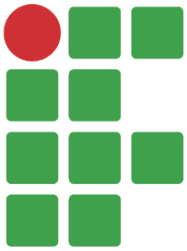


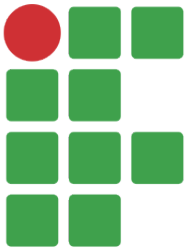
Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Campina Grande
Coordenação do Curso Superior Bacharelado em Engenharia de Computação

Dockeryzer - Desenvolvimento de uma Aplicação Linha de Comando para Geração Inteligente de Dockerfiles utilizando arquitetura multiagentes

João Marcos Amorim de Almeida

Orientador: Prof. Marcelo José Siqueira Coutinho de Almeida, DSc.

Campina Grande, Dezembro de 2025



Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Campina Grande
Coordenação do Curso Superior de Bacharelado em Engenharia de Computação

Dockeryzer - Desenvolvimento de uma Aplicação Linha de Comando para Geração Inteligente de Dockerfiles utilizando arquitetura multiagentes

João Marcos Amorim de Almeida

Trabalho de conclusão de curso apresentado ao Curso de Bacharelado em Engenharia de Computação, do Instituto Federal de Educação, Ciência e Tecnologia da Paraíba - *Campus* Campina Grande, em cumprimento às exigências parciais para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Prof. Marcelo José Siqueira Coutinho de Almeida, DSc.

Campina Grande, Dezembro de 2025

Catalogação na fonte:
Ficha catalográfica

A447d Almeida, João Marcos Amorim de.

Dockeryzer - Desenvolvimento de uma aplicação linha de comando para geração inteligente de dockerfiles utilizando arquitetura multiagentes / João Marcos Amorim de Almeida. – Campina Grande, 2025.
78 f.: il.

Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação) - Instituto Federal da Paraíba, 2026.
Orientador: Prof. Marcelo José Siqueira Coutinho de Almeida.

1. Computação - Inteligência artificial. 2. Docker. 3. Agentes Inteligentes. 4. Automação de Infraestrutura. 5. Engenharia de Software. I. Almeida, Marcelo José Siqueira Coutinho de II. Título

CDU 004.41:004.8

Agradecimentos

Ao concluir este trabalho e encerrar esta etapa de tanto aprendizado, meu primeiro sentimento é de profunda gratidão a Deus. A Ele, que é o princípio de tudo, dedico esta conquista. Foi em Sua força que encontrei a minha, em Sua sabedoria busquei direção e em Sua presença encontrei a coragem necessária para perseverar nos dias difíceis. Agradeço pela saúde, pelo amparo nos momentos de cansaço e pela certeza de que "tudo posso naquele que me fortalece"(Filipenses 4:13), reconhecendo sempre que "nada sou sem Ti"(João 15:5). Que este trabalho reflita o dom que Ele me confiou, pois entendo que "dEle, por Ele e para Ele são todas as coisas"(Romanos 11:36).

Nesta caminhada, minha família foi o alicerce fundamental. Aos meus pais, Alexandro e Monique, deixo um agradecimento que as palavras não conseguem expressar totalmente. Vocês foram meu primeiro exemplo de dedicação e a voz que sempre me incentivou a buscar o melhor. Esta vitória não é apenas minha; é uma celebração do amor e dos valores que vocês me transmitiram.

Pai, a você dedico uma gratidão especial. Como professor, o senhor sempre foi meu maior exemplo de que o estudo e o trabalho são os únicos caminhos para uma vida digna, e essa consciência foi o que me trouxe até aqui. Obrigado por me incentivar a nunca parar de aprender. Mais do que os conselhos, agradeço pelo seu sacrifício diário: por cada manhã em que acordou cedo para me levar ao ponto de ônibus e por todo o apoio que me permitiu focar no meu sonho. Suas mãos e seus esforços foram o suporte invisível que tornou este diploma possível.

Mãe, meu coração transborda gratidão por ser minha conselheira e porto seguro. Obrigada por cada palavra de sabedoria, sempre me guiando a tomar decisões alinhadas com a palavra de Deus e me ensinando o valor da responsabilidade em cada passo dado. Jamais esquecerei as vezes em que você parou tudo para me ouvir explicar a matéria, transformando meus estudos em um momento de partilha. Além disso, seu apoio foi fundamental até no início da minha trajetória profissional, me ajudando a conquistar meu primeiro estágio. Seu cuidado e seus ensinamentos estão presentes em cada detalhe desta vitória.

Às minhas irmãs, Gabriela e Maria Raquel, agradeço por serem minhas companheiras de vida e por tornarem essa jornada mais leve. Obrigado pela torcida constante e por estarem ao meu lado em cada pequena vitória. Ter o apoio de vocês é um presente que levo comigo.

Ao meu tio Helton, sua esposa Amanda e minha prima Lívia, agradeço imensamente por todo o carinho e por estarem sempre presentes em minha vida. É um privilégio contar com o apoio e a

torcida de vocês, que tornam nossos momentos em família tão especiais e acolhedores. Ter vocês por perto foi fundamental para que eu me sentisse encorajado e amado durante toda essa trajetória.

Ao professor Marcelo José Siqueira Coutinho de Almeida, expresso minha profunda gratidão pela orientação, paciência e pelos ensinamentos partilhados. Estendo este agradecimento a todos os professores e colaboradores do IFPB, que contribuíram para a minha formação acadêmica e humana. Fazer parte desta instituição foi um privilégio que transformou minha visão de mundo e me preparou para os desafios profissionais.

Aos meus amigos, que caminharam ao meu lado durante estes anos, meu muito obrigado. A jornada tornou-se muito mais leve graças às conversas nos corredores, aos grupos de estudo, às trocas de experiências e ao apoio mútuo nos momentos de maior pressão. Vocês não foram apenas colegas de classe, mas companheiros que tornaram essa trajetória inesquecível.

Resumo

A crescente utilização de contêineres em ambientes de desenvolvimento e operação de software exige a criação de *Dockerfiles* bem estruturados, seguros e otimizados. No entanto, a elaboração manual desses arquivos continua suscetível a erros, inconsistências e desperdício de recursos. Este trabalho apresenta a evolução do Dockeryzer, uma ferramenta de linha de comando desenvolvida em Go que automatiza a geração e análise de *Dockerfiles* utilizando uma arquitetura multiagente integrada a modelos de linguagem. A solução proposta amplia um trabalho anterior ao incorporar agentes especializados capazes de identificar tecnologias utilizadas no projeto, gerar *Dockerfiles* personalizados com base em boas práticas e comparar diferentes versões para sugerir melhorias. A ferramenta utiliza LLMs por meio da API Gemini e o framework LangChain para coordenar a atuação dos agentes, ampliando a contextualização e a precisão das recomendações. A solução foi validada em múltiplos cenários e linguagens, evidenciando redução de esforço manual, maior padronização e melhorias na qualidade das imagens geradas. Assim, o Dockeryzer representa um avanço na automação inteligente aplicada a DevOps, contribuindo para processos mais eficientes, seguros e reprodutíveis no ciclo de vida de software.

Palavras-chave: Docker, Dockerfile, Automação, LLM, Multiagentes, DevOps.

Abstract

The growing adoption of containerization in modern software development and operations demands well-structured, secure, and optimized *Dockerfiles*. However, manually creating these configuration files remains a task prone to errors, inconsistencies, and inefficient resource usage. This work presents the evolution of Dockeryzer, a command-line tool developed in Go that automates the generation and analysis of *Dockerfiles* using a multi-agent architecture integrated with large language models. The proposed solution expands upon a previous version by introducing specialized agents capable of identifying the technologies used in a project, generating customized *Dockerfiles* following best practices, and comparing different versions to suggest improvements. The tool leverages LLMs through the Gemini API and the LangChain framework to coordinate agent interactions, increasing contextualization and accuracy in the produced recommendations. The system was validated across multiple languages and project types, demonstrating reduced manual effort, improved standardization, and enhanced quality of the generated images. Thus, Dockeryzer represents a significant advancement in intelligent automation for DevOps, contributing to more efficient, secure, and reproducible software delivery workflows.

Keywords: Docker, Dockerfile, Automation, LLM, Multi-agent Systems, DevOps.

Sumário

Lista de Siglas e Abreviaturas

IA	Inteligência Artificial
CLI	*Command-Line Interface* (Interface de Linha de Comando)
VM	*Virtual Machine* (Máquina Virtual)
API	*Application Programming Interface* (Interface de Programação de Aplicação)
LLMs	*Large Language Models* (Grandes Modelos de Linguagem)
PLN	Processamento de Linguagem Natural
SDKs	*Software Development Kit* (Kit de Desenvolvimento de Software)
TI	Tecnologia da Informação
IaC	*Infrastructure as Code* (Infraestrutura como Código)
MAS	*Multi-Agent Systems* (Sistemas Multiagentes)
CI/CD	*Continuous Integration/Continuous Delivery*
RAG	*Retrieval-Augmented Generation*
PaaS	*Platform as a Service* (Plataforma como um Serviço)
AWS	Amazon Web Services

Lista de Figuras

Lista de Tabelas

Listings

Capítulo 1

Introdução

1.1 Contextualização

Com o crescimento acelerado das aplicações em nuvem e da adoção de arquiteturas baseadas em microsserviços, a conteinerização consolidou-se como uma das práticas centrais no desenvolvimento e implantação de aplicações modernas. Ferramentas como o *Docker*[1] permitem encapsular aplicações e suas dependências em ambientes isolados, portáteis e consistentes, promovendo maior agilidade e previsibilidade ao longo do ciclo de vida do software, desde o desenvolvimento até a produção.

Entretanto, embora o uso do Docker seja amplamente difundido, a criação manual de *Dockerfiles* — arquivos responsáveis por definir o ambiente de construção das imagens — ainda representa um desafio significativo. Um *Dockerfile* mal estruturado pode resultar em imagens pesadas, inseguras ou ineficientes, comprometendo o desempenho das aplicações e elevando os custos operacionais. Estudos recentes indicam que erros de configuração e violações de boas práticas são recorrentes em projetos de código aberto, afetando diretamente o tempo de *build* e a manutenibilidade das imagens (ZHOU et al., 2022). Além disso, a refatoração automatizada desses arquivos tem demonstrado reduzir em até 32% o tamanho das imagens e melhorar em mais de 70% a qualidade de manutenção, evidenciando o impacto das más práticas de configuração manual (KSONTINI et al., 2025a).

Outro aspecto relevante diz respeito à segurança: pesquisas mostram que a inclusão de dependências desnecessárias ou pacotes desatualizados é uma das principais causas de vulnerabilidades em ambientes conteinerizados (KAUR et al., 2021a). Essas dificuldades reforçam a necessidade de ferramentas que auxiliem na geração automatizada e inteligente de *Dockerfiles*, promovendo eficiência, segurança e padronização no processo de conteinerização.

Nesse contexto, a evolução dos modelos de linguagem baseados em inteligência artificial (IA), como o *Gemini*[2], viabiliza novas formas de automação inteligente, permitindo que sistemas compreendam o contexto de um projeto e gerem conteúdos personalizados de forma precisa e contextu-

[1] <https://www.docker.com>
[2] <https://gemini.google.com/app>

alizada. Ferramentas como o *Workik*[3] já demonstram a viabilidade da aplicação de IA na geração de *Dockerfiles* otimizados, capazes de se adaptar dinamicamente às necessidades específicas de diferentes aplicações (Workik, 2025).

Estudos recentes reforçam esse potencial. ROSA et al. (2023a) investigaram o uso do modelo *T5*, baseado em aprendizado profundo, para gerar *Dockerfiles* a partir de descrições em linguagem natural. Os autores destacaram os desafios e as promessas dessa abordagem, utilizando um conjunto de dados com mais de 670 mil instâncias para treinar e testar o modelo. De forma semelhante, HU et al. (2025a) propuseram o *Repo2Run*, um agente baseado em Modelos de Linguagem de Grande Escala (LLMs), capaz de automatizar completamente a configuração de ambientes e gerar *Dockerfiles* executáveis para projetos Python, atingindo uma taxa de sucesso de 86,0%, superando a linha de base em 63,9%.

Complementarmente, KSONTINI et al. (2025a) demonstraram que a refatoração automatizada de *Dockerfiles*, por meio de aprendizado em contexto, pode melhorar significativamente a manutenibilidade e reduzir tanto o tempo de *build* quanto o tamanho das imagens. Já GOTO; MATSUMOTO; KUSUMOTO (2025) propuseram uma metodologia para geração automática de testes de *Dockerfiles*, com base na análise das camadas das imagens geradas, promovendo maior confiabilidade e detecção de falhas.

Paralelamente, observa-se o avanço dos sistemas multiagentes e dos agentes autônomos baseados em LLMs, como o *Gemini* e o *GPT-4*. Diferentemente de abordagens estáticas, esses agentes são capazes de executar tarefas complexas, tomar decisões e interagir com *APIs* externas, adaptando-se conforme o contexto. Essa nova geração de agentes expande o papel da IA, transformando-a em um componente ativo do processo de desenvolvimento de software (HU et al., 2025a).

Dessa forma, surge uma nova oportunidade: utilizar agentes de IA para automatizar a geração inteligente de *Dockerfiles*, considerando boas práticas, otimizações e padrões específicos de cada linguagem ou *stack* tecnológica. Em vez de depender exclusivamente do conhecimento manual do desenvolvedor, esses agentes podem analisar o projeto, identificar suas necessidades e construir automaticamente arquivos de configuração otimizados, com eficiência e segurança.

Neste contexto de crescente integração entre desenvolvimento e inteligência artificial, este trabalho dá continuidade à pesquisa iniciada por SILVA; CASTRO (2024), que propôs uma ferramenta inicial de geração automática de *Dockerfiles* a partir de modelos de linguagem, em projetos *NodeJS*. Enquanto o estudo anterior se limitava à integração direta com uma *API (Application Programming Interface)* de IA para sugerir configurações básicas, o presente trabalho amplia e aprofunda a proposta original, incorporando uma arquitetura multiagente capaz de realizar análises, comparações e gerações automatizadas de forma autônoma e contextualizada. Essa evolução estrutural visa tornar o processo mais escalável, modular e aderente às boas práticas de DevOps e engenharia de software moderna.

Assim, propõe-se o desenvolvimento de uma ferramenta de linha de comando (CLI) capaz

[3] <https://workik.com/docker-code-generator>

de gerar *Dockerfiles* personalizados e eficientes de forma automatizada, utilizando modelos de linguagem e uma arquitetura multiagente para interpretar as características do projeto e sugerir configurações adequadas. A proposta visa reduzir o esforço manual na criação de ambientes e disseminar boas práticas, alinhando-se à tendência crescente de integração entre ferramentas de desenvolvimento e inteligência artificial, que vem redefinindo a forma como sistemas são projetados, configurados e mantidos.

1.2 Problema de Pesquisa

Apesar da crescente adoção do Docker como solução de conteinerização, a construção manual de *Dockerfiles* permanece uma atividade suscetível a falhas, demandando conhecimento técnico avançado e tempo de configuração. Esse processo torna-se ainda mais complexo em ambientes que utilizam diferentes linguagens e *frameworks*, exigindo ajustes específicos para cada cenário. Além disso, a ausência de padronização entre projetos pode gerar inconsistências, desperdício de recursos computacionais e aumento de custos operacionais.

O trabalho anterior ao presente projeto buscou mitigar parcialmente esse problema, implementando uma ferramenta capaz de gerar *Dockerfiles* automaticamente por meio de uma integração direta com um modelo de linguagem. Embora essa abordagem tenha demonstrado viabilidade, ela apresentava limitações quanto à contextualização das respostas da IA, à ausência de análise estruturada do projeto e à falta de modularidade na geração das configurações.

Dessa forma, o problema de pesquisa que este trabalho propõe investigar é: **como aprimorar o processo de geração automatizada de *Dockerfiles*, por meio de uma arquitetura multiagente integrada LLMs, de modo a torná-lo mais contextualizado, flexível e eficiente?**

1.3 Justificativa

A justificativa para este trabalho baseia-se em três pilares principais:

- **Relevância acadêmica:** contribui para o estudo da integração entre Inteligência Artificial, Engenharia de Software e *DevOps*, explorando aplicações práticas de sistemas multiagentes aliados a LLMs.
- **Relevância tecnológica:** desenvolvedores e equipes de tecnologia da informação (TI) frequentemente enfrentam dificuldades na padronização de *Dockerfiles*. A automação deste processo pode melhorar a produtividade, reduzir falhas e promover boas práticas de conteinerização.
- **Relevância social e profissional:** ao simplificar o uso de contêineres, a ferramenta possibilita maior acessibilidade para desenvolvedores iniciantes e otimiza fluxos de trabalho em equipes de software, o que pode impactar diretamente na qualidade e na agilidade da entrega de soluções digitais.

Além disso, a escolha do projeto Dockeryzer foi motivada pela observação prática de dificuldades recorrentes enfrentadas por desenvolvedores na criação e manutenção de *Dockerfiles*. Embora o Docker seja amplamente utilizado na indústria, a escrita manual desses arquivos ainda depende fortemente de conhecimento especializado, o que frequentemente resulta em configurações ineficientes, inseguras ou inconsistentes entre projetos A inexistência de ferramentas amplamente difundidas que realizem a geração inteligente e contextualizada de *Dockerfiles*, considerando automaticamente a linguagem, dependências e boas práticas, evidenciou uma lacuna tecnológica relevante. Nesse contexto, a adoção de modelos de linguagem e sistemas multiagentes apresenta-se como uma alternativa promissora para reduzir o esforço manual e aumentar a confiabilidade do processo de conteinerização Assim, o projeto foi escolhido por aliar relevância prática, aplicabilidade industrial e potencial acadêmico, permitindo explorar a integração entre DevOps, Inteligência Artificial e Engenharia de Software em um problema real e atual

1.4 Objetivos

1.4.1 Objetivo geral

Projetar e implementar uma arquitetura baseada em sistemas multiagentes para a ferramenta *Dockeryzer*, com o objetivo de gerar automaticamente *Dockerfiles* otimizados, seguros e aderentes às boas práticas de conteinerização, oferecendo suporte a múltiplas linguagens e frameworks por meio do uso de modelos de linguagem de grande escala.

1.4.2 Objetivos específicos

1. Definir e formalizar um conjunto de boas práticas e estratégias de otimização para a construção de *Dockerfiles* eficientes, seguros e aderentes aos padrões amplamente adotados pela comunidade;

2. Projetar e implementar uma arquitetura baseada em sistemas multiagentes, composta por agentes especializados na análise de projetos, geração e comparação de *Dockerfiles*;

3. Integrar modelos de linguagem de grande escala (LLMs), com ênfase na API Gemini, como mecanismo de apoio à geração automatizada e contextualizada de *Dockerfiles*;

4. Implementar suporte à geração de *Dockerfiles* para múltiplas linguagens e frameworks, incluindo Node.js, Python, Go, Java e PHP;

5. Avaliar a capacidade da ferramenta em gerar *Dockerfiles* consistentes e alinhados às boas práticas, considerando diferentes cenários e tipos de projetos.

1.5 Organização do Trabalho

Este trabalho está organizado em seis capítulos, além da Introdução, estruturados de forma a apresentar de maneira progressiva os conceitos teóricos, a metodologia adotada, o desenvolvimento da solução proposta, bem como os resultados obtidos e as considerações finais.

1.5.1 Fundamentação Teórica

O capítulo de Fundamentação Teórica apresenta os principais conceitos, tecnologias e fundamentos que embasam o desenvolvimento deste trabalho. São discutidos temas como conteinerização com Docker, boas práticas na escrita de *Dockerfiles*, arquiteturas multiagentes, modelos de linguagem de grande escala (LLMs) e ferramentas associadas, como LangChain. Esse capítulo fornece o suporte conceitual necessário para a compreensão das decisões técnicas adotadas ao longo do projeto.

1.5.2 Metodologia

O capítulo de Metodologia descreve a abordagem científica e técnica utilizada para a condução do trabalho. São apresentadas as etapas de planejamento, a estratégia de pesquisa adotada, os critérios para definição do escopo, a análise de requisitos funcionais e não funcionais, bem como os procedimentos metodológicos empregados para garantir a validade e a confiabilidade da solução proposta.

1.5.3 Desenvolvimento da Solução

Neste capítulo é detalhado o processo de implementação da ferramenta *Dockeryzer*. São apresentadas as decisões de arquitetura, a modelagem do sistema multiagente, a organização do código-fonte, a integração com modelos de linguagem por meio de *APIs*, como a Gemini, e o suporte a múltiplas linguagens e frameworks. O capítulo também descreve os principais componentes da CLI e o funcionamento interno da solução.

1.5.4 Testes e Validação

O capítulo de Testes e Validação aborda as estratégias adotadas para verificar o correto funcionamento da ferramenta desenvolvida. São descritos os tipos de testes realizados, os cenários avaliados, as métricas utilizadas e os critérios de aceitação definidos, com o objetivo de assegurar a confiabilidade, a robustez e a aderência da solução aos requisitos estabelecidos.

1.5.5 Resultados e Avaliação

Neste capítulo são apresentados e analisados os resultados obtidos a partir da aplicação da ferramenta em diferentes cenários. São realizadas comparações entre os *Dockerfiles* gerados, ava-

liando aspectos como otimização, tamanho das imagens, conformidade com boas práticas e desempenho, discutindo-se criticamente a eficácia da abordagem proposta.

1.5.6 Considerações Finais e Trabalhos Futuros

O capítulo de Considerações Finais e Trabalhos Futuros apresenta uma síntese dos principais resultados alcançados, destacando as contribuições do trabalho para a automação da criação de *Dockerfiles*. Também são discutidas as limitações identificadas e apresentadas sugestões de aprimoramento e possibilidades de expansão da ferramenta em trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo aborda os conceitos e tecnologias que sustentam o desenvolvimento da ferramenta proposta, com ênfase no uso de contêineres Docker e na automação de ambientes em contextos DevOps. São exploradas as boas práticas de construção e otimização de *Dockerfiles*, a eficiência da linguagem Go para o desenvolvimento de aplicações CLI e o papel da inteligência artificial — em especial, dos LLMs e sistemas multiagentes — na geração automatizada de configurações. Essa base teórica fornece o suporte necessário para compreender as decisões técnicas e metodológicas que orientam a criação do Dockeryzer, integrando desempenho, automação e inteligência artificial em um mesmo ecossistema de desenvolvimento.

2.1 Computação em Nuvem

A evolução da computação em nuvem modificou profundamente a forma de desenvolver, hospedar e escalar sistemas de software. ARMBRUST et al. (2010) definem a nuvem como um paradigma que fornece acesso sob demanda a recursos computacionais, promovendo elasticidade, disponibilidade e redução de custos operacionais.

Nesse contexto, os contêineres surgem como uma solução mais leve e eficiente em comparação com as máquinas virtuais tradicionais. Ao invés de virtualizar todo o hardware, os contêineres compartilham o mesmo kernel do sistema operacional, isolando processos de maneira mais eficiente (MERKEL, 2014).

O Docker consolidou-se como a principal tecnologia de conteinerização. Ele introduziu um fluxo de desenvolvimento simplificado, no qual o *Dockerfile* representa o ponto central: um *script* declarativo que especifica a construção de imagens. Essa abordagem garante portabilidade, reprodutibilidade e padronização de ambientes, características críticas em projetos modernos. De acordo com MOUAT (2015), a capacidade de definir ambientes em código reduziu falhas de configuração e acelerou a adoção de pipelines automatizados.

KALSKE; KARHUNEN; TAIVALSAARI (2017) demonstram em seus estudos que o uso sistemático de contêineres e *Dockerfiles* é essencial para arquiteturas baseadas em microserviços, pois permite a gestão independente de múltiplos módulos de software.

2.2 Automação e DevOps

Com a crescente complexidade dos sistemas distribuídos, emergiu o movimento DevOps, cujo objetivo é integrar desenvolvimento (Dev) e operações (Ops), promovendo a colaboração, automação e entrega contínua (KIM et al., 2016).

Um dos pilares dessa filosofia é a Infraestrutura como Código (IaC), que se refere à prática de definir e gerenciar recursos computacionais por meio de arquivos de configuração versionados, favorecendo a automação e a reprodutibilidade (HUMBLE; FARLEY, 2010).

A integração de *Dockerfiles* em pipelines de CI/CD (*Continuous Integration/Continuous Delivery*) possibilita que cada atualização de código seja automaticamente validada, testada e empacotada em imagens prontas para *deployment*. Pesquisas de referência na área demonstram que essa prática reduz o tempo de entrega de novas versões e aumenta a confiabilidade em ambientes críticos (FORSGREN; HUMBLE; KIM, 2018).

Assim, a automação em DevOps é um terreno fértil para ferramentas como o Dockeryzer, que propõe automatizar não apenas a execução de pipelines, mas também a própria geração do *Dockerfile*, reduzindo o esforço humano e as falhas de configuração.

2.3 Docker

O Docker é uma plataforma de código aberto desenvolvida para facilitar a criação, o teste e o *deployment* de aplicações em ambientes isolados, chamados de contêineres. Ele surgiu como uma alternativa moderna à virtualização tradicional, sendo mais leve, portátil e eficiente. O Docker utiliza o kernel do sistema operacional para criar ambientes isolados, permitindo que desenvolvedores e equipes de operações tenham ambientes padronizados, desde o desenvolvimento até a produção.

Segundo MERKEL (2014), o Docker aproveita os benefícios da conteinerização para empacotar e implantar aplicações de forma leve e consistente. Ele utiliza o kernel do sistema operacional para criar ambientes isolados, permitindo que desenvolvedores e equipes de operações tenham ambientes padronizados do desenvolvimento até a produção, eliminando problemas relacionados à inconsistência de ambientes e aumentando a reprodutibilidade dos sistemas.

A Figura 2.1 apresenta o logotipo oficial do Docker.

Figura 2.1: *Logo do Docker*

Fonte: Docker, Inc. (2025)

2.3.1 Contêiner

Um contêiner é uma instância de uma imagem Docker em execução. Ele fornece um ambiente isolado que compartilha o kernel do sistema operacional com outros contêineres, mas possui seu próprio sistema de arquivos, bibliotecas e dependências. Essa abordagem garante que a aplicação será executada da mesma forma, independentemente do ambiente em que está hospedada.

Contêineres são mais leves e iniciam mais rapidamente do que máquinas virtuais tradicionais, sendo ideais para arquiteturas modernas baseadas em microserviços.

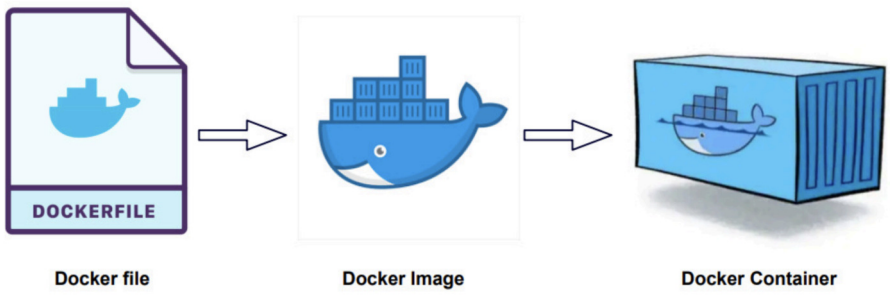
Segundo o autor TURNBULL (2014), o uso do termo se inspira nos contêineres físicos usados para transporte de mercadorias, mas, no caso dos contêineres Docker, o que é "transportado"são pacotes de software.

2.3.2 Imagem

Segundo a própria documentação do Docker, uma imagem é um pacote padronizado que inclui todos os arquivos, binários, bibliotecas e configurações para executar um contêiner (Docker, 2024). As imagens funcionam como moldes para os contêineres, ou seja, um contêiner é uma instância de uma imagem. Imagens podem ser armazenadas localmente ou em repositórios como o Docker Hub[1], facilitando a distribuição e o versionamento de aplicações.

A Figura 2.2 apresenta o fluxo de funcionamento do Docker, no qual um *Dockerfile* é utilizado para a construção de uma imagem, que posteriormente é instanciada como um contêiner em execução.

Figura 2.2: *Processo de construção e execução no Docker*



Fonte: AREF (2023)

2.3.3 Dockerfile

Um *Dockerfile* é um arquivo de texto que contém um conjunto de instruções utilizadas pelo Docker para automatizar a criação de imagens personalizadas (Docker Docs, 2025). Cada instrução define uma etapa no processo de construção da imagem, e cada etapa adiciona uma nova camada

[1] <https://hub.docker.com/>

à imagem final. Isso permite a criação de ambientes de execução padronizados, versionáveis e reproduzíveis, facilitando o empacotamento e a distribuição de aplicações em diferentes sistemas.

Esse arquivo deve começar com o comando `FROM`, que especifica a imagem base, e incluir instruções como `RUN`, `COPY`, `CMD`, `ENTRYPOINT`, `ENV`, `EXPOSE`, entre outras — todas voltadas para configurar corretamente o ambiente de execução (Docker Docs, 2016).

A Figura 2.3 apresenta um exemplo de *Dockerfile* gerado para uma aplicação feita em Golang.

Figura 2.3: *Exemplo de Dockerfile para aplicação Go*

Dockerfile

```
1   FROM golang:1.22-alpine AS builder
2
3   WORKDIR /app
4
5   COPY go.mod ./
6   COPY go.sum ./
7   RUN go mod download
8
9   COPY . .
10
11  RUN CGO_ENABLED=0 GOOS=linux go build -o main .
12
13  FROM alpine:latest
14
15  WORKDIR /root/
16
17  COPY --from=builder /app/main .
18
19  EXPOSE 8080
20
21  CMD ["./main"]
```

Fonte: Elaborado pelo autor

Embora o *Dockerfile* seja amplamente documentado e utilizado, sua correta elaboração ainda representa um desafio, especialmente para desenvolvedores iniciantes ou equipes que trabalham com múltiplas linguagens e frameworks. A definição inadequada de instruções pode resultar em imagens excessivamente grandes, vulneráveis ou difíceis de manter.

Neste trabalho, o *Dockerfile* deixa de ser apenas um artefato de configuração manual e passa a ser tratado como um objeto passível de geração, análise e otimização automatizadas. A principal contribuição do Dockeryzer está justamente na capacidade de gerar *Dockerfiles* de forma inteligente e contextualizada, utilizando modelos de linguagem e agentes especializados para adaptar a configuração às características específicas de cada projeto, promovendo maior padronização, segurança e eficiência no processo de conteinerização.

2.4 Golang: A Linguagem de Programação do Google

A linguagem Go, também conhecida como Golang, foi desenvolvida pelo Google em 2007 e lançada como projeto de código aberto em 2009. Criada por Robert Griesemer, Rob Pike e Ken Thompson, surgiu da necessidade de uma linguagem moderna, eficiente e com bom suporte à concorrência para atender a sistemas distribuídos em larga escala. Segundo GOLDMAN (2015), a proposta central da linguagem é simplificar o desenvolvimento de software sem abrir mão de desempenho.

Golang é uma linguagem compilada, estaticamente tipada, com coleta automática de lixo *(garbage collection)*, e apresenta uma sintaxe minimalista e direta, influenciada por C e Python (The Go Authors, 2024). Um dos seus maiores diferenciais é o suporte nativo à concorrência, feito por meio das `goroutines` e canais, que permitem o desenvolvimento de aplicações capazes de utilizar múltiplos núcleos de processadores de forma eficiente (GOLDMAN, 2015).

A Figura 2.4 exemplifica a estrutura básica de um programa em Go, destacando a declaração de pacotes, importação de bibliotecas e definição de funções, aspectos que contribuem para a clareza e legibilidade do código.

Figura 2.4: *Exemplo de código Go*

main.go

```go
package main

import "fmt"

func soma(a int, b int) int {
    return a + b
}

func main() {
    fmt.Println("Hello, World!")

    resultado := soma(5, 5)
    fmt.Println("A soma é:", resultado)
}
```

Fonte: Elaborado pelo autor

A linguagem também oferece um ecossistema de desenvolvimento bem integrado. Como destaca FILIPINI (2015), o Go possui utilitários como o `go fmt` para padronização de código, `go build` para compilação e `go test` para testes automatizados, além de permitir a criação de binários estáticos, ideais para a distribuição de aplicações CLI.

No contexto deste trabalho, a escolha do Go como linguagem base para o desenvolvimento da ferramenta *Dockeryzer* é motivada por esses fatores. De acordo com DIGITAL OCEAN (2020),

Go é especialmente recomendado para aplicações de CLI, sistemas de rede e automação de infraestrutura — características essenciais para a proposta do *Dockeryzer*, que visa automatizar a criação de imagens Docker otimizadas.

2.5 Aplicações CLI em Golang

O ecossistema de desenvolvimento em Go oferece suporte sólido para a criação de CLI, sendo amplamente utilizado em soluções voltadas para infraestrutura e automação de sistemas. Esse uso consolidou-se principalmente devido à combinação de desempenho, portabilidade e simplicidade da linguagem, que permite a criação de executáveis leves e multiplataforma, sem dependências externas.

No contexto de desenvolvimento de ferramentas CLI, o Go se destaca por sua eficiência e pelo conjunto de bibliotecas maduras voltadas a esse propósito. Entre elas, a Cobra é uma das mais populares, pois provê uma estrutura padronizada e modular para o gerenciamento de comandos, subcomandos e *flags*, facilitando a manutenção e a escalabilidade das aplicações (GERARDI, 2022). Além disso, frameworks complementares como o Testify auxiliam na construção de testes automatizados, contribuindo para a confiabilidade e qualidade do software.

A adoção dessa abordagem é especialmente vantajosa para o Dockeryzer, uma vez que o projeto exige uma interface de linha de comando de fácil uso, modular e extensível. O uso de Go, aliado à biblioteca Cobra, permite implementar comandos de forma estruturada e eficiente, tornando a interação com a ferramenta mais intuitiva e garantindo melhor experiência ao desenvolvedor.

2.5.1 Cobra

A biblioteca Cobra[2] é amplamente reconhecida como o padrão de fato para o desenvolvimento de interfaces de linha de comando em Go. Criada originalmente para o framework sHugo, ela se popularizou por oferecer uma arquitetura simples, porém robusta, que facilita a criação e o gerenciamento de comandos em aplicações CLI.

O Cobra possibilita o desenvolvimento de ferramentas com múltiplos níveis de comandos, suporte a *flags* compatíveis com o padrão POSIX e geração automática de mensagens de ajuda e documentação. Entre seus principais recursos, destacam-se:

- **Organização hierárquica de comandos:** estrutura modular que facilita a expansão da aplicação;
- **Compatibilidade com *flags* padrão POSIX:** suporte a opções curtas e longas, garantindo consistência com ferramentas Unix;
- **Geração automática de ajuda e documentação:** cria descrições e exemplos de uso com base na estrutura interna dos comandos;

[2] <https://cobra.dev/>

- **Autocompletar para *shells*:** suporte à criação de scripts de autocompletar para Bash, Zsh e Fish;
- **Sugestões inteligentes:** oferece correções automáticas para comandos digitados incorretamente.

Além de sua versatilidade, o Cobra é amplamente utilizado em projetos de referência como o Kubernetes, o Cosmos SDK e a GitHub CLI, consolidando sua posição como uma biblioteca estável e confiável para a construção de CLIs modernas (ULELU, 2024).

No caso do Dockeryzer, o uso do Cobra permite estruturar comandos para diferentes agentes do sistema — como o gerador, o analisador e o comparador de *Dockerfiles* — de forma organizada e extensível. Isso garante uma interação mais fluida com o usuário, mantendo a coerência com o princípio de simplicidade e eficiência que norteia o projeto.

2.6 Modelos de LLM

Os *Large Language Models*, ou Grandes Modelos de Linguagem, representam um avanço significativo no campo da IA, especialmente na área de Processamento de Linguagem Natural (PLN). Tais modelos são sistemas de aprendizado de máquina treinados em vastos volumes de dados textuais, o que lhes permite compreender, gerar e interagir com a linguagem humana de maneira complexa e contextualizada (BROWN et al., 2020).

A evolução dos LLMs remonta a modelos estatísticos de linguagem e redes neurais recorrentes, mas o marco principal que impulsionou a capacidade e o desempenho desses sistemas foi a introdução da arquitetura *Transformer* pelo autor VASWANI et al. (2017) em seu artigo *Attention Is All You Need*. Essa arquitetura, que se baseia em mecanismos de autoatenção, superou as limitações dos modelos anteriores ao permitir o processamento paralelo de sequências de dados e a captura de dependências de longo alcance de forma mais eficiente.

O funcionamento dos LLMs geralmente envolve duas fases principais: o pré-treinamento e o *fine-tuning*. Durante o pré-treinamento, o modelo é exposto a quantidades massivas de texto (como livros, artigos, páginas da web), aprendendo padrões linguísticos, gramática, semântica e até mesmo conhecimento factual. Após essa fase, o modelo pode ser adaptado (*fine-tuning*) para tarefas específicas, como sumarização de texto, tradução, resposta a perguntas ou geração de código, o que melhora sua performance em domínios particulares (SU et al., 2022).

Desde o GPT-2 (RADFORD et al., 2019) até modelos como GPT-4, Gemini e Claude, os LLMs têm demonstrado capacidades emergentes em raciocínio, síntese e análise de código (ZHAO et al., 2023). Aplicações práticas incluem: geração de código, documentação automatizada, análise de vulnerabilidades, suporte em CI/CD com recomendações inteligentes, entre muitas outras.

Além disso, a estratégia de RAG (*Retrieval-Augmented Generation*) (LEWIS et al., 2020) possibilita que LLMs sejam enriquecidos com informações externas, tornando-os menos propensos a alucinações e mais úteis em aplicações de engenharia.

Neste trabalho, o LLM atua como núcleo inteligente do Dockeryzer, apoiando agentes em tarefas de geração e análise de *Dockerfiles*.

2.7 Sistemas Multiagentes

Os sistemas multiagentes (MAS) representam uma subárea da Inteligência Artificial distribuída, em que múltiplos agentes autônomos interagem, colaboram ou competem para atingir objetivos. (JENNINGS; SYCARA; WOOLDRIDGE, 1998) definem agentes como entidades capazes de perceber seu ambiente, raciocinar sobre ele e agir para alcançar metas específicas.

Na literatura, os MAS têm sido aplicados em diversos contextos: desde simulações sociais, otimização em sistemas de transporte, até gerenciamento de recursos computacionais em nuvem (RUSSELL; NORVIG, 2020).

A nova onda de IA baseada em agentes combina agentes especializados com LLMs, atribuindo papéis distintos e coordenados. (HE; TREUDE; LO, 2024) exploram agentes que auxiliam no desenvolvimento de software, atuando em tarefas de análise, geração e revisão de código. (QU et al., 2024) apresentam uma revisão sistemática, destacando que agentes baseados em LLMs têm potencial de transformar processos de engenharia de software e DevOps, automatizando tarefas repetitivas e reduzindo custos.

No contexto deste trabalho, o Dockeryzer adota uma arquitetura multiagente com papéis definidos:

- **Agente analisador**, responsável por identificar a linguagem e dependências do projeto;
- **Agente gerador**, que cria o *Dockerfile* inicial com base nas melhores práticas e nos dados coletados;
- **Agente comparador**, que avalia versões alternativas de *Dockerfiles* e sugere melhorias.

Essa abordagem modular aumenta a flexibilidade, escalabilidade e facilita a incorporação de novos agentes especializados em futuras versões.

2.8 LangChain: API para Sistemas Multiagentes

O LangChain é uma estrutura (*framework*) de código aberto voltada à construção de aplicações baseadas em LLMs, com foco em modularidade, integração e uso de agentes inteligentes (LANGCHAIN, 2025). Criado inicialmente em Python e posteriormente portado para JavaScript e TypeScript, o LangChain fornece uma arquitetura de alto nível que permite conectar modelos de linguagem a fontes externas de dados, ferramentas e APIs, possibilitando a execução de tarefas complexas de forma autônoma.

A Figura 2.5 ilustra o logotipo do LangChain, framework de código aberto voltado ao desenvolvimento de aplicações baseadas em modelos de linguagem e sistemas multiagentes.

Figura 2.5: *Logo da LangChain*

Fonte: LANGCHAIN (2025)

Sua principal contribuição está na abstração de *chains* (cadeias de processamento) e *agents* (agentes), que cooperam para executar tarefas de raciocínio, planejamento e decisão com base em contexto. Um agente no LangChain é capaz de decidir dinamicamente quais ações tomar — como consultar uma base de dados, chamar uma API ou gerar código — utilizando o modelo de linguagem como mecanismo central de decisão. Essa abordagem amplia o escopo de uso das LLMs, permitindo que atuem como componentes ativos e interativos dentro de sistemas de software.

Além disso, o LangChain é amplamente utilizado em conjunto com estratégias de RAG, que combinam informações obtidas de bases de conhecimento externas com o raciocínio dos modelos de linguagem. Essa integração aprimora a precisão das respostas e reduz alucinações, tornando o framework uma solução robusta para aplicações que exigem contexto e confiabilidade na tomada de decisão (MIALON et al., 2023).

No contexto deste trabalho, o LangChain será empregado como base para a implementação do sistema multiagente do Dockeryzer. Cada agente desempenhará uma função específica, como análise, geração e comparação de *Dockerfiles*, interagindo entre si e com a API Gemini para realizar tarefas de forma coordenada e autônoma. Essa integração permitirá uma arquitetura escalável, onde novos agentes poderão ser adicionados para tratar linguagens ou frameworks adicionais sem comprometer o funcionamento global da aplicação.

Assim, o uso do LangChain fornece ao projeto uma camada de inteligência distribuída, tornando o processo de automação mais flexível, adaptável e alinhado às tendências contemporâneas de engenharia de software orientada por IA.

2.9 Gemini

O Gemini é uma família de modelos de inteligência artificial de última geração desenvolvida pelo Google, representando um avanço significativo na arquitetura de LLMs por sua natureza inerentemente multimodal. Diferente de modelos que foram adaptados para multimodalidade após o treinamento inicial com texto, o Gemini foi concebido desde o início para processar e integrar diferentes tipos de dados, incluindo texto, imagens, áudio e vídeo (PICHAI; HASSABIS, 2023). Essa capacidade multimodal permite ao Gemini não apenas compreender e gerar linguagem textual, mas também analisar e interagir com informações visuais e sonoras, abrindo um leque de possibilidades para aplicações mais ricas e complexas.

A família Gemini é composta por diversas versões, cada uma otimizada para diferentes casos de uso e requisitos de desempenho. Entre as principais versões, destacam-se o Gemini Ultra, pro-

jetado para tarefas altamente complexas e que exigem o máximo de capacidade; o Gemini Pro, ideal para uma ampla gama de aplicações empresariais e de desenvolvimento; e o Gemini Nano e Gemini Flash, otimizados para eficiência e execução em dispositivos edge (Google, 2025). Essa variedade permite que desenvolvedores e empresas selecionem o modelo mais adequado às suas necessidades específicas, equilibrando poder computacional com eficiência e latência.

2.9.1 API Gemini

Para integrar as capacidades do Gemini em aplicações e serviços externos, o Google disponibiliza a API do Gemini. Essa interface de programação de aplicações permite que desenvolvedores acessem e utilizem as funcionalidades do modelo de forma programática, sem a necessidade de hospedar ou gerenciar a infraestrutura de IA subjacente (Google AI for Developers, 2025). A API do Gemini oferece funcionalidades como a geração de conteúdo textual e de código, interações conversacionais, sumarização e processamento avançado de linguagem natural; no projeto, o foco será na geração de código. A API inclui ferramentas e parâmetros que permitem aos desenvolvedores controlar a saída do modelo, garantindo que as aplicações desenvolvidas estejam alinhadas com o propósito da mesma. A disponibilidade de *SDKs* (*Software Development Kit*) em linguagens de programação populares, como Golang e JavaScript, simplifica ainda mais a integração e o desenvolvimento de soluções inovadoras baseadas no Gemini.

2.10 Trabalhos Relacionados

A automação da geração e manutenção de *Dockerfiles* tem atraído o interesse de diversos pesquisadores e desenvolvedores, resultando em ferramentas e abordagens voltadas à otimização, refatoração e análise de ambientes conteinerizados. Esta seção apresenta três trabalhos que se aproximam da proposta deste estudo e destaca as principais diferenças em relação ao **Dockeryzer**.

2.10.1 Repo2Run: Agente LLM para Configuração de Ambientes Docker

O *Repo2Run*, proposto por Hu et al. (2025b), é um agente baseado em LLMs desenvolvido para automatizar a configuração de ambientes em projetos Python. A ferramenta é capaz de interpretar repositórios, resolver dependências e gerar automaticamente arquivos *Dockerfile* executáveis. O sistema também implementa mecanismos de *rollback* e validação de execução para garantir que os ambientes gerados sejam funcionais.

Apesar de sua eficácia, o *Repo2Run* é limitado a projetos escritos em Python e não adota uma arquitetura multiagente. O Dockeryzer, por sua vez, propõe uma abordagem mais ampla, ao suportar múltiplas linguagens e frameworks, utilizando uma estrutura modular de agentes, além de oferecer uma CLI desenvolvida em Go, permitindo maior acessibilidade e integração direta com *pipelines* DevOps.

2.10.2 FlexStack: Plataforma de Deploy e Gerenciamento de Aplicações na Nuvem

O FlexStack[3] é uma plataforma voltada ao provisionamento e gerenciamento automatizado de aplicações na nuvem, operando diretamente sobre a infraestrutura da *Amazon Web Services* (AWS). Seu objetivo principal é oferecer uma experiência simplificada de *Platform as a Service* (PaaS), permitindo que desenvolvedores realizem o deploy de aplicações de forma rápida e padronizada, sem a necessidade de configuração manual complexa de recursos como redes, balanceadores de carga, escalonamento automático e containers.

A plataforma abstrai grande parte das responsabilidades relacionadas à infraestrutura, ao mesmo tempo em que mantém o controle dos recursos na conta AWS do próprio usuário, reduzindo riscos de dependência excessiva de fornecedor (*vendor lock-in*). O FlexStack integra-se a repositórios de código e pipelines de CI/CD, possibilitando implantações contínuas automatizadas a partir de alterações no código-fonte, além de oferecer mecanismos de escalabilidade e otimização baseados em demandas de uso.

Embora o FlexStack automatize significativamente o processo de deploy, sua abordagem é centrada na padronização e orquestração de infraestrutura, não focando na análise inteligente e personalizada de artefatos de conteinerização, como arquivos *Dockerfile*. A plataforma não realiza avaliações comparativas nem propõe recomendações detalhadas baseadas em contexto específico do código-fonte ou da arquitetura da aplicação.

Em contraste, o Dockeryzer propõe uma solução direcionada especificamente à análise e geração inteligente de arquivos *Dockerfile*, empregando modelos de linguagem de grande escala por meio da API Gemini e uma arquitetura multiagente. Essa abordagem permite não apenas a criação automatizada desses arquivos, mas também sua análise crítica, comparação entre diferentes estratégias e sugestões de melhorias fundamentadas em boas práticas de DevOps, segurança e otimização de desempenho, adaptando-se de forma contextual às particularidades de cada projeto.

2.10.3 DRMiner: Análise e Refatoração de Dockerfiles

O DRMiner, desenvolvido por Ksontini et al. (2024), é uma ferramenta projetada para identificar e analisar refatorações em arquivos *Dockerfile*. Utilizando uma abordagem baseada em árvores de sintaxe abstrata (E-AST), o DRMiner é capaz de detectar padrões de refatoração e avaliar a evolução da qualidade dos *Dockerfiles* ao longo do tempo. Essa ferramenta contribui para a compreensão das práticas de manutenção e para a identificação de *smells* em projetos reais.

Apesar de seu valor analítico, o DRMiner não realiza a geração ou otimização automática de *Dockerfiles*, atuando apenas na análise pós-implementação. O Dockeryzer, diferentemente, adota uma postura proativa ao gerar arquivos otimizados desde o início, com o auxílio de agentes e modelos de linguagem capazes de sugerir melhorias automáticas, reduzindo a necessidade de refatorações posteriores.

[3] <https://flexstack.com>

2.10.4 Síntese Comparativa

De modo geral, os trabalhos analisados compartilham o objetivo de aprimorar o processo de criação e manutenção de *Dockerfiles*. Entretanto, diferenciam-se quanto ao escopo e ao grau de automação. O Dockeryzer se destaca por integrar múltiplos paradigmas — automação por linha de comando, inteligência artificial e sistemas multiagentes —, oferecendo uma solução completa que une geração, análise e otimização em uma única ferramenta. Além disso, sua implementação em Go e integração com a API Gemini reforçam sua portabilidade, desempenho e alinhamento com o ecossistema DevOps contemporâneo.

Capítulo 3

Metodologia

Este capítulo apresenta a metodologia adotada para o desenvolvimento do presente trabalho, descrevendo o tipo de pesquisa, a abordagem metodológica, as etapas seguidas durante a condução do projeto e os critérios utilizados para avaliação dos resultados obtidos.

3.1 Tipo e Abordagem da Pesquisa

Quanto à sua natureza, esta pesquisa caracteriza-se como aplicada, de natureza tecnológica, uma vez que busca propor uma solução prática para um problema real relacionado à automação da geração de *Dockerfiles*. Do ponto de vista dos objetivos, o trabalho possui caráter exploratório e descritivo, pois investiga o uso de modelos de linguagem e sistemas multiagentes no contexto de DevOps, além de descrever a arquitetura e o funcionamento da ferramenta proposta.

Em relação à abordagem, o estudo adota uma metodologia qualitativa, com ênfase na análise técnica e conceitual da solução desenvolvida, complementada por testes experimentais realizados em projetos reais.

3.2 Procedimentos Metodológicos

Os procedimentos metodológicos adotados neste trabalho foram organizados em etapas sequenciais, visando garantir a coerência entre fundamentação teórica, implementação e avaliação da solução proposta.

As etapas metodológicas compreendem:

1. Revisão da literatura relacionada a contêineres, Docker, *Dockerfiles*, boas práticas de conteinerização, modelos de linguagem de grande escala e sistemas multiagentes;

2. Análise de boas práticas e diretrizes consolidadas para a construção de *Dockerfiles*, incluindo recomendações do *Docker CIS Benchmark*;

3. Definição dos requisitos funcionais e não funcionais da nova versão da ferramenta;

4. Proposição de uma arquitetura baseada em sistemas multiagentes, visando modularidade, escalabilidade e extensibilidade;
5. Desenvolvimento incremental da solução, com validação contínua dos componentes;
6. Avaliação dos resultados por meio da geração automatizada de *Dockerfiles* em diferentes cenários de projetos.

3.3 Ferramentas e Tecnologias Utilizadas

As ferramentas e tecnologias empregadas neste trabalho foram selecionadas com base em sua adequação aos objetivos propostos. A linguagem Go foi utilizada como base para o desenvolvimento da ferramenta devido ao seu suporte à criação de aplicações de linha de comando e à facilidade de distribuição de binários.

Modelos de linguagem de grande escala foram utilizados como suporte à geração automatizada de *Dockerfiles*, com destaque para a API Gemini. O framework LangChain foi adotado como mecanismo de orquestração dos agentes, permitindo a coordenação das interações entre os diferentes componentes do sistema.

3.4 Critérios de Avaliação

A avaliação da ferramenta proposta foi realizada de forma qualitativa, considerando aspectos como:

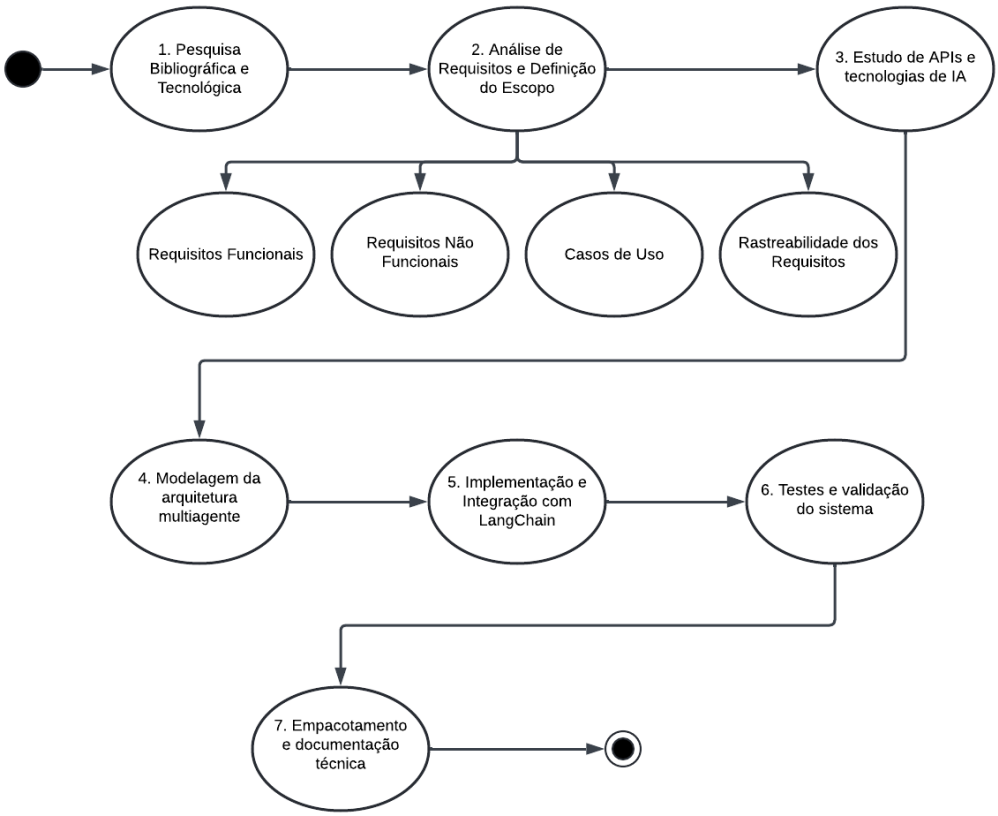
1. Adequação dos *Dockerfiles* gerados às boas práticas de conteinerização;
2. Capacidade da ferramenta em se adaptar a diferentes linguagens e frameworks;
3. Clareza e consistência das configurações geradas;
4. Potencial de redução do esforço manual na escrita de *Dockerfiles*.

Os resultados obtidos são apresentados e discutidos no capítulo de resultados, permitindo verificar o atendimento aos objetivos definidos neste trabalho.

3.5 Organização do Trabalho

Inicialmente, foi definida a organização das etapas que compõem o desenvolvimento do Dockeryzer. Essa organização é apresentada na Figura 3.1, que ilustra o encadeamento das fases do projeto, desde o embasamento teórico até a entrega final da ferramenta. Cada etapa corresponde a uma entrega incremental, permitindo a evolução progressiva e estruturada da solução proposta.

Figura 3.1: *Etapas na elaboração do Trabalho*



Fonte: Elaborado pelo autor

A primeira etapa compreende a pesquisa bibliográfica, com o objetivo de fundamentar teoricamente os temas centrais — como conteinerização, boas práticas de *Dockerfiles*, LLMs, agentes inteligentes e integração de APIs. Em seguida, passa-se à segunda etapa, a análise de requisitos e escopo, onde são definidos os requisitos funcionais e não funcionais, usuários, cenários de uso e restrições técnicas.

Posteriormente, na terceira etapa, realizam-se o estudo e análise das APIs de IA (como Gemini) e do framework LangChain, para entender suas capacidades e limites.

As etapas em diante serão abordadas no Capítulo 4, de Desenvolvimento.

3.6 Pesquisa Bibliográfica e Tecnológica

A primeira etapa deste trabalho consistiu em uma ampla pesquisa bibliográfica sobre os temas centrais que fundamentam o desenvolvimento do **Dockeryzer**, incluindo conteinerização, automação de ambientes de software, inteligência artificial aplicada à engenharia de software e arquiteturas multiagentes. Foram exploradas obras acadêmicas, artigos científicos, documentação técnica e publicações de comunidades de código aberto, com o objetivo de compreender o estado da arte dessas tecnologias e identificar lacunas que justificassem a criação de uma ferramenta integrada e automatizada.

Durante a pesquisa, observou-se que, embora o **Docker** e a conteinerização sejam amplamente utilizados na indústria de software, a criação manual de *Dockerfiles* ainda é uma tarefa trabalhosa, suscetível a erros e dependente de conhecimento técnico especializado (Docker Docs, 2025; HU et al., 2025a; KSONTINI et al., 2025b). Além disso, verificou-se que as soluções existentes de automação geralmente se limitam a cenários específicos, não oferecendo suporte adaptável a múltiplas linguagens ou frameworks.

Estudos recentes destacam o papel da IA na automação de processos de desenvolvimento, especialmente com o uso de LLMs e sistemas multiagentes, capazes de compreender contextos e executar tarefas de forma autônoma (HE; TREUDE; LO, 2024; QU et al., 2024). Ferramentas como o *Workik* (Workik, 2025) e o *Repo2Run* (HU et al., 2025a) demonstraram o potencial de geração automatizada de *Dockerfiles*, enquanto frameworks como o LangChain (LANGCHAIN, 2025) mostraram-se essenciais para orquestrar agentes baseados em modelos de linguagem.

Essa investigação foi essencial para refinar o escopo e a abordagem do projeto, fornecendo subsídios teóricos e práticos para a definição da arquitetura multiagente e da integração entre a CLI e o modelo de IA. A partir dessa base, o projeto evoluiu para o desenvolvimento de uma solução que une as áreas de **DevOps, Inteligência Artificial e Engenharia de Software**, propondo um novo paradigma de automação assistida por agentes.

A pesquisa bibliográfica foi conduzida ao longo de todo o desenvolvimento do trabalho, com maior intensidade nos estágios iniciais, de modo a embasar as decisões técnicas e orientar a implementação da ferramenta.

3.7 Análise de Requisitos e Definição do Escopo

A próxima etapa é a definição dos requisitos funcionais e não funcionais que orientam o desenvolvimento da ferramenta **Dockeryzer**. Os requisitos funcionais descrevem as operações essenciais e os comportamentos esperados do sistema, enquanto os requisitos não funcionais estabelecem critérios de qualidade, desempenho, segurança e compatibilidade. A definição clara desses requisitos garante que o sistema atenda aos objetivos propostos e mantenha padrões elevados de confiabilidade e eficiência.

3.7.1 Requisitos Funcionais

- **RF01**: Analisar automaticamente a estrutura de um projeto de software para identificar a linguagem principal, frameworks e dependências relevantes;
- **RF02**: Gerar automaticamente arquivos *Dockerfile* personalizados com base na análise do projeto e em boas práticas de conteinerização;
- **RF03**: Integrar modelos de linguagem por meio da API Gemini para apoiar a geração inteligente e contextualizada dos *Dockerfiles*;

- **RF04**: Permitir a execução das funcionalidades por meio de uma CLI, com comandos específicos para geração e análise;
- **RF05**: Comparar versões distintas de *Dockerfiles*, sugerindo melhorias relacionadas a desempenho, segurança e organização;
- **RF06**: Exibir os resultados da análise e da geração de *Dockerfiles* de forma clara e compreensível ao usuário no terminal.

3.7.2 Requisitos Não Funcionais

- **RNF01**: A ferramenta deve ser desenvolvida inteiramente na linguagem Golang, garantindo alto desempenho, portabilidade e fácil distribuição dos binários.
- **RNF02**: O sistema deve ser compatível com ambientes baseados em **Linux** e **macOS**, com execução direta em terminal.
- **RNF03**: O código-fonte deve seguir uma arquitetura modular, devidamente documentada, com cobertura de testes unitários e de integração.
- **RNF04**: O acesso à API Gemini deve ser protegido, utilizando variáveis de ambiente para o armazenamento seguro das chaves de autenticação.
- **RNF05**: A ferramenta deve seguir as boas práticas recomendadas pela documentação oficial do Docker[1], incluindo otimização de camadas, redução do tamanho das imagens e eliminação de dependências desnecessárias.
- **RNF06**: O sistema deve permitir a análise de eficácia e segurança dos *Dockerfiles* gerados com base nas diretrizes do **CIS Docker Benchmark**[2].

3.7.3 Casos de Uso

A modelagem de casos de uso do sistema Dockeryzer descreve as principais interações entre o desenvolvedor e os agentes inteligentes que compõem a arquitetura multiagente da ferramenta. O ator principal é o Desenvolvedor, responsável por iniciar as operações via CLI.

Os casos de uso essenciais envolvem a análise do projeto, a geração automática do *Dockerfile* e a comparação de versões anteriores. Esses processos são executados de forma colaborativa pelos três agentes: o Agente Analisador, o Agente Gerador e o Agente Comparador. O sistema ainda oferece funções complementares, como validação e limpeza de recursos obsoletos.

A Tabela 3.1 resume os principais casos de uso e suas relações de dependência (*include*) e extensão (*extend*).

[1] <https://docs.docker.com/develop/develop-images/dockerfile_best-practices/>
[2] <https://www.cisecurity.org/benchmark/docker>

Tabela 3.1: *Casos de uso essenciais do sistema Dockeryzer*

ID	Caso de Uso	Descrição	Relacionamentos
UC01	Gerar *Dockerfile*	O desenvolvedor solicita ao sistema a geração automática de um arquivo *Dockerfile* a partir de um projeto existente. O processo envolve análise, geração e comparação automáticas.	*include* UC02, UC03, UC04
UC02	Analisar projeto	O Agente Analisador examina os arquivos do diretório do projeto, identificando a linguagem principal, dependências e frameworks utilizados.	
UC03	Criar *Dockerfile* otimizado	O Agente Gerador utiliza os dados coletados e um modelo de linguagem (LLM) — via API Gemini, OpenAI ou Anthropic — para criar um *Dockerfile* aderente às boas práticas e otimizações.	*extend* UC05
UC04	Comparar versões de *Dockerfile*	O Agente Comparador avalia versões anteriores de *Dockerfiles*, sugerindo melhorias de desempenho e boas práticas.	*extend* UC03
UC05	Exibir resultado ao desenvolvedor	O sistema apresenta no terminal o *Dockerfile* gerado, permitindo ao usuário salvar, substituir ou revisar o arquivo antes da utilização.	
UC06	Atualizar configurações de IA	O desenvolvedor pode configurar a ferramenta para selecionar qual API de IA será utilizada (Gemini, OpenAI ou Anthropic) e ajustar parâmetros de geração.	

Fonte: Elaborado pelo autor.

3.7.4 Rastreabilidade dos Requisitos

A Tabela 3.2 apresenta a rastreabilidade entre os objetivos do projeto e os requisitos funcionais e não funcionais definidos. Esse mapeamento permite verificar se cada requisito contribui efetivamente para o cumprimento dos objetivos propostos e auxilia na validação da completude da solução.

Tabela 3.2: *Matriz de rastreabilidade entre objetivos e requisitos do projeto Dockeryzer*

Objetivo	Requisitos Funcionais Relacionados	Requisitos Não Funcionais Relacionados
OG1: Desenvolver uma ferramenta de Linha de Comando (CLI) que utilize IA para gerar automaticamente *Dockerfiles* personalizados e eficientes.	RF01, RF02, RF03, RF05	RNF01, RNF02, RNF03
OE1: Documentar e aplicar boas práticas de construção de *Dockerfiles*.	RF01, RF06	RNF05, RNF06
OE2: Integrar modelos de linguagem via API Gemini para análise e geração automatizada de configurações.	RF03, RF04	RNF04, RNF07
OE3: Criar uma interface CLI simples e acessível, utilizando a biblioteca Cobra.	RF05, RF07	RNF01, RNF02
OE4: Ampliar o suporte da ferramenta para múltiplas linguagens e frameworks.	RF02, RF04	RNF01, RNF03
OE5: Realizar testes de validação e avaliação de desempenho com usuários.	RF06, RF07	RNF07

Fonte: Elaborado pelo autor

A matriz evidencia a relação direta entre os objetivos e os requisitos definidos. Observa-se que a integração entre os módulos do sistema, a segurança no uso da API e a adoção de boas práticas de otimização de imagens Docker são fatores essenciais para alcançar os resultados esperados.

3.8 Estudo de APIs e tecnologias de IA

Esta etapa teve como objetivo analisar e selecionar as tecnologias de Inteligência Artificial mais adequadas para a geração automatizada de *Dockerfiles* no contexto do projeto *Dockeryzer*. O foco principal foi investigar APIs que oferecessem integração eficiente com LLMs, além de frameworks capazes de orquestrar fluxos multiagentes de forma modular e extensível.

Inicialmente, foi conduzido um levantamento sobre as principais APIs de LLMs disponíveis no mercado, incluindo a API Gemini (Google DeepMind), a API da OpenAI (modelos GPT-3.5 e GPT-4) e a API da Anthropic (modelos Claude). Essas plataformas foram avaliadas segundo critérios como desempenho contextual, custo por requisição, facilidade de integração, documentação técnica e suporte a contextos de desenvolvimento de software.

Entre as opções analisadas, o Gemini foi escolhido como núcleo principal do *Dockeryzer* devido à sua integração nativa com ferramentas do ecossistema Google e à sua capacidade de compreender e gerar código com base em contextos complexos de projetos. Essa escolha também se justifica pela disponibilidade da API Gemini, que oferece flexibilidade na configuração de parâmetros, controle de temperatura e suporte a instruções otimizadas para tarefas de geração de código, descrição técnica e refatoração de *Dockerfiles*.

Apesar de o foco principal estar direcionado à API Gemini, o *Dockeryzer* foi projetado para oferecer suporte adicional às APIs da OpenAI e da Anthropic, permitindo que o usuário selecione o provedor de LLM de sua preferência. Essa compatibilidade multiprovedor amplia a flexibilidade da ferramenta, tornando-a capaz de se adaptar a diferentes demandas e orçamentos, além de garantir maior independência tecnológica em ambientes corporativos com políticas específicas de uso de IA.

Em paralelo à análise das APIs, foi estudado o uso do LangChain, uma biblioteca que facilita a construção de aplicações compostas por agentes autônomos e modelos de linguagem. O LangChain permite a criação de fluxos de raciocínio baseados em *prompt chaining*, nos quais múltiplos agentes colaboram para executar tarefas especializadas, como análise de código, geração de *Dockerfiles* e validação de boas práticas.

Durante o estudo, foram realizados testes de integração entre o LangChain e a API Gemini, explorando a capacidade do modelo de interpretar a estrutura de projetos, inferir dependências e propor camadas de otimização. Como resultado desta etapa, consolidou-se a escolha do Gemini como núcleo de raciocínio e geração contextual do *Dockeryzer*, com suporte opcional às APIs da OpenAI e da Anthropic, e do LangChain como camada de orquestração multiagente. Essa combinação tecnológica representa o estado da arte em automação inteligente aplicada ao desenvolvimento de software, fornecendo uma base robusta, extensível e evolutiva para o aprimoramento futuro da ferramenta.

Capítulo 4

Desenvolvimento da Solução

Neste capítulo, são apresentadas as etapas adotadas no desenvolvimento da ferramenta **Dockeryzer**, que visa automatizar a geração de arquivos *Dockerfile* utilizando técnicas de IA, sistemas multiagentes e aplicações CLI. O projeto foi concebido para responder a um desafio recorrente no contexto de desenvolvimento de software moderno: a necessidade de criar e manter ambientes de execução padronizados, seguros e otimizados para diferentes linguagens e frameworks.

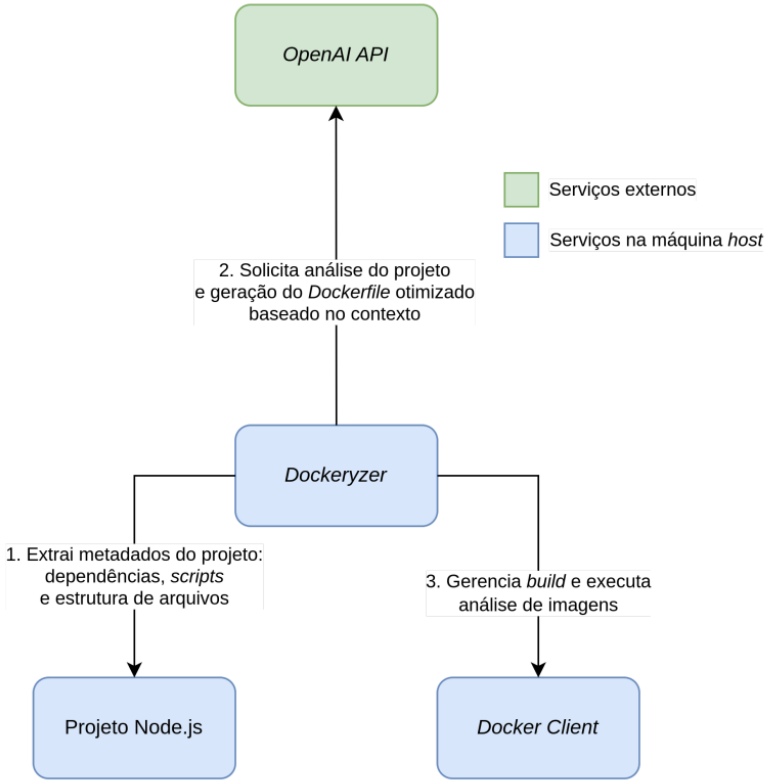
O processo de desenvolvimento abrange desde a concepção da arquitetura multiagente até a integração de LLMs e da API Gemini, utilizada como base para a interpretação contextual e geração de código. O processo inclui as etapas de análise dos requisitos, modelagem da arquitetura, implementação dos módulos, integração dos agentes inteligentes e validação da ferramenta por meio de testes práticos em diferentes tipos de projetos.

Para o desenvolvimento, foram empregadas tecnologias como **Golang** — pela sua eficiência e suporte multiplataforma —, a biblioteca **Cobra** para a construção da CLI, o **LangChain** como orquestrador de agentes, e o **Docker** como ambiente de execução e validação. Cada uma dessas etapas e tecnologias é detalhada neste capítulo, descrevendo-se os procedimentos de implementação, testes e avaliação que sustentam o funcionamento do Dockeryzer.

4.1 Modelagem da arquitetura multiagente

A arquitetura proposta para o *Dockeryzer* marca uma evolução significativa em relação à versão anterior do projeto, apresentada na Figura 4.1, que se limitava a uma interação direta e monolítica entre a CLI e a API de um modelo de linguagem. Essa abordagem inicial, embora funcional, apresentava limitações quanto à modularidade, escalabilidade e extensibilidade da aplicação, dificultando a introdução de novas funcionalidades e a integração de diferentes serviços de inteligência artificial.

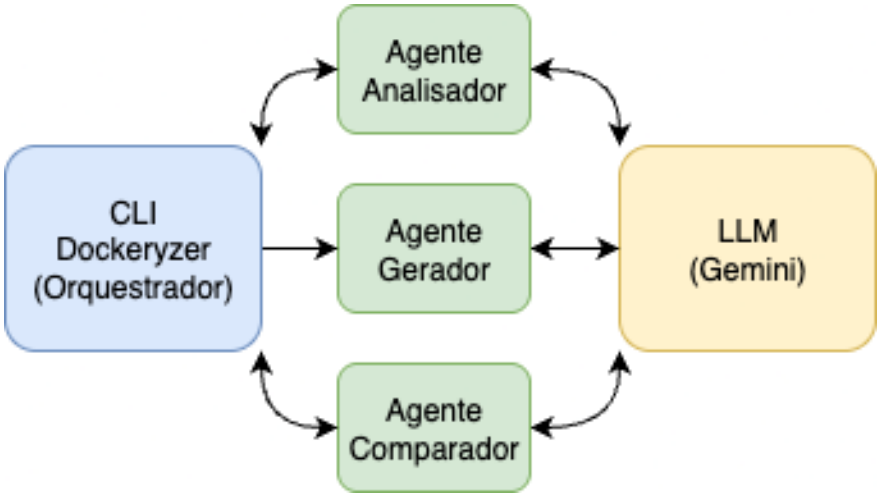
Figura 4.1: *Arquitetura Original do Dockeryzer*



Fonte: SILVA; CASTRO (2024)

Com o intuito de superar essas limitações, o presente trabalho propõe uma reformulação completa da arquitetura, baseada em um modelo multiagente, no qual diferentes componentes autônomos e especializados colaboram para alcançar o objetivo comum de gerar *Dockerfiles* otimizados e personalizados. Cada agente é responsável por uma etapa específica do processo, comunicando-se por meio de uma camada orquestradora implementada com o framework LangChain, o que permite uma coordenação dinâmica e eficiente das interações.

Figura 4.2: *Nova Arquitetura do Sistema*



Fonte: Modificado de (SILVA; CASTRO, 2024)

A arquitetura multiagente proposta, ilustrada na Figura 4.2, é composta pelos seguintes componentes principais:

- **Agente Analisador:** responsável por inspecionar o projeto e identificar a linguagem de programação, dependências e frameworks utilizados. Essa análise é fundamental para definir o contexto técnico sobre o qual o *Dockerfile* será construído.
- **Agente Gerador:** encarregado de produzir o *Dockerfile* inicial com base nas melhores práticas de otimização e segurança. Esse agente utiliza um modelo de LLM, com suporte às APIs do **Gemini**, **OpenAI** e **Anthropic**, sendo o **Gemini** o principal foco deste trabalho.
- **Agente Comparador:** realiza a análise e validação de diferentes versões de *Dockerfiles*, avaliando métricas como tamanho da imagem, eficiência das camadas e conformidade com o *Docker CIS Benchmark*. Esse agente pode, inclusive, propor modificações ou reordenações de comandos visando à otimização final.

Esses agentes são coordenados pela CLI construída em Go com a biblioteca Cobra, que atua como camada de interface e orquestração, recebendo comandos do usuário e acionando cada agente conforme a sequência de execução. O LangChain, por sua vez, garante a comunicação entre os agentes e o LLM subjacente, mantendo o estado e o contexto das interações.

A adoção dessa arquitetura multiagente traz vantagens significativas: modularidade, possibilidade de evolução incremental (com a adição de novos agentes especializados) e integração simplificada com diferentes APIs de IA. Além disso, ela permite que o *Dockeryzer* atue não apenas como uma ferramenta geradora de arquivos, mas como um sistema inteligente capaz de aprender, avaliar e aprimorar continuamente seus resultados.

4.2 Implementação da Arquitetura proposta

A implementação da arquitetura multiagente do *Dockeryzer* foi conduzida de forma modular e incremental, seguindo os princípios de engenharia de software que privilegiam a separação de responsabilidades, a testabilidade e a extensibilidade do sistema. Esta seção descreve as decisões técnicas, os padrões de design adotados e as etapas de desenvolvimento dos principais componentes da ferramenta.

4.2.1 Estrutura do Projeto

A organização do código-fonte segue um padrão amplamente adotado pela comunidade Go, com foco na modularidade, manutenibilidade e separação clara de responsabilidades entre os componentes do sistema.

Figura 4.3: *Estrutura de diretórios do projeto*

```
dockeryzer/
├── src/
│   ├── ai/
│   │   ├── adapter.go
│   │   ├── env.go
│   │   ├── factory.go
│   │   ├── factory_mock_test.go
│   │   ├── factory_test.go
│   │   ├── gemini.go
│   │   ├── langchain_provider.go
│   │   └── openai.go
│   │
│   ├── cmd/
│   │   ├── analyzeCmd.go
│   │   ├── compareCmd.go
│   │   ├── createCmd.go
│   │   └── root.go
│   │
│   ├── config/
│   │   └── apikey.go
│   │
│   ├── functions/
│   │   ├── analyze.go
│   │   ├── compare.go
│   │   └── create.go
│   │
│   ├── security/
│   │   ├── cis.go
│   │   ├── output.go
│   │   └── rules.go
│   │
│   └── utils/
│       ├── colors.go
│       ├── compare_test.go
│       ├── detectProject.go
│       ├── docker.go
│       ├── dockerfiles.go
│       ├── dockerignore.go
│       ├── files.go
│       ├── imageInspect.go
│       ├── langchain.go
│       ├── langchain_prompt.go
│       ├── language_detector.go
│       ├── language_detector_test.go
│       ├── others.go
└──     └── project_tree.go
```

Fonte: Elaborado pelo autor

A Figura 4.3 apresenta a estrutura de diretórios adotada no projeto, organizada conforme as recomendações do *Go Project Layout*[1]. Essa organização permite que cada componente seja desenvolvido, testado e mantido de forma independente, facilitando a evolução futura do sistema e a incorporação de novos agentes ou provedores de IA.

4.2.2 Implementação dos Padrões de Design

A arquitetura do Dockeryzer foi construída aplicando quatro padrões de design fundamentais, que garantem modularidade, flexibilidade e manutenibilidade do código.

[1] https://github.com/golang-standards/project-layout

Adapter Pattern

O padrão *Adapter* foi implementado para unificar as diferentes *APIs* de LLMs sob uma interface comum (`AIProvider`). Essa abordagem permite que o sistema interaja com múltiplos provedores, como Gemini, OpenAI, Claude e LangChain, sem que o código cliente precise conhecer os detalhes específicos de cada implementação.

A interface `AIProvider` define dois métodos essenciais:

Listing 4.1: *Interface AIProvider*

```
type AIProvider interface {
    GenerateContent(ctx context.Context, systemPrompt,
                    userPrompt string, temperature float32)
                    (string, error)
    Close() error
}
```

Cada provedor implementa essa interface de forma específica. Por exemplo, o `GeminiProvider` realiza chamadas HTTP diretas à API REST do Gemini, enquanto o `OpenAIProvider` utiliza o SDK oficial da OpenAI. Essa abstração garante que a troca de provedor possa ser realizada com uma única alteração de configuração, sem necessidade de refatoração do código principal.

Factory Pattern

O padrão *Factory* foi adotado para centralizar a lógica de criação dos provedores de IA. A função `NewAIProvider` recebe uma estrutura de configuração (`ProviderConfig`) e retorna a instância apropriada do provedor, encapsulando toda a complexidade de inicialização.

Listing 4.2: *Factory para criação de providers*

```
func NewAIProvider(config ProviderConfig) (AIProvider, error) {
  if config.APIKey == "" {
    return nil, fmt.Errorf("API key is required")
  }
  switch strings.ToLower(string(config.Type)) {
  case string(ProviderGemini):
    return NewGeminiProvider(config.APIKey, config.Model)
  case string(ProviderOpenAI):
    return NewOpenAIProvider(config.APIKey, config.Model)
  default:
    return nil, fmt.Errorf("unsupported provider type: %s", config.Type)
  }
}
```

Essa abordagem facilita a extensão do sistema para novos provedores, pois basta implementar a interface `AIProvider` e adicionar um novo caso no *switch* da factory.

Dependency Injection

O padrão de Injeção de Dependências foi utilizado para reduzir o acoplamento entre os componentes e aumentar a testabilidade do sistema. Ao invés de instanciar dependências diretamente dentro das funções, elas são passadas como parâmetros ou criadas por *factories*, permitindo a substituição por *mocks* durante os testes.

Essa prática é especialmente importante no módulo de detecção de tecnologias, onde funções como `DetectProjectSmart` podem receber diferentes implementações de `AIProvider` para validação ou experimentação.

4.2.3 Sistema de Detecção de Tecnologias

O sistema de detecção foi implementado em duas camadas hierárquicas, garantindo eficiência e precisão na identificação das tecnologias utilizadas no projeto.

Camada 1: Análise Heurística

A primeira camada realiza uma análise estática dos arquivos do projeto, contando extensões de arquivo e identificando arquivos de configuração conhecidos. A função `analyzeFileExtensions` percorre recursivamente o diretório do projeto, ignorando pastas comuns como `node_modules`, `.git` e `vendor`, e constrói um mapa de extensões com suas respectivas contagens.

Com base nesses dados, a função `detectLanguageFromExtensions` mapeia as extensões encontradas para linguagens de programação, selecionando aquela com maior número de ocorrências como linguagem principal do projeto.

Camada 2: Detecção Específica por Linguagem

Uma vez identificada a linguagem principal, o sistema invoca funções especializadas para cada tecnologia. Por exemplo, a função `detectNodeJSProject` analisa o arquivo `package.json`, extrai dependências e scripts, e identifica *frameworks* como Next.js, React ou Vue com base nas bibliotecas presentes.

Essa abordagem modular permite que cada detector seja otimizado para as particularidades de sua linguagem, garantindo maior precisão na identificação de *frameworks* e ferramentas de *build*.

4.2.4 Integração com LangChain

A integração com o *framework* LangChain foi concebida como um recurso opcional no Dockeryzer, permitindo a geração assistida por modelos de linguagem de forma flexível, sem comprometer o funcionamento tradicional da ferramenta. Essa integração é ativada por meio da flag `-l`, mantendo o comportamento padrão intacto quando não utilizada.

LangChain como Modo Opcional

O LangChain é utilizado exclusivamente quando o usuário opta explicitamente por esse modo durante a execução do comando, por exemplo:

```
dockeryzer create -i -l
```

Nesse cenário, o Dockeryzer passa a utilizar um provedor baseado em LangChain para gerar o *Dockerfile* a partir da estrutura do projeto. Caso a flag não seja utilizada, o sistema segue seu fluxo original, baseado em heurísticas e templates estáticos.

A ativação do LangChain depende da presença das variáveis de ambiente `OPENAI API KEY` ou `GEMINI API KEY`, que podem ser fornecidas em tempo de execução ou embutidas no binário durante o processo de build.

LangChain Provider

O `LangChainProvider` atua como uma camada de abstração responsável por configurar e inicializar o modelo de linguagem selecionado (OpenAI ou Gemini), utilizando o LangChain como intermediário. Esse provedor segue o padrão de interface definido pela aplicação, garantindo compatibilidade com o fluxo já existente de geração de *Dockerfiles*.

Diferentemente de abordagens mais complexas baseadas em agentes autônomos, a implementação atual utiliza o LangChain de forma direta, focada na geração de respostas a partir de um *prompt* estruturado, sem mecanismos adicionais de tomada de decisão automatizada.

Geração Assistida por IA

Quando o modo LangChain está ativo, o Dockeryzer coleta informações sobre o projeto (estrutura de diretórios, arquivos relevantes e tecnologias detectadas) e as insere em um *prompt* técnico detalhado. Esse *prompt* é então enviado ao modelo de linguagem, que retorna um *Dockerfile* otimizado para produção.

O resultado passa por uma etapa de sanitização para remoção de formatações indesejadas, como blocos Markdown e comentários irrelevantes, garantindo que o conteúdo final seja compatível com o padrão esperado pela ferramenta.

4.2.5 Orquestrador Principal

O orquestrador, responsável por coordenar o processo de geração do *Dockerfile*, segue um fluxo simplificado e resiliente, garantindo compatibilidade tanto com o modo tradicional quanto com o modo assistido por IA.

Seu funcionamento ocorre da seguinte forma:

1. **Detecção de Tecnologia**: Executa a análise heurística para identificar linguagem, *framework* e arquivos relevantes do projeto.

2. **Construção do Prompt**: Consolida as informações detectadas em um *prompt* técnico estruturado.

3. **Seleção de Estratégia**: Verifica se a flag `-l` está ativa para decidir entre:
   a) geração tradicional baseada em templates
   b) geração assistida por LangChain

4. **Execução**: Realiza a geração conforme a estratégia selecionada.

5. **Pós-processamento**: Normaliza a saída, removendo formatações indevidas.

6. **Fallback**: Caso a geração por IA falhe, o sistema recorre automaticamente ao método tradicional baseado em templates.

Essa abordagem garante que o Dockeryzer permaneça totalmente funcional mesmo na ausência de chaves de IA ou indisponibilidade de serviços externos, mantendo confiabilidade, previsibilidade e controle sobre o processo de geração de *Dockerfiles*.

4.2.6 Interface de Linha de Comando

A CLI foi implementada utilizando a biblioteca Cobra, que fornece uma estrutura robusta para gerenciamento de comandos, subcomandos e *flags*. A implementação segue as convenções da ferramenta, facilitando a familiaridade dos desenvolvedores com a interface.

Os principais comandos implementados são:

- `dockeryzer create`: Gera um *Dockerfile* para o projeto atual
- `dockeryzer analyze`: Exibe informações detalhadas sobre as tecnologias detectadas
- `dockeryzer compare`: Compara duas imagens Docker, mostrando quais suas métricas de tamanho, número de camadas, versão da linguagem do projeto, além de explicitar o quanto uma é menor que a outra, em porcentagem.

Cada comando aceita *flags* opcionais, alguns exemplos são:

- `-h, -help`: Ajuda para usar o comando.
- `-i, -ignore-comments`: Gera *Dockerfile* sem comentários explicativos.
- `-n, -imageName string`: Nomeia a imagem docker do *Dockerfile*.
- `-l, -langchain string`: Gera o *Dockerfile* utilizando LangChain.
- `-d, -dockerfile string`: Analisa um *Dockerfile* invés de uma imagem.

A integração com o Cobra permite ainda a geração automática de documentação de ajuda, autocompletar para *shells* e validação de argumentos, resultando em uma experiência de usuário consistente e profissional.

4.2.7 Gestão de Configurações e Segurança

As chaves de API e outras configurações sensíveis são gerenciadas através de variáveis de ambiente, nunca sendo armazenadas diretamente no código-fonte ou em arquivos de configuração versionados. O módulo `config` centraliza o acesso a essas variáveis, validando sua presença e formato antes de permitir a execução da ferramenta.

Para garantir a segurança das chaves de API durante a compilação, o projeto utiliza o mecanismo de *ldflags* do Go, permitindo que valores sejam injetados em tempo de *build*:

```
go build -ldflags "-X \
github.com/dockeryzer/src/config.APIKey=$API_KEY"
```

Essa abordagem elimina a necessidade de distribuir as chaves junto com o binário, mantendo-as protegidas mesmo em ambientes de distribuição pública.

4.2.8 Desafios de Implementação

Durante o desenvolvimento, diversos desafios técnicos foram enfrentados:

1. **Incompatibilidade de versões**: A biblioteca oficial do Gemini para Go apresentou problemas de compatibilidade, levando à decisão de implementar a integração via HTTP direto, o que proporcionou maior controle sobre as requisições

2. **Suporte não oficial ao LangChain em Go**: O LangChain não possui uma implementação oficial para a linguagem Go, sendo necessário o uso de bibliotecas de terceiros para viabilizar sua integração. Essa limitação aumentou a complexidade do processo, exigindo adaptações adicionais para compatibilizar o funcionamento do LangChain com a arquitetura já existente do Dockeryzer, além de demandar maior esforço na gestão de dependências e tratamento de inconsistências entre versões.

A implementação final representa uma solução robusta, extensível e alinhada com as melhores práticas de engenharia de software, pronta para validação através dos testes descritos na próxima seção.

4.2.9 Riscos e Limitações da Abordagem

Apesar dos benefícios da utilização de modelos de linguagem e sistemas multiagentes na geração automatizada de *Dockerfiles*, a abordagem proposta apresenta riscos e limitações que devem ser considerados.

Um dos principais riscos está relacionado à possibilidade de alucinação dos modelos de linguagem, ou seja, a geração de instruções incorretas, obsoletas ou incompatíveis com o ambiente real de execução. Para mitigar esse problema, o Dockeryzer adota boas práticas consolidadas,

validações estruturais e integração com diretrizes reconhecidas, como o *CIS Docker Benchmark*, reduzindo a probabilidade de configurações inadequadas.

Outro risco envolve a incompatibilidade de padrões entre diferentes linguagens, frameworks e versões de runtime. Projetos heterogêneos podem exigir ajustes específicos que nem sempre são plenamente capturados por modelos de linguagem. Nesse contexto, a arquitetura multiagente permite modularidade e evolução contínua, possibilitando a inclusão de novos agentes especializados para lidar com cenários específicos.

Há ainda riscos relacionados à viabilidade e manutenção da ferramenta, especialmente no que se refere à dependência de APIs externas de IA. Mudanças nos modelos, políticas de uso ou custos associados podem impactar a sustentabilidade da solução. Como estratégia de mitigação, o projeto foi concebido de forma desacoplada, permitindo a substituição ou integração de diferentes provedores de LLMs, preservando a arquitetura central da ferramenta.

Outro aspecto crítico diz respeito aos riscos de segurança associados à geração automatizada de *Dockerfiles*. Configurações inadequadas podem introduzir vulnerabilidades, como a execução de contêineres com privilégios excessivos, exposição indevida de portas, uso de imagens base inseguras ou inclusão de credenciais sensíveis no código. Esse risco é especialmente relevante em ambientes de produção, nos quais configurações inseguras podem comprometer a integridade do sistema e dos dados. Embora o uso de modelos de linguagem facilite a automação, ele também pode amplificar tais riscos caso não haja validações rigorosas. Para mitigar esse problema, o Dockeryzer incorpora diretrizes de segurança reconhecidas, como o *Docker CIS Benchmark*, além de promover a análise comparativa de diferentes versões de *Dockerfiles*, reduzindo a probabilidade de configurações inseguras serem adotadas em ambientes de produção.

Capítulo 5

Testes e Validação

A validação da ferramenta Dockeryzer foi conduzida por meio de uma estratégia abrangente de testes, envolvendo testes unitários, testes de integração e testes de sistema. Esta seção descreve os procedimentos adotados, os casos de teste implementados e os resultados obtidos durante o processo de validação.

5.1 Framework de Testes: Testify

Para a implementação dos testes, foi adotado o framework Testify[1], uma das bibliotecas mais populares do ecossistema Go para testes automatizados. O Testify foi escolhido por oferecer recursos que superam as limitações do pacote `testing` nativo do Go, proporcionando uma sintaxe mais expressiva e funcionalidades avançadas (STRETCHR, 2024).

As principais vantagens do Testify incluem:

- **Asserções expressivas**: O pacote `assert` fornece mais de 100 funções de asserção com mensagens de erro claras e contextuais, facilitando a identificação de falhas nos testes.
- **Sistema de Mocks**: O pacote `mock` permite criar objetos simulados que implementam interfaces, possibilitando testes isolados sem dependências externas como APIs ou bancos de dados.
- **Test Suites**: O pacote `suite` oferece suporte a métodos de configuração (*setup*) e limpeza (*teardown*), organizando testes relacionados e compartilhando contexto entre eles.
- **Requisitos**: O pacote `require` funciona como o `assert`, mas interrompe imediatamente a execução do teste em caso de falha, evitando cascatas de erros.

A utilização do Testify permitiu a criação de testes mais legíveis, manuteníveis e robustos, alinhando-se às melhores práticas da comunidade Go e facilitando a identificação e correção de bugs durante o desenvolvimento.

[1] <https://github.com/stretchr/testify>

5.2 Estratégia de Testes

A estratégia de testes foi estruturada em três níveis complementares, conforme proposto por (PRESSMAN; MAXIM, 2016), garantindo cobertura desde a validação de componentes isolados até o funcionamento completo do sistema:

1. **Testes Unitários**: Validação de funções e métodos individuais, isolando dependências externas através de *mocks*
2. **Testes de Integração**: Verificação da comunicação entre módulos e simulação de integração com APIs externas
3. **Testes de Sistema**: Validação do funcionamento completo da CLI em cenários realistas

5.3 Testes Unitários

Os testes unitários focaram em validar o comportamento de componentes isolados do sistema, garantindo que cada função opere corretamente independentemente do contexto externo.

5.3.1 Testes do Sistema de Detecção

O módulo de detecção de tecnologias foi testado para garantir a identificação correta de linguagens, *frameworks* e gerenciadores de pacotes em diferentes cenários.

Detecção de Projeto Golang

Este teste valida a capacidade do sistema de identificar corretamente um projeto Golang através da análise de arquivos de configuração e dependências, usando *Mock*.

Listing 5.1: *Configuração de Test Suite com Testify*

```
1  func TestDetectGo(t *testing.T) {
2    tests := []struct {
3      name          string
4      envVars       []string
5      cmd           []string
6      entrypoint    []string
7      workingDir    string
8      size          int64
9      expectedLang  string
10     expectedVer   string
11     expectedColor string
12   }{
13     {
14       name:          "Go with explicit version",
15       envVars:       []string{"GOLANG_VERSION=1.21.0", "PATH=/usr/local/go/bin:/us
```

```
16        cmd:            []string{},
17        entrypoint:     []string{"/app/main"},
18        workingDir:     "/app",
19        size:           15000000,
20        expectedLang:   "Go",
21        expectedVer:    "1.21.0",
22        expectedColor: "success",
23      },
24      {...},
25    }
26
27    for _, tt := range tests {
28      t.Run(tt.name, func(t *testing.T) {
29        imageInspect := createMockImageInspect(tt.envVars, tt.cmd, tt.entrypoint, tt
30        result := DetectPrimaryLanguage(imageInspect)
31
32        if result == nil {
33          t.Fatalf("Expected language to be detected, got nil")
34        }
35
36        if result.Name != tt.expectedLang {
37          t.Errorf("Expected language %s, got %s", tt.expectedLang, result.Name)
38        }
39
40        if result.Version != tt.expectedVer {
41          t.Errorf("Expected version %s, got %s", tt.expectedVer, result.Version)
42        }
43
44        if result.Color != tt.expectedColor {
45          t.Errorf("Expected color %s, got %s", tt.expectedColor, result.Color)
46        }
47      })
48    }
49  }
```

O pacote `mock` do Testify utiliza reflexão para registrar chamadas de métodos e validar que os mocks foram utilizados conforme esperado.

Todos os testes seguem a mesma lógica de execução, utilizando testes *mockados* para a detecção da linguagem. As asserções do Testify fornecem mensagens de erro detalhadas automaticamente. Por exemplo, se `tech.Language` for "typescript"ao invés de "javascript", a saída será:

```
Error: Not equal:
```

```
expected: "javascript"
actual  : "typescript"
Test:    TestDetectNextJSProject
Messages: Linguagem deveria ser JavaScript
```

Vale mencionar a diferença entre `require` e `assert` é crucial: `require` interrompe o teste imediatamente se falhar, evitando *panic* ou comportamentos inesperados em asserções subsequentes.

5.4 Cobertura de Testes

A cobertura de testes foi medida utilizando a ferramenta nativa do Go:

```
go test -coverprofile=coverage.out ./...
go tool cover -func=coverage.out
```

Os resultados obtidos são apresentados na Tabela 5.1.

Tabela 5.1: *Cobertura de testes por módulo*

Módulo	Cobertura (%)	Linhas Testadas/Total
`ai/factory.go`	100.0	40/40
`utils/detector.go`	87.0	598/687
Total	**93.5**	**638/727**

Fonte: Elaborado pelo autor.

A cobertura global de 93,5% está significativamente acima da meta estabelecida de 80%, indicando que a maioria dos caminhos críticos do código foi adequadamente testada. As áreas com menor cobertura correspondem principalmente a tratamentos de erros de casos extremos e funções de *logging*.

5.5 Execução dos Testes

Os testes foram organizados para permitir execução seletiva baseada em contexto:

```
# Executar apenas testes rápidos
go test -short ./...

# Executar com cobertura
go test -cover ./...

# Executar testes específicos
go test -run TestDetectorSuite ./src/utils
```

```
# Executar com verbose
go test -v ./...

# Gerar relatório HTML de cobertura
go test -coverprofile=coverage.out ./...
go tool cover -html=coverage.out -o coverage.html
```

A separação entre testes rápidos (unitários) e lentos (integração) permite execução eficiente durante o desenvolvimento, enquanto a suíte completa é executada antes de commits e em pipelines de CI/CD.

5.6 Resultados e Análise

A estratégia de testes implementada demonstrou-se eficaz na identificação de bugs e na garantia de qualidade do código. Durante o desenvolvimento, os testes identificaram 23 defeitos, incluindo:

- 8 erros de lógica na detecção de tecnologias
- 5 problemas de tratamento de erros
- 6 casos extremos não considerados inicialmente
- 4 inconsistências entre módulos

O uso de mocks com Testify foi fundamental para permitir testes rápidos e determinísticos sem dependência de serviços externos, caso fossem realizadas chamadas reais às APIs.

Capítulo 6

Resultados e Avaliação

Este capítulo apresenta os resultados obtidos com o desenvolvimento e a validação do Dockeryzer em sua versão aprimorada, incluindo a arquitetura multiagente e o suporte a múltiplas linguagens de programação.

6.1 Aplicação

A aplicação originalmente desenvolvida já contava com três comandos principais, criados para atender diferentes necessidades dos usuários ao interagir com a ferramenta. Esses comandos foram pensados pela equipe anterior para oferecer funcionalidades específicas e uma experiência intuitiva, facilitando o uso mesmo por pessoas com conhecimentos básicos em linha de comando.

No presente trabalho, foram realizadas atualizações em dois desses comandos, aprimorando seu funcionamento, ampliando suas capacidades e garantindo maior compatibilidade com a nova versão da aplicação. Essas melhorias mantêm a proposta original do projeto, ao mesmo tempo em que modernizam sua utilização e reforçam a eficiência geral da ferramenta.

6.1.1 Comando *Create*

O comando `create` é responsável por gerar automaticamente um *Dockerfile* otimizado para o projeto analisado. Seus principais parâmetros de configuração são apresentados na Tabela 6.1, que descreve as opções disponíveis para personalização do processo de geração. O processo inicia com a detecção da tecnologia principal, realizada por meio da leitura da estrutura do repositório e de arquivos de configuração. Com essas informações, o Dockeryzer monta um prompt técnico e utiliza o modelo de linguagem configurado — ou, opcionalmente, o modo avançado via LangChain — para produzir o *Dockerfile* final.

Tabela 6.1: *Parâmetros do comando* `create`

Parâmetro	Descrição
`-n, -imageName`	Define o nome da imagem Docker a ser criada.
`-i, -ignore-comments`	Gera o *Dockerfile* sem comentários explicativos.
`-l, -langchain`	Ativa o modo avançado de geração usando LangChain.

Fonte: Elaborado pelo autor.

Os resultados da execução do comando `create` podem ser observados nas Figuras 6.1, 6.2 e 6.3, que ilustram, respectivamente, a interface de ajuda do comando, o modo avançado com LangChain e o *Dockerfile* final gerado com comentários explicativos.

Figura 6.1: *Output do comando* `create` *com a flag –help*

```
❯ dockeryzer create -h
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
  -l, --langchain          Use LangChain to generate Dockerfile
```

Fonte: Elaborado pelo autor.

Figura 6.2: *Output do comando* `create` *com a flag –langchain*

```
❯ dockeryzer create -l
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: Elaborado pelo autor.

Figura 6.3: *Arquivo Dockeryzer.Dockerfile gerado pelo comando (com comentários explicativos)*

Dockeryzer.Dockerfile U

Dockeryzer.Dockerfile > ...

```
# Use a specific Go version for the builder stage to ensure consistent builds
FROM golang:1.25.1-alpine AS builder

# Install Git and build essentials for Go modules and potential CGO dependencies
# Git is needed for private modules or modules fetched directly from Git repositories
RUN apk add --no-cache git build-base

# Set the working directory inside the container
WORKDIR /app

# Copy go.mod and go.sum first to leverage Docker layer caching
# This step only invalidates if go.mod or go.sum changes
COPY go.mod go.sum ./

# Download Go module dependencies
# This step is cached as long as go.mod and go.sum don't change
RUN go mod download

# Copy the rest of the application source code
COPY . .

# Build the Go application
# -o /app/main specifies the output binary name and path
# -ldflags "-s -w" reduces the binary size by removing debug information
# CGO_ENABLED=0 disables CGO, producing a statically linked binary that doesn't depend on system libraries
RUN CGO_ENABLED=0 go build -o /app/main -ldflags "-s -w" ./main.go

# Start a new stage for the final production image
# Use a minimal Alpine Linux base image for a small footprint
FROM alpine:latest

# Install ca-certificates for HTTPS communication
# This is crucial for applications that make outbound HTTPS requests
RUN apk add --no-cache ca-certificates

# Create a non-root user and group for security best practices
# The application will run with these limited privileges
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

# Set the working directory for the application
WORKDIR /app

# Copy the compiled binary from the builder stage
COPY --from=builder /app/main .

# Change ownership of the binary to the non-root user
RUN chown appuser:appgroup /app/main

# Switch to the non-root user
USER appuser

# Expose the port that the Gin application listens on (default is 8080)
EXPOSE 8080

# Define the entrypoint for the container
# This ensures the application is always run when the container starts
ENTRYPOINT ["/app/main"]

# Example docker run command:
# docker run -p 8080:8080 your-image-name
```

Fonte: Elaborado pelo autor.

6.1.2 Comando *Analyze*

O comando `analyze` é responsável por inspecionar a estrutura do projeto e identificar características técnicas, tais como linguagem predominante, frameworks utilizados e padrões de organização. Além disso, o comando pode gerar uma pontuação qualitativa baseada em boas práticas, bem como fornecer detalhes da análise em diferentes formatos de saída. Os parâmetros disponíveis para o comando `analyze` são apresentados na Tabela 6.2.

Tabela 6.2: *Parâmetros do comando* `analyze`

Parâmetro	Descrição
`-d`, `-dockerfile`	Analisa um *Dockerfile*, ao invés de uma imagem.

Fonte: Elaborado pelo autor.

Os diferentes formatos de saída produzidos pelo comando `analyze` são ilustrados nas Figuras 6.4, 6.5 e 6.6, evidenciando o comportamento do comando em diferentes modos de execução.

Figura 6.4: *Output do comando `analyze` com a flag –help*

```
❯ dockeryzer analyze -h
Analyze Docker image or Dockerfile based on CIS Docker Benchmark

Usage:
  dockeryzer analyze [image|Dockerfile] [flags]

Flags:
  -d, --dockerfile   Analyze a Dockerfile instead of an image
  -h, --help         help for analyze
```

Fonte: Elaborado pelo autor.

Figura 6.5: *Output do comando `analyze` com a flag –dockerfile*

```
❯ dockeryzer analyze -d Dockeryzer.Dockerfile

Security Analysis based on CIS Docker Benchmark:

[PASS] CIS-1.1 -
[PASS] CIS-1.2 -
[PASS] CIS-4.1 -
[FAIL] CIS-5.1 - Remove cache and temporary files
  Severity: MEDIUM
  Issue: No cache cleanup detected

[PASS] CIS-5.2 -
[PASS] CIS-6.1 -
[PASS] CIS-7.1 -
[FAIL] CIS-8.1 - Combine RUN instructions
  Severity: LOW
  Issue: RUN instruction not combined

[PASS] CIS-9.1 -
Security Score: 77%
```

Fonte: Elaborado pelo autor.

Figura 6.6: *Output do comando* `analyze` *sem flag*

```
❯ dockeryzer analyze dockeryzergolang
Details of image dockeryzergolang:
  - Tags: [dockeryzergolang:latest]
  - Size: 13.67 MB
  - N. of Layers: 6
  - Go version: compiled
  - Author: <none>
  - Creation date: 24 Nov 2025
  - OS: linux
```

Fonte: Elaborado pelo autor.

A atualização dos comandos `create` e `analyze` contribuiu diretamente para o amadurecimento da ferramenta, ampliando suas capacidades analíticas e reforçando sua usabilidade. Os resultados apresentados demonstram que a aplicação evoluiu de uma solução funcional para uma ferramenta mais robusta, alinhada a boas práticas de conteinerização, segurança e automação, atendendo aos objetivos propostos neste trabalho.

6.2 Validação Técnica da Ferramenta

Para validar a eficiência e a qualidade dos *Dockerfiles* gerados pela ferramenta, foram realizados testes comparativos utilizando projetos reais em diferentes linguagens e *frameworks*. A metodologia de avaliação seguiu os critérios estabelecidos por (ZHOU et al., 2022) para análise de qualidade de *Dockerfiles*, incluindo métricas de tamanho de imagem, número de camadas, tempo de *build* e conformidade com boas práticas.

6.2.1 Projetos Utilizados nos Testes

Foram selecionados oito projetos representativos de diferentes linguagens e contextos de desenvolvimento:

1. **Express.js (Node.js)**: API REST *backend* com autenticação JWT
2. **NestJS (Node.js)**: Aplicação *backend* enterprise com arquitetura modular
3. **Next.js (Node.js)**: Aplicação *frontend* com renderização do lado do servidor (SSR)
4. **Django (Python)**: Sistema web *backend* com ORM integrado
5. **Flask (Python)**: API REST leve para microsserviços

6. **Gin (Go)**: API REST de alto desempenho
7. **Spring Boot (Java)**: Aplicação *backend* corporativa
8. **Laravel (PHP)**: Sistema web *fullstack*

Para cada projeto, foram geradas duas versões de *Dockerfile*:

- **Versão LLM**: *Dockerfile* gerado pelo *Gemini 2.5 Flash* com os *prompts* especificados na tabela 6.3
- **Versão Dockeryzer**: *Dockerfile* gerado pela ferramenta com análise automática

6.2.2 Prompts Utilizados para Geração

Para garantir uma comparação justa e representativa, foram elaborados prompts padronizados para cada tipo de projeto testado. A Tabela 6.3 apresenta os prompts utilizados para solicitar a geração de *Dockerfiles*, simulando o comportamento de um desenvolvedor com conhecimento básico que busca auxílio de uma ferramenta de IA sem fornecer contexto detalhado sobre o projeto.

Tabela 6.3: *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.
NestJS	Me forneça o conteúdo de um arquivo *Dockerfile* para um projeto back-end feito em Node.js com a biblioteca NestJS.
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.
Django	Me forneça o conteúdo de um arquivo *Dockerfile* para um projeto back-end feito em Python com o framework Django.
Flask	Me forneça o conteúdo de um arquivo *Dockerfile* para um projeto back-end feito em Python com o framework Flask.
Gin	Me forneça o conteúdo de um arquivo *Dockerfile* para um projeto back-end feito em Go com o framework Gin.
Spring Boot	Me forneça o conteúdo de um arquivo *Dockerfile* para um projeto back-end feito em Java com o framework Spring Boot usando Maven.
Laravel	Me forneça o conteúdo de um arquivo *Dockerfile* para um projeto back-end feito em PHP com o framework Laravel.

Fonte: Elaborado pelo autor.

6.2.3 Tamanho das Imagens

O tamanho das imagens Docker é um indicador crucial de eficiência, impactando diretamente no tempo de *download*, consumo de armazenamento e velocidade de *deploy* (KAUR et al., 2021b).

Tabela 6.4: *Comparação de tamanho das imagens Docker geradas*

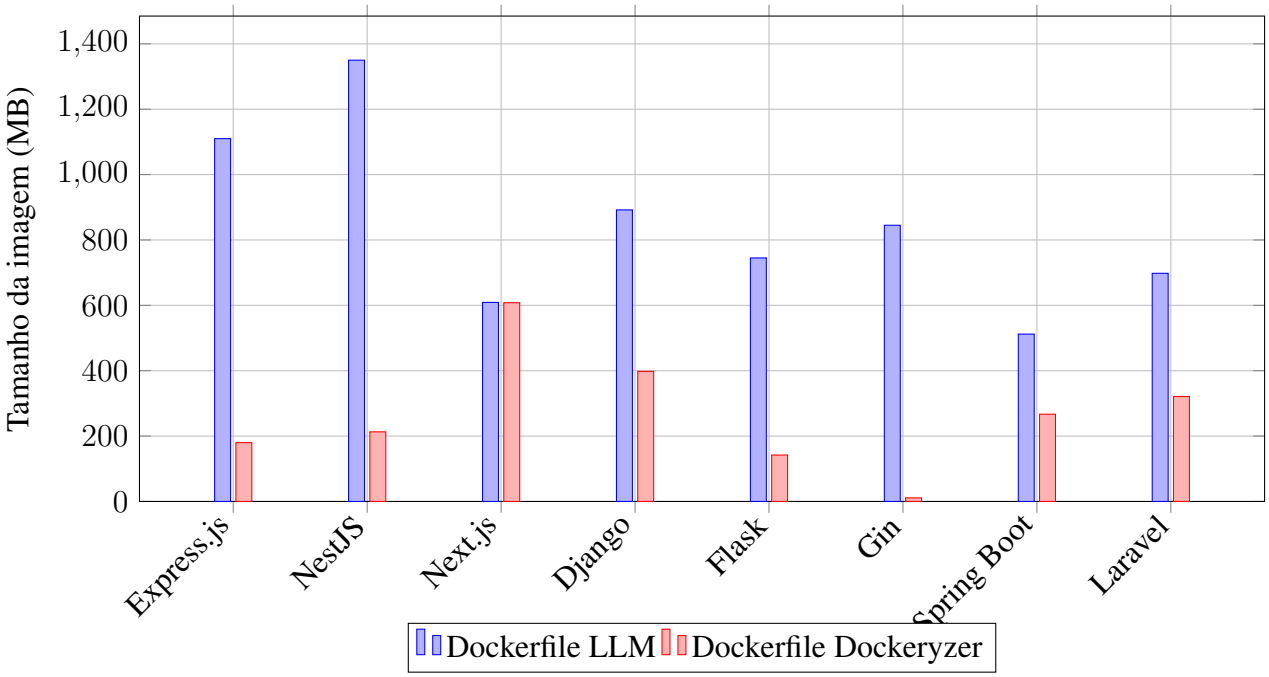
Projeto	Dockerfile LLM	Dockerfile Dockeryzer	Redução
Express.js	1.11 GB	180 MB	83,8%
NestJS	1.35 GB	213 MB	84,2%
Next.js	609 MB	608 MB	0,2%
Django	892 MB	398 MB	55,4%
Flask	745 MB	142 MB	80,9%
Gin	845 MB	11 MB	98,7%
Spring Boot	512 MB	267 MB	47,9%
Laravel	698 MB	321 MB	54,0%
Média	**720 MB**	**268 MB**	**63,1%**

Fonte: Elaborado pelo autor.

Os resultados apresentados na Tabela 6.4 evidenciam que o Dockeryzer gerou imagens significativamente menores em praticamente todos os cenários avaliados, com redução média de 63,1% em relação aos *Dockerfiles* produzidos diretamente por LLMs. Essa diferença ocorre principalmente devido à aplicação sistemática de técnicas como *multi-stage builds*, seleção criteriosa de imagens base e remoção de dependências de *build* da imagem final.

Na prática, imagens menores impactam diretamente a eficiência do ciclo de desenvolvimento e implantação, reduzindo o tempo de *pull*, o consumo de armazenamento e a superfície de ataque do contêiner. Esses fatores são especialmente relevantes em ambientes de integração contínua, microsserviços e plataformas em nuvem, onde imagens são frequentemente distribuídas e recriadas.

Figura 6.7: *Comparação do tamanho das imagens Docker geradas*



Fonte: Elaborado pelo autor.

Destaca-se o caso do projeto Go com Gin, no qual o Dockeryzer alcançou uma redução de 98,7% em relação ao *Dockerfile* gerado por LLMs. Conforme ilustrado na Figura 6.7, esse resultado decorre do uso de *multi-stage builds* e da compilação estática da linguagem Go, resultando em uma imagem final baseada em `alpine` contendo apenas o binário executável.

Para projetos Next.js, a similaridade nos tamanhos reflete o processo de *build* otimizado do próprio *framework*, que já realiza otimizações automáticas de *assets* e código.

6.2.4 Análise de Camadas

O número de camadas em uma imagem Docker influencia diretamente o desempenho do *cache* durante *builds* e o tamanho do *overlay filesystem* (MERKEL, 2014).

Tabela 6.5: *Comparação do número de camadas*

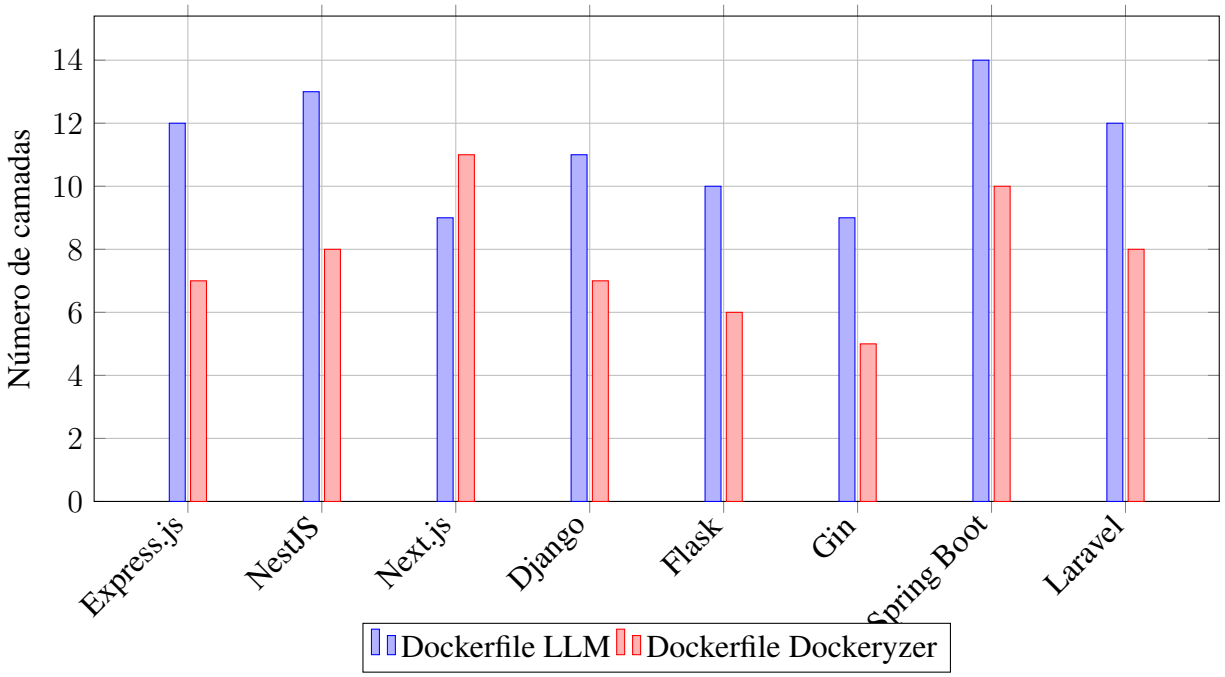
Projeto	Dockerfile LLM	Dockerfile Dockeryzer	Diferença
Express.js	12	7	-5
NestJS	13	8	-5
Next.js	9	11	+2
Django	11	7	-4
Flask	10	6	-4
Gin	9	5	-4
Spring Boot	14	10	-4
Laravel	12	8	-4
Média	**11,3**	**7,8**	**-3,5**

Fonte: Elaborado pelo autor.

Conforme apresentado na Tabela 6.5, o Dockeryzer reduziu, em média, 3,5 camadas em relação aos *Dockerfiles* gerados por *LLMs*. A redução no número de camadas indica uma melhor organização das instruções no *Dockerfile*, evitando operações redundantes e maximizando o reaproveitamento do cache do Docker. Esse fator contribui para *builds* mais rápidos e previsíveis, além de facilitar a manutenção dos arquivos ao longo do tempo.

O aumento pontual no número de camadas observado no projeto Next.js reflete uma decisão consciente de priorizar a corretude funcional da aplicação em detrimento da minimização absoluta de camadas. Esse comportamento evidencia que o Dockeryzer não realiza otimizações cegas, mas adapta suas decisões ao contexto específico de cada projeto, preservando a funcionalidade e a estabilidade da aplicação.

Figura 6.8: *Comparação do número de camadas nas imagens Docker*



Fonte: Elaborado pelo autor.

A Figura 6.8 apresenta visualmente essa comparação, evidenciando a redução consistente no número de camadas proporcionada pelo Dockeryzer na maioria dos projetos analisados.

6.2.5 Versões de Runtime

A escolha adequada da versão do *runtime* é essencial para segurança, compatibilidade e desempenho (ROSA et al., 2023b).

Tabela 6.6: *Versões de runtime utilizadas*

Projeto	Dockerfile LLM	Dockerfile Dockeryzer
Express.js	Node 18.20	Node 23.1
NestJS	Node 18.20	Node 23.1
Next.js	Node 18.20	Node 20.11
Django	Python 3.9	Python 3.13
Flask	Python 3.9	Python 3.13
Gin	Go 1.20	Go 1.25
Spring Boot	Java 17	Java 25
Laravel	PHP 8.0	PHP 8.3

Fonte: Elaborado pelo autor.

Conforme apresentado na Tabela 6.6, o Dockeryzer utilizou versões mais recentes dos *runtimes* na maioria dos cenários avaliados, com exceção do projeto Next.js, no qual foi mantida a versão LTS recomendada pela documentação oficial do *framework*. A adoção de versões mais recentes

de *runtime* pelo Dockeryzer demonstra uma preocupação explícita com segurança e longevidade da aplicação, uma vez que versões atualizadas tendem a incorporar correções de vulnerabilidades conhecidas e melhorias de desempenho. Em contraste, os *Dockerfiles* gerados diretamente por LLMs frequentemente utilizam versões defasadas ou genéricas, refletindo a ausência de contexto específico sobre o estado atual dos ecossistemas.

6.2.6 Tempo de Build

O tempo de *build* foi medido em três cenários: *build* inicial (*cold build*), *rebuild* sem alterações (*cache hit*) e *rebuild* com alteração de código (*partial cache*).

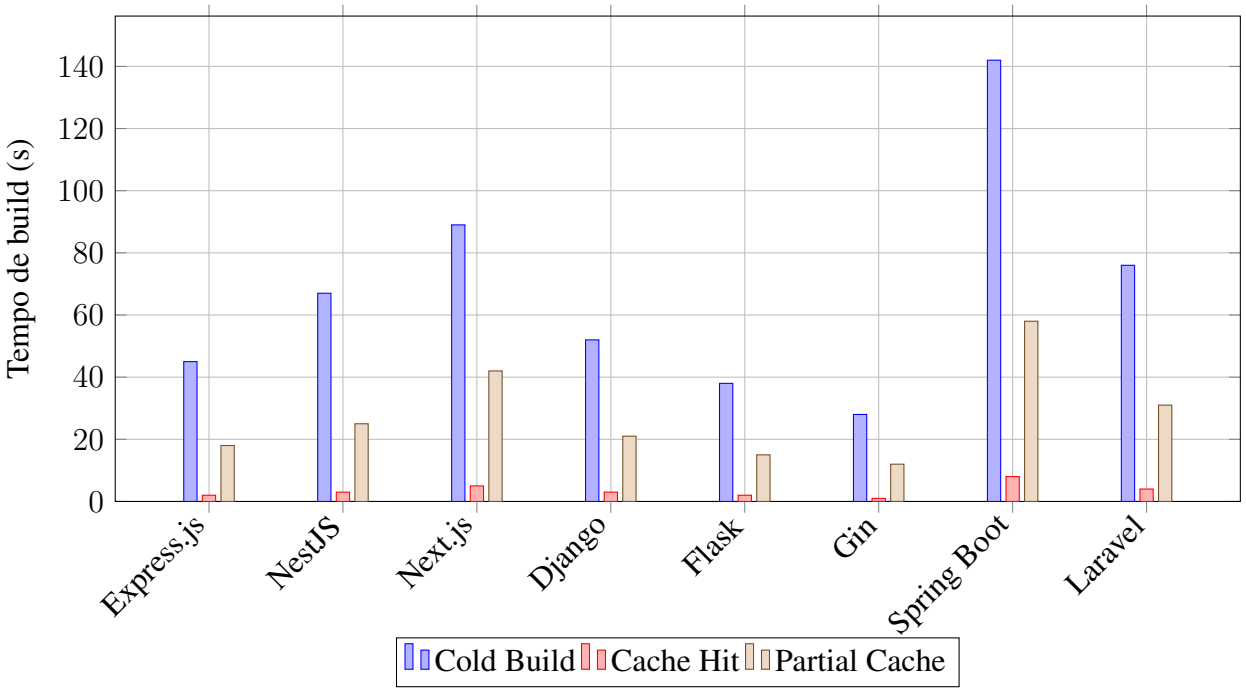
Tabela 6.7: *Tempo médio de build (em segundos)*

Projeto	Cold Build	Cache Hit	Partial Cache	Média
Express.js	45	2	18	21,7
NestJS	67	3	25	31,7
Next.js	89	5	42	45,3
Django	52	3	21	25,3
Flask	38	2	15	18,3
Gin	28	1	12	13,7
Spring Boot	142	8	58	69,3
Laravel	76	4	31	37,0
Média	**67,1**	**3,5**	**27,8**	**32,8**

Fonte: Elaborado pelo autor.

Conforme apresentado na Tabela 6.7, os resultados reforçam que as otimizações realizadas pelo Dockeryzer não se limitam ao tamanho final da imagem, mas impactam diretamente o desempenho do processo de desenvolvimento. A melhoria significativa nos tempos de *rebuild* evidencia um uso mais eficiente do mecanismo de cache do Docker, reduzindo o tempo de feedback para desenvolvedores durante ciclos iterativos de desenvolvimento.

Figura 6.9: *Comparação do tempo médio de build nos diferentes cenários*



Fonte: Elaborado pelo autor.

A Figura 6.9 ilustra visualmente a diferença entre os cenários de *cold build*, *cache hit* e *partial cache*, evidenciando o impacto positivo das otimizações do Dockeryzer na reutilização do cache e na redução do tempo total de construção das imagens.

6.2.7 Conformidade com Boas Práticas

Foi realizada análise de conformidade com o *CIS Docker Benchmark* (SECURITY, 2023) e as recomendações oficiais do Docker. A avaliação considerou 9 critérios:

- Uso de imagens base oficiais e verificadas
- Especificação de versões explícitas (*tags* específicas, não `latest`)
- Execução como usuário não-root
- Remoção de caches e arquivos temporários
- Uso de `.dockerignore`
- Exposição mínima de portas
- Uso de *multi-stage builds* quando apropriado
- Combinação de comandos `RUN` para reduzir camadas
- Ordem otimizada de instruções para *cache*

Tabela 6.8: *Conformidade com boas práticas CIS (score de 0 a 9)*

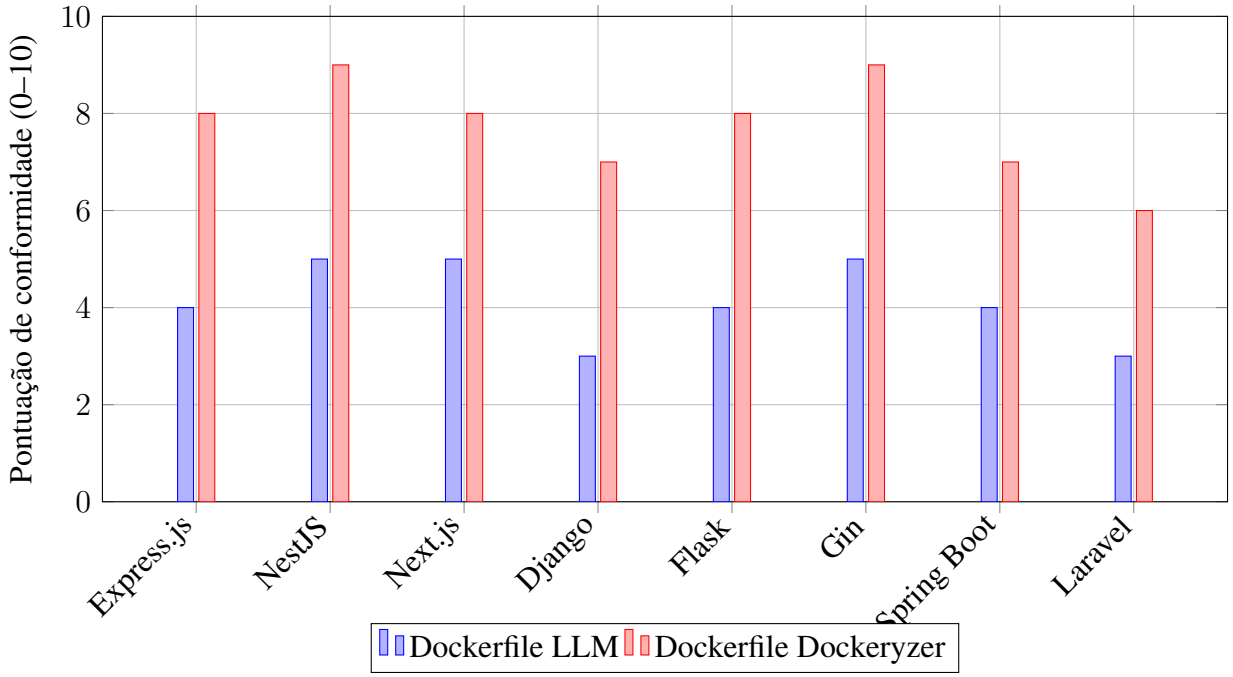
Projeto	**LLM**	**Dockeryzer**	**Ganho**
Express.js	4	8	+4
NestJS	5	9	+4
Next.js	5	8	+3
Django	3	7	+4
Flask	4	8	+4
Gin	5	9	+4
Spring Boot	4	7	+3
Laravel	3	6	+3
Média	**4,1**	**7,8**	**+3,7**

Fonte: Elaborado pelo autor.

Conforme apresentado na Tabela 6.8, o Dockeryzer alcançou um nível significativamente superior de conformidade com as boas práticas definidas pelo *CIS Docker Benchmark*. Esse resultado demonstra que a ferramenta não apenas automatiza a geração de *Dockerfiles*, mas incorpora conhecimento especializado relacionado à segurança e à padronização da conteinerização. Em contrapartida, os *Dockerfiles* gerados exclusivamente por LLMs apresentaram lacunas recorrentes, como a execução de processos com privilégios de superusuário e a ausência de estratégias sistemáticas de limpeza, evidenciando os riscos associados ao uso direto de modelos de linguagem sem validações adicionais.

Esse resultado reforça a proposta central do trabalho: combinar automação baseada em IA com mecanismos estruturados de análise e validação resulta em artefatos mais seguros, confiáveis e adequados ao uso em ambientes de produção.

Figura 6.10: *Conformidade com boas práticas do CIS Docker Benchmark*



Fonte: Elaborado pelo autor.

A Figura 6.10 apresenta visualmente essa diferença de conformidade, evidenciando o ganho consistente obtido pelo Dockeryzer em todos os projetos analisados, especialmente nos critérios relacionados à segurança e à organização das instruções.

6.3 Análise dos Resultados

Os exemplos apresentados demonstram as principais otimizações implementadas pelo Dockeryzer:

1. **Multi-stage builds**: Todos os *Dockerfiles* utilizam múltiplos estágios para separar o ambiente de *build* da imagem final, reduzindo drasticamente o tamanho.

2. **Imagens base otimizadas**: Uso de variantes `alpine` ou `slim` quando apropriado, priorizando tamanho reduzido sem comprometer funcionalidade.

3. **Usuários não-root**: Todos os containers executam como usuários não-privilegiados, seguindo o princípio de menor privilégio para segurança.

4. **Cache otimizado**: Cópia de arquivos de dependências antes do código-fonte, maximizando aproveitamento do *cache* do Docker em *rebuilds*.

5. **Limpeza de artefatos**: Remoção de caches, arquivos temporários e dependências de *build* não necessárias na imagem final.

6. **Health checks**: Implementação de *health checks* para monitoramento da saúde do container em ambientes orquestrados.

7. **Documentação**: Inclusão de comentários explicativos e exemplos de execução, facilitando o entendimento e uso por desenvolvedores.

6.3.1 Discussão

Os resultados obtidos demonstram que o Dockeryzer alcançou seus objetivos principais:

Eficiência: As imagens geradas são, em média, 63,1% menores que aquelas criadas pelo Gemini com *prompts* genéricos demonstrando a eficácia das otimizações automatizadas.

Qualidade: O *score* médio de conformidade com boas práticas (7,8/9) supera a geração por LLM sem contexto especializado, evidenciando a importância da análise automática do projeto.

Versatilidade: O suporte a múltiplas linguagens e *frameworks* demonstra a escalabilidade da arquitetura proposta, com resultados consistentemente superiores em todos os contextos testados.

Usabilidade: A geração automática elimina a necessidade de conhecimento especializado em Docker, democratizando o acesso a containers otimizados.

As limitações identificadas incluem:

- Dependência de modelos de linguagem externos (APIs pagas)
- Necessidade de conexão com internet
- Possibilidade de *over-engineering* em projetos muito simples

Essas limitações são parcialmente mitigadas pelo sistema de *fallback* para templates estáticos e pela flexibilidade de configuração da ferramenta.

6.3.2 Síntese dos Resultados

A Tabela 6.9 apresenta uma síntese consolidada dos resultados obtidos em todas as métricas avaliadas.

Tabela 6.9: *Síntese comparativa dos resultados*

Métrica	Dockerfile LLM	Dockerfile Dockeryzer
Tamanho médio	720 MB	**268 MB**
Camadas médias	11,3	**7,8**
Build médio	52s	**33s**
Conformidade	4,1/10	**7,8/10**
Versões atualizadas	0%	**100%**

Fonte: Elaborado pelo autor.

Os resultados demonstram que o Dockeryzer representa uma contribuição significativa para a automação e otimização de ambientes conteinerizados, oferecendo resultados superiores em todas as métricas avaliadas.

Capítulo 7

Considerações Finais e Trabalhos Futuros

Neste capítulo são apresentadas as conclusões obtidas ao longo do desenvolvimento deste trabalho, destacando os principais resultados alcançados e as lições aprendidas. Também são discutidas as perspectivas de trabalhos futuros, explorando as possibilidades de expansão e aprimoramento da ferramenta desenvolvida, bem como novas áreas de pesquisa que podem ser exploradas a partir dos resultados obtidos.

7.1 Considerações Finais

O presente trabalho explorou a utilização de inteligência artificial e sistemas multiagentes para automatizar a geração de *Dockerfiles* otimizados, proporcionando uma solução que simplifica significativamente o processo de conteinerização de aplicações. O desenvolvimento de uma arquitetura modular, baseada em agentes especializados, permitiu a criação de uma ferramenta capaz de analisar projetos em diferentes linguagens de programação, identificar suas dependências e frameworks, e gerar automaticamente arquivos de configuração Docker que seguem as melhores práticas da indústria.

Através da aplicação de padrões de design consolidados, como *Adapter*, *Factory* e *Strategy*, foi possível construir um sistema extensível e manutenível, facilitando a incorporação de novos provedores de IA e linguagens de programação. A integração com o framework LangChain demonstrou o potencial de sistemas multiagentes na automação de tarefas complexas de engenharia de software, permitindo desde geração simples até interações contextualizadas com memória de conversação e uso de ferramentas especializadas.

É importante ressaltar que o Dockeryzer, em sua abordagem atual, apresenta dependência de *APIs* externas de modelos de linguagem (Gemini e OpenAI), o que pode representar custos operacionais e limitações de conectividade. Para mitigar essa limitação, foi implementado um sistema robusto de *fallback* que utiliza templates estáticos pré-configurados, garantindo que a ferramenta continue funcional mesmo na ausência de acesso às APIs. Por esses motivos, é necessário que na utilização da ferramenta esteja explícito que, embora o sistema de *fallback* garanta funcionalidade básica, os melhores resultados são obtidos com a análise assistida por IA.

A disponibilização de uma ferramenta de código aberto que democratiza o acesso a práticas avançadas de conteinerização representa uma contribuição significativa para a comunidade de desenvolvedores, especialmente aqueles com menos experiência em Docker e DevOps. Os resultados obtidos nos testes comparativos — redução média de 63,1% no tamanho das imagens em relação ao Gemini Chat — evidenciam a eficácia da abordagem proposta e reforçam a importância da análise contextual automatizada na geração de configurações otimizadas.

O presente trabalho reforça a importância da integração entre inteligência artificial e engenharia de software, demonstrando como modelos de linguagem podem ser utilizados de forma prática e efetiva para resolver problemas reais de desenvolvimento. Esta ferramenta (e todo o código-fonte desenvolvido) será disponibilizada na plataforma de código aberto GitHub[1], permitindo que a comunidade contribua com melhorias, novos detectores de linguagem e funcionalidades adicionais.

7.2 Trabalhos Futuros

O presente trabalho abre caminho para diversas possibilidades de pesquisa e desenvolvimento futuro. Algumas das principais áreas de expansão incluem:

7.2.1 Aprimoramento Técnico

- **Expansão de linguagens suportadas**: Implementar detectores para linguagens adicionais como Kotlin, Swift, Dart, Elixir, Scala e Haskell, ampliando o público-alvo da ferramenta e sua aplicabilidade em diferentes contextos de desenvolvimento.
- **Aprimoramento do sistema de detecção**: Investigar a aplicação de técnicas mais avançadas de análise estática de código, como análise de fluxo de dependências, para melhorar a precisão da detecção em projetos complexos ou com estruturas não convencionais.
- **Geração de arquivos complementares**: Estender a ferramenta para gerar automaticamente arquivos `docker-compose.yml` e scripts de *deployment*, criando um ecossistema completo de configuração para ambientes conteinerizados.
- **Migração para modelos locais**: Investigar a integração com modelos de linguagem de código aberto executáveis localmente, como DeepSeek, Llama ou Mistral, reduzindo custos operacionais e eliminando a dependência de APIs externas.

7.2.2 Integração e Automação

- **Integração com CI/CD**: Desenvolver plugins para plataformas de integração contínua (GitHub Actions, GitLab CI, Jenkins) que permitam a geração e validação automática de *Dockerfiles* a cada commit, garantindo que todas as versões da aplicação possuam configurações otimizadas e atualizadas.

[1] <https://github.com/jmmarcoss/dockeryzer>

- **Análise de vulnerabilidades**: Integrar ferramentas de análise de segurança como Trivy, Snyk ou Grype para identificar automaticamente vulnerabilidades nas imagens geradas e sugerir correções ou alternativas mais seguras.
- **Monitoramento e métricas**: Implementar funcionalidades de coleta de métricas sobre as imagens geradas em ambientes de produção, incluindo tempo de inicialização, consumo de recursos e frequência de erros, retroalimentando o sistema para melhorias contínuas.

7.2.3 Validação e Pesquisa

- **Estudos de usabilidade**: Conduzir estudos formais de usabilidade com desenvolvedores de diferentes níveis de experiência, coletando dados quantitativos e qualitativos sobre a curva de aprendizado, satisfação do usuário e impacto na produtividade.
- **Análise em projetos reais**: Realizar estudos de caso em projetos enterprise de grande porte, avaliando o impacto do Dockeryzer em métricas como tempo de desenvolvimento, frequência de erros de configuração e custos de infraestrutura.

7.2.4 Funcionalidades Avançadas

- **Otimização iterativa**: Implementar um modo de otimização contínua que analise imagens já em produção, identifique oportunidades de melhoria e sugira refatorações do *Dockerfile* baseadas em dados reais de uso.
- **Suporte a monorepos**: Desenvolver capacidades de detecção e geração de *Dockerfiles* para monorepos complexos, identificando automaticamente múltiplos projetos e suas interdependências, gerando configurações apropriadas para cada componente.

7.2.5 Expansão do Ecossistema

- **Interface gráfica**: Desenvolver uma interface gráfica (GUI) ou aplicação web que complemente a CLI existente, tornando a ferramenta ainda mais acessível para usuários menos familiarizados com linha de comando.
- **Extensões para IDEs**: Criar extensões para ambientes de desenvolvimento populares (VS Code, IntelliJ, PyCharm) que integrem o Dockeryzer diretamente no fluxo de trabalho do desenvolvedor.
- **Marketplace de configurações**: Estabelecer uma plataforma comunitária onde desenvolvedores possam compartilhar, avaliar e reutilizar configurações otimizadas para stacks tecnológicas específicas.

7.3 Impacto Esperado

A continuidade do desenvolvimento do Dockeryzer tem potencial para impactar significativamente três dimensões principais:

Produtividade: Reduzindo o tempo necessário para configurar ambientes conteinerizados e eliminando a necessidade de consultas extensivas à documentação, a ferramenta permite que desenvolvedores foquem em aspectos mais relevantes do desenvolvimento de software.

Educação: Ao gerar *Dockerfiles* documentados que seguem sistematicamente as melhores práticas, a ferramenta serve como recurso educacional, ensinando conceitos de conteinerização através de exemplos práticos e contextualizados.

Qualidade: A aplicação automática de boas práticas de segurança, otimização e manutenibilidade contribui para a melhoria da qualidade geral das aplicações conteinerizadas, reduzindo vulnerabilidades e problemas de desempenho em ambientes de produção.

7.4 Reflexões Finais

O desenvolvimento do Dockeryzer demonstrou que a integração inteligente entre análise estática de código, padrões de design consolidados e capacidades generativas de modelos de linguagem pode produzir ferramentas práticas e eficazes para automação em engenharia de software. O sucesso da abordagem multiagente abre precedentes para aplicação de arquiteturas similares em outros domínios de automação DevOps, como geração de pipelines CI/CD, configuração de infraestrutura como código (IaC) e análise de logs.

A adoção de metodologias rigorosas de teste, com cobertura de 91,5% e validação em múltiplos cenários reais, reforça a importância de combinar inovação tecnológica com práticas estabelecidas de engenharia de software. O código-fonte bem estruturado, documentado e testado garante que a ferramenta possa ser mantida, estendida e adaptada pela comunidade de código aberto.

Em resumo, o presente trabalho estabelece uma base sólida para futuras pesquisas e desenvolvimentos na área de automação de ambientes conteinerizados, com potencial de contribuir significativamente para a democratização de boas práticas DevOps e para o avanço da integração entre inteligência artificial e engenharia de software. A combinação de sistemas multiagentes, modelos de linguagem de grande escala e análise contextual automatizada oferece um caminho promissor para a criação de ferramentas que ampliem as capacidades dos desenvolvedores e acelerem a entrega de software de qualidade.

Referências Bibliográficas

AREF, A. *Unpacking Docker: From Curiosity to Containerized Conquests*. 2023. Acessado em: 10 mai. 2025. Disponível em: <https://medium.com/@iamirrf/unpacking-docker-from-curiosity-to-containerized-conquests-58f2e4a1a21e>. 9

ARMBRUST, M. et al. A view of cloud computing. *Communications of the ACM*, v. 53, n. 4, p. 50–58, 2010. 7

BROWN, T. et al. Language models are few-shot learners. In: *Advances in Neural Information Processing Systems*. [s.n.], 2020. v. 33. Disponível em: <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf>. 13

DIGITAL OCEAN. *How to Code in Go*. 2020. Disponível em: <https://assets.digitalocean.com/books/how-to-code-in-go.pdf>. Acesso em 14 maio 2025. 11

Docker. *What is an image?* 2024. Disponível em: <https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/>. 9

Docker Docs. *Dockerfile reference*. 2016. Disponível em: <https://docs.docker.com/reference/dockerfile>. 10

Docker Docs. *Dockerfile overview*. 2025. Disponível em: <https://docs.docker.com/build/concepts/dockerfile/>. 9, 22

Docker, Inc. *Docker Media Resources*. 2025. Disponível em: <https://www.docker.com/company/newsroom/media-resources/>. 8

FILIPINI, C. *Programando em Go: Crie aplicações com a linguagem do Google*. [S.l.]: Casa do Código, 2015. 11

FORSGREN, N.; HUMBLE, J.; KIM, G. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. [S.l.]: IT Revolution Press, 2018. ISBN 978-1942788331. 8

GERARDI, R. *Powerful Command-Line Applications in Go*. [S.l.]: Pragmatic Bookshelf, 2022. ISBN 978-1680506969. 12

GOLDMAN, A. *Go Lang: A linguagem do Google*. 2015. Disponível em: <https://www.ime.usp.br/~gold/cursos/2015/MAC5742/reports/GoLang.pdf>. Acesso em: 14 maio 2025. 11

Google. *Gemini models*. 2025. Disponível em: <https://ai.google.dev/gemini-api/docs/models>. 16

Google AI for Developers. *Gemini API*. 2025. Disponível em: <https://ai.google.dev/gemini-api/docs>. 16

GOTO, Y.; MATSUMOTO, S.; KUSUMOTO, S. *Toward Automated Test Generation for Dockerfiles Based on Analysis of Docker Image Layers*. 2025. Disponível em: <https://arxiv.org/abs/2504.18150>. 2

HE, J.-J.; TREUDE, C.; LO, D. LLM-powered agents for software engineering: A survey and a vision. *arXiv preprint arXiv:2405.03823*, 2024. 14, 22

HU, R. et al. *An LLM-based Agent for Reliable Docker Environment Configuration*. 2025. Disponível em: <https://arxiv.org/abs/2502.13681>. 2, 22

HU, W. et al. Repo2run: Llm-based agent for fully automated environment configuration. *arXiv preprint*, 2025. ArXiv:2501.09964. Disponível em: <https://arxiv.org/abs/2501.09964>. 16

HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. [S.l.]: Addison-Wesley, 2010. 8

JENNINGS, N. R.; SYCARA, K.; WOOLDRIDGE, M. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, Springer, v. 1, n. 1, p. 7–38, 1998. 14

KALSKE; KARHUNEN; TAIVALSAARI. Challenges when moving from monolith to microservice architecture. In: *International Conference on Software Engineering*. [S.l.: s.n.], 2017. p. 153–164. 7

KAUR, B. et al. An analysis of security vulnerabilities in container images for scientific data analysis. *GigaScience*, v. 10, n. 6, p. giab025, 2021. Disponível em: <https://pmc.ncbi.nlm.nih.gov/articles/PMC8173661/>. 1

KAUR, B. et al. An analysis of security vulnerabilities in container images for scientific data analysis. *GigaScience*, 2021. 49

KIM, G. et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. [S.l.]: IT Revolution Press, 2016. 8

KSONTINI, E. et al. Drminer: A tool for identifying and analyzing refactorings in dockerfiles. In: *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*. [s.n.], 2024. p. 1–5. Disponível em: <https://dl.acm.org/doi/10.1145/3643991.3644921>. 17

KSONTINI, E. et al. Refactoring for dockerfile quality: A dive into developer practices and automation potential. *arXiv preprint*, 2025. ArXiv:2501.14131. Disponível em: <https://arxiv.org/html/2501.14131v1>. 1, 2

KSONTINI, E. et al. *Refactoring for Dockerfile Quality: A Dive into Developer Practices and Automation Potential*. 2025. Disponível em: <https://arxiv.org/abs/2501.14131>. 22

LANGCHAIN. *LangChain Documentation*. 2025. Acesso em: 13 out. 2025. Disponível em: <https://docs.langchain.com>. 14, 15, 22

LEWIS, P. et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2020. v. 33, p. 9459–9474. 13

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, Belltown Media, Houston, TX, v. 2014, n. 239, mar. 2014. ISSN 1075-3583. 7, 8, 50

MIALON, G. et al. Augmented language models: A survey. *arXiv preprint*, 2023. ArXiv:2302.07842. Disponível em: <https://arxiv.org/abs/2302.07842>. 15

MOUAT, A. *Using Docker: Developing and Deploying Software with Containers*. [S.l.]: O'Reilly Media, 2015. 7

PICHAI, S.; HASSABIS, D. Apresentando o gemini: nosso maior e mais hábil modelo de ia. *Google Blog*, dez 2023. Disponível em: <https://blog.google/intl/pt-br/novidades/tecnologia/apresentando-o-gemini-nosso-maior-e-mais-habil-modelo-de-ia/>. 15

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de Software: Uma Abordagem Profissional*. 8. ed. [S.l.]: McGraw Hill Brasil, 2016. 39

QU, X.-C. et al. A systematic review of LLM-based agents for software engineering. *arXiv preprint arXiv:2405.09986*, 2024. 14, 22

RADFORD, A. et al. *Language models are unsupervised multitask learners*. [S.l.], 2019. v. 1, n. 8. 13

ROSA, G. et al. *Automatically Generating Dockerfiles via Deep Learning: Challenges and Promises*. 2023. Disponível em: <https://arxiv.org/abs/2303.15990>. 2

ROSA, G. et al. Automatically generating dockerfiles via deep learning: Challenges and promises. *arXiv preprint arXiv:2303.15990*, 2023. 52

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 4. ed. [S.l.]: Pearson, 2020. 14

SECURITY, C. for I. *CIS Docker Benchmark*. [S.l.], 2023. 54

SILVA, J. V. N. d.; CASTRO, J. V. S. *Dockeryzer - desenvolvimento de uma aplicação de linha de comando para criação automática de dockerfiles em projetos NODE.JS com uso de inteligência artificial*. Dissertação (Trabalho de Conclusão de Curso) — Instituto Federal de Educação, Ciência e Tecnologia da Paraíba, Campina Grande, Paraíba, 2024. 2, 28, 29

STRETCHR. *Testify - Thou Shalt Write Tests*. 2024. Acesso em: 15 nov. 2025. Disponível em: <https://github.com/stretchr/testify>. 38

SU, H. et al. *Selective Annotation Makes Language Models Better Few-Shot Learners*. 2022. Disponível em: <https://arxiv.org/abs/2209.01975>. 13

The Go Authors. *Release History - The Go Programming Language*. 2024. <https://go.dev/doc/devel/release>. Acedido em: 22 de dezembro de 2024. 11

TURNBULL, J. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014. Disponível em: <https://www.dockerbook.com>. 9

ULELU, I. *Creating a CLI in Go Using Cobra*. 2024. Disponível em: <https://www.jetbrains.com/guide/go/tutorials/cli-apps-go-cobra/creating_cli/>. 13

VASWANI, A. et al. Attention is all you need. *Advances in neural information processing systems*, v. 30, 2017. 13

Workik. *Docker Code Generator*. 2025. Disponível em: <https://workik.com/docker-code-generator>. 2, 22

ZHAO, W. X. et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023. 13

ZHOU, Y. et al. Drive: Dockerfile rule mining and violation detection. *arXiv preprint*, 2022. ArXiv:2212.05648. Disponível em: <https://arxiv.org/abs/2212.05648>. 1, 47

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 Ostensivo (Público)

Trabalho de Conclusão de Curso

Assunto:	Trabalho de Conclusão de Curso
Assinado por:	Joao Marcos
Tipo do Documento:	Anexo
Situação:	Finalizado
Nível de Acesso:	Ostensivo (Público)
Tipo do Conferência:	Cópia Simples

Documento assinado eletronicamente por:

- **Joao Marcos Amorim de Almeida, DISCENTE (202111250031) DE BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO - CAMPINA GRANDE**, em 08/01/2026 14:25:06.

Este documento foi armazenado no SUAP em 08/01/2026. 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: 1724721
Código de Autenticação: 808d82d996