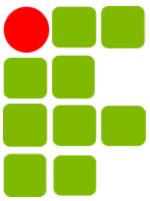


INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA – IFPB
CAMPUS CAMPINA GRANDE
COORDENAÇÃO DO CURSO SUPERIOR DE ENGENHARIA DE COMPUTAÇÃO

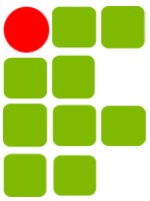
DOCKERYZER - DESENVOLVIMENTO DE UMA APLICAÇÃO DE LINHA DE COMANDO PARA CRIAÇÃO AUTOMÁTICA DE DOCKERFILES EM PROJETOS NODE.JS COM USO DE INTELIGÊNCIA ARTIFICIAL

JOÃO VICTOR NEGREIROS DA SILVA

JORGE VINÍCIUS SANTOS CASTRO

Orientador: Cesar Rocha Vasconcelos

Campina Grande, 2025
João Victor Negreiros da Silva
Jorge Vinícius Santos Castro



INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA – IFPB
CAMPUS CAMPINA GRANDE
COORDENAÇÃO DO CURSO SUPERIOR DE ENGENHARIA DE COMPUTAÇÃO

DOCKERYZER - DESENVOLVIMENTO DE UMA APLICAÇÃO DE LINHA DE COMANDO PARA CRIAÇÃO AUTOMÁTICA DE DOCKERFILES EM PROJETOS NODE.JS COM USO DE INTELIGÊNCIA ARTIFICIAL

JOÃO VICTOR NEGREIROS DA SILVA

JORGE VINÍCIUS SANTOS CASTRO

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação, do Instituto Federal da Paraíba – Campus Campina Grande, em cumprimento às exigências parciais para a obtenção do título de Bacharel em Engenharia da Computação.

Orientador: Cesar Rocha Vasconcelos

Campina Grande, 2025

Catalogação na fonte:
Ficha catalográfica elaborada por Gustavo César Nogueira da Costa - CRB 15/479

S586c Silva, João Victor Negreiros da
DOCKERYZER - Desenvolvimento de uma aplicação de linha de comando para criação automática de dockerfiles em projetos NODE.JS com uso de inteligência artificial / João Victor Negreiros da Silva, Jorge Vinícius Santos Castro. - Campina Grande, 2025.
28 f. : il.

Trabalho de Conclusão de Curso (Curso Superior de Engenharia de Computação) - Instituto Federal da Paraíba, 2025.
Orientador: Prof. Cesar Rocha Vasconcelos

1. Engenharia de computação 2. Inteligência artificial - OpenAI 3. Software open source - Node.js I. Castro, Jorge Vinícius Santos II. Vasconcelos, Cesar Rocha III.Título.

CDU 004.8

SUMÁRIO

1. Introdução **6**
1.1. Relevância do Tema 6
1.2. Objeto de Pesquisa 7
1.3. Objetivos de Estudo 7
1.4. Justificativa 8
1.5. Estrutura do Trabalho 9
2. Referencial Teórico **10**
2.1. Go 10
2.2. Docker 10
2.3. Node.js 12
2.4. OpenAI 12
3. Metodologia **13**
3.1. Pesquisa 13
3.3. Inteligência artificial 15
3.4. Trabalhos semelhantes 16
4. Dockeryzer **17**
4.1. Arquitetura 17
4.2. Aplicação 17
5. Resultados **22**
5.1. Comparação de Tamanho das Imagens 24
5.2. Número de Camadas das Imagens 25
5.3. Versão do Node.js 25
6. Conclusão e Melhorias Futuras **26**
6.1. Conclusão 26
6.2. Melhorias Futuras 26
Referências **28**

Resumo

Este trabalho automatiza e simplifica a criação de imagens *Docker* por meio de uma aplicação de alto nível e um *CLI* (*Command-Line Interface*) especializado, utilizando inteligência artificial, ajudando desenvolvedores a otimizar desempenho sem a necessidade de lidar com comandos complexos. A solução proposta visa reduzir a complexidade do processo de criação de imagens *Docker*, tornando-o mais automatizado e eficiente para desenvolvedores menos experientes. Além disso, melhora a confiança e a satisfação na implantação de aplicações em containers, simplificando a experiência do usuário.

Palavras-chave: *Docker*, *Dockerfile*, *Containers*, *Node.js*, *OpenAI*.

Abstract

This work presents a solution, called Dockerizer, that automates and simplifies the creation of Docker images through a high-level application and a specialized CLI that leverages artificial intelligence to help developers optimize performance without the need for complex commands. Dockerizer allows developers to easily create Dockerfiles, while reducing the complexity of the process, making it automated, and simple for less experienced developers. Additionally, it promotes confidence and satisfaction in the deployment of containerized applications, simplifying the user experience.

Keywords: *JavaScript, Docker, Command-Line Interface, OpenAI.*

1. Introdução

1.1. Relevância do Tema

No contexto do desenvolvimento de software, a criação de imagens de *containers Docker*[1] (DOCKER, 2024) eficientes pode se tornar um desafio, especialmente para projetos que demandam escalabilidade e rapidez na sua implantação. Hoje em dia, os desenvolvedores têm enfrentado dificuldades na criação e otimização dessas imagens de aplicações. Não é incomum que estes profissionais tenham, em muitos casos, de realizar processos manuais complexos, ineficazes e que produzem imagens finais que consomem mais recursos do que o necessário.

Atualmente, percebe-se uma ausência de ferramentas que possam automatizar essa tarefa de forma simples e eficaz, o que força os desenvolvedores a utilizar métodos manuais, como digitação de instruções em terminal de linha de comando e a edição de arquivos de configuração. Além disso, é comum ver desenvolvedores realizarem a pesquisa em documentações, fóruns e até o uso de inteligência artificial para formular boas soluções para criar estes tipos de imagens. Embora utilizar um modelo de linguagem para gerar comandos necessários para criação de imagens de aplicações possa parecer uma alternativa mais simples, essa tarefa ainda exige um esforço considerável para formular o *prompt* para IA (Inteligência Artificial) adequado e interpretar as respostas recebidas. Em particular, o nível de detalhe e clareza no *prompt* é crucial, posto que um *prompt* vago ou mal estruturado pode gerar respostas inadequadas, produzindo imagens *Docker* que não atendem às necessidades específicas do projeto, o que pode ser tanto demorado quanto complexo.

Para resolver esse problema, este trabalho apresenta uma solução alternativa, o *Dockeryzer* (DOCKERYZER, 2024), que surge como uma ferramenta inovadora, capaz de automatizar e simplificar a criação de imagens *Docker* de aplicações *Node.js*[2] (NODE, 2024). Especificamente desenvolvida

---

[1] Pacotes imutáveis que contêm o código, dependências e configurações necessárias para executar uma aplicação em qualquer ambiente compatível com Docker.

[2] Ambiente de execução JavaScript assíncrono e baseado em eventos, utilizado principalmente no desenvolvimento de aplicações backend.

com a linguagem *Golang* (GO, 2024), a solução apresentada neste trabalho utiliza o *Docker* (DOCKER, 2024) para gerenciar contêineres e o *Git* (GIT, 2024) para controle de versões. O diferencial do *Dockeryzer* é a integração com a inteligência artificial da *OpenAI*, que permite a geração automática de arquivos *Dockerfiles* otimizados, seguindo boas práticas de desenvolvimento.

Em particular, o leitor verá que, com apenas um comando, o *Dockeryzer* elimina a necessidade de pesquisas extensivas e da formulação de *prompts* complexos, tornando o processo de criação de imagens mais ágil e menos propenso a erros.

1.2. Objeto de Pesquisa

O problema em análise apresenta a necessidade de simplificar e otimizar a criação de imagens *Docker* para projetos *Node.js*, sobretudo para desenvolvedores menos experientes. Atualmente, a criação de imagens ainda envolve a formulação e digitação manual de comandos complexos, o que aumenta o risco de erros e pode reduzir a eficiência do resultado final. Uma possível solução para esse desafio é o desenvolvimento de uma aplicação de linha de comando que automatize esse processo, gerando imagens otimizadas e reduzindo a complexidade da configuração. Dessa forma, espera-se que essa abordagem contribua para aumentar a produtividade dos desenvolvedores, favorecendo um desenvolvimento mais ágil e também seguro.

1.3. Objetivos de Estudo

Nesta seção, são expostos os objetivos gerais e específicos do estudo, para alcançar o resultado esperado.

1.3.1. Objetivo Geral

O objetivo geral deste trabalho é o desenvolvimento de uma aplicação de linha de comando que automatize a criação de imagens *Docker* para aplicações *Node.js*, seguindo as melhores práticas para garantir a geração de imagens funcionais leves e com uma performance satisfatória.

1.3.2. Objetivos Específicos

Abaixo, são elencados os seguintes objetivos específicos:

- Identificar os desafios e limitações na criação manual de imagens *Docker* para aplicações *Node.js*;
- Analisar artigos e trabalhos relacionados que abordam a criação e otimização de imagens *Docker*;
- Selecionar as tecnologias e ferramentas estado da arte mais apropriadas para o desenvolvimento da aplicação de linha de comando e automação de *Dockerfiles*;
- Documentar e explicar detalhadamente as melhores práticas e estratégias de otimização incorporadas na ferramenta.
- Integrar tecnologias de inteligência artificial no processo de geração e análise de arquivos de configuração *Docker* (*Dockerfiles*), permitindo que o sistema utilize modelos de linguagem para sugerir configurações otimizadas e personalizadas para a criação de imagens otimizadas.

1.4. Justificativa

Este trabalho, na forma de uma aplicação de linha de comando, tem como objetivo consolidar os conceitos adquiridos durante a graduação em Engenharia da Computação, aplicando-os ao desenvolvimento de uma ferramenta para automatizar a criação de imagens *Docker* otimizadas para aplicações *Node.js*. Através do uso das melhores práticas de criação de imagens *Docker* e técnicas de otimização, pretende-se demonstrar a importância de seguir diretrizes que garantam a geração de imagens leves e com desempenho satisfatório.

A criação manual de imagens *Docker* é uma tarefa não trivial, que exige um conhecimento aprofundado e a elaboração de várias instruções específicas no arquivo *Dockerfile*. Um erro comum entre desenvolvedores menos experientes é utilizar a imagem base errada, como *node:latest*, que pode resultar em imagens grandes e desnecessariamente pesadas. Além disso, o mau uso das instruções *RUN*, *COPY* e *ADD* pode gerar camadas redundantes,

aumentando o tamanho da imagem. Outro problema frequente é a falta de otimização no gerenciamento de dependências, como copiar todos os arquivos do projeto antes de instalar os pacotes, o que impede o aproveitamento do *cache* da *build* e torna o processo mais lento. Essas falhas podem levar a imagens que ocupam mais recursos da máquina e apresentam um desempenho inferior.

Com o *Dockeryzer*, busca-se mitigar esses problemas ao automatizar a geração de um *Dockerfile* otimizado, reduzindo a complexidade, o tempo e o esforço necessários para criar imagens eficientes. Além disso, a ferramenta simplifica o aprendizado de desenvolvedores menos experientes, pois oferece uma documentação clara e explicações detalhadas sobre todas as otimizações realizadas.

Portanto, a criação desta ferramenta se justifica como uma solução essencial para melhorar o desenvolvimento de projetos que utilizam *Docker*, garantindo que as imagens finais sigam as melhores práticas e sejam otimizadas desde o início.

1.5. Estrutura do Trabalho

Este trabalho é dividido em seis seções, sendo esta a primeira. Nela, são abordadas as considerações iniciais, a relevância e a contextualização do estudo, incluindo os objetivos e a justificativa.

A Seção 2 aborda o referencial teórico definido para o desenvolvimento deste projeto, destacando as tecnologias e ferramentas utilizadas, além de trabalhos relacionados ao tema.

Na Seção 3, é descrita a metodologia aplicada na formulação do trabalho para atingir os objetivos propostos.

A Seção 4 apresenta a arquitetura do projeto e os comandos implementados na ferramenta.

A Seção 5 apresenta um estudo experimental realizado, além dos resultados alcançados e comparações que demonstram a eficiência da solução proposta.

Por fim, a Seção 6 traz as conclusões e sugestões de melhorias para a continuidade futura.

2. Referencial Teórico

2.1. Go

De acordo com o repositório oficial do *Golang* (2024), *Golang*, ou simplesmente *Go,* é uma linguagem de programação de código aberto que torna fácil construir *softwares* simples, confiáveis e eficientes.

Go é uma linguagem de propósito geral e encontra uso em domínios como *drivers* e gráficos, aplicativos móveis e aprendizado de máquina. “Os programas *Go* normalmente são executados mais rápido do que os programas escritos em linguagens dinâmicas e sofrem muito menos travamentos devido a erros de digitação inesperados”, conforme visto em (DONOVAN, 2015).

A facilidade da linguagem permite o desenvolvimento de um grande número de bibliotecas (também conhecidas por *packages*) que expandem suas capacidades. Dentre elas, podemos destacar a biblioteca *Cobra*, projetada para facilitar a criação de interfaces de linha de comando (*CLI*).

2.1.1. Cobra

Cobra é uma biblioteca desenvolvida para *Golang* que facilita a criação de soluções *CLI*, oferecendo suporte a comandos aninhados, opções (*flags)* compatíveis com *POSIX (Portable Operating System Interface)* e um autocompletar com suporte para *shell*. Segundo a documentação oficial, a biblioteca foi desenvolvida para criar “aplicativos *CLI* modernos e poderosos” (COBRA, 2024).

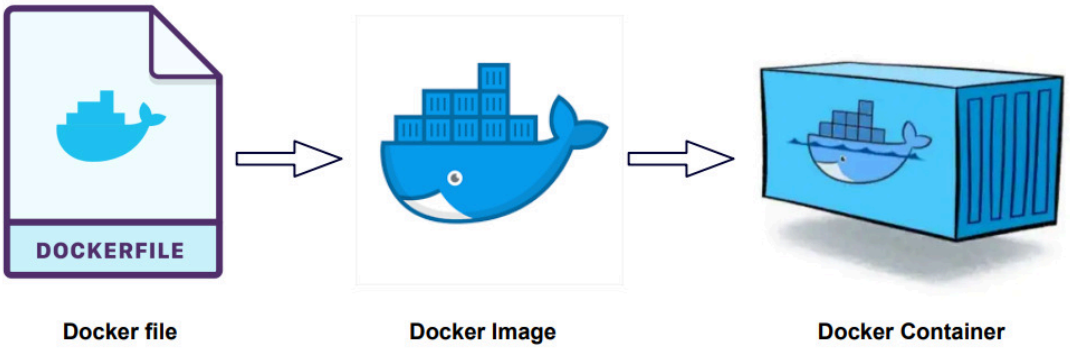
2.2. Docker

Docker é uma plataforma de código aberto que facilita a virtualização no nível do sistema operacional (SO). Com ela é possível isolar e “empacotar” aplicações junto com suas dependências, assegurando que o *software* seja executado de forma consistente em diferentes ambientes. O *Docker* objetiva,

segundo Turnbull (2014), a diminuição do ciclo de tempo entre a escrita do código e seu processo de teste, implantação e uso.

Entre os principais componentes do *Docker* estão as imagens e os *containers*. A Figura 1 ilustra o fluxo de trabalho básico do *Docker*.

Figura 1 - Processo de construção e execução no Docker



Fonte: (AREF, 2023)

O processo inicia-se com o *Dockerfile*, que funciona como um conjunto de instruções para a construção da aplicação. A partir deste arquivo, é gerada uma imagem *Docker*, que contém todos os componentes necessários para a aplicação, mas ainda não está em execução. Por fim, ao executar a imagem, obtemos um *container Docker*, que é a aplicação encapsulada e em pleno funcionamento.

2.2.1. Container

Um *container Docker* é um espaço isolado para executar aplicações e serviços, que pode conter um ou mais processos em andamento. São semelhantes a máquinas virtuais, porém, de acordo com Fernando (2023), são mais leves e mais integrados ao sistema operacional da máquina *host*, o que proporciona melhor desempenho por conta do gerenciamento único dos recursos.

O autor Turnbull (2014) afirma que o *Docker* se inspira nos *containers* físicos usados para transporte de mercadorias, mas, no caso de seus *containers*, o que é "transportado" são pacotes de *software*.

Cada *container* é criado a partir de uma imagem, que denota a etapa de construção (*build*), enquanto o *container* em si corresponde à etapa de execução (*deploy*).

2.2.2. Imagem

A documentação oficial do *Docker*, define uma imagem como “um pacote padronizado que inclui todos os arquivos, binários, bibliotecas e configurações para executar um *container*.” (DOCKER, 2024). Turnbull (2014) complementa, definindo uma imagem como um composto de sistemas de arquivos dispostos em camadas.

De forma geral, a imagem atua como o plano de construção de um *container*, reunindo todas as informações e recursos necessários para se criar a instância do serviço.

2.3. Node.js

Criado em 2009, pelo engenheiro de *software* Ryan Dahl, o *Node.js* é “um ambiente de execução *JavaScript* gratuito, de código aberto e multiplataforma que permite aos desenvolvedores criar servidores, aplicativos da *web*, ferramentas de linha de comando e *scripts*.” (NODE.JS, sd).

A praticidade dessa ferramenta contribui significativamente para posicionar o *JavaScript* no topo das linguagens mais utilizadas pelos desenvolvedores, permitindo a criação fácil e integrada de soluções *front-end* e *back-end*.

2.4. OpenAI

A *OpenAI* é uma organização dedicada à pesquisa e ao desenvolvimento de inteligência artificial, com o objetivo de garantir que seus avanços sejam seguros e amplamente acessíveis. Fundada em 2015, a empresa busca criar sistemas de IA que possam beneficiar a sociedade

como um todo, com foco em segurança, ética e transparência. Entre suas principais contribuições está o desenvolvimento do modelo *GPT* (*Generative Pre-trained Transformer*), amplamente utilizado em diversas aplicações por sua capacidade de interpretar o contexto e gerar textos com elevado grau de precisão e relevância (BROWN et al., 2020).

2.4.1. GPT

O modelo *GPT* (*Generative Pre-trained Transformer*), desenvolvido pela *OpenAI*, é um avanço no processamento de linguagem natural, utilizando uma arquitetura de *Transformers* para análise e geração de texto de forma contextual e coerente. Sua versatilidade permite aplicações em áreas como automação de atendimento, suporte ao desenvolvimento de *software* e geração de relatórios, demonstrando o potencial da inteligência artificial para aprimorar processos em diferentes contextos.

É importante ressaltar que o *Dockeryzer* depende do *ChatGPT/OpenAI*, que não são totalmente gratuitos. Como alternativa futura, considera-se a migração para o *DeepSeek*, uma ferramenta de código aberto, que pode oferecer uma solução viável e econômica para a automação de criação de *Dockerfiles* otimizados.

3. Metodologia

Neste capítulo, são detalhados os procedimentos adotados para o desenvolvimento deste trabalho, abrangendo as etapas de planejamento, as ferramentas utilizadas, a definição do ambiente experimental e as técnicas aplicadas para a obtenção e análise dos dados.

3.1. Pesquisa

Inicialmente, para fundamentar as melhores práticas na criação de *containers Docker*, foram analisados artigos que forneceram orientações técnicas e estratégicas indispensáveis para uma configuração otimizada e segura de *Docker* para aplicações *Node.js*.

O primeiro trabalho (*Specfy*, 2023), explora aspectos como a otimização de camadas de imagem, o uso de instruções específicas para minimizar o tamanho final da imagem e a relevância de ambientes de construção isolados para aprimorar o desempenho e a segurança dos *containers* .

Complementando essas recomendações, o trabalho realizado por Eduardo Rabelo, em (Rabelo, 2023), apresenta dez práticas essenciais, abordando desde a seleção de uma imagem base leve e apropriada até o uso eficaz de variáveis de ambiente e estratégias de *caching*[3], elementos fundamentais para otimização de desempenho. Adicionalmente, o artigo enfatiza a importância de estabelecer limites de recursos, como *CPU (Central Processing Unit)* e memória, para assegurar a eficiência e a estabilidade das aplicações em ambientes de produção.

3.2. Implementação

Para o desenvolvimento da aplicação em linha de comando (*CLI*), utilizou-se a linguagem de programação *Golang* em conjunto com a biblioteca *Cobra*, destinada à construção da interface de linha de comando. O ambiente de desenvolvimento escolhido foi o *Visual Studio Code* (MICROSOFT, 2024), uma vez que oferece suporte avançado tanto para *Go* quanto para *Docker*, facilitando o processo de codificação e depuração. O *Docker* foi utilizado na construção e gerenciamento dos contêineres, com ênfase nas melhores práticas de otimização e segurança previamente estudadas.

A pesquisa realizada resultou na aplicação de uma série de processos de boas práticas com o objetivo de garantir a criação de imagens *Docker* mais leves e performáticas para aplicações *Node.js*. Em primeiro lugar, optou-se pela utilização de imagens *Alpine* oficiais do *Node.js*. Essa escolha foi fundamentada na necessidade de reduzir o tamanho das imagens, o que não só facilita o *download* e a distribuição, mas também melhora o desempenho em tempo de execução, uma vez que menos recursos da máquina são utilizados. Em seguida, a implementação de *multi-stage builds*[4] foi adotada como uma

[3] Técnica de armazenamento temporário de dados frequentemente acessados para melhorar o desempenho e reduzir o tempo de processamento.

[4] Técnica do Docker que permite a criação de imagens mais leves ao dividir o processo de construção em múltiplos estágios, descartando arquivos desnecessários na imagem final.

estratégia crucial para a redução do tamanho final da imagem. Conforme descrito na documentação oficial do *Docker*, essa técnica permite que diferentes estágios de construção sejam utilizados, de modo que as dependências de desenvolvimento sejam instaladas em uma fase inicial e, posteriormente, apenas os artefatos necessários sejam copiados para a imagem final, eliminando a inclusão de pacotes desnecessários (DOCKER, 2024).

Adicionalmente, é realizada a remoção automática de arquivos temporários e *caches* que, se não fossem tratados, poderiam aumentar consideravelmente o tamanho da imagem final. Essa etapa é vital para manter a imagem leve e eficiente, uma vez que muitos processos de construção geram arquivos que não são necessários para a execução da aplicação. Por fim, a configuração otimizada da imagem foi ajustada para garantir que somente as dependências de produção fossem incluídas. Isso não apenas minimiza o tamanho da imagem, mas também assegura que a aplicação esteja livre de dependências desnecessárias que poderiam comprometer a segurança e a estabilidade em ambientes de produção. Juntas, essas práticas formam uma metodologia robusta que não apenas atende às recomendações da literatura, mas também resulta em imagens *Docker* mais eficientes e seguras para aplicações *Node.js*.

3.3. Inteligência artificial

O projeto incorpora uma *API (Application Programming Interface)* de inteligência artificial, utilizando o modelo *GPT-4o* para a geração automática de *Dockerfiles* otimizados. O sistema inicia com uma pré-análise do projeto no qual o comando é executado, realizando uma leitura detalhada de todas as dependências e estruturas associadas ao projeto. Este processo permite ao código identificar os requisitos específicos e as melhores práticas a serem consideradas, resultando na formulação de um *prompt* altamente personalizado para a IA. Dessa forma, a inteligência artificial é capaz de gerar um Dockerfile que não apenas atende às demandas do projeto, mas também garante funcionalidade e segurança adequadas. Nos casos em que ocorra algum erro

na comunicação com com a *API* da *OpenAI*, o sistema recorre à utilização de templates alternativos, *Dockerfiles* previamente escritos, os quais também foram configurados para seguir as boas práticas de desenvolvimento.

Por motivos de segurança, o projeto foi desenvolvido de forma a garantir a privacidade e a proteção dos dados manipulados. Nesse sentido, os conteúdos completos dos arquivos do projeto não são enviados para a *API* de inteligência artificial; apenas o arquivo *package.json*, que contém as dependências do projeto, é utilizado na formulação do *prompt* para a geração do *Dockerfile*. Essa abordagem minimiza o risco de exposição de informações sensíveis, permitindo que o sistema utilize a IA de maneira eficiente e responsável, mantendo a confidencialidade dos dados do usuário enquanto otimiza a criação de imagens *Docker*.

3.4. Trabalhos semelhantes

Durante a pesquisa, foi encontrada uma ferramenta semelhante chamada *DockerSlim*. Embora ambos os projetos tenham como objetivo a otimização de imagens *Docker*, suas abordagens diferem significativamente. O *DockerSlim* reduz o tamanho de imagens existentes ao remover arquivos e dependências não utilizados durante a execução do *container*, sem exigir modificações no *Dockerfile* original. Já o *Dockeryzer* se concentra na geração automatizada de *Dockerfiles* otimizados desde o início, garantindo boas práticas sem necessidade de ajustes manuais. Além disso, o *Dockeryzer* inclui um comando para comparar imagens, recurso não disponível no *DockerSlim*. Outra limitação do *DockerSlim* é a falta de um mecanismo alternativo caso a otimização não seja bem-sucedida, enquanto o *Dockeryzer* garante que um *Dockerfile* funcional seja gerado mesmo se a criação otimizada falhar.

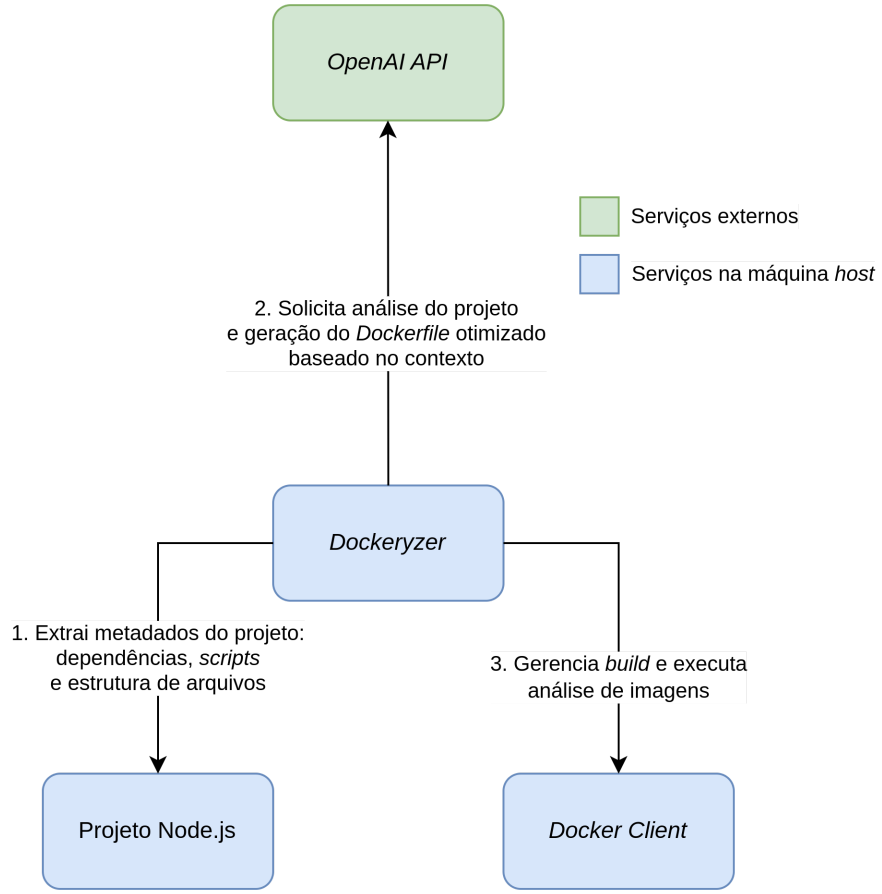
O *Dockeryzer* é uma opção mais acessível para desenvolvedores iniciantes, pois automatiza a criação de um *Dockerfile* otimizado sem exigir conhecimento avançado. Já o *DockerSlim* requer uma imagem funcional prévia e um entendimento sobre a remoção de arquivos e camadas, tornando seu uso mais complexo.

4. Dockeryzer

Neste capítulo, são apresentados os detalhes da ferramenta *Dockeryzer*, incluindo a arquitetura do projeto e suas funcionalidades.

4.1. Arquitetura

Figura 2 - Processo de construção e execução no Docker



Fonte: Os próprios autores (2024)

4.2. Aplicação

A aplicação desenvolvida inclui a implementação de três comandos principais, projetados para atender a diferentes necessidades dos usuários ao interagir com a ferramenta. Esses comandos foram elaborados para oferecer funcionalidades específicas e intuitivas, tornando o uso da aplicação eficiente e

acessível mesmo para quem possui conhecimentos básicos em linha de comando.

4.2.1. Comando *Create*

O comando *create* é responsável por gerar automaticamente um *Dockerfile* otimizado para um projeto *Node.js*. O processo começa com a análise da estrutura do projeto (arquivos de configuração e de dependências). Em seguida, é gerado um prompt com base nas informações coletadas e é utilizado um modelo de linguagem para criar o *Dockerfile* final, denominado *Dockeryzer*, que é salvo na raiz do projeto. A ferramenta implementa um robusto tratamento de erros. Se a geração do *Dockerfile* com IA falhar, o Dockeryzer automaticamente recorre a um *Dockerfile* padrão, que contém configurações básicas e funcionais para o projeto. Isso garante que o usuário não fique sem um *Dockerfile*, permitindo que o desenvolvimento continue sem interrupções, mesmo em situações imprevistas.

Tabela 1: Parâmetros disponíveis para o comando *create*

Parâmetro	Descrição
-n, --imageName	Define o nome da imagem a ser criada.
-i, --ignore-comments	Gera o *Dockerfile* sem comentários explicativos.

Fonte: Autores (2024)

As imagens a seguir mostram os possíveis *outputs* gerados pelo comando *create*.

Figura 3: *Output* do comando *create* com a *flag --help*

```
→ dockeryzer create --help
Command to generate a Dockerfile and create an Docker image (optional)

Usage:
  dockeryzer create [flags]

Flags:
  -h, --help               help for create
  -i, --ignore-comments    No include comments to Dockerfile
  -n, --imageName string   Image imageName to create
```

Fonte: Autores (2024)

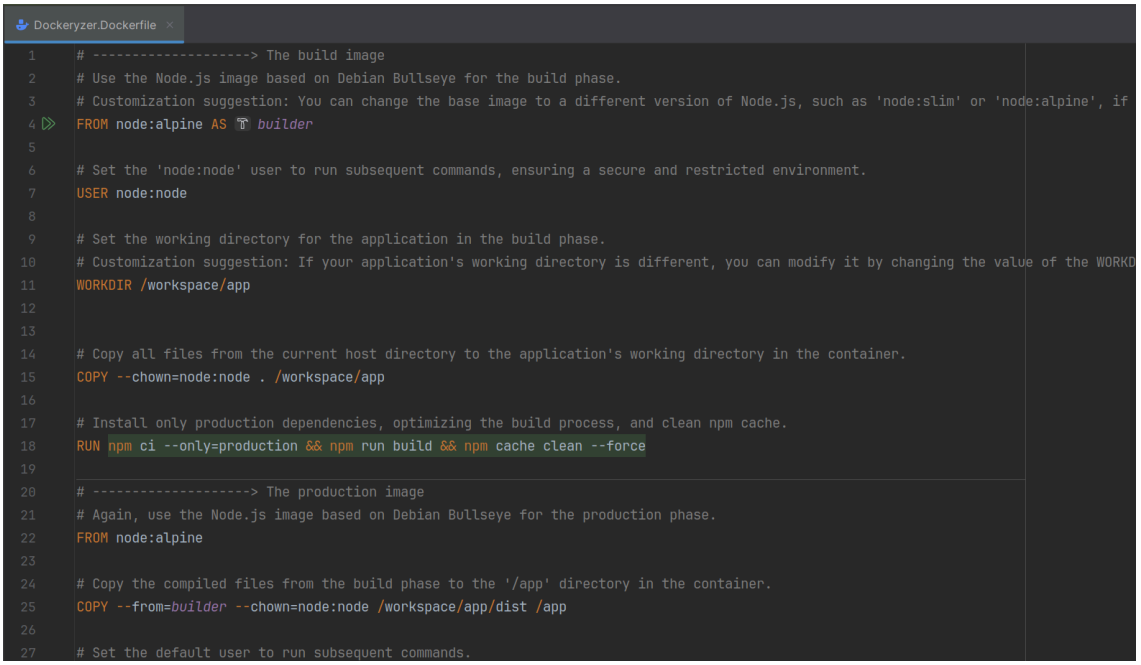
Figura 4: *Output* do comando *create* sem nenhuma *flag* informada

```
→ dockeryzer create
New files:
        Dockeryzer.Dockerfile
        .dockerignore

To build your image, run one of the following commands::
- To specify a imageName for the image:
        docker build -t <image-imageName> -f Dockeryzer.Dockerfile .
- To build without specifying a imageName:
        docker build -f Dockeryzer.Dockerfile .
```

Fonte: Autores (2024)

Figura 5: Arquivo *Dockeryzer.Dockerfile* gerado pelo comando (com comentários explicativos)

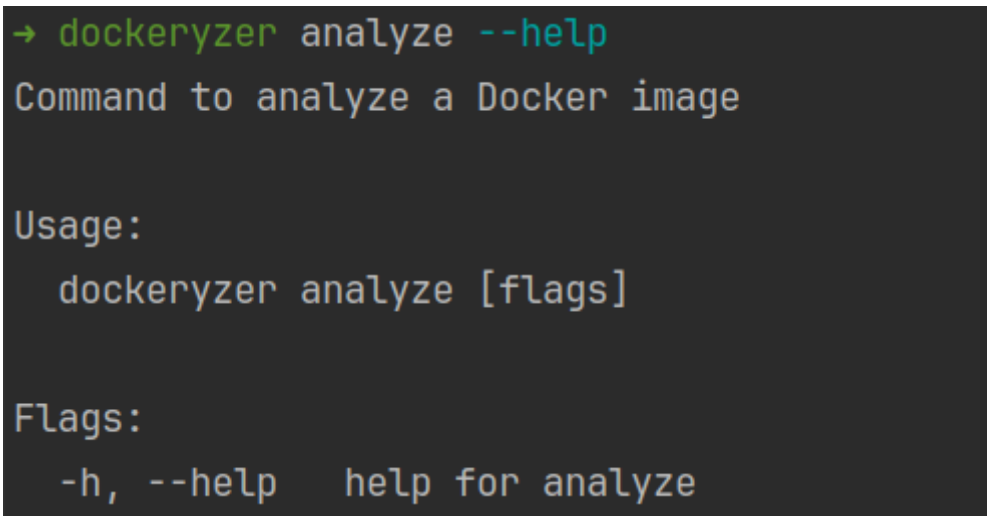


Fonte: Autores (2024)

4.2.2. Comando *Analyze*

O comando *analyze* permite ao usuário examinar detalhadamente uma imagem *Docker*, fornecendo informações como: tamanho total da imagem, número de camadas, versão do *Node.js* e pontos de otimização possíveis.

Figura 6: *Output* do comando *analyze* com a *flag --help*



Fonte: Autores (2024)

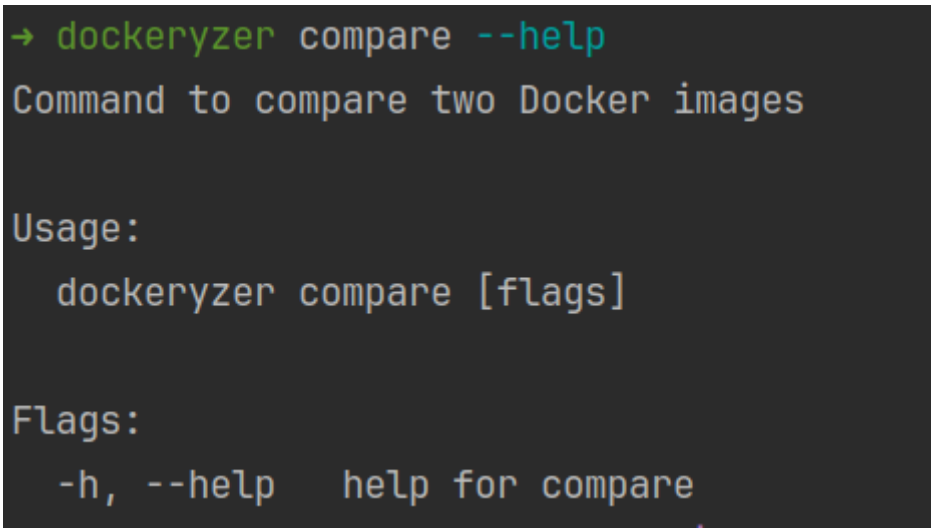
Figura 7: *Output* do comando *analyze* com o nome de uma imagem informada como parâmetro

Fonte: Autores (2024)

4.2.3. Comando *Compare*

O comando *compare* possibilita a comparação entre duas imagens Docker, destacando diferenças em aspectos como: tamanho das imagens, número de camadas, versão do Node.js e comparações entre ambas.

Figura 8: *Output do comando compare com a flag --help*



Fonte: Autores (2024)

Figura 9: *Output* do comando *compare* com o nome de duas imagem informada como parâmetro

Fonte: Autores (2024)

5. Resultados

Para validar a eficiência da aplicação, foram realizados testes comparativos entre as imagens geradas pela ferramenta e aquelas criadas manualmente com *Dockerfiles* convencionais, utilizando o modelo de linguagem *GPT-4o*.

Os prompts gerados para a comparação com a *CLI* consistiram em instruções simples que não especificam nenhum arquivo do projeto ou quais dependências cada um possuía. Em vez disso, o comando apenas indicava se a aplicação era do tipo *back-end* ou *front-end*, juntamente com a biblioteca de *Node.js* utilizada. A motivação para utilizar um comando tão básico foi simular a situação de um desenvolvedor iniciante que, sem experiência prévia, tentaria gerar um *Dockerfile*. Essa comparação não apenas elucidou a eficiência da aplicação *CLI*, mas também ressaltou a relevância de um *prompt* bem elaborado para garantir a geração de *Dockerfiles* que atendam às necessidades específicas de cada projeto.

Os testes foram realizados utilizando quatro projetos diferentes: uma aplicação *back-end* feita com *Express.js* (NODE_JWT, 2024), uma aplicação *back-end* feita com *Nest.js* (API_JWT_NEST, 2024), uma aplicação *front-end*

feita com *Vite.js* (LOCAL_LIST, 2024) e uma aplicação *front-end* feita com *Next.js* (YOUTUBENEXT, 2024).

Os *prompts* utilizados para a geração dos *Dockerfiles* de cada projeto foram os seguintes:

Tabela 2: *Prompts* utilizados para a geração dos *Dockerfiles*

Projeto	Prompt
Express.js	*Me forneça o conteúdo de um arquivo Dockerfile para um projeto back-end feito em Node.js com a biblioteca Express.js.*
Nest.js	*Me forneça o conteúdo de um arquivo Dockerfile para um projeto back-end feito em Node.js com a biblioteca Nest.js.*
Vite.js	*Me forneça o conteúdo de um arquivo Dockerfile para um projeto front-end feito em Node.js com a biblioteca Vite.js.*
Next.js	*Me forneça o conteúdo de um arquivo Dockerfile para um projeto front-end feito em Node.js com a biblioteca Next.js. Use o pnpm como package manager.*

Fonte: Autores (2024)

Os *prompts* utilizados para a geração dos *Dockerfiles* apresentaram algumas inconsistências que impactaram a eficiência dos resultados. Um dos principais problemas foi observado no *prompt* destinado à aplicação *Next.js*. Este *prompt* teve que ser adaptado, pois a versão genérica do *ChatGPT* não conseguia nem mesmo realizar o *build* da aplicação corretamente. Isso evidenciou a necessidade de um maior detalhamento nas instruções fornecidas ao modelo, uma vez que prompts vagos podem levar a resultados insatisfatórios ou até inviáveis.

Além disso, durante o processo de avaliação, alguns dos *Dockerfiles* gerados também precisaram ser adaptados. Um exemplo notável foi o comando de execução do *container*, que não estava configurado de maneira ideal em alguns casos. Essas adaptações foram necessárias para assegurar que as aplicações pudessem ser executadas corretamente em ambientes de desenvolvimento e produção. Essa experiência destacou a importância de testar e ajustar os arquivos gerados, mesmo quando originados de uma ferramenta de inteligência artificial.

Por fim, os *Dockerfiles* utilizados pelo *ChatGPT* não refletiram consistentemente a capacidade do modelo de linguagem. Em algumas situações, foram gerados *Dockerfiles* bem otimizados, enquanto em outras, as soluções propostas eram subótimas. Essa abordagem permitiu uma análise mais crítica da geração automática de arquivos *Docker* e enfatizou a necessidade de um *prompt* mais detalhado e específico para garantir a criação de imagens *Docker* que atendam às melhores práticas de otimização e segurança.

Todos os arquivos *Dockerfiles* utilizados podem ser encontrados no repositório de cada projeto.

5.1. Comparação de Tamanho das Imagens

Tabela 3: Resultado do tamanho das imagens

Projeto	*Dockerfile* GPT	*Dockerfile* *Dockeryzer*	Resultado
Express.js	1.11 GB	180.58 MB	83.80% menor
Nest.js	1.35 GB	213.48 MB	84.15% menor
Vite.js	47.41 MB	47.41 MB	<0%
Next.js	608.85 MB	608.87 MB	<0%

Fonte: Autores (2024)

5.2. Número de Camadas das Imagens

Tabela 4: Resultado do número de camadas das imagens

Projeto	*Dockerfile* GPT	*Dockerfile Dockeryzer*	Resultado
Express.js	12	7	5 camadas a menos
Nest.js	13	8	5 camadas a menos
Vite.js	9	9	-
Next.js	9	11	2 camadas a mais

Fonte: Autores (2024)

5.3. Versão do Node.js

Tabela 5: Resultado da versão do Node.js

Projeto	*Dockerfile* GPT	*Dockerfile Dockeryzer*	Resultado
Express.js	18.20.4	23.1.0	5 versões mais recentes
Nest.js	18.20.4	23.1.0	5 versões mais recentes
Vite.js	Não identificado	Não identificado	Versão Node.js não identificada

Next.js	18.20.4	18.20.4	Mesma versão

Fonte: Autores (2024)

6. Conclusão e Melhorias Futuras

6.1. Conclusão

Conclui-se que o *Dockeryzer* demonstrou significativa eficiência em relação aos *Dockerfiles* gerados manualmente via *ChatGPT*, especialmente em aspectos como otimização de tamanho das imagens e número de camadas. Nos projetos de *back-end*, o *Dockeryzer* gerou imagens até 84% menores e com menos camadas, refletindo uma estrutura de container mais enxuta e eficiente. Em projetos *front-end*, a similaridade no tamanho das imagens é atribuída ao processo de *build* que já realiza otimizações inerentes, embora o *Dockeryzer* se destaque pela inclusão de documentação *inline* e exemplos de execução. No caso específico do *Next.js*, o *Dockeryzer* implementa camadas adicionais essenciais para a cópia da pasta *public* e do arquivo *next.config.js*, elementos críticos ignorados pelo *ChatGPT*. A não identificação da versão do *Node.js* em determinados projetos é resultado da sua ausência na imagem final, visto que este não é necessário para a execução da aplicação em produção. Dessa forma, a aplicação mostrou-se uma solução prática e robusta para desenvolvedores, promovendo a otimização dos *Dockerfiles* de maneira automatizada, eficiente e alinhada às melhores práticas de criação de *containers*.

6.2. Melhorias Futuras

Durante o desenvolvimento deste trabalho, foram identificadas diversas oportunidades de evolução e aprimoramento da ferramenta *Dockeryzer*. Uma das principais melhorias propostas é a expansão da capacidade de reconhecimento de contexto da aplicação, permitindo que ela identifique e otimize *Dockerfiles* para outros *frameworks* além do *Node.js*, como *Python* com *Django* ou *Flask*, *Java* com *Spring Boot*, e *Ruby on Rails*. Esta adaptação

permitiria que a ferramenta atendesse uma gama mais ampla de desenvolvedores e projetos.

Outra melhoria significativa envolve a integração do *Dockeryzer* com ferramentas de *CI/CD (Continuous Integration/Continuous Delivery)*, permitindo a automação completa da geração e otimização de *Dockerfiles* no fluxo de desenvolvimento, o que promove eficiência e padronização nos ambientes de *build* e *deploy*. Essa integração possibilita que o *Dockeryzer* seja executado a cada nova alteração no código, gerando *Dockerfiles* otimizados automaticamente, que são posteriormente testados e implantados nos ambientes de produção. Dessa forma, a ferramenta assegura que todas as versões da aplicação estejam configuradas de maneira consistente, com *Dockerfiles* alinhados às melhores práticas de segurança e desempenho, reduzindo o tempo de *deploy* e minimizando o risco de erros manuais no processo. A implementação em *pipelines* de *CI/CD* proporciona um ciclo de desenvolvimento mais ágil e seguro, sendo compatível com ferramentas como *GitHub Actions*, *GitLab CI* e *Jenkins*, o que amplia a adaptabilidade da solução para diversos contextos de desenvolvimento.

Para validar a eficácia da ferramenta, propõe-se sua aplicação em *benchmarks*[5] padrões do *Docker*, como o *CIS Benchmark*, permitindo uma análise detalhada das otimizações geradas. Além disso, uma avaliação em um projeto real, utilizando banco de dados e múltiplas imagens, possibilitaria medir o impacto do *Dockeryzer* em métricas como tempo de *build* e consumo de memória. Esses testes forneceriam dados concretos sobre os benefícios da ferramenta e possíveis ajustes para aprimorar seu desempenho.

[5] Técnica utilizada para avaliar e comparar o desempenho de sistemas, softwares ou processos por meio de testes padronizados, permitindo a identificação de melhorias e otimizações.

Referências

TURNBULL, James. The Docker Book. 2014. Disponível em: https://www.dockerbook.com. Acesso em: 18 de outubro de 2024.

COBRA. Disponível em: https://github.com/spf13/cobra. Acesso em: 18 de outubro de 2024.

GO. Disponível em: https://github.com/golang/go. Acesso em: 18 de outubro de 2024.

NODE.JS. Disponível em: https://nodejs.org/en. Acesso em: 27 de outubro de 2024.

DOCKERYZER. Disponível em: https://github.com/jorgevvs2/dockeryzer. Acesso em: 30 de outubro de 2024.

DOCKER. Disponível em: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/. Acesso em: 27 de outubro de 2024.

DONOVAN, Alan A. The Go Programming Language, 2015.

GIT. Disponível em: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F. Acesso em: 18 de outubro de 2024.

FERNANDO, Jeferson. Descomplicando Docker. Disponível em: https://github.com/badtuxx/descomplicandoDocker. Acesso em: 21 de outubro de 2024.

AREF, Amir. Unpacking Docker: From Curiosity to Containerized Conquests. 2023. Disponível em: https://medium.com/@iamirrf/unpacking-docker-from-curiosity-to-containerized-conquests-58f2e4a1a21e. Acesso em: 2 de março de 2025.

BESSA, André. Node.js: o que é, como funciona esse ambiente de execução JavaScript e um Guia para iniciar. 18 de setembro de 2023. Disponível em: https://www.alura.com.br/artigos/node-js. Acesso em: 27 de outubro de 2024.

PEREIRA, João Vitor; SILVÉRIO, Rafaela Petelin. Guia de JavaScript: o que é e como aprender a linguagem mais popular do mundo? 18 de setembro de 2023. Disponível em: https://www.alura.com.br/artigos/node-js. Acesso em: 27 de outubro de 2024.

SPECFY. Efficient Dockerfile for Node.js in 7 Steps. 3 de maio de 2023. Disponível em: https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps. Acesso em: 27 de outubro de 2024.

RABELO, Eduardo. 10 melhores práticas para aplicações Node.js em containers com Docker. 10 de agosto de 2023. Disponível em: https://oieduardorabelo.medium.com/10-melhores-pr%C3%A1ticas-para-aplica%C3%A7%C3%B5es-node-js-em-containers-com-docker-b52a3e27ba3c. Acesso em: 27 de outubro de 2024.

BROWN, Tom. Language Models are Few-Shot Learners. 2020. Disponível em: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf. Acesso em: 10 de novembro de 2024.

MICROSOFT. Visual Studio Code. Disponível em: https://code.visualstudio.com/. Acesso em: 05 dez. 2024.

NODE_JWT. Disponível em: https://github.com/joaovictornsv/node_jwt. Acesso em: 5 de dezembro de 2024.

API_JWT_NEST. Disponível em: https://github.com/joaovictornsv/api-jwt-nest. Acesso em: 5 de dezembro de 2024.

LOCAL_LIST. Disponível em: https://github.com/joaovictornsv/local-list. Acesso em: 5 de dezembro de 2024.

YOUTUBENEXT. Disponível em: https://github.com/ovinidev/YoutubeNext. Acesso em: 5 de dezembro de 2024.

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
Campus Campina Grande - Código INEP: 25137409
R. Tranqüilino Coelho Lemos, 671, Dinamérica, CEP 58432-300, Campina Grande (PB)
CNPJ: 10.783.898/0003-37 - Telefone: (83) 2102.6200

Documento Digitalizado Restrito

Versão Final do TCC

Assunto:	Versão Final do TCC
Assinado por:	Joao Victor
Tipo do Documento:	Comprovante
Situação:	Finalizado
Nível de Acesso:	Restrito
Hipótese Legal:	Controle Interno (Art. 26, $ 3o, da Lei no 10.180/2001)
Tipo do Conferência:	Cópia Simples

Documento assinado eletronicamente por:

- **João Victor Negreiros da Silva**, **ALUNO (202011250033) DE BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO - CAMPINA GRANDE**, em 11/03/2025 18:49:37.

Este documento foi armazenado no SUAP em 11/03/2025. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifpb.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 1415497
Código de Autenticação: f2a78c8dd1