UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RAYRISSON VINICIUS ALVES DE LIMA

ANÁLISE COMPARATIVA DE GERENCIADORES DE PACOTES JAVASCRIPT: NPM, PNPM E YARN

QUIXADÁ

2025

RAYRISSON VINICIUS ALVES DE LIMA

ANÁLISE COMPARATIVA DE GERENCIADORES DE PACOTES JAVASCRIPT: NPM, PNPM E YARN

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da computação.

Orientador: Prof. Dr. Jefferson de Carvalho Silva.

QUIXADÁ

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

L71a Lima, Rayrisson Vinicius Alves de.
Análise comparativa de gerenciadores de pacotes JavaScript : NPM, PNPM e Yarn / Rayrisson Vinicius Alves de Lima. – 2025.
63 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2025.
Orientação: Prof. Dr. Jefferson de Carvalho Silva.

1. Gerenciador de pacotes. 2. JavaScript. 3. NPM. 4. Yarn. 5. PNPM. I. Título.

CDD 004

RAYRISSON VINICIUS ALVES DE LIMA

ANÁLISE COMPARATIVA DE GERENCIADORES DE PACOTES JAVASCRIPT: NPM, PNPM E YARN

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da computação.

Aprovada em: 24/07/2025.

BANCA EXAMINADORA

Prof. Dr. Jefferson de Carvalho Silva (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. João Marcelo Uchôa de Alencar
Universidade Federal do Ceará (UFC)

Prof. Dr. Marcio Espíndola Freire Maia
Universidade Federal do Ceará (UFC)

À minha família, pelo apoio incondicional.

AGRADECIMENTOS

Agradeço, em primeiro lugar, a Deus, por ter me concedido forças e sabedoria para concluir mais esta etapa da minha vida.

À minha família, minha eterna gratidão pelo apoio incondicional ao longo de toda a jornada. Em especial, à minha mãe, por todo o esforço, dedicação e pelos ensinamentos que me inspiraram a lutar pelos meus objetivos com coragem e perseverança.

Aos meus amigos de longa data — e também colegas de apartamento — Deyvidson, João Victor, Narcelio, Romes, Ryan e Vito, obrigado por compartilharem tantos momentos marcantes e por tornarem a caminhada mais leve e divertida.

Aos bons amigos que a universidade me proporcionou — Almir, Alysson, Elano, Mariana e Tiago — agradeço pela convivência, companheirismo e pelas inúmeras conversas que tornaram o cotidiano acadêmico mais leve e prazeroso.

Por fim, agradeço sinceramente a todos que, de alguma forma, contribuíram para que eu pudesse chegar até aqui. Cada gesto de apoio, por menor que tenha sido, fez diferença.

RESUMO

Com o crescimento da complexidade no desenvolvimento de aplicações JavaScript modernas, o uso de gerenciadores de pacotes tornou-se essencial para lidar com a instalação, atualização e organização de dependências. Nesse contexto, ferramentas como NPM, Yarn e PNPM surgem como alternativas amplamente adotadas, cada uma com abordagens distintas que impactam diretamente o fluxo de trabalho dos desenvolvedores. Este trabalho tem como objetivo realizar uma análise comparativa entre os gerenciadores de pacotes NPM, PNPM e Yarn, avaliando suas características, abordagens e comportamentos frente a aspectos como desempenho, uso de armazenamento, segurança e compatibilidade. Para isso, foi desenvolvido um projeto prático com elevado número de dependências e conduzida uma série de experimentos em diferentes sistemas operacionais, com foco em métricas objetivas e observações empíricas. A análise proposta visa fornecer um guia prático que auxilie desenvolvedores na escolha da ferramenta mais adequada às necessidades específicas de seus projetos.

Palavras-chave: Gerenciador de pacotes; Dependências; NPM; Yarn; PNPM; JavaScript.

ABSTRACT

With the increasing complexity in the development of modern JavaScript applications, the use of package managers has become essential to handle the installation, updating, and organization of dependencies. In this context, tools such as NPM, Yarn, and PNPM have emerged as widely adopted alternatives, each offering distinct approaches that directly impact developers' workflows. This work aims to perform a comparative analysis of the NPM, PNPM, and Yarn package managers, evaluating their characteristics, strategies, and behavior regarding aspects such as performance, storage usage, security, and compatibility. For this purpose, a practical project with a high number of dependencies was developed, and a series of experiments were conducted on different operating systems, focusing on objective metrics and empirical observations. The proposed analysis seeks to provide a practical guide to help developers choose the tool that best suits the specific needs of their projects.

Keywords: Package manager; Dependencies; NPM; Yarn; PNPM; JavaScript.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

CLI	Command Line Interface
CSS	Cascading Style Sheets
DLL	Dynamic Link Library
HTML	HyperText Markup Language
NPM	Node Package Manager
PNPM	Performant NPM

SUMÁRIO

1 INTRODUÇÃO

O gerenciamento de pacotes é um problema antigo e frequente na computação, usuários de versões antigas do Windows já sofriam com o chamado "inferno de DLL", condição essa que fazia com que a instalação de algum programa tornasse outros inutilizáveis devido à incompatibilidade entre bibliotecas vinculadas dinamicamente (DLL, do inglês *Dynamic Link Library*). Problemas semelhantes aconteciam com desenvolvedores que instalavam manualmente bibliotecas que possuem módulos auxiliares aos seus projetos. Felizmente, esse é um problema antigo, graças aos gerenciadores de pacotes, ferramentas que simplificam a instalação e manutenção de pacotes, padronizando e organizando a produção e consumo dos mesmos (Spinellis, 2012). Um gerenciador de pacotes é uma ferramenta que gerencia as dependências do seu projeto, fornecendo métodos para instalação, remoção e atualização de pacotes. Além disso, permite gerenciar onde os pacotes serão armazenados e oferece recursos para a publicação (MDN Web Docs, 2023a).

O JavaScript é uma linguagem de programação desenvolvida por Brendan Eich em 1995 para o navegador Netscape 2 (W3Schools, 2023). Caracterizada por ser leve, interpretada, baseada em objetos, multi-paradigma e dinâmica é conhecida como a linguagem de *script* para páginas Web, apesar de também poder ser utilizada em ambientes sem navegador (MDN Web Docs, 2022).

Com o passar dos anos e os rápidos avanços tecnológicos, o JavaScript e os navegadores tornaram-se cada vez mais poderosos, tornando também os projetos mais complexos. Devido a essa complexidade e a fim de auxiliar no desenvolvimento, tornando-o mais ágil, bibliotecas ganharam bastante popularidade, pois elas permitem a utilização de uma coleção de recursos, funções e/ou ferramentas desenvolvidas e disponibilizadas por terceiros - desenvolvedores, equipes, empresas e entre outros - que resolvem problemas específicos (MDN Web Docs, 2023a). As bibliotecas também são chamadas de dependências ou pacotes.

Conforme destacado por Spinellis (2012), a gestão meticulosa e bem organizada de pacotes é um componente crucial de um processo de desenvolvimento de software eficaz. A tarefa de instalar dependências tornou-se habitual, porém, também complexa para os desenvolvedores JavaScript. Encontrar os arquivos corretos, verificar a presença de vulnerabilidades e lidar com sub-dependências acabou por se tornar um desafio significativo. Para enfrentar essas questões, surgiram os gerenciadores de pacotes (MDN Web Docs, 2023a). No ecossistema JavaScript, existem várias opções de gerenciadores de pacotes. Neste contexto, exploraremos três das

alternativas mais populares: o NPM[1], o PNPM[2] e o Yarn[3].

O Node Package Manager (NPM) é o gerenciador de pacotes padrão do Node.js, foi criado em 2009 como um projeto de código aberto e lançado em 2010, sendo assim o primeiro gerenciador de pacotes JavaScript (npm, 2023a; Goldwater, 2019). O Yarn foi desenvolvido em 2016 por Sebastian McKenzie e contava com um algoritmo mais rápido para obtenção de dados em cache, acesso *offline* a pacotes já baixados e segurança refinada utilizando um hash gerado a partir de cada pacote (Goldwater, 2019). O Performant NPM (PNPM) foi desenvolvido em 2017 por Zoltan Kochan, seu principal trunfo está no uso de links simbólicos direcionados a um armazenamento global, o que acaba economizando espaço utilizado em disco (pnpm, 2023; Goldwater, 2019). Mesmo com o lançamento do Yarn e do PNPM com novas abordagens e excelentes tempos na instalação de pacotes, o NPM se manteve sendo o gerenciador mais utilizado, de acordo com o NPM Trends (2023).

De acordo com o índice TIOBE (2023), o JavaScript é a sexta linguagem de programação mais popular do mundo. Com essa grande popularidade é natural que exista uma grande quantidade de ferramentas, pacotes e frameworks disponíveis, somente no registro npm existem mais de dois milhões de pacotes, sendo assim o maior registro de software do mundo (npm, 2023b). Esse grande catálogo de possibilidades acaba acarretando em um não consenso da comunidade sobre qual utilizar em seu projeto. Esse problema acaba acontecendo também com os gerenciadores de pacotes, apesar do NPM ser o gerenciador mais utilizado pela comunidade, Chodorowski (2021) prova que o Yarn obtém melhores resultados em alguns testes, trazendo novamente questionamentos na hora de definir qual gerenciador escolher.

Portanto, este trabalho visa analisar e comparar de forma abrangente esses três gerenciadores de pacotes - NPM, PNPM e Yarn -, nas suas versões mais recentes, explorando suas características, desempenhos e vantagens individuais. Ao compreender as particularidades de cada um, será possível fornecer um guia prático para auxiliar os desenvolvedores na escolha do gerenciador de pacotes mais adequado para seus projetos.

[1] NPM - https://www.npmjs.com/
[2] PNPM - https://pnpm.io/
[3] Yarn - https://yarnpkg.com/

1.1 Objetivos

1.1.1 Objetivos gerais

Comparar os gerenciadores de pacotes JavaScript, analisando as abordagens utilizadas, afim de entender suas vantagens e desvantagens e como elas afetam questões como performance, armazenamento em disco, segurança e compatibilidade.

1.1.2 Objetivos específicos

- Comparar e avaliar a abordagem adotada por cada gerenciador de pacotes.
- Comparar e avaliar a performance de cada gerenciador de pacotes.
- Comparar e avaliar o armazenamento em disco de cada gerenciador de pacotes.
- Comparar e avaliar a segurança de cada gerenciador de pacotes.
- Comparar e avaliar a compatibilidade de cada gerenciador de pacotes.

Os próximos capítulos deste trabalho estão organizados da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica, abordando os principais conceitos relacionados a gerenciadores de pacotes, seu papel no ecossistema JavaScript e as tecnologias envolvidas. O Capítulo 3 trata dos trabalhos relacionados, descrevendo e comparando estudos e projetos que analisam ferramentas semelhantes às avaliadas neste trabalho. No Capítulo 4, é apresentada a metodologia utilizada para a realização dos testes comparativos, com a descrição das ferramentas, métricas e procedimentos adotados. O Capítulo 5 expõe os experimentos realizados e os resultados obtidos a partir das análises práticas. Por fim, o Capítulo 6 apresenta as conclusões, destacando as principais observações do estudo e possíveis direções para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os conceitos necessários para o desenvolvimento deste trabalho. Na Seção 2.1 é apresentada a história e características do JavaScript. Na Seção 2.2 são apresentados e discutidos conceitos e problemas relacionados ao uso de dependências. Por último, na Seção 2.3 é apresentado o funcionamento e uma breve descrição dos principais gerenciadores de pacotes que serão utilizados nesta pesquisa.

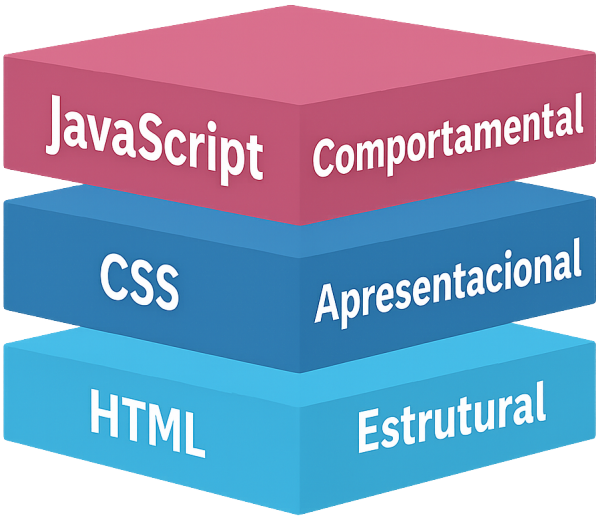
2.1 JavaScript

Em 1995, a Web e os navegadores estavam em estágios iniciais e a Netscape Communications Corporation era uma das principais impulsionadoras dessa evolução, liderando o desenvolvimento de navegadores Web. Dentro deste contexto, o JavaScript foi concebido e desenvolvido por Brendan Eich, em maio de 1995 e anunciado pela Netscape Communications Corporation e a Sun Microsystems em 4 de dezembro de 1995. O JavaScript foi criado com o propósito de ser uma linguagem simples, de fácil utilização e dinâmica, que possibilitasse a inserção de trechos de código diretamente nas definições das páginas web, para que fossem interpretados na fase de renderização permitindo que a apresentação das páginas pudesse ser personalizada de forma dinâmica e respondesse às interações do usuário. Foi inicialmente planejada para operar ao lado do Java nos navegadores, mas rapidamente ultrapassou seu companheiro, tornando-se a principal linguagem para páginas Web - e não somente para páginas Web (Wirfs-Brock; Eich, 2020).

O JavaScript é uma linguagem de programação leve, interpretada, dinâmica, baseada em protótipos e multi-paradigma, conhecida como a linguagem de *script* para páginas Web, mas que atualmente também pode ser utilizada em vários ambientes sem navegador, como por exemplo, Node.js[1], Deno[2] e Bun[3] (MDN Web Docs, 2022). É a terceira camada da pilha das tecnologias padrões da Web, juntamente com o HyperText Markup Language (HTML) e o Cascading Style Sheets (CSS) (MDN Web Docs, 2023b). Essas camadas estão representadas na Figura 1.

[1] Node.js - https://nodejs.org/
[2] Deno - https://deno.com/
[3] Bun - https://bun.sh/

Figura 1 – Pilha das tecnologias padrões da web



Fonte: Rathore (2020)

2.1.1 ECMAScript

ECMAScript é a especificação padrão oficial do JavaScript, desenvolvida e mantida pela Ecma International[4], tendo sua primeira versão emitida em 1997 (Wirfs-Brock; Eich, 2020). Essa especificação é responsável por definir quais funcionalidades, recursos e/ou correções serão adicionadas posteriormente aos motores de compilação JavaScript (Jacobs, 2019).

Suas três primeiras edições foram lançadas respectivamente em junho de 1997, agosto de 1998 e dezembro de 1999. O desenvolvimento da 4ª edição foi abandonando após 10 anos de tentativas frustradas. Após anos sem nenhuma nova edição, em dezembro de 2009 foi lançada a 5ª versão e em junho de 2011 tivemos uma edição 5.1 (Wirfs-Brock; Eich, 2020; International, 2023). Em junho de 2015 foi lançada sua 6ª edição, também chamada de *EcmaScript 2015*, esta edição foi extremamente impactante, pois adicionou diversos novos recursos, como *arrow functions*, *Promises*, desestruturação e o tão importante suporte nativo a módulos, adicionando ao JavaScript uma forma padrão de consumir e empacotar código para reutilização (International, 2015). Na Figura 2, temos um exemplo da utilização do sistema de módulos implementado na 6ª edição do *ECMAScript* sendo utilizado para importação e exportação.

Desde então, uma nova edição do *ECMAScript* é lançada todo ano, sendo a 14ª edição a mais recente, tendo sido lançada em junho de 2023 (International, 2023).

[4] Ecma International - https://www.ecma-international.org/

Figura 2 – Exemplo de importação e exportação utilizando o sistema de módulos ES6

Fonte: Elaborado pelo autor

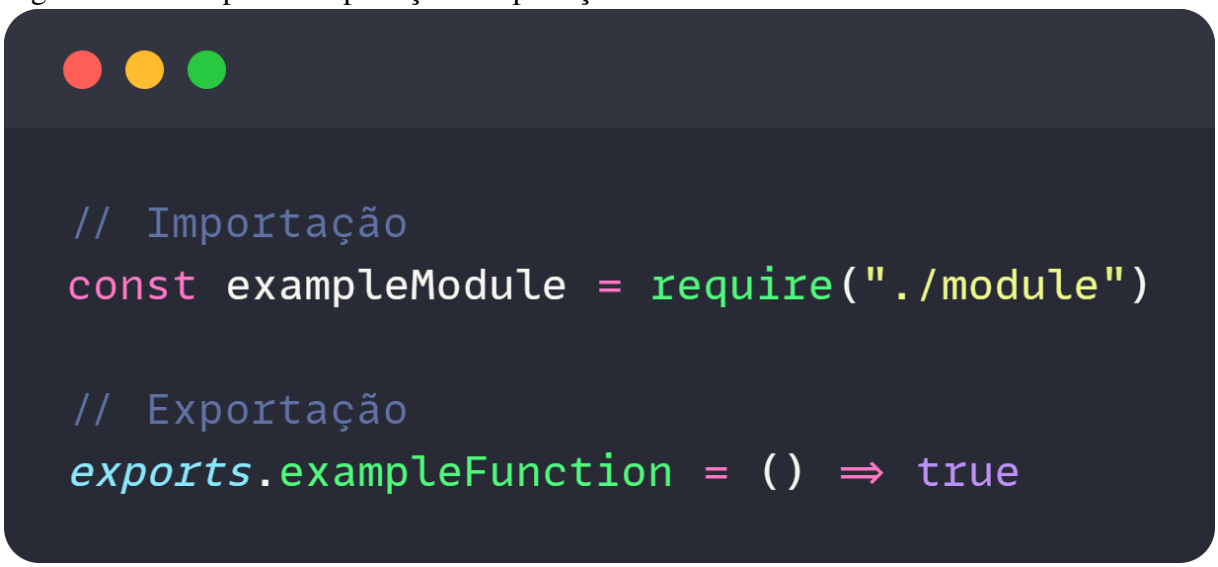
2.1.2 Node.js

O Node.js foi desenvolvido por Ryan Dahl em 2009 e concebido inicialmente como uma plataforma de código aberto baseado em JavaScript para construção de aplicações que rodam no lado do servidor, também chamadas de aplicações *server-side* (Wirfs-Brock; Eich, 2020).

O Node.js é baseado no motor V8 do navegador Google Chrome[5], uma ferramenta de alto desempenho que compila e executa código-fonte JavaScript (V8, 2018). O Node.js é *single-thread* e orientado a eventos, utiliza de loop de eventos para executar operações de E/S sem bloqueios - mesmo sendo *single-thread* - delegando a responsabilidade sobre estas operações para o kernel do sistema sempre que possível. Implementa também um conjunto de módulos integrados e um carregador de módulos *CommonJS* - posteriormente também ofereceria suporte aos módulos nativos introduzidos no *EcmaScript 2015* (Wirfs-Brock; Eich, 2020). Um módulo *CommonJS* é essencialmente um bloco de código JavaScript encapsulado em uma função, cujo escopo abrange várias associações que possibilitam a interação do código com outros módulos. Isso é viabilizado por meio de um carregador de módulos síncrono, que busca o código-fonte de um módulo, envolve esse código com uma definição de função esqueleto e, posteriormente, analisa e invoca a função sintetizada para inicializar o módulo e estabelecer suas conexões com outros módulos (Wirfs-Brock; Eich, 2020). Na Figura 3, temos exemplos de importação e exportação utilizando o sistema de módulos *CommonJS*.

[5] Google Chrome - https://www.google.com/intl/en-US/chrome/

Figura 3 – Exemplo de importação e exportação utilizando o sistema de módulos CommonJS



Fonte: Elaborado pelo autor

Apesar de ser concebido como uma tecnologia para construção de aplicações *server-side*, o Node.js tornou-se uma plataforma que permitia que o JavaScript fosse utilizado como uma linguagem de uso geral nas mais diversas plataformas. Permitindo, desta forma, que desenvolvedores que utilizavam JavaScript em clientes web, utilizassem seus conhecimentos e habilidades em outros aplicativos e ambientes que não fossem o navegador (Wirfs-Brock; Eich, 2020).

2.2 Dependência

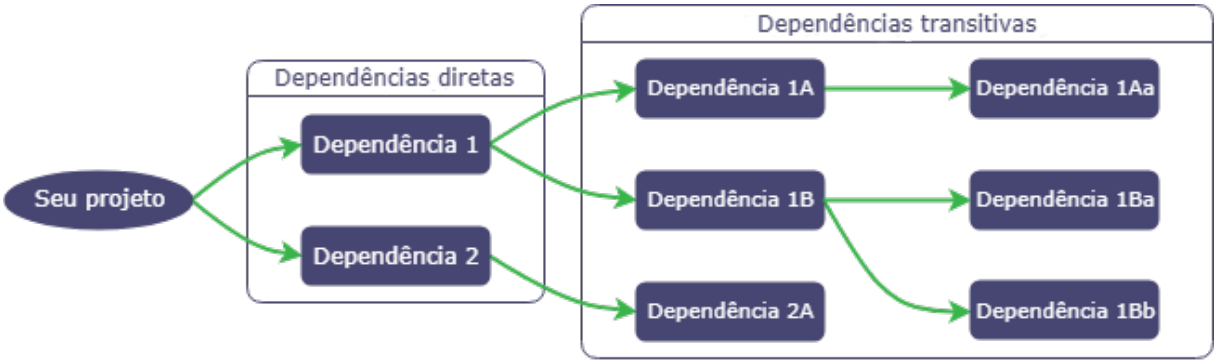
Durante anos, a discussão sobre reuso de software era mais comum do que o seu reuso de fato, situação muito diferente do que ocorre atualmente, na qual desenvolvedores utilizam códigos de terceiros todos os dias, na forma de dependências (Cox, 2019). Segundo Cox (2019), uma dependência é um código suplementar que o programador deseja invocar, a sua utilização evita que o desenvolvedor tenha que projetar, escrever, testar, depurar e manter um código que já foi feito anteriormente. Essa unidade de código também pode ser chamada de pacote, biblioteca ou módulo.

Um pacote - ou dependência -, contém o código fonte ou compilado de um software, e também seus metadados, que incluem informações importantes relacionadas aquele código, como nome, descrição, versão, dependências - sim, dependências também podem possuir dependências -, fornecedor, licença e especificações de instalação (Spinellis, 2012).

Existem dois tipos de dependências, as diretas e as transitivas. Dependências diretas

são os pacotes utilizados explicitamente pelo seu código. Já as dependências transitivas são os pacotes utilizados pelas dependências diretas do projeto, ou seja, dependências das dependências (Snyk, 2023). Uma representação de como as dependências se comportam é apresentada na Figura 4.

Figura 4 – Árvore de dependências



Fonte: Ion Channel (2023).

A principal vantagem na utilização de dependências está na rapidez que ela proporciona na construção de software, mas existem alguns riscos na sua utilização. Ao utilizar uma dependência, o desenvolvedor terceiriza a responsabilidade de desenvolvimento daquele código para outra pessoa, muitas das vezes desconhecida. A execução da aplicação agora depende de um completo desconhecido, podendo trazer riscos a segurança da aplicação (Cox, 2019; Snyk, 2023).

Além dos possíveis problemas de segurança, as dependências também podem conter bugs ou ficarem desatualizadas, casos que podem afetar o desempenho da aplicação. A utilização excessiva de dependências também pode afetar o tamanho do *bundle* da aplicação. Devido a estas questões é importante sempre que possível checar a procedência do pacote utilizado, verificando se existe uma documentação clara, se o código está bem escrito, se possui testes, se o pacote está sendo mantido e atualizado - histórico de *commits* é uma boa forma de validação -, a popularidade, entre diversos outros métodos de checagem de qualidade de um pacote (Cox, 2019; Snyk, 2023).

2.2.1 Inferno de dependências

A medida que a quantidade de dependências aumenta em um projeto, a dificuldade em lidar com essa complexa rede de dependências também aumenta. O inferno de dependências ocorre quando em um projeto existem vários pacotes que possuem dependências compartilhadas,

mas necessitam de versões diferentes das mesmas (Pedroso, 2022). O inferno de dependências pode assumir várias formas e ocorrer de diversas maneiras, sendo algumas delas apresentadas a seguir.

2.2.1.1 Muitas dependências

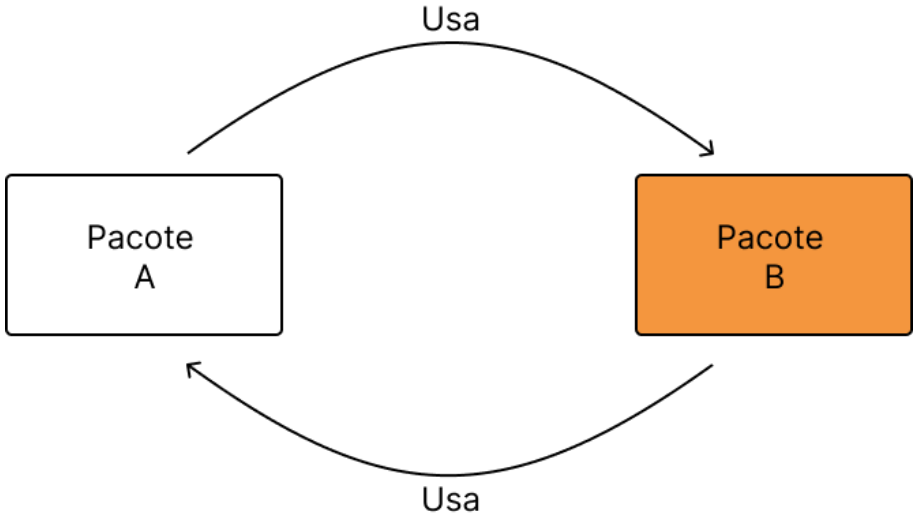
O uso excessivo de dependências pode acarretar diversos problemas, como downloads demorados, compilações lentas, repositórios enormes e diminuição da portabilidade (Rickard, 2021; Pedroso, 2022). Ressaltando também que mesmo que em um projeto seja utilizado somente uma pequena parte do pacote, ele ainda assim será completamente baixado (Pedroso, 2022). Por isso é sempre importante ponderar se de fato o pacote é necessário para o seu projeto.

2.2.1.2 Dependências circulares

O problema das dependências circulares ocorre quando dois pacotes distintos dependem um do outro para funcionar de forma adequada. Geralmente surgem de divisão excessiva de pacotes, em situações em que dois componentes foram separados logicamente, mas na verdade deviam estar juntos (Rickard, 2021).

Na Figura 5, temos uma exemplo de como ocorre o problema das dependências circulares. Nela é possível notar que o pacote A depende do pacote B e o pacote B também depende do pacote A.

Figura 5 – Problema de dependências circulares



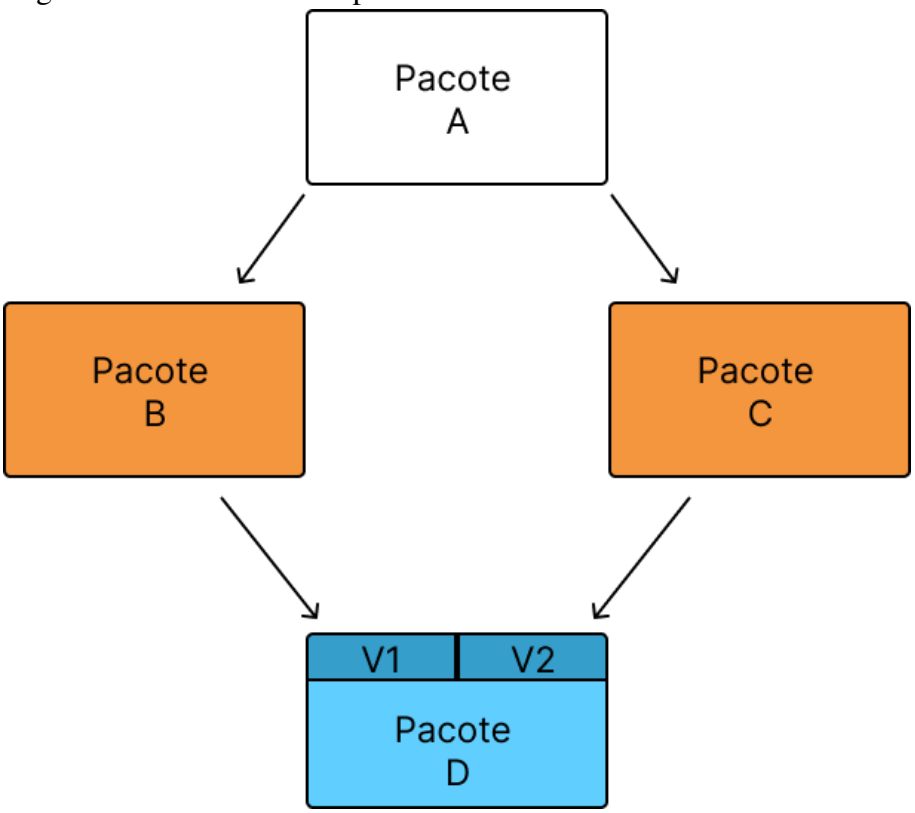
Fonte: Elaborado pelo autor.

2.2.1.3 Dependência de diamantes

O problema da dependência de diamantes ocorre quando dois ou mais pacotes dependem de versões diferentes de um terceiro pacote. Ele é considerando para a maior parte das implementações um problema NP-Completo, ou seja, pode levar muito tempo para ser resolvido (Rickard, 2021).

Por exemplo, existe um pacote A que depende dos pacotes B e C. Tanto o pacote B quanto o C dependem do pacote D, mas B requer D na versão 1 enquanto C requer D na versão 2. Este exemplo está representado na Figura 6.

Figura 6 – Problema de dependência de diamantes



Fonte: Elaborado pelo autor.

Existem algumas maneiras de resolver este problema. Uma delas é simplesmente permitir o *download* e a utilização de versões diferentes do pacote D, sendo este método utilizado na maior parte dos gerenciadores de pacotes JavaScript. A sua desvantagem é que o projeto pode acabar com uma grande quantidade de cópias do mesmo pacote, apenas em versões diferentes. Outras linguagens, como Go, permitem apenas definir a versão mínima de um pacote. Se mais de uma versão for detectada, então será realizada uma tentativa de utilização da versão *minor* mais recente de acordo com o versionamento semântico. Caso nenhum dos pacotes compartilhem de

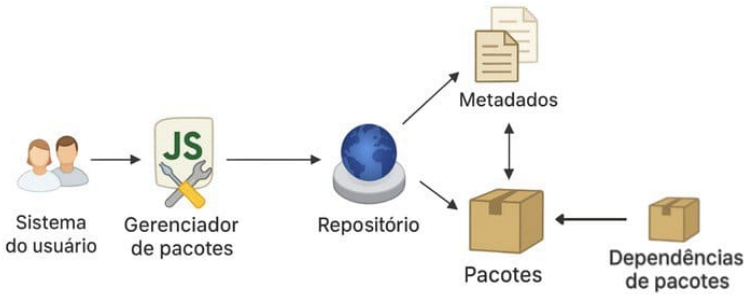
uma mesma versão *major*, então acontecerá um erro (Rickard, 2021).

2.3 Gerenciador de pacotes JavaScript

Gerenciador de pacotes é uma ferramenta que facilita a instalação, manutenção e o gerenciamento de dependências de um projeto, software ou sistema, buscando resolver o problema do inferno de dependências. É responsável por padronizar e organizar a produção e o consumo de coleções de softwares, além de simplificar a utilização, buscando e instalando automaticamente o pacote necessário e também as suas dependências (Spinellis, 2012; MDN Web Docs, 2023a).

Os gerenciadores de pacotes acessam repositórios que hospedam todos os pacotes, que são devolvidos para o sistema do usuário juntamente com seus metadados e suas dependências (openSUSE, 2010; Spinellis, 2012). Na Figura 7, temos a representação do comportamento dos gerenciadores de pacotes.

Figura 7 – Comportamento dos gerenciadores de pacotes



Fonte: openSUSE (2010)

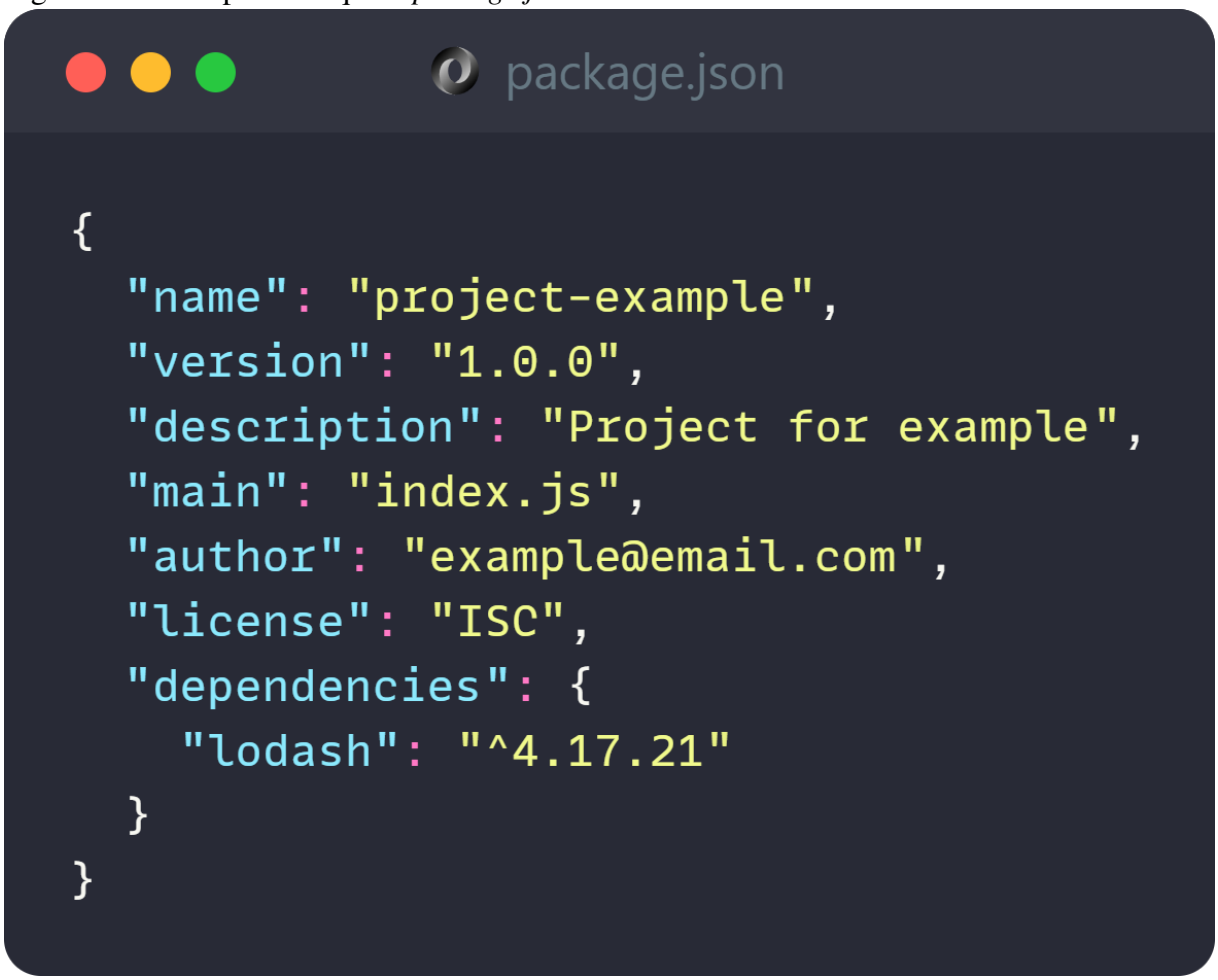
Estas ferramentas se tornaram bastante populares no ecossistema JavaScript, principalmente após a popularização do Node.js, que mais tarde introduziu ao seu ecossistema o NPM, o primeiro gerenciador de pacotes JavaScript (Goldwater, 2019).

A interação com a maioria dos gerenciadores de pacotes JavaScript é realizada através de uma interface de linha de comando (CLI, do inglês *Command Line Interface*), que permite que o desenvolvedor realize todas as operações necessárias, como inicialização, instalação, atualização, remoção e entre outras (Jacobs, 2019).

Ao inicializar um projeto JavaScript com qualquer um dos principais gerenciadores de pacotes é criado um arquivo, chamado *package.json* (PNPM, 2019; Jacobs, 2019). O arquivo *package.json* contém informações do projeto como nome, versão, descrição, licença, autor e

dependências (Jacobs, 2019). Na Figura 8, é mostrado um exemplo de como se parece o arquivo *package.json*.

Figura 8 – Exemplo de arquivo *package.json*



Fonte: Elaborado pelo autor.

Ao realizar a primeira operação de instalação de dependências é criado um arquivo *lockfile*, que possui nomes diferentes para cada gerenciador. Este arquivo lista todas as dependências do projeto, juntamente com suas respectivas versões exatas, com o intuito de garantir que o gerenciador sempre baixe a versão correta do pacote (Jacobs, 2019).

Dentro do ecossistema JavaScript existem diversos gerenciadores de pacotes JavaScript com características próprias disponíveis, a seguir serão apresentados três das soluções mais utilizadas: NPM, Yarn e PNPM.

2.3.1 NPM

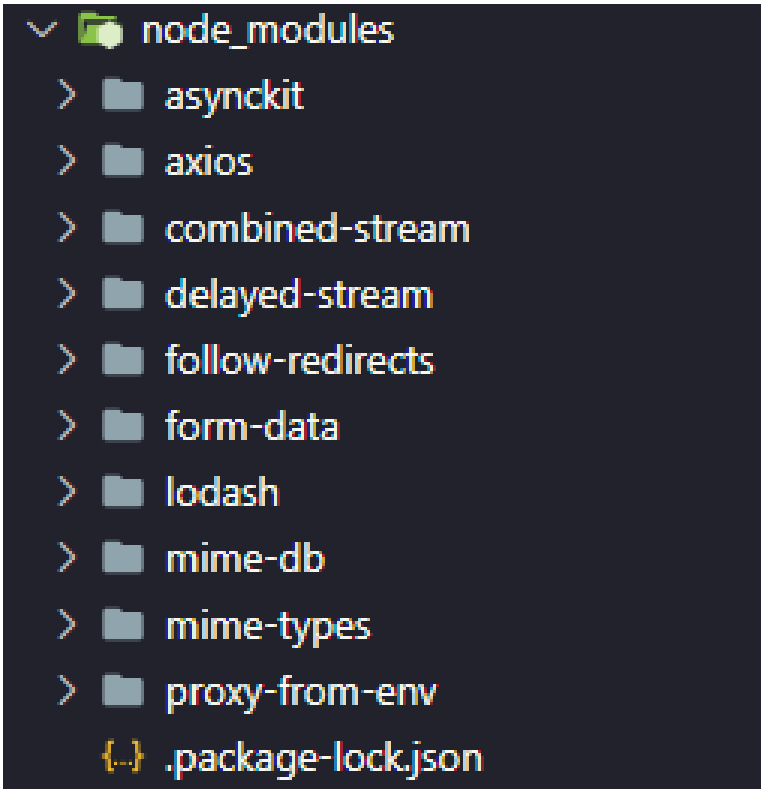
O NPM é o gerenciador de pacotes padrão do Node.js, foi criado em 2009 como um projeto de código aberto e lançado em 2010, sendo assim o primeiro gerenciador de pacotes

JavaScript. Atualmente é mantido pela NPM Inc., empresa fundada em 2014 e comprada em 2020 pelo GitHub (npm, 2023a; Goldwater, 2019).

Além do gerenciador de pacotes, a NPM Inc. também é responsável por manter o registro NPM, uma coleção pública de pacotes JavaScript, que atualmente possui mais de 2 milhões de pacotes, sendo assim o maior registro de software do mundo (npm, 2023a; npm, 2023b). O NPM usa este registro por padrão, mas permite a utilização de qualquer outro registro compatível.

No NPM, as dependências são baixadas em uma pasta chamada *node_modules* que fica localizada na raiz do projeto. Ao instalar um pacote, ele é carregado no cache e descompactado em uma pasta com o seu nome na raiz de *node_modules*. Na Figura 9 é possível ver a estrutura da pasta *node_modules*, onde cada pasta representa uma dependência direta e/ou transitiva do projeto (npm Docs, 2023a).

Figura 9 – Estrutura da pasta *node_modules* no NPM

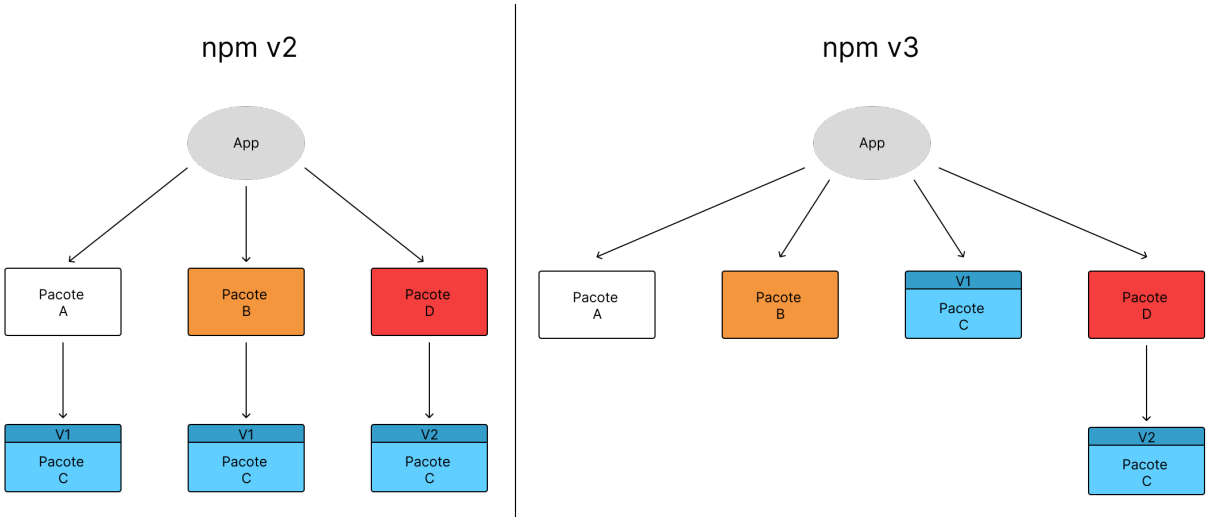


Fonte: Elaborado pelo autor.

Desde a versão 3, o NPM eleva as dependências por padrão, tentando sempre instalar as dependências, sejam elas diretas ou transitivas, no nível mais alto possível. A instalação usa a propriedade do sistema de módulos do node, que percorre os diretórios em busca de pastas *node_modules*, assim, se a dependência já estiver instalada em outra pasta *node_modules*

ancestral e for a mesma versão requisitada, ela não será instalada novamente (npm Docs, 2023a). A Figura 10 mostra como eram salvas as dependências na versão 2 e como passaram a ser salvas a partir da versão 3, exemplificando a elevação de dependências citada anteriormente.

Figura 10 – Comparação entre o comportamento do NPM na versão 2 e comportamento que passou a ser utilizado a partir versão 3



Fonte: Elaborado pelo autor.

O NPM tem um desenvolvimento bastante ativo e atualmente se encontra na versão 11.4.2, lançada em 12 de junho de 2025 (NPM, 2018).

2.3.1.1 Instalação

Como o NPM é o gerenciador de pacotes padrão do Node.js, a sua instalação consiste basicamente na instalação do Node.js (npm Docs, 2023b).

Para instalação do Node.js é necessário entrar em seu site oficial[6] e baixar a versão referente ao seu sistema operacional ou utilizar um gerenciador de versões de Node, que permite a instalação e alternância entre diversas versões do Node.js (npm Docs, 2023b).

2.3.1.2 Interface de linha de comando

A interação com o NPM é realizada através do CLI, no Quadro 1 temos exemplos de alguns comandos básicos do NPM.

[6] Node.js - https://nodejs.org/

Quadro 1 – Comandos básicos NPM

Comando	**Descrição**
`npm init`	Inicialização do projeto
`npm install`	Instalação de todos os pacotes do projeto
`npm install nome-do-pacote`	Instalação de novo(s) pacote(s)
`npm update`	Atualização de pacote(s)
`npm uninstall`	Desinstalação de pacote(s)
`npm audit`	Auditoria de pacotes (busca de vulnerabilidades de segurança)

Fonte: Elaborado pelo autor.

2.3.2 Yarn

O Yarn foi desenvolvido em 2016 por Sebastian McKenzie, funcionário do Facebook - atualmente, conhecido como Meta -, em colaboração com Exponent, Google e Tilde (Nakazawa *et al.*, 2016). Surgiu como uma forma do Meta lidar com os constantes problemas de consistência, segurança e desempenho que tinham ao utilizar o NPM (Nakazawa *et al.*, 2016). O Meta continua investindo na empresa, mas atualmente, o projeto está em sua própria organização no GitHub e a equipe é composta exclusivamente por pessoas não relacionadas as empresas fundadoras (Yarn, 2023d).

A fim de resolver os problemas enfrentados pelo Meta, o Yarn contava com um algoritmo mais rápido para obtenção de dados em cache, acesso offline a pacotes já baixados, arquivo *lockfile* para consistência de dependências e segurança refinada utilizando um hash gerado a partir de cada pacote (Yarn, 2016; Goldwater, 2019). Soluções estas que, a partir da versão 5, também seriam utilizadas pelo NPM (Goldwater, 2019).

Em 2020, o Yarn recebeu uma grande atualização para a versão 2, uma nova versão que reestruturaria o design do Yarn. Essa nova versão e todas as seguintes foram nomeadas de Yarn Modern, enquanto isso a versão 1 - renomeada, como Yarn Classic -, que continua sendo um pilar do ecossistema JavaScript, devido a grande quantidade de projetos que a utilizam, entrou em modo de manutenção, recebendo atualizações somente no caso de existência de vulnerabilidades (Nison, 2020; Yarn, 2023d).

O Yarn Modern foi escrito em TypeScript, oferece suporte a plugins e a *workspaces* (Yarn, 2019). Além destas e outras características, a versão moderna do Yarn, oferece suporte a três estratégias de instalação de dependências, sendo elas: *node_modules*, Yarn PnP e cache endereçado a conteúdo (Yarn, 2023a).

O *node_modules* é a estratégia descrita anteriormente, na subseção 2.3.1, que é utilizada pelo NPM. O cache endereçado a conteúdo é uma estratégia semelhante a utilizada pelo

PNPM, que utiliza links simbólicos e físicos para um armazenamento endereçável de conteúdo global - esta abordagem será explicada com mais detalhes na subseção 2.3.3. O Yarn Plug'n'Play - ou, Yarn PnP -, é a estratégia de instalação de dependências padrão das versões modernas do Yarn, nele é gerado um arquivo carregador com o nome de *.pnp.cjs* que contém todas as informações sobre a árvore de dependências do projeto, incluindo a localização dos pacotes no disco e a forma de resolver chamadas de importação - *require* e *import* (Yarn, 2023c). Na Figura 11, é apresentado um trecho do arquivo *.pnp.cjs* que contém informações do projeto, incluindo a localização em disco das dependências.

Figura 11 – Trecho do arquivo *.pnp.cjs*

Fonte: Elaborado pelo autor.

O Yarn Classic atualmente está em modo de manutenção na versão 1.22.21, tendo sido lançada em 14 de novembro de 2023 (Yarn, 2016). Já o Yarn Modern se encontra na versão 4.9.2, lançada em 03 de junho de 2025 (Yarn, 2019).

2.3.2.1 Instalação

A maneira recomendada para instalação do Yarn Modern é através do *Corepack*, uma ferramenta para controle de versão dos gerenciadores de pacotes, fornecida por padrão pelo Node.js (Yarn, 2023b). Para habilitar o *Corepack*, é só utilizar o comando mostrado na Figura 12.

Figura 12 – Comando para ativação do *Corepack*

```
# Ativando corepack
corepack enable

# Inicializando projeto com Yarn Modern
yarn init -2
```

Fonte: Elaborado pelo autor.

2.3.2.2 Interface de linha de comando

No Quadro 2, são apresentados alguns comandos básicos que podem ser utilizados através do Yarn Classic e do Yarn Modern.

Quadro 2 – Comandos básicos Yarn

Comando		**Descrição**
Yarn Classic	**Yarn Modern**	
`yarn init`	`yarn init`	Inicialização do projeto
`yarn install`	`yarn install`	Instalação de todos os pacotes do projeto
`yarn add nome-do-pacote`	`yarn add nome-do-pacote`	Instalação de novo(s) pacote(s)
`yarn up`	`yarn up`	Atualização de pacote(s)
`yarn remove`	`yarn remove`	Desinstalação de pacote(s)
`yarn audit`	`yarn npm audit`	Auditoria de pacotes (busca de vulnerabilidades de segurança)

Fonte: Elaborado pelo autor.

2.3.3 PNPM

O PNPM foi lançado em 2017 por Zoltan Kochan, com a proposta de resolver problemas causados pela abordagem simples com *node_modules* utilizada pelo NPM e pelo Yarn Classic (Kochan, 2017; Goldwater, 2019).

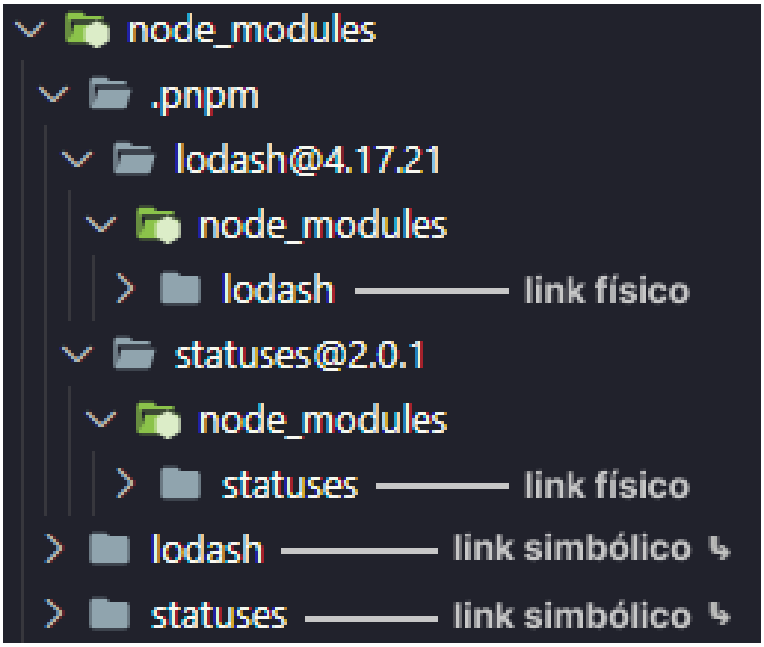
De acordo com Kochan (2017), a árvore de dependências achatada utilizada pela

abordagem simples com *node_modules*, possuía problemas como: permissão para que módulos acessassem pacotes que não dependiam, algoritmo para nivelamento de árvore de dependências - responsável pela elevação de dependências - complexo e a necessidade de alguns pacotes serem copiados para dentro da pasta *node_modules* de uma dependência específica.

A fim de resolver estes problemas, o PNPM utiliza links simbólicos e físicos para um armazenamento endereçável de conteúdo global, ou seja, dependências são baixadas globalmente e depois são somente referenciadas no projeto. Esta abordagem funciona de forma adequada devido à natureza do Node.js, que ignora links simbólicos e executa o caminho real (Kochan, 2017).

Ao inicializar um projeto com PNPM, é criada uma pasta *node_modules*. Na raiz de *node_modules* existe uma pasta *.pnpm* e os links simbólicos - que estão vinculados a pastas que estão em *.pnpm* - de todas as dependências diretas. Dentro da pasta *.pnpm*, existe uma pasta com o nome de cada dependência - direta ou transitiva - e sua respectiva versão, e subpastas *node_modules* dentro de cada uma delas. Cada arquivo dentro destas subpastas representa um link físico para o pacote necessário (PNPM, 2021). Na Figura 13, existe um exemplo da estrutura da pasta *node_modules* no PNPM.

Figura 13 – Estrutura da pasta *node_modules* no PNPM



Fonte: Elaborado pelo autor.

Atualmente o PNPM possui desenvolvimento ativo, tendo sua última versão, a 10.12.1, sido lançada em 08 de junho de 2025 (PNPM, 2016).

2.3.3.1 Instalação

O PNPM pode ser instalado de diversas formas, sendo elas, através de *scripts* independentes, *Corepack*, NPM, *Homebrew*, *winget*, *Scoop*, *Chocolatey* e *Volta* (PNPM, 2023). Na Figura 14, temos um exemplo de como instalar o PNPM, através do NPM.

Figura 14 – Comando para instalação do PNPM

Fonte: Elaborado pelo autor.

2.3.3.2 Interface de linha de comando

No Quadro 3, temos exemplos de alguns comandos básicos utilizados através do PNPM.

Quadro 3 – Comandos básicos PNPM

Comando	**Descrição**
`pnpm init`	Inicialização do projeto
`pnpm install`	Instalação de todos os pacotes do projeto
`pnpm add nome-de-pacote`	Instalação de novo(s) pacote(s)
`pnpm update`	Atualização de pacote(s)
`pnpm remove`	Desinstalação de pacote(s)
`pnpm audit`	Auditoria de pacotes (busca de vulnerabilidades de segurança)

Fonte: Elaborado pelo autor.

3 TRABALHOS RELACIONADOS

Neste capítulo são apresentados trabalhos que contribuem para o desenvolvimento desta pesquisa.

3.1 Comparison of JavaScript package managers (Jacobs, 2019)

Jacobs (2019) analisa três dos mais proeminentes gerenciadores de pacotes JavaScript, o NPM, o PNPM e o Yarn. Este trabalho tem como objetivo avaliar cada gerenciador individualmente e em seguida compará-los, observando seus benefícios, desvantagens e potenciais problemas de segurança.

O NPM e o Yarn foram escolhidos devido a terem uma participação maior no mercado em comparação com as outras opções disponíveis. Já o PNPM, foi escolhido devido as suas gritantes diferenças de design quando comparado com as soluções anteriores.

Dois métodos foram utilizados por Jacobs (2019) para o desenvolvimento deste trabalho, primeiramente foi elaborada e realizada uma pesquisa com desenvolvedores JavaScript, a fim de entender quais experiências eles tem tido com os gerenciadores de pacotes que utilizam. O segundo método utilizado foi a realização de testes para monitorar o desempenho de cada gerenciador de pacotes.

A pesquisa consistiu em um formulário com 19 perguntas relacionadas ao uso de JavaScript com gerenciadores de pacotes e foi respondido por 39 pessoas, como o tamanho da amostra foi limitado, os resultados obtidos não podem ser utilizados para generalizar a comunidade.

O teste de desempenho foi realizado em três sistemas operacionais - *Windows 10 Education 10.0.17763 Build 17763, Ubuntu Linux 19.04 e MacOS 10.11 El Capitan* - utilizando as mesmas especificações de hardware em condições semelhantes.

Com base nos dados coletados, Jacobs (2019) percebe como os gerenciadores de pacotes se tornaram essenciais no mundo do desenvolvimento JavaScript. Os testes de desempenho indicam que exceto em certas circunstâncias, nenhum dos gerenciadores foi claramente mais rápido que o outro.

Em conclusão, Jacobs (2019) cita que embora cada gerenciador tenha realizado um esforço para melhorar o desempenho, aparentemente não existe nenhum benefício significativo ao escolher um ao invés de outro.

O presente trabalho, assim como Jacobs (2019), busca analisar e comparar os gerenciadores de pacotes anteriormente citados. Entretanto, dessa vez, realizando este processo em suas versões mais recentes, atualizando assim a pesquisa, e também utilizando mais métricas comparativas, como abordagem de gerenciamento de dependências e armazenamento em disco.

3.2 Comparative analysis of JavaScript package managers - yarn and npm (Chodorowski, 2021)

Chodorowski (2021) afirma que apesar de parecer uma questão insignificante, a escolha de um gerenciador de pacotes adequado poderá economizar segundos em operações realizadas com frequência durante um período longo, o que poderá resultar em um ganho de horas que poderiam ser utilizadas programando.

A fim de ajudar nessa decisão é proposta uma análise comparativa para as duas soluções líderes para gerenciamento de pacotes no ambiente JavaScript, o NPM e o Yarn. Chodorowski (2021) parte da hipótese que apesar do NPM ser o gerenciador mais utilizado, o Yarn é a melhor escolha quando é necessário eficiência de tempo em projetos com grande quantidade de pacotes.

O estudo é realizado levando em consideração critérios associados a instalação, desempenho, segurança, popularidade, comandos CLI, *logs* e espaço em disco. Analisando e descrevendo cada uma das características de cada gerenciador individualmente e comparando-as.

Para testar a eficiência, foram utilizados dois projetos Angular, um contendo somente as dependências básicas e outro contendo 100 dependências adicionais aleatórias. Foram utilizados os comandos de instalação e atualização, sendo cada comando rodado 100 vezes em cada projeto, totalizando um total de 800 utilizações.

Os resultados obtidos, indicam que o Yarn é quase três vezes mais eficiente que o NPM na instalação em projetos pequenos, e pouco mais de duas vezes e meia mais eficiente em grandes projetos. Já o NPM se provou mais eficiente em atualizações, tanto em pequenos projetos, onde foi mais de três vezes mais eficiente, quanto em grandes projetos, onde foi mais de duas vezes mais eficiente.

Por fim, Chodorowski (2021) conclui que apesar do NPM ser mais popular, o Yarn é a melhor escolha em termos de eficiência de tempo para projetos com grande número de pacotes, confirmando assim a hipótese proposta.

Chodorowski (2021) assim como o presente trabalho busca de forma similar analisar

e comparar o Yarn e o NPM. Entretanto, o trabalho proposto utiliza as versões mais recentes dos mesmos e adiciona mais um gerenciador ao comparativo, o PNPM.

3.3 On the impact of security vulnerabilities in the npm package dependency network (Decan *et al.*, 2018)

Decan *et al.* (2018) destaca que as vulnerabilidades de segurança em *softwares* estão entre os problemas mais impactantes na comunidade de desenvolvimento. Isso se dá, devido a crescente dependência de *softwares* de terceiros e os impactos potencialmente devastadores que essas vulnerabilidades de segurança podem causar.

O objetivo do estudo era realizar um estudo empírico da propagação de vulnerabilidades de segurança e suas correções no histórico de distribuição do NPM.

Para a realização do estudo foi utilizado um conjunto de dados fornecidos pelo serviço de monitoramento contínuo do Snyk.io, que contém uma lista de relatórios de segurança de diversos gerenciadores de pacotes, incluindo o NPM. Foram utilizados 399 relatórios de segurança, afetando 269 pacotes distintos e 6752 versões desses pacotes.

Após a realização do estudo, Decan *et al.* (2018) observa que na maior parte dos casos as vulnerabilidades levam muito tempo para serem descobertas. Um terço das vulnerabilidades são corrigidas antes da data de descoberta, metade das vulnerabilidades demoram mais, porém são corrigidas antes da data de publicação, enquanto o restante delas são consideradas de alto risco, pois são corrigidas somente após a publicação do pacote, e em alguns casos nem são corrigidas.

Decan *et al.* (2018) também destaca a importância da conscientização dos mantenedores de pacotes sobre os riscos causados pelas vulnerabilidades de segurança, riscos estes que não afetam somente o pacote, mas também todo o ecossistema, levando em consideração as dependências de outros pacotes. Decan *et al.* (2018) recomenda que os mantenedores façam uso de políticas de segurança e ferramentas automatizadas de detecção de vulnerabilidades, para que elas possam ser corrigidas mais rapidamente, assim reduzindo o seu impacto em pacotes dependentes.

Este trabalho, assim como Decan *et al.* (2018), realizará uma análise de segurança no ecossistema de pacotes NPM. Entretanto, este trabalho, foca esta análise nas ferramentas de segurança e na capacidade de detecção de vulnerabilidades fornecidas pelos gerenciadores de pacotes NPM, Yarn e PNPM.

3.4 Análise comparativa

Todos os trabalhos descritos realizam estudos em cima do ecossistema de pacotes JavaScript. Jacobs (2019) e Chodorowski (2021), assim como o trabalho proposto, realizam um comparativo entre os gerenciadores de pacotes JavaScript. Porém, em Jacobs (2019), diferentemente deste trabalho, não são avaliadas as abordagens de gerenciamento de dependências e nem o espaço em disco utilizado. Já em Chodorowski (2021), foram avaliados e comparados somente os gerenciadores de pacotes NPM e Yarn.

Assim como Decan *et al.* (2018), que realiza um estudo sobre o impacto das vulnerabilidades de segurança na rede de pacotes NPM, este trabalho também analisará questões de segurança, porém essa análise será realizada em cima das ferramentas de segurança fornecidas pelos gerenciadores de pacotes.

No quadro 4 é apresentado um comparativo entre este trabalho e os trabalhos citados nesta seção.

Quadro 4 – Comparativo entre os trabalhos

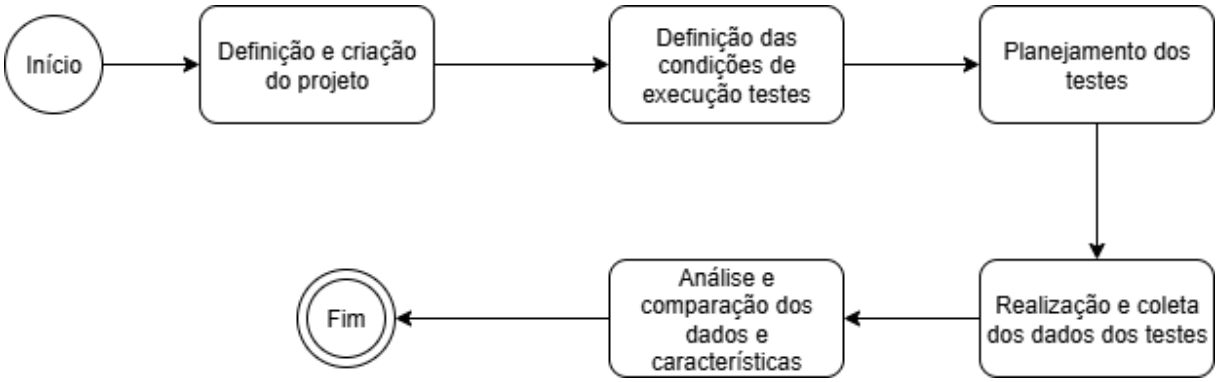
Trabalho	**Comparativo entre os gerenciadores de pacotes**	**Análise do NPM**	**Análise do Yarn**	**Análise do PNPM**	**Análise da abordagem**	**Análise do desempenho**	**Análise da segurança**	**Análise do espaço em disco**
Jacobs (2019)	Sim	Sim	Sim	Sim	Não	Sim	Sim	Não
Chodorowski (2021)	Sim	Sim	Sim	Não	Não	Sim	Sim	Sim
Decan *et al.* (2018)	Não	Sim	Não	Não	Não	Não	Sim	Não
Este trabalho	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim

Fonte: Elaborado pelo autor.

4 METODOLOGIA

Neste capítulo são apresentados um conjunto de passos a serem seguidos, a fim de alcançar os objetivos propostos neste trabalho. Cada um destes passos estão representados na Figura 15.

Figura 15 – Passos metodológicos



Fonte: Elaborado pelo autor.

4.1 Definição e criação dos projetos

Na primeira etapa metodológica deste trabalho será definida uma aplicação com escopo amplo e um elevado número de dependências, com o objetivo de simular um cenário realista e representativo do uso de gerenciadores de pacotes em projetos modernos.

Após isso será decidido se serão utilizados *frameworks* ou bibliotecas, como React, Angular ou Vue e quantos e quais pacotes comporão estes projetos. Após a definição de cada uma das características, os projetos serão inicializados e as dependências instaladas utilizando os comandos mostrados na Seção 2.3.

4.2 Definição das condições de execução testes

Nesta etapa serão definidas as condições e cenários em que os testes serão executados. Serão definidos qual o hardware, os sistemas operacionais e a largura de banda que serão utilizados durante os testes.

A definição destas condições é importante, pois garante que cada um dos testes seja executado em cenários similares, permitindo assim que a análise e o comparativo entre eles sejam realizados da forma mais justa possível.

4.3 Planejamento dos testes

Nesta etapa é necessário organizar o que será executado a seguir. Determinar quais e como os testes serão realizados, baseando-se nas características de desempenho, armazenamento, segurança, compatibilidade e abordagem, a fim de obter-se resultados significativos em todas as métricas. Também será definida a quantidade de vezes que os testes serão executados, baseada no tipo de dado obtido - quantitativo ou qualitativo.

4.4 Realização e coleta dos dados dos testes

Após a conclusão do planejamento, cada um dos testes definidos na etapa anterior será executado e seus resultados serão observados e armazenados para serem utilizados na etapa seguinte. Os testes que retornam dados quantitativos - como o de desempenho - serão executados múltiplas vezes e seu resultado final será baseado na média dos resultados obtidos.

4.5 Análise e comparação dos dados e características

Com os dados coletados, será realizada uma análise que consiste na avaliação dos resultados obtidos em cada teste realizado na etapa anterior e nos comportamentos encontrados durante a construção da aplicação. Avaliação esta que levará em conta como a abordagem de gerenciamento de dependências de cada gerenciador influencia nas outras métricas.

Após analisados, os resultados de cada teste serão comparados. Com base neste comparativo será possível concluir se existe algum gerenciador significativamente superior ao outro nos quesitos avaliados.

5 RESULTADOS

Nesta Seção são apresentados os resultados obtidos a partir da execução das etapas apresentadas na metodologia.

5.1 Desenvolvimento da aplicação web

Com o objetivo de simular um cenário realista e proporcionar uma análise prática comparativa entre os gerenciadores de pacotes JavaScript, nesta etapa foi desenvolvida uma aplicação web na forma de um dashboard para uma loja utilizando React com TypeScript. A escolha do React se deve à sua alta popularidade na comunidade de desenvolvimento front-end e à grande diversidade de pacotes disponíveis para complementar funcionalidades essenciais.

O dashboard implementado possui funcionalidades abrangentes, incluindo acompanhamento geral das operações, controle de estoque, gerenciamento detalhado de vendas, produtos e clientes, bem como a geração de relatórios diversos. Além disso, recursos adicionais foram implementados para melhorar a experiência do usuário, como suporte a multilinguagem e opções de temas - claro e escuro.

O projeto desenvolvido foi estruturado utilizando ao todo 103 dependências, divididas em 74 dependências diretas do projeto e outras 29 dependências destinadas ao ambiente de desenvolvimento - dependências de desenvolvimento. Entre os pacotes utilizados destacam-se bibliotecas bastante populares no ecossistema React, tais como *react-router-dom* para o gerenciamento eficiente de rotas, *tailwind* como framework CSS para estilização ágil, *axios* para comunicação simplificada com APIs externas, *TanStack Query* para gerenciamento avançado de estados assíncronos, *zustand* para gerenciamento global de estados e *zod* para validação robusta dos dados.

As dependências, incluindo aquelas não citadas explicitamente, foram escolhidas por serem essenciais para o desenvolvimento eficiente e organizado de aplicações modernas, abordando questões importantes como gerenciamento global de estado, navegação, requisições, estilização, acessibilidade, padronização de código, testes automatizados e manipulação de datas. Além disso, algumas dependências adicionais foram definidas durante o desenvolvimento conforme surgiam novas necessidades específicas do projeto, tais como *zxcvbn* para validação de segurança de senhas, *react-timeago* para exibição dinâmica de datas relativas e *react-hotkeys-hook* para gerenciamento avançado de atalhos.

Essa configuração diversificada de dependências proporciona uma base sólida para executar testes rigorosos e representativos com cada um dos gerenciadores analisados, garantindo uma comparação abrangente em relação ao desempenho, utilização de armazenamento, segurança e compatibilidade, aspectos fundamentais abordados nas subseções seguintes.

5.2 Realização de experimentos

Nesta Subseção são descritos os procedimentos realizados para execução dos experimentos comparativos entre os gerenciadores de pacotes analisados. Os experimentos foram elaborados para avaliar as características previamente definidas na metodologia, tais como desempenho, armazenamento e segurança. Cada experimento foi projetado para reproduzir situações comuns enfrentadas por desenvolvedores durante o ciclo de desenvolvimento de aplicações JavaScript modernas, proporcionando uma base empírica consistente e representativa dos desafios reais enfrentados ao utilizar essas ferramentas.

5.2.1 Ambiente de testes

A fim de garantir consistência nos resultados, todos os testes serão executados no mesmo ambiente com as mesmas condições. Os testes serão realizados em uma conexão Wi-Fi na frequência de 5 GHz, com largura de banda de 700 Mbps, em um equipamento recém-formatado com as seguintes configurações de hardware:

- Processador: Intel® Core™ i5-8250U.
- Placa de vídeo: NVIDIA GeForce MX110.
- Memória RAM: 8GB.
- Armazenamento: SSD 500GB M2 NVMe.

Além disso, com o objetivo de compreender como cada gerenciador lida e otimiza os recursos disponibilizados, os testes de desempenho e armazenamento serão realizados em dois sistemas operacionais distintos:

- Windows 11 Home Single Language Build 26100.4351.
- Pop!_OS 22.04 LTS.

5.2.2 Desempenho

Para o teste de desempenho, foi avaliado o tempo de execução, em segundos, de algumas das operações mais comumente utilizadas durante o desenvolvimento de uma aplicação. As operações analisadas foram: instalação completa e remoção de pacotes.

Para a realização dos testes, foi desenvolvido um *script* em Python responsável por executar cada um dos comandos relacionados às operações mencionadas anteriormente, repetindo cada execução 50 vezes. Os resultados apresentados a seguir correspondem à média dos tempos de execução obtidos.

Uma análise aprofundada dos dados obtidos a partir dos testes realizados nesta Subseção será apresentada na Subseção 5.3.1.

5.2.2.1 Instalacão completa

No teste de instalação completa, foi realizada uma instalação limpa, ou seja, sem a presença do *lockfile* e da pasta *node_modules* - quando aplicável, de acordo com o comportamento de cada gerenciador. O objetivo foi simular o cenário inicial de instalação em um projeto recém-clonado. Para uma análise mais abrangente, foram considerados dois cenários distintos:

1. Instalação com cache limpo, em que nenhuma dependência está previamente armazenada.
2. Instalação com cache populado, em que as dependências já se encontram previamente armazenadas nos mecanismos de cache utilizados por cada gerenciador de pacotes

O Quadro 5 apresenta os resultados obtidos nos testes realizados nos sistemas operacionais Windows e Pop!_OS, respectivamente.

Quadro 5 – Comparação de tempo (em segundos) de instalação no Windows e Pop!_OS

Gerenciador de pacotes	**Windows**		**Pop!_OS**	
	Sem cache	**Com cache**	**Sem cache**	**Com cache**
NPM	163,962s	66,311s	107,311s	36,102s
Yarn Classic	190,609s	141,537s	169,947s	79,949s
Yarn Modern com *node_modules*	57,548s	38,586s	39,415s	22,83s
Yarn Modern com links endereçáveis	102,582s	43,809s	40,158s	25,495s
Yarn Modern com Plug'n'Play (PnP)	23,838s	9,384s	23,32s	8,097s
PNPM	66,348s	34,329s	22,81s	12,642s

Fonte: Elaborado pelo autor.

Observando os resultados obtidos na instalação completa, percebe-se que o Yarn Modern com PnP apresentou o melhor desempenho tanto no Windows quanto no Pop!_OS,

sendo significativamente mais rápido que os outros gerenciadores em ambos os cenários - com e sem cache. O PNPM também apresentou resultados notáveis, especialmente no Pop!_OS. Em contrapartida, o Yarn Classic obteve os piores resultados. É possível notar ainda que, em geral, as operações realizadas no sistema operacional Pop!_OS foram mais rápidas que as executadas no Windows.

5.2.2.2 Remoção

Para o teste de remoção de pacotes, foram selecionadas três bibliotecas amplamente utilizadas em projetos front-end modernos — *axios*, *zod* e *TanStack Query*. A avaliação consistiu na execução do comando de remoção destes pacotes do projeto, com o objetivo de medir o tempo que cada gerenciador de pacotes leva para realizar esta operação.

O Quadro 6 apresenta os resultados obtidos nos testes realizados nos sistemas operacionais Windows e Pop!_OS, respectivamente.

Quadro 6 – Comparação de tempo (em segundos) de remoção no Windows e Pop!_OS

Gerenciador de pacotes	**Windows**	**Pop!_OS**
NPM	3,324s	3,196s
Yarn Classic	7,603s	5,854s
Yarn Modern com *node_modules*	4,303s	4,884s
Yarn Modern com links endereçáveis	21,139s	16,273s
Yarn Modern com Plug'n'Play (PnP)	3,492s	4,049s
PNPM	4.558	3,689s

Fonte: Elaborado pelo autor.

Nos testes de remoção, o NPM mostrou os melhores resultados em ambos os sistemas operacionais. O PNPM e o Yarn Modern com *node_modules* e com PnP também apresentaram bons resultados, com tempos próximos aos do NPM - com diferença máxima de aproximadamente 1,5 segundos. Em contrapartida, o Yarn Classic obteve tempos ligeiramente maiores - mas ainda razoáveis -, enquanto Yarn Modern com links endereçáveis se destaca negativamente, obtendo tempos muito altos, podendo levar até mais de 20 segundos para realizar a operação de remoção. Nota-se também que, neste caso, embora o Pop!_OS proporcione tempos de execução ligeiramente menores em relação ao Windows na maior parte dos testes, a diferença foi menos acentuada em boa parte dos resultados.

5.2.3 Armazenamento

Para a avaliação do uso de armazenamento, foram consideradas três métricas principais: o tamanho da pasta *node_modules*, o tamanho do cache utilizado por cada gerenciador de pacotes - no caso do PNPM, em vez do cache tradicional, foi considerado o tamanho do *store* global, que é onde os pacotes são realmente armazenados e referenciados - e o tamanho do *lockfile* correspondente. Para garantir que apenas os pacotes efetivamente utilizados no projeto estivessem presentes no cache ou *store*, seu conteúdo foi previamente apagado, seguido de uma nova instalação completa dos pacotes, realizada exclusivamente para fins de medição.

O Quadro 7 apresenta os resultados obtidos nos testes realizados nos sistemas operacionais Windows e Pop!_OS, respectivamente.

Quadro 7 – Comparação do espaço ocupado (em MBs) no Windows e Pop!_OS

Gerenciador de pacotes	**Windows**			**Pop!_OS**		
	node_modules	**Lockfile**	**Cache**	***node_modules***	**Lockfile**	**Cache**
NPM	993,34MB	0,44MB	265,46MB	1065,12MB	0,43MB	296,06MB
Yarn Classic	909,8MB	0,3MB	2373,09MB	984,04MB	0,29MB	2397,91MB
Yarn Modern com *node_modules*	904,19MB	0,35MB	928,73MB	910,9MB	0,34MB	917,96MB
Yarn Modern com links endereçáveis	0,01MB	0,35MB	928,73MB	0,05MB	0,34MB	917,96MB
Yarn Modern com Plug'n'Play (PnP)	Não possui	0,35MB	928,73MB	Não possui	0,34MB	917,96MB
PNPM	0,02MB	0,31MB	881,94MB	0,06MB	0,3MB	948,36MB

Fonte: Elaborado pelo autor.

Ao utilizar o Yarn Modern com PnP não é criada uma pasta *node_modules*, mas são criados dois arquivos responsáveis pela resolução das dependências, o *.pnp.cjs* e o *.pnp.loader.mjs*, que possuem 1,01MB e 0,07MB, respectivamente, em ambos os sistemas operacionais.

Já quando o Yarn Modern é utilizado com a estratégia de links endereçáveis, além do cache, é criada também uma pasta de índices que representa o *store* global, neste caso, possuindo 856,72MB no Windows e 851,04MB no PoP!_OS.

Além disso, outro comportamento notado, foi que ao utilizar ferramentas de gerenciamento de arquivos na pasta *node_modules* gerada pelo PNPM, foi indicado um tamanho de 874,87 MB no Windows e de 954,12 MB no Pop!_OS, o que não é um valor condizente com o armazenamento que realmente é utilizado - levando em conta que possui consumo compartilhado com o store devido aos links físicos. Comportamento semelhante ocorre com o Yarn Modern com

links endereçáveis - pois utiliza uma estratégia semelhante - indicando 857,1MB no Windows e 872,12MB no PoP!_OS. Para medição de um tamanho próximo do real, foi desconsiderado o tamanho das pastas *.pnpm* - no caso do PNPM - e *.store* - no caso do Yarn Modern - dentro da pasta *node_modules* que é responsável por manter os links físicos.

Os resultados apresentados nesta Subseção serão discutidos mais profundamente na Subseção 5.3.2.

5.2.4 Segurança

Para a análise de segurança dos gerenciadores de pacotes, foram avaliadas as ferramentas de auditoria disponibilizadas por cada um deles. Essas ferramentas são responsáveis por escanear as dependências do projeto em busca de vulnerabilidades conhecidas, consultando bancos de dados de falhas de segurança públicas.

Além da detecção, também foi analisada a capacidade de correção automática, observando se as vulnerabilidades identificadas podiam ser resolvidas diretamente pelo gerenciador. A análise considerou tanto a quantidade quanto a gravidade das falhas encontradas e corrigidas, permitindo comparar a eficácia das abordagens adotadas por cada ferramenta. O Quadro 8 apresenta os resultados obtidos nos testes.

Quadro 8 – Vulnerabilidades encontradas e corrigidas

Gerenciador de pacotes	**Quantidade de vulnerabilidades (nível de severidade)**	**Capacidade de correção**	**Pacotes corrigidos**
NPM	18 (1 baixa, 4 moderada e 13 alta)	Sim	1 (baixa severidade)
Yarn Classic	14 (10 moderada e 4 alta)	Não	-
Yarn Modern	17 (2 moderada e 15 alta)	Não	-
PNPM	12 (10 moderada e 2 alta)	Sim	Todos

Fonte: Elaborado pelo autor.

Nos testes de segurança, o PNPM demonstrou claramente uma maior eficácia na correção automática de vulnerabilidades, sendo capaz de corrigir todas as vulnerabilidades detectadas. Já o NPM, embora tenha identificado mais vulnerabilidades, apresentou uma capacidade limitada de correção automática. Enquanto o Yarn Classic e o Yarn Modern não possuem ferramentas para a realização de correções automáticas das vulnerabilidades. Estes resultados serão discutidos mais profundamente na Subseção 5.3.3.

5.3 Análise e comparação dos dados e características

Nesta Subseção, serão analisadas e comparadas as principais características observadas durante a execução dos testes descritos anteriormente. A avaliação considera tanto os resultados quantitativos obtidos — como tempos de execução e uso de armazenamento — quanto aspectos qualitativos identificados ao longo da utilização prática de cada gerenciador.

As subseções a seguir abordam individualmente os pontos de maior relevância para a escolha e adoção de uma ferramenta de gerenciamento de pacotes em projetos JavaScript: desempenho, armazenamento, segurança, compatibilidade e impacto da abordagem adotada. Cada uma dessas características é discutida com base nos dados levantados, nos comportamentos observados e nas implicações técnicas associadas às decisões de projeto de cada ferramenta.

5.3.1 Desempenho

Ao aprofundar a análise do desempenho dos gerenciadores de pacotes com base nos dados apresentados na Subseção 5.2.2, foi possível observar diferenças significativas nos tempos de execução das operações de instalação completa e remoção de pacotes.

Na operação de instalação completa, o Yarn Modern com Plug'n'Play – PnP – destacou-se claramente, apresentando os melhores tempos de instalação em ambos os sistemas operacionais e em quase todos os cenários testados – com e sem cache - obtendo um empate técnico em somente um caso: com o PNPM no cenário sem cache no Pop!_OS. No cenário sem cache, foi quase sete vezes mais rápido que o NPM no Windows e cerca de cinco vezes mais rápido no Pop!_OS. Com o cache populado, a proporção se manteve semelhante, chegando a ser também aproximadamente sete vezes mais rápido que o NPM no Windows e quatro vezes mais rápido no Pop!_OS. Esses resultados se devem à abordagem do PnP, que evita a criação de pastas *node_modules* e utiliza um carregador customizado – *.pnp.cjs* e *.pnp.loader.mjs* – o que elimina a sobrecarga de I/O e resolução de módulos em disco.

O PNPM também obteve resultados notáveis, especialmente no Pop!_OS, onde chegou a obter valores semelhantes ao Yarn Modern com PnP e superou todos os demais no tempo de instalação. Já no Windows, foi o terceiro mais rápido no cenário sem cache e o segundo mais rápido no cenário com cache. Esta diferença se deve à sua arquitetura baseada em links físicos para um store global e à eficiência dos sistemas de arquivos no Linux, o que permite reutilização e minimiza a cópia de arquivos. Com o cache populado, o PNPM foi

aproximadamente duas vezes mais rápido que o NPM no Windows e três vezes mais rápido no Pop!_OS.

Em contraste, os gerenciadores com arquiteturas menos otimizadas – como o NPM e o Yarn Classic – apresentaram os piores desempenhos gerais nos testes de instalação. O Yarn Classic, em especial, demonstrou ser significativamente mais lento que as versões modernas do Yarn, refletindo limitações estruturais em sua forma de resolver e instalar dependências. O NPM, apesar de seu uso amplamente difundido e contínuas atualizações, também apresentou desempenho inferior, embora o cache tenha contribuído para uma leve melhora no tempo de instalação – particularmente necessária no ambiente Windows, onde seu desempenho foi mais afetado.

Já o Yarn Modern com links endereçáveis apresentou desempenho intermediário, mas competitivo. Apesar de ser mais lento do que o Yarn Modern com PnP e o PNPM, sua performance foi superior ao Yarn Classic e ao NPM em todos os testes. No Windows com cache – por exemplo – chegou a ficar próximo do PNPM, levando 9 segundos a mais com um tempo de instalação de 43,809s. Porém, no Windows sem cache, apresentou um aumento acentuado, sendo quase o dobro do tempo do PNPM – 102,582s contra 66,348s. Essa discrepância pode ser atribuída à forma como o Yarn cria e gerencia os links físicos e índices na estratégia de links endereçáveis, que parece ser menos otimizada no sistema de arquivos do Windows.

O Yarn Modern com *node_modules* também demonstrou desempenho sólido, ficando atrás apenas do Yarn Modern com PnP e do PNPM. Um ponto que chama a atenção é o fato de ele ter obtido resultados semelhantes e até melhores do que o Yarn Modern com links endereçáveis em praticamente todos os testes, com destaque para o ambiente Windows. Essa diferença de desempenho entre as duas abordagens do Yarn pode ser explicada pelas particularidades da estratégia de links endereçáveis, que, embora inspirada no funcionamento do PNPM, adota uma estrutura distinta. Nessa abordagem, os pacotes são armazenados em um *store* global e os arquivos são organizados por meio de um índice global. Durante a instalação, o Yarn monta os diretórios *node_modules* do projeto com base nesses metadados, utilizando uma estrutura de links físicos e simbólicos que precisa ser reconstruída a cada nova instalação. Essa operação envolve uma lógica mais complexa de resolução de caminhos, verificação de integridade e criação de links físicos – operações que apresentam desempenho consideravelmente inferior no sistema de arquivos do Windows, o que explica a diferença mais acentuada nesse ambiente. Além disso, o Yarn mantém um cache adicional contendo os arquivos compactados originais

dos pacotes, o que também contribui para o aumento do tempo de preparação do ambiente. A combinação destes fatores ajuda a explicar o seu desempenho quando comparado à abordagem tradicional aplicada pelo Yarn Modern com *node_modules*.

Esse comportamento evidencia uma tendência mais ampla observada nos testes: de forma geral, os tempos de instalação e remoção foram consistentemente menores no Pop!_OS em comparação ao Windows, para quase todos os gerenciadores e estratégias testadas. Isso se deve principalmente às vantagens do sistema de arquivos ext4 – utilizado no Pop!_OS – que oferece desempenho superior em operações com grande volume de arquivos pequenos, criação de diretórios e manipulação de links simbólicos ou físicos. Tais operações são fundamentais para ferramentas como NPM, PNPM e Yarn com *node_modules* ou links endereçáveis. Além disso, o Linux tende a apresentar menor latência de acesso a disco e melhor gerenciamento de I/O – especialmente em dispositivos SSD – o que favorece diretamente esse tipo de *workload*.

A única exceção significativa foi o Yarn Modern com Plug'n'Play – PnP –, cujo desempenho se manteve praticamente idêntico entre os dois sistemas operacionais. Isso ocorre porque o PnP não depende do sistema de arquivos tradicional para resolver dependências, já que dispensa o uso da pasta *node_modules* e não cria links ou diretórios adicionais. Toda a resolução é feita em memória, a partir de arquivos mapeadores – *.pnp.cjs* e *.pnp.loader.mjs* – interpretados diretamente pelo *runtime* - tempo de execução. Com isso, o impacto das diferenças entre sistemas de arquivos e desempenho de disco é minimizado, resultando em uma performance mais estável e uniforme entre Windows e Linux.

Já na operação de remoção de pacotes, o NPM foi o mais eficiente em ambos os sistemas operacionais, onde levou cerca de 3 segundos para remover os três pacotes selecionados. O PNPM teve um desempenho próximo ao NPM no Linux, ficando apenas 0,5 segundos atrás, e também se saiu bem no Windows, mantendo-se competitivo. O Yarn Modern com PnP apresentou tempos igualmente bons, ficando na faixa de 3,5 a 4 segundos, o que reforça sua consistência geral mesmo em cenários de desinstalação.

O Yarn Classic, apesar de sua arquitetura mais tradicional, apresentou tempos de remoção razoáveis – 7,603 segundos no Windows e 5,854 segundos no Pop!_OS – posicionando-se abaixo do NPM, PNPM e Yarn Modern com *node_modules* e com PnP, mas ainda com desempenho significativamente melhor que o Yarn Modern com links endereçáveis, que apresentou os piores resultados nesse cenário: cerca de 21 segundos no Windows e 16 segundos no Pop!_OS. Esse desempenho inferior pode ser atribuído à necessidade de desmontar estruturas complexas

de links físicos e reorganizar os índices utilizados pelo Yarn, o que envolve uma sequência mais custosa de operações de leitura e escrita em disco durante a desinstalação.

De forma geral, os resultados indicam que, se o principal objetivo é obter o melhor desempenho bruto, o Yarn Modern com Plug'n'Play é a escolha mais adequada. No entanto, é importante destacar que, apesar de sua excelente performance, essa abordagem pode apresentar questões de compatibilidade com ferramentas e bibliotecas que esperam a presença de uma pasta *node_modules*, como será discutido com mais detalhes na Subseção 5.3.4. Para quem busca um equilíbrio entre desempenho e compatibilidade com o ecossistema tradicional do Node.js, o PNPM e o Yarn Modern com *node_modules* se mostram alternativas robustas e eficientes. O Yarn com links endereçáveis, apesar de não ter liderado os testes, apresenta uma abordagem moderna e gerenciamento organizado dos pacotes – embora possa sofrer penalizações no tempo de instalação sem cache e remoção, especialmente em ambientes Windows, devido à sobrecarga envolvida na manutenção de sua estrutura interna.

5.3.2 Armazenamento

A análise do espaço ocupado por cada gerenciador de pacotes levou em consideração o tamanho total das pastas *node_modules*, cache – ou *store*, quando aplicável – e também o *lockfile*, nos dois sistemas operacionais. Esses dados permitem avaliar não apenas o impacto da instalação das dependências no projeto, mas também o custo de armazenamento adicional que cada ferramenta gera ao longo do tempo. Para esta análise foram utilizados os dados obtidos na Subseção 5.2.3.

De forma geral, o PNPM apresentou os melhores resultados em termos de economia de espaço. Isso é consequência direta de sua arquitetura baseada em um *store* global, com links físicos para os projetos, o que evita a duplicação de pacotes. O conteúdo real das dependências é instalado apenas uma vez no *store*, e os diretórios *node_modules* de cada projeto passam a conter apenas referências físicas – resultando em ocupações mínimas, como 0,02 MB no Windows e 0,06 MB no Pop!_OS.

No entanto, essa abordagem - também utilizada pelo Yarn Modern com links endereçáveis - também implica em diferenças na forma como o espaço é medido: como os links físicos não duplicam efetivamente os arquivos no disco, ferramentas que contabilizam o uso de espaço sem considerar isso (como o Windows Explorer ou comandos como `du` no Linux) podem apresentar valores inflados. Nos testes, ao utilizar ferramentas de gerenciamento de arquivos

diretamente na pasta *node_modules*, foi indicado um tamanho de 874,87 MB no Windows e 954,12 MB no Pop!_OS – valores que não condizem com o uso real. Para obter uma medida próxima do real, foi desconsiderado o tamanho da pasta *.pnpm*, responsável por manter os links físicos.

O Yarn Modern com Plug'n'Play – PnP – também se destacou positivamente. Por não gerar uma pasta *node_modules*, ele evita a replicação de arquivos em disco e mantém todo o controle de dependências em dois arquivos de mapeamento: *.pnp.cjs* e *.pnp.loader.mjs*, que possuem, respectivamente, 1,01 MB e 0,07 MB em ambos os sistemas operacionais. Esses arquivos substituem completamente a estrutura tradicional de diretórios e resolvem todas as dependências de forma virtual. Dessa forma, o espaço ocupado por projeto é significativamente reduzido, sendo composto apenas pelo cache global e os arquivos de mapeamento. Essa arquitetura torna o Yarn Modern com PnP uma das soluções mais eficientes em termos de armazenamento local – com vantagens especialmente visíveis no Windows, onde o gerenciamento de múltiplos arquivos pequenos tende a ser menos eficiente.

O Yarn Modern com *node_modules*, por sua vez, adota uma abordagem mais tradicional ao manter uma cópia completa das dependências diretamente no projeto, sem o uso de links físicos ou de um store global compartilhado. Embora utilize um cache global com arquivos compactados, o conteúdo das dependências é efetivamente descompactado e duplicado dentro da pasta *node_modules*, o que contribui para o aumento do uso de espaço. Nos testes, essa pasta atingiu 904,19 MB no Windows e 910,9 MB no Pop!_OS, enquanto o cache ocupou 928,73 MB e 917,96 MB, respectivamente. Essa estrutura garante ampla compatibilidade com a maioria dos pacotes, mas resulta em um uso de espaço mais elevado – superando, inclusive, o NPM em um cenário de armazenamento total, principalmente devido ao seu cache mais inflado. Ainda que o sistema de cache e o comportamento moderno do Yarn tragam vantagens em termos de resolução, essa estratégia não oferece ganhos concretos em termos de economia de espaço, sendo menos econômica que soluções baseadas em links físicos ou resolução virtual.

O Yarn Modern com links endereçáveis também apresentou um consumo considerável de armazenamento. Ao utilizar uma estrutura moderna com montagem baseada em links simbólicos, os arquivos reais das dependências não são armazenados diretamente no diretório *node_modules*, mas sim em um *store* global - estratégia semelhante à utilizada pelo PNPM. O diretório *node_modules*, por sua vez, contém apenas links, resultando em um uso direto de apenas 0,01 MB no Windows e 0,05 MB no Pop!_OS. O *store* global, que armazena os pacotes

efetivamente baixados, ocupou 856,72 MB no Windows e 851,04 MB no Pop!_OS. A sua desvantagem em termos de armazenamento é que o Yarn além do *store* global também vai manter sua estrutura de cache adicional, contendo os arquivos compactados originais dos pacotes, o que aumenta o consumo total de armazenamento — diferentemente do PNPM, por exemplo, que utiliza apenas a *store* e não mantém um cache redundante. Apesar desse volume ser significativo mesmo com apenas um projeto, a abordagem do Yarn se mostrou mais eficiente que a do Yarn Classic na economia geral de espaço e também possui as vantagens de evitar cópias físicas no *node_modules* e permitir reutilização entre projetos.

O Yarn Classic apresentou os piores resultados de armazenamento, com valores consideravelmente mais altos nos dois sistemas operacionais. Isso ocorre porque a ferramenta mantém uma cópia completa das dependências na pasta *node_modules* de cada projeto – 909,8 MB no Windows e 984,04 MB no Pop!_OS – além de um cache extremamente volumoso localizado no diretório do usuário. Esse cache foi o maior entre todos os gerenciadores avaliados, totalizando 2.373,09 MB no Windows e 2.397,91 MB no Pop!_OS – mais que o dobro do consumo observado no PNPM e no Yarn Modern, ambos com aproximadamente 900 MB, e quase dez vezes maior que o cache do NPM. Essa estrutura redundante, somada ao cache excessivamente volumoso, contribui significativamente para o alto consumo total de armazenamento, mesmo em projetos relativamente simples.

Já o NPM, embora adote uma arquitetura semelhante ao Yarn Classic, mostrou-se mais econômico nos testes realizados – apresentando um consumo total de armazenamento inferior, inclusive, ao do Yarn Modern com *node_modules* e ao do Yarn com links endereçáveis. Essa economia se deve principalmente ao baixo volume do seu cache, devido ao uso da biblioteca de cache *cacache*, que organiza os arquivos de forma segmentada e eficiente. Nessa estrutura, os pacotes são armazenados em um formato segmentado baseado em hash e os metadados – como informações sobre integridade e origem dos arquivos – são mantidos separadamente em um índice dedicado. Essa abordagem reduz a redundância, pois impede a duplicação desnecessária de arquivos ou dados auxiliares, ao contrário do que ocorre com estratégias baseadas em arquivos compactados. Como resultado, o NPM consegue manter um cache mais leve e direto – cerca de 265,46 MB no Windows e 296,06 MB no Pop!_OS –, contribuindo para um uso mais econômico do armazenamento, mesmo utilizando um modelo tradicional de instalação.

Ao analisar os valores totais de armazenamento, nota-se que não há uma superioridade clara entre os dois sistemas operacionais. Em muitos casos, como no Yarn Classic e no

PNPM, o Pop!_OS apresentou maior consumo. Isso contraria uma suposição comum de que o sistema de arquivos NTFS, presente no Windows, sempre gera maior ocupação de espaço. Embora seja verdade que o NTFS lide de forma menos eficiente com grandes quantidades de arquivos pequenos – situação comum em pastas *node_modules* –, os resultados indicam que, na prática, essa diferença é menos significativa do que o esperado.

Já o sistema de arquivos ext4, utilizado pelo Pop!_OS, é conhecido por sua maior eficiência na alocação de arquivos pequenos, o que pode favorecer alguns cenários. Ainda assim, os dados coletados demonstram que o sistema operacional não foi, isoladamente, o principal fator determinante no consumo de espaço – sendo a arquitetura do próprio gerenciador a principal variável nesse contexto.

Por fim, os *lockfiles* também apresentam diferenças estruturais entre os gerenciadores, o que resulta em tamanhos ligeiramente distintos. No entanto, como esses arquivos são relativamente pequenos – variando entre 0,3 MB e 0,5 MB na maioria dos casos – seu impacto no consumo total de armazenamento é mínimo e não altera significativamente os resultados observados.

De maneira geral, os resultados mostram que as ferramentas que adotam estratégias modernas de gerenciamento de pacotes – como links físicos, índices centralizados e ausência de *node_modules* – tendem a ser mais eficientes em termos de armazenamento, com destaque para o PNPM e o Yarn Modern com PnP. Esses dois gerenciadores se tornam ainda mais vantajosos em cenários com múltiplos projetos que compartilham dependências entre si, pois evitam a duplicação de arquivos e otimizam o uso de espaço. O Yarn com links endereçáveis também representa um avanço considerável e possui as mesmas vantagens de deduplicação citadas anteriormente, entretanto, a redundância ao manter tanto o *store* global quanto o cache tradicional causa um aumento considerável no consumo total de armazenamento. O NPM, apesar de sua arquitetura tradicional baseada em cópias completas na pasta *node_modules*, obteve resultados mais econômicos que o Yarn Classic e até mesmo que algumas abordagens modernas – como o Yarn Modern com *node_modules* –, demonstrando eficiência no uso de cache e estruturação de arquivos. No entanto, mesmo com essa eficiência, gerenciadores que seguem essa abordagem tradicional – baseada na duplicação local das dependências – apresentam desvantagens em cenários com múltiplos projetos, já que os mesmos pacotes precisam ser armazenados repetidamente em cada projeto, o que compromete a economia de espaço. Já o Yarn Classic permanece como a opção mais onerosa nesse aspecto, pois adota a mesma abordagem tradicional de duplicação local das

dependências e ainda possui um cache extremamente volumoso, o que amplia consideravelmente o consumo total de armazenamento.

5.3.3 Segurança

Com base nos dados apresentados na Subseção 5.2.4, que detalha os resultados da auditoria de vulnerabilidades realizada com os diferentes gerenciadores de pacotes, foi possível realizar uma análise comparativa da eficácia de cada ferramenta na identificação e correção de falhas. A investigação considerou não apenas a quantidade de vulnerabilidades detectadas, mas também a gravidade atribuída a elas e a capacidade de correção automática oferecida por cada solução.

Nos testes realizados, o PNPM destacou-se como o gerenciador com melhor desempenho em termos de eficácia na correção automática. Apesar de ter identificado um número inferior de vulnerabilidades - 12 falhas, sendo 10 moderadas e 2 altas -, conseguiu aplicar correções para todas elas. Isso foi possível graças ao mecanismo de *overrides* utilizado pelo PNPM, que permite forçar o uso de versões seguras de dependências, mesmo quando bibliotecas intermediárias exigem versões vulneráveis. Essa estratégia oferece um controle fino sobre as correções, sem comprometer a previsibilidade da instalação ou o versionamento semântico do projeto.

O NPM, por outro lado, foi o gerenciador que mais identificou falhas em sua análise padrão - 18 no total, sendo 13 classificadas como de alta severidade. Sua capacidade de correção automática, porém, mostrou-se limitada - apenas uma falha, de baixa severidade, foi corrigida. Buscando ampliar esse resultado, também foi testado o uso do comando *npm audit fix –force* - que tenta resolver vulnerabilidades por meio da atualização forçada de dependências, inclusive com mudanças de versão *major*.

No entanto, o resultado obtido foi contraproducente: após a aplicação da *flag* –*force*, o número total de vulnerabilidades aumentou de 18 para 32 - revelando que a tentativa de correção automática introduziu novas falhas ao atualizar pacotes para versões instáveis ou com problemas conhecidos. Esse comportamento evidencia a ineficiência e os riscos dessa estratégia, que pode comprometer a segurança e a estabilidade do projeto ao modificar o grafo de dependências de forma indiscriminada. Isso ocorre porque o *npm audit fix –force* ignora as restrições de versionamento definidas no *package.json*, permitindo a instalação de versões *major* - que possuem *breaking changes* - de dependências, o que pode resultar na inclusão de pacotes

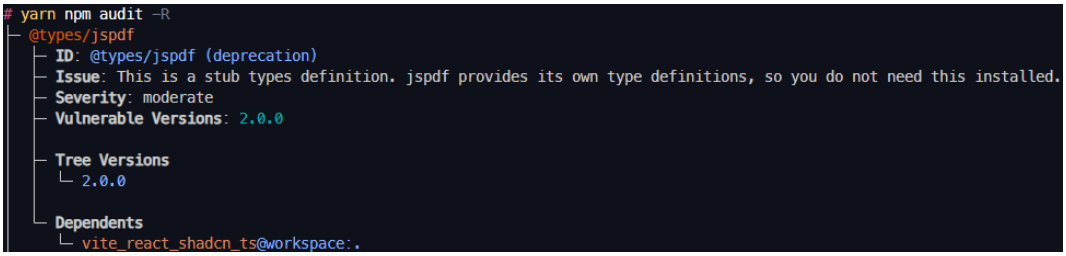
incompatíveis ou ainda vulneráveis. Por isso recomenda-se que o uso da *flag –force* deve ser feito com cautela - sendo recomendado apenas em cenários de teste ou emergência.

Em contraste, a estratégia de correção do PNPM se mostrou mais eficaz e segura - baseada exclusivamente no uso controlado de *overrides* para resolver vulnerabilidades, tanto em dependências transitivas quanto diretas. O comando *pnpm audit –fix* aplica automaticamente esse mecanismo quando é possível resolver a falha utilizando versões *patch* ou *minor* - desde que a atualização não viole as restrições de compatibilidade semântica estabelecidas pelas dependências intermediárias ou pelo próprio projeto.

Quando a correção exige uma atualização *major*, o PNPM não realiza alterações automáticas - sendo necessário que o desenvolvedor edite manualmente o campo *overrides* no arquivo *pnpm-workspace.yaml* - arquivo de configurações do espaço de trabalho. Essa abordagem evita modificações diretas no grafo de dependências e garante previsibilidade e controle mesmo em situações críticas. Ao adotar exclusivamente o mecanismo de substituição controlada, o PNPM elimina o risco de atualizações indesejadas e reforça sua proposta de integridade e estabilidade nas instalações.

Os demais gerenciadores - Yarn Classic e Yarn Modern - também identificaram um número significativo de vulnerabilidades - 14 e 17, respectivamente -, mas não apresentaram qualquer capacidade de correção automática. No caso do Yarn Modern, a ferramenta de auditoria utilizada se baseia na infraestrutura do NPM, mas não implementa comandos de correção. Além disso, foram relatadas vulnerabilidades em pacotes de tipo - como *@types/** - e dependências descontinuadas, que normalmente não afetam a segurança real da aplicação. Um exemplo observado durante os testes foi o alerta para o pacote *@types/jspdf*, mostrado na Figura 16, marcado como vulnerável apenas por estar obsoleto - já que a biblioteca *jspdf* passou a fornecer suas próprias definições de tipo. Esse tipo de ocorrência inflaciona os resultados, sem necessariamente representar um maior nível de risco.

Figura 16 – Alerta de vulnerabilidade do pacote *jspdf*



Fonte: Elaborado pelo autor.

É importante observar ainda que todas as ferramentas analisadas utilizam essencialmente a mesma base de dados de vulnerabilidades - geralmente derivada do *GitHub Advisory Database*. No entanto, os resultados variam significativamente, pois cada gerenciador interpreta, estrutura e apresenta esses dados de forma diferente. Essa diferença está relacionada à forma como o grafo de dependências é resolvido, aos critérios adotados para definir o que constitui uma falha relevante e ao tratamento de pacotes específicos - como dependências de desenvolvimento, pacotes de tipos ou bibliotecas obsoletas. Esse fator reforça que o número de falhas reportadas não deve ser interpretado isoladamente como um indicativo de maior ou menor segurança, mas sim considerado à luz da arquitetura e das decisões de design de cada ferramenta.

Além das diferenças no conteúdo reportado, observou-se também uma variação significativa na forma como os resultados da auditoria são apresentados por cada ferramenta. O Yarn Classic e o PNPM, por exemplo, exibem os dados em formato de tabela - com foco em clareza e concisão -, permitindo uma leitura rápida e objetiva das vulnerabilidades detectadas. Já o NPM e o Yarn Modern adotam uma abordagem que tenta emular a estrutura da árvore de dependências, com indentação hierárquica - o que pode ser útil para visualizar o caminho até a dependência vulnerável, mas acaba tornando a saída mais verbosa e, em alguns casos, de leitura mais difícil em projetos com muitas camadas. Essa diferença no formato de saída pode influenciar diretamente a experiência do desenvolvedor durante a análise de vulnerabilidades - principalmente quando é necessário revisar manualmente múltiplos pacotes ou aplicar correções específicas. As Figuras 17, 18, 19 e 20 ilustram os diferentes formatos de saída gerados por cada ferramenta de auditoria.

Figura 17 – Trecho de saída da ferramenta de auditoria do NPM

Fonte: Elaborado pelo autor.

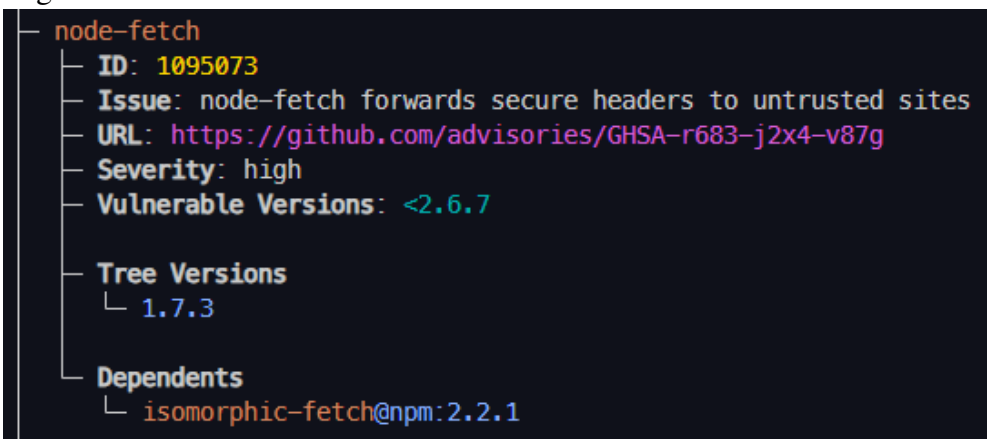
Por fim, a eficácia de uma ferramenta de auditoria está relacionada à sua capacidade de identificar falhas relevantes e aplicar correções seguras e sustentáveis. Ferramentas que adotam estratégias agressivas de atualização podem resolver mais falhas superficialmente - mas

Figura 18 – Trecho de saída da ferramenta de auditoria do Yarn Classic

high	node-fetch forwards secure headers to untrusted sites
Package	node-fetch
Patched in	>=2.6.7
Dependency of	ml5
Path	ml5 > @tensorflow/tfjs-vis > glamor > fbjs > isomorphic-fetch > node-fetch
More info	https://www.npmjs.com/advisories/1095073

Fonte: Elaborado pelo autor.

Figura 19 – Trecho de saída da ferramenta de auditoria do Yarn Modern



Fonte: Elaborado pelo autor.

Figura 20 – Trecho de saída da ferramenta de auditoria do PNPM

high	node-fetch forwards secure headers to untrusted sites
Package	node-fetch
Vulnerable versions	<2.6.7
Patched versions	>=2.6.7
Paths	.>ml5>@tensorflow/tfjs-vis>glamor>fbjs>isomorphic-fetch>node-fetch
More info	https://github.com/advisories/GHSA-r683-j2x4-v87g

Fonte: Elaborado pelo autor.

às custas da estabilidade do sistema. Além disso, a forma como essas ferramentas apresentam os resultados também influencia a experiência de uso: saídas mais concisas - como as oferecidas pelo PNPM e pelo Yarn Classic - tendem a facilitar o diagnóstico e a tomada de decisão, enquanto saídas mais verbosas e hierárquicas - como as do NPM e do Yarn Modern - podem dificultar a análise em projetos com grandes cadeias de dependência.

Nesse cenário, o PNPM apresentou o melhor equilíbrio entre precisão na detecção,

abrangência da análise, capacidade efetiva de correção e clareza na apresentação dos dados - consolidando-se como a opção mais segura, eficiente e tecnicamente adequada entre os gerenciadores avaliados.

5.3.4 Compatibilidade

A compatibilidade de um gerenciador de pacotes está diretamente relacionada à sua capacidade de interagir corretamente com o ecossistema JavaScript como um todo - incluindo bibliotecas de terceiros, ferramentas de build, ambientes de execução e editores de código. Essa compatibilidade é fortemente influenciada pela forma como cada gerenciador organiza e disponibiliza os pacotes no projeto, especialmente no que diz respeito à estrutura de diretórios e ao mecanismo de resolução de módulos.

O NPM e o Yarn Classic adotam a abordagem tradicional baseada na criação física da pasta *node_modules*, onde as dependências são dispostas em uma hierarquia de diretórios que segue o modelo de resolução nativo do Node.js. Esse formato é amplamente suportado, pois corresponde à estrutura esperada pela grande maioria das ferramentas e bibliotecas do ecossistema. Como consequência, ambos gerenciadores oferecem ampla compatibilidade e são raramente afetados por problemas de integração.

Entretanto, apesar dessa compatibilidade generalizada, o Yarn Classic começa a apresentar limitações ao lidar com bibliotecas mais recentes que adotam requisitos específicos da nova geração de ferramentas. Um exemplo prático dessa situação foi identificado durante o desenvolvimento do projeto utilizado neste trabalho, ao tentar instalar a biblioteca *ml5* com o Yarn Classic. A operação resultou no erro apresentado na Figura 21. Esse tipo de erro está relacionado ao campo *engines*, presente no *package.json* de diversas bibliotecas modernas, que permite declarar quais versões de Node.js ou de gerenciadores de pacotes são compatíveis com o módulo. No caso em questão, a biblioteca *ml5* exige explicitamente uma versão do Yarn a partir da 4.3.1 - pertencente à linha do Yarn Modern. Como o Yarn Classic - versão 1.x - não atende a esse requisito, o gerenciador bloqueia a instalação do pacote, mesmo que tecnicamente ele ainda seja compatível.

Figura 21 – Erro apresentado pelo Yarn Classic ao instalar a biblioteca *ml5*

```
error ml5@1.2.1: The engine "yarn" is incompatible with this module. Expected version "^4.3.1". Got "1.22.22"
```

Fonte: Elaborado pelo autor.

Para contornar esse bloqueio, foi utilizada a *flag –ignore-engines*, que desativa a

verificação do campo *engines* durante o processo de instalação. Com esse ajuste, o pacote pôde ser instalado normalmente e, neste caso específico, funcionou como esperado. No entanto, é importante destacar que essa abordagem não é recomendada como prática padrão. Ignorar verificações de compatibilidade pode resultar em falhas de execução ou comportamento inesperado quando o pacote realmente depende de funcionalidades específicas da versão exigida. Portanto, o uso da *flag –ignore-engines* deve ser reservado a situações pontuais e sempre acompanhado de validação manual do funcionamento da biblioteca.

Esse episódio também reforça a importância de considerar o status de manutenção das ferramentas utilizadas. O Yarn Classic encontra-se atualmente em modo de manutenção, o que significa que novas funcionalidades não estão sendo desenvolvidas e apenas correções críticas são aplicadas esporadicamente. Apesar de sua ampla compatibilidade com projetos legados, essa limitação o torna uma escolha menos indicada para projetos novos, especialmente quando há dependência de bibliotecas mais recentes que já adotaram recursos ou requisitos específicos do Yarn Modern. Dessa forma, embora ainda amplamente funcional, o Yarn Classic tende a apresentar uma compatibilidade cada vez mais restrita com bibliotecas que acompanham as versões mais recentes do ecossistema.

Seguindo para um modelo mais moderno e com proposta estrutural distinta, o PNPM também utiliza a pasta *node_modules*, porém adota uma arquitetura diferenciada ao empregar uma estrutura de links simbólicos e diretórios virtualizados que apontam para um repositório de armazenamento centralizado. Essa abordagem cria ambientes mais previsíveis e restritos, nos quais cada pacote acessa apenas suas dependências declaradas. Embora essa rigidez seja benéfica para a integridade do projeto e ajude a evitar o uso de dependências fantasmas, ela pode causar problemas de compatibilidade com bibliotecas mal projetadas que assumem implicitamente o acesso a dependências transitivas. Ainda assim, o PNPM vem ganhando adoção significativa, o que tem incentivado a comunidade a corrigir esse tipo de problema em muitas bibliotecas populares.

Um ponto que vale a pena pontuar é que durante o desenvolvimento dos *scripts* automatizados utilizados para realização dos testes de desempenho, implementados em Python, foi identificado um problema ao tentar remover a pasta *node_modules* de projetos instalados com o PNPM no sistema operacional Windows. A operação, realizada com a função `shutil.rmtree`, falhou com a mensagem de erro “O sistema não pode encontrar o caminho especificado”, mesmo que o caminho indicado fosse válido. Após investigação, constatou-se que a falha está

relacionada à forma como o Python lida com links simbólicos no Windows - que são utilizados pelo PNPM para estruturar as dependências no diretório *node_modules*. Esse comportamento não foi observado com o NPM, com o Yarn utilizando o *node_modules*, nem mesmo ao utilizar o Node.js - com `fs.rmSync` - para remover a mesma estrutura gerada pelo PNPM, o que reforça a hipótese de que o problema está na compatibilidade entre a API de arquivos do Python e os mecanismos de link utilizados pelo PNPM no ambiente Windows.

Esse tipo de incompatibilidade, embora pontual, evidencia uma limitação importante da arquitetura do PNPM: sua estrutura interna, embora eficiente e segura dentro do ecossistema JavaScript, pode causar falhas silenciosas quando manipulada por ferramentas externas – como *scripts* em Python, sistemas de backup, soluções de monitoramento ou pipelines de DevOps baseadas em linguagens que não tratam corretamente links no Windows. Isso reforça a importância de avaliar o impacto da arquitetura do gerenciador de pacotes não apenas no projeto em si, mas também em todas as ferramentas e processos que interagem com ele ao longo do ciclo de vida do desenvolvimento.

O Yarn Modern, entre os gerenciadores apresentados, é o que oferece maior flexibilidade ao permitir a configuração do comportamento de resolução de dependências por meio do campo `nodeLinker` definido no arquivo de configurações do Yarn Modern - *.yarnrc.yml*. Quando configurado como `node-modules`, seu funcionamento se assemelha ao do Yarn Classic e do NPM, garantindo total compatibilidade com ferramentas legadas, editores e bibliotecas diversas. Essa configuração preserva a estrutura física tradicional e, por isso, é amplamente suportada.

Já ao utilizar a opção `pnpm`, o Yarn adota uma estratégia inspirada diretamente na arquitetura do PNPM, passando a utilizar links endereçáveis que oferecem benefícios semelhantes - como previsibilidade e controle de escopo. No entanto, essa configuração também herda as mesmas restrições, especialmente no que se refere ao acesso implícito a dependências transitivas, o que pode limitar a compatibilidade com bibliotecas que não seguem boas práticas de declaração.

Por fim, a configuração mais distinta é a `pnp` - Plug'n'Play -, na qual o Yarn Modern elimina completamente a pasta *node_modules* e passa a resolver as dependências por meio dos arquivos *.pnp.cjs* e *.pnp.loader.mjs*. Esses arquivos atuam como um mapeamento centralizado, indicando exatamente onde cada pacote está localizado. Essa abordagem oferece vantagens importantes, como maior previsibilidade, isolamento e desempenho na resolução de módulos. No entanto, justamente por romper com a convenção tradicional de estrutura física no disco, ela pode

gerar incompatibilidades com ferramentas que esperam encontrar os arquivos das dependências no diretório *node_modules* - como *linters*, *bundlers* ou extensões de editores de código.

Embora o suporte ao PnP tenha evoluído consideravelmente nos últimos anos, sendo hoje compatível com diversas ferramentas modernas, seu uso ainda pode exigir ajustes adicionais em muitos casos. Em especial, ferramentas que não foram projetadas com suporte nativo ao PnP podem apresentar falhas em funcionalidades, exigir configurações específicas ou até mesmo deixar de funcionar corretamente. Por isso, sua integração com o ecossistema não é totalmente automática ou livre de fricções, especialmente em ambientes complexos ou projetos legados. Um exemplo concreto dessa limitação foi observado no próprio projeto utilizado neste trabalho, que faz uso da biblioteca *patch-package* para aplicar correções temporárias em dependências. Como essa ferramenta depende do acesso direto aos arquivos físicos das bibliotecas instaladas, ela se torna incompatível com a estratégia PnP.

Para contornar essa limitação, o Yarn oferece seu próprio sistema de patches, por meio dos comandos *yarn patch* e *yarn patch-commit*, que operam de forma integrada ao ambiente Plug'n'Play. Apesar disso, essa diferença de abordagem entre ferramentas reforça a necessidade de cautela ao migrar projetos legados para a arquitetura PnP - especialmente quando o ecossistema envolvido ainda depende fortemente da presença física da estrutura *node_modules*.

Considerando essas diferenças, observa-se que quanto mais rígido é o controle sobre a resolução de dependências e maior a busca por isolamento e previsibilidade, menor tende a ser a compatibilidade com ferramentas legadas ou bibliotecas que não seguem estritamente as boas práticas do ecossistema. Nesse contexto, configurações como a do Yarn com PnP apresentam maiores restrições de compatibilidade - especialmente em projetos legados ou com ferramentas que exigem acesso físico à pasta *node_modules*. O PNPM e o Yarn Modern com links endereçáveis, por sua vez, representam um meio-termo interessante: mantém a estrutura tradicional da *node_modules*, o que assegura boa compatibilidade com a maior parte das ferramentas do ecossistema, mas sua arquitetura baseada em *links* pode gerar problemas em ambientes que não lidam bem com esses mecanismos. Já soluções como o NPM, o Yarn Classic e o Yarn Modern com *node_modules* oferecem a maior compatibilidade prática, sendo especialmente indicadas quando a prioridade é a integração imediata e ampla com o ecossistema atual. Entretanto, apesar desta melhor compatibilidade prática, o Yarn Classic não é recomendado para novos projetos, devido ao se encontrar, atualmente, em modo de manutenção, podendo assim apresentar restrições em bibliotecas ou ferramentas mais recentes.

5.3.5 Impacto das abordagens adotadas

Ao longo das análises realizadas nesta Subseção, observou-se que os comportamentos e os resultados obtidos nos testes de desempenho, armazenamento, segurança e compatibilidade estão intimamente ligados às abordagens adotadas por cada gerenciador de pacotes. Essa abordagem se refere à forma como cada ferramenta estrutura o ambiente do projeto, resolve e instala dependências, armazena pacotes e interage com o sistema de arquivos. Nesta Subseção, discute-se de forma integrada como essas escolhas arquiteturais impactam diretamente as demais características avaliadas.

O desempenho, por exemplo, está diretamente relacionado à forma como os gerenciadores manipulam o sistema de arquivos. O Yarn Modern com PnP elimina completamente a pasta *node_modules*, resolvendo os módulos por meio de um sistema próprio baseado em um manifesto e arquivos *.pnp.cjs*. Essa abordagem reduz drasticamente as operações de leitura e escrita no sistema de arquivos, o que explica seu bom desempenho, especialmente em ambientes onde o acesso ao disco representa um gargalo. Já o NPM, o Yarn Classic e o Yarn Modern com *node_modules*, ao seguirem a estrutura tradicional com cópias físicas das dependências, geram um volume significativamente maior de arquivos e operações no sistema de arquivos, o que tende a impactar negativamente o desempenho geral do processo. O PNPM e o Yarn Modern com links endereçáveis, por sua vez, apostam na criação de links físicos para evitar duplicações e acelerar a instalação, embora a criação desses links possa ser mais custosa em determinados sistemas, como o Windows.

No que diz respeito ao armazenamento, as decisões arquiteturais afetam não apenas o espaço ocupado no disco, mas também a forma como os arquivos são organizados. O NPM, o Yarn Classic e o Yarn Modern com *node_modules*, por utilizarem cópias físicas completas das dependências em cada projeto, tendem a consumir mais espaço. Já o PNPM, com sua estrutura centralizada de *store* e o uso de *links*, oferece uma solução mais eficiente, enquanto o Yarn Modern com PnP armazena os pacotes de forma compacta e evita redundâncias ao não utilizar a estrutura tradicional de pastas. O Yarn Modern com links endereçáveis, por sua vez, adota uma abordagem intermediária — próxima à do PNPM —, mas mantém além do índice centralizado responsável pelo store global, um cache, o que resulta em uma organização distinta e em um uso de armazenamento também considerável.

No aspecto da segurança, a abordagem adotada por cada gerenciador impacta diretamente sua capacidade de realizar correções proativas de vulnerabilidades. O modo como

o grafo de dependências é estruturado — centralizado, virtualizado ou duplicado — define o grau de controle que o gerenciador consegue exercer sobre as versões instaladas. O PNPM, por exemplo, adota uma arquitetura baseada em links e uma representação explícita e previsível da árvore de dependências, o que viabiliza a aplicação seletiva e segura de substituições por meio do mecanismo de *overrides*. Essa abordagem, mais estável e controlada, permite corrigir falhas sem comprometer o versionamento semântico ou a integridade do projeto. Em contraste, o NPM, mesmo adotando uma estrutura tradicional, limita-se a atualizações automáticas convencionais ou ao uso de comandos mais agressivos — como o *npm audit fix –force* — que atuam sobre o grafo de dependências de forma indiscriminada. Já o Yarn Modern, especialmente em modo Plug'n'Play, abstrai a resolução física das dependências, o que dificulta a implementação de correções automatizadas e limita o controle direto sobre o ambiente. Essas diferenças mostram que a capacidade de agir proativamente frente a falhas não está apenas na existência de uma ferramenta de auditoria, mas sim na arquitetura adotada por cada gerenciador e na flexibilidade que ela oferece para operar sobre o grafo de dependências.

Por fim, a compatibilidade é diretamente impactada pela abordagem de resolução e armazenamento. Gerenciadores que mantêm a estrutura *node_modules*, como NPM e PNPM, tendem a apresentar maior compatibilidade com bibliotecas do ecossistema, uma vez que muitos pacotes ainda assumem a existência dessa pasta para funcionar corretamente. Em contrapartida, abordagens alternativas como a do Yarn com PnP podem gerar incompatibilidades com bibliotecas legadas, exigindo configurações adicionais ou até mesmo inviabilizando seu uso em determinados projetos.

Assim, observa-se que a abordagem adotada por cada gerenciador não apenas define sua identidade técnica, mas também condiciona os resultados obtidos nas demais características avaliadas. Compreender essas abordagens é essencial para interpretar corretamente os resultados dos testes e, principalmente, para orientar a escolha da ferramenta mais adequada às necessidades específicas de cada projeto.

6 CONCLUSÕES E TRABALHOS FUTUROS

Os gerenciadores de pacotes são ferramentas indispensáveis durante o desenvolvimento de aplicações JavaScript, estas ferramentas simplificam a instalação e gerenciamento de pacotes, padronizando e organizando a produção e consumo dos mesmos. Este trabalho teve como principal objetivo analisar e comparar alguns dos gerenciadores de pacotes mais populares do ecossistema JavaScript - o NPM, o Yarn e o PNPM - e algumas de suas principais características, a fim de entender suas nuances, vantagens, desvantagens e como se comportam - fornecendo um guia prático que auxilia na escolha da alternativa mais adequada.

Os resultados apresentados mostram a forma como cada gerenciador se destaca e fornece uma análise geral que pode facilitar na tomada de decisão sobre qual ferramenta escolher. O PNPM se destaca como a alternativa mais balanceada, apresentando bom desempenho, menor consumo de memória em disco, boa taxa de captação e resolução de vulnerabilidades e boa compatibilidade geral com o ecossistema JavaScript. O Yarn Modern com PnP apresentou um excelente - e consistente - desempenho geral e também baixo consumo de memória em disco, mas não é capaz de corrigir vulnerabilidades de forma automática e pode apresentar problemas de compatibilidade - e requerer configurações adicionais -, características que devem ser levadas em consideração no momento da escolha. Já o NPM e o Yarn Modern com *node_modules* se destacam principalmente pela compatibilidade, pois resolvem as dependências de forma tradicional, o que é amplamente compatível com todo o ecossistema JavaScript. Por outro lado, o Yarn Classic e Yarn Modern com links endereçáveis não apresentam vantagens expressivas, o Yarn Classic obteve os piores resultados em quase todos os aspectos analisados - o que pode ser justificado pelo seu status atual -, já o Yarn Modern com links endereçáveis utiliza uma estratégia similar à do PNPM, mas com resultados gerais piores, o que não justifica sua escolha em detrimento deste.

Como propostas para trabalhos futuros, sugere-se a análise de outras características, como a experiência com a interface de linha de comando - CLI -, funcionalidades oferecidas, nível de suporte pela comunidade, entre outras. Além disso, pode-se também incluir outros gerenciadores de pacotes na comparação, como o Bun, que tem ganhado bastante destaque e popularidade. Uma outra faceta que foi citada durante a análise e pode ser aprofundada é como cada um dos gerenciadores é afetado pelo sistema de arquivos do sistema operacional.

REFERÊNCIAS

CHODOROWSKI, M. Comparative analysis of javascript package managers - yarn and npm. **Journal of Computer Sciences Institute**, v. 19, p. 75–80, Jun. 2021. Disponível em: https://ph.pollub.pl/index.php/jcsi/article/view/2460. Acesso em: 12 set. 2023.

COX, R. Surviving software dependencies: Software reuse is finally here but comes with risks. **Queue**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 2, p. 24–47, apr 2019. ISSN 1542-7730. Disponível em: https://doi.org/10.1145/3329781.3344149. Acesso em: 23 nov. 2023.

DECAN, A.; MENS, T.; CONSTANTINOU, E. On the impact of security vulnerabilities in the npm package dependency network. In: INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES, 15. **Proceedings**. New York, NY, USA: Association for Computing Machinery, 2018. (MSR '18), p. 181–191. ISBN 9781450357166. Disponível em: https://doi.org/10.1145/3196398.3196401. Acesso em: 26 nov. 2023.

GOLDWATER, M. **An abbreviated history of JavaScript package managers**. 2019. Disponível em: https://javascript.plainenglish.io/an-abbreviated-history-of-javascript-package-managers-f9797be7cf0e. Acesso em: 13 set. 2023.

INTERNATIONAL, E. **ECMAScript® 2015 Language Specification**. 2015. Disponível em: https://262.ecma-international.org/6.0/?_gl=1*g91f84*_ga*ODUxMzA2MjIxLjE3MDA3NjM0NTI.*_ga_TDCK4DWEPP*MTcwMDc3ODk3My4zLjEuMTcwMDc3OTE0NC4wLjAuMA..&_ga=2.208592471.47722604.1700763453-851306221.1700763452. Acesso em: 23 nov. 2023.

INTERNATIONAL, E. **ECMA-262**. 2023. Disponível em: https://ecma-international.org/publications-and-standards/standards/ecma-262/. Acesso em: 23 nov. 2023.

Ion Channel. **Dependency Detection**. 2023. Disponível em: https://docs.ionchannel.io/docs/dependency-detection. Acesso em: 31 out. 2023.

JACOBS, A. **Comparison of JavaScript package managers**. 2019. Disponível em: https://www.theseus.fi/handle/10024/227945. Acesso em: 12 set. 2023.

KOCHAN, Z. **Why should we use pnpm?** 2017. Disponível em: https://medium.com/pnpm/why-should-we-use-pnpm-75ca4bfe7d93. Acesso em: 16 nov. 2023.

MDN Web Docs. **JavaScript**. 2022. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript. Acesso em: 12 set. 2023.

MDN Web Docs. **Package management basics**. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Package_management. Acesso em: 12 set. 2023.

MDN Web Docs. **What is JavaScript?** 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript. Acesso em: 30 out. 2023.

NAKAZAWA, C.; MCKENZIE, S.; KYLE, J. **Yarn**: A new package manager for javascript. 2016. Disponível em: https://engineering.fb.com/2016/10/11/web/yarn-a-new-package-manager-for-javascript/. Acesso em: 14 nov. 2023.

NISON, M. **Release**: Yarn 2.0. 2020. Disponível em: https://yarnpkg.com/blog/release/2.0. Acesso em: 15 nov. 2023.

NPM. **NPM GitHub Repository**. 2018. Disponível em: https://github.com/npm/cli/. Acesso em: 13 nov. 2023.

NPM. **About npm**. 2023. Disponível em: https://www.npmjs.com/about. Acesso em: 13 set. 2023.

NPM. **npm**. 2023. Disponível em: https://www.npmjs.com. Acesso em: 20 set. 2023.

npm Docs. **folders**. 2023. Disponível em: https://docs.npmjs.com/cli/v10/configuring-npm/folders. Acesso em: 13 nov. 2023.

npm Docs. **install**. 2023. Disponível em: https://docs.npmjs.com/cli/v10/configuring-npm/install. Acesso em: 13 nov. 2023.

NPM Trends. **NPM vs PNPM vs Yarn**. 2023. Disponível em: https://npmtrends.com/npm-vs-pnpm-vs-yarn. Acesso em: 13 set. 2023.

OPENSUSE. **Package management**. 2010. Disponível em: https://en.opensuse.org/Package_management. Acesso em: 24 nov. 2023.

PEDROSO, M. G. **O que é o inferno de dependências e como utilizar o Dependabot**. 2022. Disponível em: https://www.alura.com.br/artigos/o-que-e-inferno-de-dependencias-como-utilizar-dependabot. Acesso em: 08 nov. 2023.

PNPM. **PNPM GitHub Repository**. 2016. Disponível em: https://github.com/pnpm/pnpm. Acesso em: 16 nov. 2023.

PNPM. **package.json**. 2019. Disponível em: https://pnpm.io/package_json. Acesso em: 24 nov. 2023.

PNPM. **Symlinked ‘node_modules‘ structure**. 2021. Disponível em: https://pnpm.io/symlinked-node-modules-structure. Acesso em: 16 nov. 2023.

PNPM. **Installation**. 2023. Disponível em: https://pnpm.io/installation. Acesso em: 16 nov. 2023.

PNPM. **pnpm vs npm**. 2023. Disponível em: https://pnpm.io/pt/pnpm-vs-npm. Acesso em: 13 set. 2023.

RATHORE, J. **Javascript for Web Development**. 2020. Disponível em: https://mnlabs.in/javascript-for-web-development-2/. Acesso em: 25 set. 2023.

RICKARD, M. **The Nine Circles of Dependency Hell (and a roadmap out)**. 2021. Disponível em: https://about.sourcegraph.com/blog/nine-circles-of-dependency-hell. Acesso em: 08 nov. 2023.

SNYK. **Software dependencies**: How to manage dependencies at scale. 2023. Disponível em: https://snyk.io/pt-BR/series/open-source-security/software-dependencies/. Acesso em: 31 out. 2023.

SPINELLIS, D. Package management systems. **IEEE Software**, v. 29, n. 2, p. 84–86, 2012.

TIOBE. **TIOBE Index**. 2023. Disponível em: https://www.tiobe.com/tiobe-index/. Acesso em: 20 set. 2023.

V8. **Documentation**. 2018. Disponível em: https://v8.dev/docs. Acesso em: 22 nov. 2023.

W3SCHOOLS. **JavaScript History**. 2023. Disponível em: https://www.w3schools.com/js/js_history.asp. Acesso em: 12 set. 2023.

WIRFS-BROCK, A.; EICH, B. Javascript: The first 20 years. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 4, n. HOPL, jun 2020. Disponível em: https://doi.org/10.1145/3386327.

YARN. **Yarn Classic GitHub Repository**. 2016. Disponível em: https://github.com/yarnpkg/yarn. Acesso em: 14 nov. 2023.

YARN. **Yarn Modern GitHub Repository**. 2019. Disponível em: https://github.com/yarnpkg/berry. Acesso em: 14 nov. 2023.

YARN. **Benefits**. 2023. Disponível em: https://yarnpkg.com/migration/overview. Acesso em: 15 nov. 2023.

YARN. **Installation**. 2023. Disponível em: https://yarnpkg.com/getting-started/install. Acesso em: 15 nov. 2023.

YARN. **Plug'n'Play**. 2023. Disponível em: https://yarnpkg.com/features/pnp. Acesso em: 15 nov. 2023.

YARN. **Questions & Answers**. 2023. Disponível em: https://yarnpkg.com/getting-started/qa. Acesso em: 14 nov. 2023.