau de interdependência e organização interna dos diferentes ecossistemas de software. Nesta seção, são comparadas métricas selecionadas de 19 plataformas monitoradas pelo Libraries.io

Tabela 1 – Comparativo entre plataformas - Densidade

Plataforma	Densidade
PyPI	0.00005820
Maven	0.00003602
CPAN	0.00025958
CRAN	0.00043041
Cargo	0.00019613
NuGet	0.00001832
Packagist	0.00002028
Rubygems	0.00003528
Pub	0.00038413
Conda	0.00177832
Dub	0.00102691
Elm	0.00174143
Haxelib	0.00216532

Hex	0.00034409
Homebrew	0.00092723
Puppet	0.00043277
Atom	0.00032653
Haxelib	0.00216532
NPM	0.00001037

Tabela 2 – Comparativo entre plataformas - Componentes FC

Plataforma	Componentes FC
PyPI	577
Maven	1.561
CPAN	32
CRAN	3
Cargo	152
NuGet	2.957
Packagist	1.630
Rubygems	366
Pub	16
Conda	63
Dub	93
Elm	1
Haxelib	49
Hex	119
Homebrew	45
Puppet	54
Atom	222
Haxelib	49
NPM	4316

Tabela 3 – Comparativo entre plataformas - Grau Médio

Plataforma	Grau Médio
PyPI	5.43
Maven	8.83
CPAN	17.12
CRAN	13.27
Cargo	9.53
NuGet	5.50
Packagist	7.56
Rubygems	9.03
Pub	6.66
Conda	4.51
Dub	2.38
Elm	5.13
Haxelib	2.76
Hex	3.93
Homebrew	3.94
Puppet	4.08
Atom	4.73
Haxelib	2.76
NPM	21.49

Tabela 4 – Comparativo entre plataformas - Clusterização

Plataforma	Clusterização
PyPI	0.0381
Maven	0.1094
CPAN	0.1396
CRAN	0.1762
Cargo	0.1143
NuGet	0.0862

Packagist	0.1421
Rubygems	0.1970
Pub	0.1103
Conda	0.1568
Dub	0.0326
Elm	0.3005
Haxelib	0.0680
Hex	0.0405
Homebrew	0.0000
Puppet	0.1401
Atom	0.0015
Haxelib	0.0680
NPM	0.1236

Tabela 5 – Comparativo entre plataformas - Modularidade

Plataforma	Modularidade
PyPI	0.5505
Maven	0.6091
CPAN	0.3815
CRAN	0.3258
Cargo	0.4845
NuGet	0.6806
Packagist	0.5905
Rubygems	0.3968
Pub	0.4839
Conda	0.5542
Dub	0.8532
Elm	0.3852
Haxelib	0.8335
Hex	0.6516
Homebrew	0.6236
Puppet	0.5360
Atom	0.6067
Haxelib	0.8335
NPM	0.4848

B. Análise dos resultados

A maioria das plataformas apresenta significativa fragmentação, com alto número de componentes fracamente conectados. Isso é particularmente evidente em NPM (4.316), NuGet (2.957), Maven (1.561) e Packagist (1.630), indicando redes amplas, mas com conectividade limitada entre grandes subconjuntos. Essas redes, apesar de populares, abrigam diversas comunidades isoladas, sugerindo uma estrutura menos coesa.

Em termos de densidade, destacam-se Haxelib (0.00216532), Elm (0.00174143) e Conda (0.00177832), com grafos proporcionalmente mais conectados. Isso aponta para ecossistemas onde os pacotes compartilham mais dependências entre si, promovendo interdependência e integração mais densa. Por outro lado, plataformas como Rubygems, Maven, NuGet e NPM apresentam densidades extremamente baixas (todas inferiores a 0.00005), reflexo da escala massiva combinada a uma arquitetura dispersa e modular.

No aspecto do grau médio, NPM se destaca com a maior média de todos os ecossistemas analisados (21.49), seguido por CPAN (17.12), CRAN (13.27) e Rubygems (9.03). Esses valores indicam ecossistemas com forte reutilização de pacotes, o que potencializa a eficiência no desenvolvimento, mas também aumenta os riscos de propagação de falhas. Em contraste, redes como Dub, Haxelib e Homebrew apresentam baixos graus médios, sugerindo menor acoplamento entre bibliotecas.

O coeficiente de clusterização médio, que representa a tendência de pacotes formarem “grupinhos” altamente conectados, atinge seus maiores valores em Elm (0.3005), Rubygems (0.1970) e CRAN (0.1762). Essas plataformas parecem conter subestruturas coesas, onde bibliotecas se organizam em comunidades densas. Por outro lado, Homebrew (0.0000), Atom (0.0015) e PyPI (0.0381) possuem estruturas mais

dispersas, com baixa tendência à formação de agrupamentos densos.

Por fim, ao observar a modularidade, que mede a separação entre comunidades na rede, destacam-se Dub (0.8532) e Haxelib (0.8335), com arquiteturas bem compartimentalizadas e comunidades fortemente definidas. NPM, apesar da baixa densidade e alta fragmentação, apresenta modularidade de 0.4848 e mais de 5.400 comunidades detectadas, sinalizando um ecossistema vasto e altamente modular.

C. Destaques

- **Plataforma mais representativa (perfil médio):**

PyPI, com grau médio, densidade e modularidade dentro da faixa central observada entre as plataformas. Também apresenta uma quantidade intermediária de componentes fracamente conectados (577) e comunidades detectadas (625), servindo como reflexo do comportamento médio das redes de dependência.

- **Plataforma mais conectada (alta reutilização):**

NPM, com grau médio de 21.49, o maior entre todas as plataformas, além de ampla diversidade interna (5.428 comunidades detectadas). Essa estrutura revela altíssima reutilização de bibliotecas, sendo um dos ecossistemas mais acoplados e interligados.

- **Plataforma mais fragmentada:**

NuGet, com 2.957 componentes fracamente conectados e uma densidade extremamente baixa (0.00001832), apresenta uma rede dispersa e pouco coesa, refletindo um alto grau de independência entre pacotes.

D. Análise Geral da Rede Unificada

A análise do grafo completo, que consolida as dependências de diversas plataformas open source, oferece uma visão abrangente da complexa estrutura de interdependência entre bibliotecas no ecossistema de software moderno. Os dados foram extraídos do dataset do Libraries.io e modelados como um grafo direcionado com mais de 1,7 milhão de nós. A seguir, apresentamos os principais resultados obtidos.

Figura 1 – Visualização do subgrafo dos 1000 nós mais conectados da rede de dependências unificada. Cada nó representa um pacote de software, e cada aresta indica uma relação de dependência.

A Figura 1 apresenta uma visualização parcial da rede geral, limitada aos 1000 nós mais conectados para melhor legibilidade. A imagem evidencia a presença de hubs altamente conectados e sub estruturas modulares, sugerindo a existência de comunidades bem definidas e pontos críticos de centralização na rede.

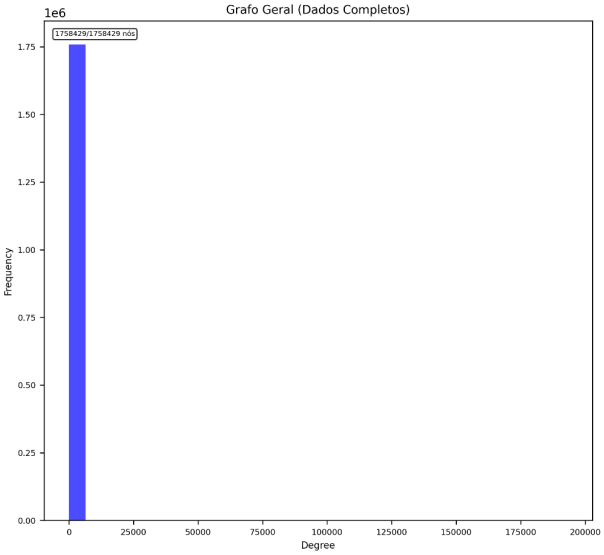

Figura 2 – Visualização do histograma de frequência de grau dos nós nas rede geral

A Figura 2 exibe o histograma da frequência de graus da rede, cuja visualização em escala linear se mostra impraticável. Isso ocorre devido à distribuição de cauda longa (long-tail) dos dados, uma característica central de redes scale-free: a vasta maioria dos nós possui um grau muito baixo, enquanto um número reduzido de hubs concentra um número extremamente alto de conexões. Tal assimetria comprime quase toda a informação visual no início do eixo. A abordagem tomada para analisar essa estrutura foi utilizar um gráfico em escala log-log, que revela o comportamento de lei de potência (*power-law*) da rede.

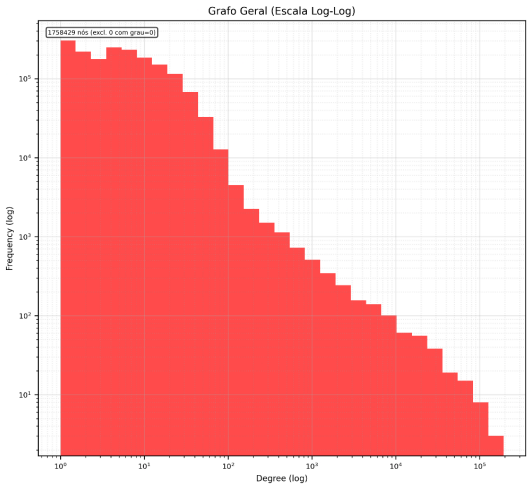

Figura 3 – Visualização do histograma de frequência de grau dos nós nas rede geral em escala logarítmica

A transformação logarítmica revela uma relação aproximadamente linear entre o grau de um nó e sua frequência, evidenciada pela tendência retilínea dos pontos no gráfico. Este padrão linear é a assinatura de uma distribuição de lei de potência (*power-law*), confirmando que a rede de dependências de software possui uma topologia *scale-free* (livre de escala). Essa característica estrutural implica que, em vez de uma conectividade homogênea, o ecossistema é dominado por um pequeno número de pacotes (hubs) que são desproporcionalmente mais conectados que a grande maioria dos outros nós, o que tem profundas implicações para a robustez e vulnerabilidade da rede.

Para aprofundar a análise da conectividade geral da rede, foi calculado o seu diâmetro. O cálculo exato desta métrica, no entanto, é computacionalmente inviável para um grafo com mais de 1.7 milhão de nós. Por isso, foi empregada uma metodologia de aproximação por amostragem no maior componente fortemente conectado da rede, que resultou em um

diâmetro aproximado de 28. Este valor significa que são necessários, no máximo, 28 saltos de dependência para conectar os dois pacotes mais distantes do ecossistema. Um diâmetro relativamente alto como este sugere que a rede, embora vasta, possui baixa integração e não exibe a propriedade de "mundo pequeno" (*small-world*), reforçando a ideia de uma estrutura hierárquica onde a propagação de atualizações ou vulnerabilidades pode percorrer longas cadeias de dependência.

E. Métricas Gerais

A Tabela 6 resume as principais características estruturais da rede geral de dependências:

Tabela 6 – Métricas Estruturais do Grafo Geral

Métrica	Valor
Nós(Projetos/Dependências)	1.758.429
Arestas (Relações de Dependência)	14.078.099
Densidade do Grafo	0.00000455
Tipo de Grafo	Direcionado

Esses valores destacam uma rede altamente esparsa, com um número muito pequeno de conexões em relação ao total possível de arestas. A densidade extremamente baixa é comum em redes reais de grande escala.

- **Análise de Conectividade**

A conectividade da rede é fortemente assimétrica. Embora haja mais de 11 mil componentes fracamente conectados, a maior parte dos nós pertence a um único componente massivo:

Tabela 7 – Conectividade do Grafo

Métrica	Valor
Componentes Fracamente Conectados (WCCs)	11.301

Nós no Maior Componente (LCC)	1.604.174 (91,23%)
Arestas no LCC	13.504.283 (95,92%)

A presença de um grande componente dominante é um indício claro de que a maioria dos pacotes está interligada por caminhos diretos ou indiretos, indicando alta coesão funcional do ecossistema.

- **Análise de Graus (Popularidade e Complexidade)**

O grau de entrada (in-degree) de um nó representa quantos pacotes dependem dele — uma métrica direta de popularidade. O grau de saída (out-degree), por outro lado, representa quantos pacotes um nó consome, refletindo sua complexidade.

Tabela 8 - Estatísticas Gerais de Grau

Métrica	In-Degree	Out-Degree	Grau Total
Média	8,01	8,01	16,01
Mediana	0,00	4,00	5,00
Máximo	192.961	1.000	193.045

O valor de mediana zero no in-degree revela uma característica interessante: grande parte dos pacotes não é utilizada por outros — muitos podem ser utilitários internos, protótipos ou obsoletos.

- **Top 10 Dependências Mais Populares**

Pacotes com maior grau de entrada são verdadeiros pilares da infraestrutura de software. Estes pacotes são reutilizados por dezenas (ou centenas) de milhares de outros projetos:

Tabela 9 – Principais Dependências (Maior In-Degree)

Rank	Dependência	In-Degree
1	mocha	192.961
2	eslint	177.661
3	typescript	128.561
4	chai	117.227
5	webpack	116.054
6	babel-core	111.311
7	babel-loader	99.137
8	react	95.553
9	lodash	93.358
10	jest	88.218

Observa-se que bibliotecas amplamente adotadas em desenvolvimento front-end, testes e transpilação (ex: mocha, webpack, babel) lideram em reutilização.

- **Top 10 Projetos Mais Complexos**

Estes são os projetos que mais dependem de outras bibliotecas. Geralmente são grandes aplicações ou ambientes altamente integrados:

Tabela 10 – Projetos Mais Complexos (Maior Out-Degree)

Rank	Projeto	Out-Degree
1	sindresorhus.js	1.000
2	npm-bomb	999
3	1000-packages	999
4	all-of-them	989
5	digital-keyboard-demos	975
6	@hawkingnetwork/react-native-tab-view	921
7	@ericmcornelius/ease	905
8	vue-compment	895
9	nois-react-toast	873
10	spring-boot-dependencies (Java)	856

A presença de pacotes experimentais ou de demonstração no ranking indica que algumas

entradas podem ter sido criadas com o propósito específico de estressar o ecossistema.

- **Análise de Centralidade**

Duas métricas clássicas de redes complexas ajudam a identificar nós centrais:

PageRank: influência na propagação da informação.

Betweenness Centrality: importância para a conectividade estrutural da rede.

- **Top 10 Nós por PageRank**

Tabela 11 – Pacotes com Maior Influência (PageRank)

Rank	Nó	PageRank Score
1	mkdirp	0.019664
2	mock-fs	0.019409
3	eslint-plugin-import	0.016975
4	jest	0.016138
5	babel-core	0.014131
6	rspec	0.013571
7	webpack	0.012042
8	sinon	0.011489
9	istanbul	0.011411
10	@babel/preset-env	0.011333

- **Top 10 Nós por Betweenness Centrality**

Tabela 12 – Pacotes com Maior Importância Estrutural (Betweenness)

Rank	Nó	Betweenness Score
1	rollup	0.000049
2	webpack	0.000046
3	async	0.000043
4	core-js	0.000027
5	eslint-plugin-import	0.000026
6	karma	0.000026
7	@babel/preset-env	0.000025
8	istanbul	0.000023
9	airtap	0.000021
10	redux	0.000019

As bibliotecas webpack e @babel/preset-env aparecem com destaque em ambas as métricas, indicando que são estrategicamente centrais e altamente reutilizadas — pontos críticos da rede.

VI. DISCUSSÃO

A análise dos grafos de dependência, tanto de forma unificada quanto por plataforma, revela uma arquitetura complexa, com propriedades estruturais que têm implicações diretas na robustez, vulnerabilidade e eficiência do ecossistema de software open source.

A. A Topologia Scale-Free e a Centralização do Ecossistema

A análise do grafo geral, com mais de 1,7 milhão de nós, confirma a natureza massivamente conectada do ecossistema de software. A característica mais marcante é sua topologia scale-free (livre de escala). A alta assimetria nos graus, indicada na Tabela 8 , é visualmente comprovada pela distribuição de lei de potência (power-law) observada no histograma em escala log-log (Figura 3).

Esta topologia tem uma consequência dual: por um lado, a rede é robusta a falhas aleatórias, pois a remoção de um dos milhões de pacotes com baixo grau de conexão tem impacto local e insignificante. Por outro lado, ela é extremamente vulnerável a ataques ou falhas direcionadas. Se um pacote-hub como mocha (com quase 200 mil dependências) ou eslint for comprometido, os impactos podem se propagar em cascata, gerando um risco sistêmico para uma vasta porção da indústria de software.

B. Conectividade, Distância e a Ausência de um "Mundo Pequeno"

Para aprofundar a análise da conectividade, foi calculado o diâmetro da rede.

Devido à escala do grafo, uma abordagem de aproximação por amostragem foi utilizada, resultando em um diâmetro aproximado de 28. Este valor, relativamente alto, indica que são necessários até 28 "saltos" de dependência para conectar os dois pacotes mais distantes. Isso sugere que a rede, embora possua um componente gigante que conecta mais de 91% dos nós, não exibe a propriedade de "mundo pequeno" (small-world) encontrada em redes sociais. A estrutura é mais alongada e hierárquica, o que pode retardar a propagação de atualizações, mas também de vulnerabilidades, através de longas cadeias de dependência.

C. A Arquitetura das Plataformas: Coesão vs. Fragmentação

A análise individual das plataformas revelou ecossistemas com "personalidades" distintas:

- **CRAN (R)** se destaca como uma rede altamente **coesiva e integrada**. Com apenas 3 componentes fracamente conectados e um dos maiores coeficientes de clusterização (0.1762), seus pacotes formam uma comunidade científica fortemente interligada.
- **NuGet (.NET)**, em contraste, representa um ecossistema **fragmentado**. Com quase 3.000 componentes desconexos e uma densidade baixíssima, sua estrutura é dispersa, sugerindo um alto grau de independência entre os projetos.
- **NPM (JavaScript)** é o caso mais complexo: apesar de altamente fragmentado (4.316 componentes), possui o maior grau médio (21.49), indicando uma reutilização de código massiva. Sua estrutura é de uma "nação" vasta, composta por milhares de "cidades-estado" (comunidades) que são muito densas internamente, mas pouco conectadas entre si.

D. Limitações do Estudo

É importante reconhecer as limitações desta análise. Primeiramente, os dados do Libraries.io representam um snapshot estático; uma análise temporal poderia revelar como essas redes evoluem. Em segundo lugar, métricas como o diâmetro foram calculadas de forma aproximada devido a restrições computacionais. Por fim, a análise focou em dependências diretas, não explorando o peso ou a natureza dessas conexões (ex: dependências de desenvolvimento vs. produção), o que representa uma oportunidade para trabalhos futuros.

VII. PARTICIPAÇÃO DOS ALUNOS

- **Emilly:** Organização dos dados e Filtragem dos resultados e Métricas;
- **Raquel:** Organização dos dados;
- **Bruno:** Criação do vídeo e Análise dos Resultados e Métricas;
- **Samuel:** Coleta de dados , processamento de dados e Filtragem de dados.

VIII. CONCLUSÃO

Este estudo modelou e analisou a rede de dependências de software open source, confirmando que ela forma uma estrutura complexa, heterogênea e com marcantes características scale-free. A construção de grafos a partir de dados reais permitiu não apenas visualizar, mas quantificar a arquitetura subjacente do software moderno.

As principais conclusões foram:

- **A confirmação da topologia scale-free:** A distribuição de graus segue uma lei de potência (Figura 3), o que implica uma alta centralização em torno de poucos pacotes-chave.
- **A identificação de infraestrutura crítica:** Pacotes como mocha, eslint, webpack e core-js emergiram como nós de altíssima centralidade em múltiplas métricas (grau, PageRank, intermediação), funcionando como pilares para dezenas de milhares de outros projetos.
- **A heterogeneidade dos ecossistemas:** As plataformas variam drasticamente em coesão, modularidade e padrões de reutilização, com CRAN sendo um exemplo de rede integrada e NuGet de um ecossistema fragmentado.

As implicações desses achados são significativas. A centralização excessiva, especialmente no ecossistema NPM, representa um risco sistêmico que pode levar a falhas em cascata. A análise gráfica, portanto, mostrou-se uma ferramenta essencial para a governança de

pacotes, orientando políticas de segurança e auditoria que devem focar nos nós mais críticos.

Para **trabalhos futuros**, sugere-se uma análise temporal para modelar a evolução dessas redes, bem como a simulação da propagação de vulnerabilidades a partir dos hubs identificados. Compreender essas dinâmicas é fundamental para garantir a resiliência e a segurança do ecossistema de software do qual todos nós, direta ou indiretamente, dependemos.

VIII. REFERÊNCIAS

[1] Libraries.io, "The Open Source Discovery Service." Disponível em: https://libraries.io. Acesso em: jul. 2025.

[2] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," Empirical Software Engineering, vol. 24, no. 1, pp. 381–416, fev. 2019.

[3] NetworkX Developers, "NetworkX: High-productivity software for complex networks." Disponível em: https://networkx.org/. Acesso em: jul. 2025.

[4] Python Software Foundation, "Python 3.11 Documentation." Disponível em: https://docs.python.org/3/. Acesso em: jul. 2025.

[5] Zenodo, "Libraries.io Open Source Dataset." DOI: 10.5281/zenodo.2536573.

[6] GitHub – samuelc254, "Repositório oficial do projeto CR-UFABC." Disponível em: https://github.com/samuelc254/cr-ufabc. Acesso em: jul. 2025.

[7] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," Phys. Rev. E, vol. 68, no. 4, p. 046116, Oct. 2003, doi: 10.1103/PhysRevE.68.046116.

[8] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," Nature, vol. 406, no. 6794, pp. 378–382, Jul. 2000, doi: 10.1038/35019019.

[9] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," Journal of Statistical Mechanics: Theory and Experiment, vol. 2008, no. 10, p. P10008, Oct. 2008, doi: 10.1088/1742-5468/2008/10/P10008.