Universidade Federal de Mato Grosso – UFMT
Campus Universitário de Várzea Grande
Faculdade de Engenharia
Engenharia de Computação

Luis Octavio Galesso Seror

DESENVOLVIMENTO DE SOFTWARE DE ASSISTENTE PESSOAL INTEGRANDO INTELIGÊNCIAS ARTIFICIAIS

Várzea Grande
2024

Luis Octavio Galesso Seror

DESENVOLVIMENTO DE SOFTWARE DE ASSISTENTE PESSOAL INTEGRANDO INTELIGÊNCIAS ARTIFICIAIS

Trabalho de Conclusão de Curso apresentado à Faculdade de Engenharia como parte dos requisitos para a obtenção do título de Bacharel em Engenharia de Computação.
Orientador: Profa. Dra. Thais Reggina Kempner

Várzea Grande
2024

Dados Internacionais de Catalogação na Fonte.

S486d Seror, Luis Octavio Galesso Seror.
Desenvolvimento de software de assistente virtual integrando inteligências artificiais [recurso eletrônico] / Luis Octavio Galesso Seror Seror. -- Dados eletrônicos (1 arquivo : 75 f., il. color., pdf). -- 2024.

Orientador: Profa. Dra. Thais Reggina Kempner Kempner.
TCC (graduação em Engenharia de Computação) - Universidade Federal de Mato Grosso, Instituto de Engenharia, Várzea Grande, 2024.
Modo de acesso: World Wide Web: https://bdm.ufmt.br.
Inclui bibliografia.

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

UNIVERSIDADE FEDERAL DE MATO GROSSO

ATA DE REUNIÃO

Nome do aluno: LUIS OCTAVIO GALESSO SEROR

Data da defesa: 07/11/2024

Banca Examinadora

Orientador: Profa. Dra. Thais Reggina Kempner

Membro: Prof. Dr. Raoni Florentino da Silva Teixeira

Membro: Prof. Dr. Fillipe Matos de Vasconcelos (FAET/UFMT)

Membro Suplente:

Título da monografia: Desenvolvimento de software de assistente pessoal integrando inteligências artificiais.

Local: Sala virtual

Horário de início: 14h06

Coordenador de Trabalho de Conclusão de Curso: Prof. Dr. Fabrício Barbosa de Carvalho

APRESENTAÇÃO, ESTRUTURA E REDAÇÃO	1. Exatidão, correção gramatical, clareza; linguagem científica adequada, objetiva e estilo direto; uso correto de terminologia;	**9,5**
APRESENTAÇÃO, ESTRUTURA E REDAÇÃO	2. Equilíbrio e estética na disposição e tamanho das partes (introdução, desenvolvimento e conclusão); organização geral;	**9,5**
ESCOLHA DO ASSUNTO	3. Relevância, importância, originalidade na área de atuação e ao nível do autor; revelação de contribuição pessoal/profissional;	**9,5**
INTRODUÇÃO	4. Delimitação do tema; apresentação da motivação, justificativa e importância do assunto escolhido; formulação do problema; apresentação de objetivos (geral e específicos) e hipóteses;	**9,5**
REVISÃO BIBLIOGRÁFICA	5. Referencial bibliográfico suficiente e adequado; quantidade, qualidade e atualidade das fontes utilizadas;	**9,5**
MÉTODO E MATERIAIS	6. Descrição detalhada do método; adequação ao problema da pesquisa e a o atendimento dos objetivos; descrição do campo de observação, amostra, variáveis e instrumentos;	**9,5**
ANÁLISE DOS RESULTADOS	7. Sequência lógica; estruturação dos itens e subitens; clareza na descrição, análise e interpretação dos dados e resultados; apresentação de discussões; equilíbrio entre teoria e prática.	**9,5**
CONCLUSÕES E SUGESTÕES	8. Conclusões relacionadas com as hipóteses e os objetivos; demonstração de capacidade de síntese; apresentação de sugestões, contribuições e possibilidades de pesquisas futuras;	**9,5**
APRESENTAÇÃO DO TRABALHO	9. forma de apresentação; estratégias e recursos audiovisuais para apresentação do trabalho. clareza e objetividade; ênfase nos resultados e contribuições; apresentação dentro do tempo;	**9,5**
APRESENTAÇÃO DO TRABALHO	10. Segurança e domínio dos conteúdos.	**9,5**
OBS.: o item 11 deve ser avaliado apenas pelo docente orientador	11. Participação, interesse e responsabilidade ao logo de todo o período (ano letivo) de orientação.	**9,5**

NOTA: 9,5

Em sessão pública, após exposição de cerca de **24** minutos, o candidato foi arguido oralmente pelos membros da banca tendo como resultado:

( ) Aprovação por unanimidade sem exigências;

(**X**) Aprovação condicionada ao atendimento dos melhoramentos conceituais da fundamentação teórica no prazo fixado pela banca de 10 (dez) dias;

( ) Reprovação.

Na forma regulamentar foi lavrada a presente ata que é assinada pelos membros da banca e pelo aluno.

Cuiabá-MT, 07 de novembro de 2024.

Documento assinado eletronicamente por **THAIS REGGINA KEMPNER**, **Docente da Universidade Federal de Mato Grosso**, em 07/11/2024, às 17:12, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **FILLIPE MATOS DE VASCONCELOS**, **Docente da Universidade Federal de Mato Grosso**, em 07/11/2024, às 17:34, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **RAONI FLORENTINO DA SILVA TEIXEIRA**, **Docente da Universidade Federal de Mato Grosso**, em 07/11/2024, às 17:54, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **LUIS OCTAVIO GALESSO SEROR**, **Usuário Externo**, em 07/11/2024, às 19:56, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

A autenticidade deste documento pode ser conferida no site http://sei.ufmt.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **7318114** e o código CRC **AA0131CA**.

Dedico este trabalho à minha família, que sempre me apoiou, e, sobretudo, à minha mãe, por sempre acreditar nos meus sonhos.

Agradecimentos

À minha mãe, Maria Paulina Galesso, por sempre acreditar nos meus sonhos e me amar acima de tudo. Ao meu pai, Roberto Teixeira Seror, pelo amor e por todo o apoio em todos os sentidos, e ao meu irmão, Kaiky, por seu amor incondicional. E à minha família, por nunca me deixar desistir.

Um grande agradecimento à professora Thais Kempner, que, além de ser minha orientadora, me ajudou a vir para a Alemanha e tem me apoiado na realização dos meus sonhos. Obrigado por tudo e por acreditar em mim. Ao professor Bruno, que também me auxiliou a vir para a Alemanha e foi meu orientador de estágio. Ao professor Fabrício Carvalho, por me ajudar desde o início do curso de Engenharia de Computação e por nunca desistir de mim — obrigado por ter lutado por mim. À professora Gracyeli, pelo suporte com questões burocráticas da UFMT e pela ajuda com as horas complementares e de extensão. Ao professor Anísio, por ser um ótimo professor, e ao professor Eduardo Theodoro, do IC, que muito me ajudou com a última matéria do curso.

Aos meus grandes amigos, que a UFMT me presenteou e que levarei para a vida: Mateus Reis, Mateus Matos, Gustavo Ale Primo, Flávia Paiva, Fernando Maeda, Caio Valério, Monize Braga e todos os outros amigos que estão comigo desde o início do curso. Vocês estão no meu coração.

Um grande agradecimento aos grandes amigos que o intercâmbio na Alemanha me proporcionou. Obrigado, Pedro Weiss, por me apoiar em tudo, e Mateus Freitas, por me fazer voltar a acreditar em Deus. Vocês dois são a melhor parte da minha estadia aqui na Alemanha. Um agradecimento também aos meus amigos e companheiros de apartamento, Renato Sonohara e Bryan Andrade.

Agradeço imensamente à equipe médica que me auxiliou nos últimos anos: ao meu psicólogo, doutor Júlio Peres, e à minha médica, doutora Tânia Alves. Sem o apoio de vocês, eu ainda estaria em condições médicas muito difíceis.

Resumo

Galesso Seror, Luis Octavio. **DESENVOLVIMENTO DE SOFTWARE DE ASSISTENTE PESSOAL INTEGRANDO INTELIGÊNCIAS ARTIFICIAIS**. 77 p. Trabalho de Conclusão de Curso – Engenharia de Computação, Faculdade de Engenharia, Universidade Federal de Mato Grosso, Brasil, 2024.

Neste trabalho de conclusão de curso, será apresentado o desenvolvimento de um *software* que integra múltiplas tecnologias de inteligência artificial (IA) em uma aplicação única, com ênfase no uso de agentes e ferramentas variadas para otimizar as funcionalidades do sistema. A crescente demanda por soluções inteligentes e automatizadas impulsiona a convergência de diferentes tecnologias de IA em plataformas integradas, potencializando o desempenho e a eficiência do sistema final. Além disso, o projeto segue os princípios de design SOLID, essenciais para garantir escalabilidade, manutenibilidade e organização do código, facilitando futuras expansões.

Para engenheiros de computação, compreender todas as fases do ciclo de vida de desenvolvimento de *software* — desde a análise de requisitos até a escolha da arquitetura e implementação — é fundamental, especialmente ao trabalhar com arquiteturas complexas e ferramentas diversificadas de IA. Este estudo contribui para o avanço do conhecimento no desenvolvimento de software com IA integrada, detalhando processos, desafios e práticas recomendadas nesse campo em rápida evolução. Com o domínio desses conceitos e práticas como o SOLID, os profissionais de engenharia estarão mais preparados para enfrentar desafios complexos e explorar as oportunidades oferecidas pelo desenvolvimento de sistemas de IA integrados e organizados.

Palavras-chave: IA, Software, Agentes, LLMs, SOLID.

Abstract

Galesso Seror, Luis Octavio. **Software development of personal assistant integrating artificial intelligence** . 77 p. Undergraduate Dissertation – Computing Engineering, Engineering Faculty, Federal University of Mato Grosso, Brazil, 2024.

In this thesis, the software development that integrates multiple artificial intelligence (AI) technologies into a single application will be presented, with an emphasis on using agents and various tools to optimize the system's functionalities. The growing demand for intelligent and automated solutions drives the convergence of different AI technologies into integrated platforms, enhancing the performance and efficiency of the final system. Furthermore, the project adheres to the SOLID design principles, which are essential for ensuring scalability, maintainability, and organization of the code, facilitating future expansions.

For computer engineers, understanding all phases of the software development life cycle—ranging from requirements analysis to architecture selection and implementation—is fundamental, especially when working with complex architectures and diverse AI tools. This study contributes to advancing knowledge in the development of software with integrated AI by detailing processes, challenges, and best practices in this rapidly evolving field. With a mastery of these concepts and practices like SOLID, engineering professionals will be better prepared to face complex challenges and explore the opportunities offered by the development of integrated and organized AI systems.

Keywords: AI, Software, agents, LLMs, SOLID.

Lista de ilustrações

Sumário

Capítulo 1

Introdução

Como resposta aos desafios crescentes de hoje na indústria de *software*, surgiu um amplo espectro de novas abordagens de desenvolvimento de *software* (Yang, 2017). Uma direção proeminente é atualmente o paradigma de desenvolvimento de *software* baseado em Inteligência Artificial (IA), que é uma área de pesquisa em constante desenvolvimento de novas metodologias e tecnologias. John McCarthy define IA como:

> a ciência e engenharia da criação de máquinas inteligentes, especialmente programas de computação inteligentes. Está relacionado a tarefa semelhante de usar computadores para entender a inteligência humana, mas a IA não precisa se limitar a métodos de observação biológica (McCarthy, 2007, p.2).

Apesar da IA possuir uma grande variedade de ramos, quatro deles se destacam atualmente: Aprendizado de Máquina ou *Machine Learning* (ML), Processamento de Linguagem Natural ou *Natural Language Processing* (NLP), Visão Computacional ou *computer vision* e Robótica Inteligente. Esses ramos têm inter-relações importantes, com o uso de algoritmos em comum, mais especificamente de ML. No entanto, ML pode ser considerado como um ramo à parte, pois existem diversas outras aplicações que não necessariamente se encaixam nas outras três categorias, como o caso de algoritmos de recomendações do Youtube e Spotify, bem como aplicações para ciência de dados (Meng, 2024; He; Li; Cai, 2024).

Com o surgimento de uma variedade de *softwares* que oferecem uma ampla gama de serviços, a IA está se tornando cada vez mais presente no cotidiano em diversas aplicações, desde aplicativos de mensagens até ferramentas de geração de imagens e áudio. Grandes empresas em todo o mundo estão desenvolvendo aplicativos com IA, como a OpenAI, responsável pelo desenvolvimento do ChatGPT e do DALL-E. A Google também lançou sua *Aplication Interface* (API) de reconhecimento de fala para texto, enquanto a ElevenLabs criou ferramentas para conversão de texto em fala e geradores de voz. Este avanço tecnológico está transformando a maneira como interagimos com a tecnologia e abre portas para uma infinidade de possibilidades inovadoras (Bellavista *et al.*, 2014).

Desta forma, o processo de desenvolvimento de *software* é algo que tem se tornado cada vez mais complexo e com cada vez mais exigências de performance e funcionalidades. Este processo, segundo Pressman (1994), é definido como uma metodologia para as atividades, ações e tarefas necessárias para desenvolver um *software* de alta qualidade.

Um *software* de alta qualidade é aquele que oferece as funcionalidades e o desempenho necessários, além de ser confiável e fácil de usar. Em resumo, para ser considerado de boa qualidade, um *software* deve satisfazer as expectativas dos clientes em termos de funcionalidade e desempenho, ao mesmo tempo em que é confiável e apresenta uma boa usabilidade (Sommerville, 2010).

Para integrar IA no desenvolvimento de *software* há a necessidade de combinar diversas áreas do conhecimento e criar sistemas que além de executar tarefas especificas tenham aprendizado com o tempo. ML é o campo da IA que se concentra no desenvolvimento de algoritmos e técnicas que permitem aos computadores aprenderem a partir de dados, reconhecer padrões e tomar decisões através de modelos preditivos ou descritivos (El Naqa; Murphy, 2015). As aplicações da ML estão em diversas áreas do conhecimento.

É importante destacar que algumas das soluções mencionadas anteriormente, como o ChatGPT, apresentam certas limitações. A versão gratuita do ChatGPT não tem acesso à ferramenta de navegação para informações em tempo real. Somente a versão paga (ChatGPT Plus, utilizando o modelo GPT-4) oferece a opção de usar o navegador para buscar dados atualizados, como eventos, notícias ou conteúdos da internet.

No entanto, mesmo com a versão paga, perguntas como "Onde estou?", "Que horas são?"ou "Qual a temperatura atual?"ainda não podem ser respondidas diretamente, pois o ChatGPT não acessa dados de localização ou sensores de dispositivos, nem monitora o horário ou clima exatos em tempo real. Além disso, ele não conta atualmente com um sistema de reconhecimento de voz, como o *speech to text* do Google. Isso pode dificultar o uso da ferramenta por pessoas cegas. Mesmo na versão paga, ele não dispõe de um sintetizador de voz natural tão avançado quanto os oferecidos pelo Google ou pela ElevenLabs.

Neste contexto, este trabalho propõe desenvolver um *software* que interaja com múltiplas IAs, incorporando funcionalidades como reconhecimento de voz, suporte a múltiplos idiomas, envio de *e-mails*, reprodução de músicas, exibição de informações como data e hora atuais, tudo em uma interface de usuário amigável e intuitiva. O desenvolvimento de um *software* passa por várias etapas, incluindo a análise de requisitos, estudo de viabilidade, *design*, codificação, testes, instalação e *deploy*, além de manutenção, atualizações e suporte. Este trabalho abordará detalhadamente a definição de cada uma dessas etapas, com foco específico no processo de desenvolvimento deste *software*.

Para integrar diversas IAs em um único *software* são necessárias diversas APIs. Neste programa, serão usadas APIs de algumas das maiores aplicações do mundo, com o objetivo de criar um produto funcional e confiável.

Concluindo, a importância de entender os princípios e práticas envolvidos no processo se faz para enfrentar os desafios e aproveitar oportunidades. A capacidade de incorporar IA em aplicativos e sistemas é fundamental para impulsionar a inovação e alcançar vantagens competitivas.

1.1 Justificativa do estudo

Neste trabalho de conclusão de curso, será abordado o processo abrangente de desenvolvimento de *software* para uma aplicação que integra diversas tecnologias de IA em um único sistema. A crescente demanda por soluções inteligentes e automatizadas tem impulsionado a convergência de diferentes tecnologias de IA em uma única plataforma, fortalecendo o resultado final. Compreender todo o ciclo de vida do desenvolvimento de *software*, desde a análise de requisitos até a implementação e escolha de arquitetura de *software*, é fundamental para um engenheiro de computação. Agregar conhecimento necessário para integrar IA, desde análise de requisitos até a implementação e escolha de arquitetura de *software*, permite compreender como essas etapas se relacionam e influenciam o desenvolvimento de uma aplicação de IA integrada. Este estudo visa contribuir para o avanço do conhecimento no campo do desenvolvimento de *software* de IA integrada, processos, desafios e práticas recomendadas envolvidos nesse domínio de rápida evolução. Ao compreender e dominar esses conceitos, os profissionais de engenharia estarão preparados para enfrentar os desafios complexos e aproveitar as oportunidades oferecidas pelo desenvolvimento de sistemas de IA integrados.

1.2 Objetivos

1.2.1 Gerais

Os objetivos gerais deste estudo serão todo o ciclo de desenvolvimento de um *software*, incluindo análises de requisitos, escolha de arquitetura da aplicação, e implementação. Este *software* tem uma interface gráfica intuitiva *Graphical User Interface* (GUI), que inclui um visualizador de áudio, um terminal e uma aba de configurações. Esta aba permiti ao usuário selecionar o idioma desejado e inserir as chaves de APIs necessárias para o funcionamento adequado do sistema. O *software* é projetado para oferecer suporte a quatro idiomas - português, inglês, espanhol e alemão - e integra várias funcionalidades. Incluindo, o reconhecimento de voz usando a API *speech-to-text da Google*, assim como sintetizadores de voz fornecidos pela ElevenLabs. Além disso, o *software* é capaz de manter conversas naturais com o usuário, utilizando a APIs de LLMs para geração de texto. Adicionalmente, o *software* possibilita ao usuário enviar e-mails, obter informações básicas como horário atual e data, fornecer cotações de moedas e até mesmo reproduzir

músicas no Spotify. Essas são apenas algumas das funcionalidades planejadas, havendo potencial para adicionar mais recursos conforme o desenvolvimento avance.

1.2.2 Específicos

- ❏ Desenvolver um assistente pessoal inteligente que integre múltiplas inteligências artificiais em uma única plataforma, capaz de realizar uma interação avançada e abrangente com o usuário.

- ❏ Implementar funcionalidades inovadoras que ampliem as possibilidades de uso do assistente, como reconhecimento de voz robusto, suporte a múltiplos idiomas, envio de e-mails, reprodução de músicas, criação e leitura de anotações, consulta a valores de criptomoedas e exibição de data e hora atuais.

- ❏ Projetar uma interface de usuário amigável e intuitiva que inclua um visualizador de áudio, terminal de *feedback* das operações e um botão central para ativar o reconhecimento de voz.

- ❏ Selecionar uma arquitetura de *software* adequada, visando garantir a separação de responsabilidades, modularidade, facilidade de manutenção e escalabilidade futura.

- ❏ Utilizar linguagens, bibliotecas e APIs apropriadas para cada funcionalidade implementada, como *Speech_recognition* para reconhecimento de voz, ElevenLabs para síntese de voz, Spotipy para integração com Spotify, entre outras, visando um desempenho eficiente e um desenvolvimento alinhado aos requisitos técnicos.

- ❏ Desenvolver o código seguindo os princípios SOLID e boas práticas de programação, de modo a garantir que o *software* seja legível, escalável e fácil de manter, considerando também a possibilidade de futuras expansões.

- ❏ Avaliar e considerar os impactos sociais e éticos do uso de assistentes pessoais baseados em IA, com foco em privacidade, transparência, acessibilidade, mitigação de vieses e impacto no mercado de trabalho, para promover uma tecnologia inclusiva e segura.

1.3 Metodologia

Para o desenvolvimento do *software* de inteligência artificial integrada, foi seguida uma metodologia estruturada em etapas sequenciais e integradas, para garantir um processo organizado e eficaz na criação do sistema. Primeiramente, foi realizado o levantamento de requisitos e funcionalidades desejadas, documentados detalhadamente em um arquivo específico. Esse levantamento visou a identificar tanto as funcionalidades principais quanto

os requisitos não funcionais, como a segurança de dados, escalabilidade, simplicidade e usabilidade. O sistema deveria incluir funcionalidades como reconhecimento de voz de alta precisão, um sintetizador de voz natural, suporte a múltiplos idiomas e operações específicas, como envio de *e-mails*, reprodução de músicas, e consulta a valores de criptomoedas. Além disso, deveria apresentar uma interface gráfica intuitiva, com recursos visuais interativos para melhorar a experiência do usuário.

Com os requisitos documentados, procedeu-se à análise e seleção da arquitetura de *software* mais adequada. Foram consideradas opções como arquitetura monolítica, microserviços, *plugins* e MVC (*Model-View-Controller*). Após uma pesquisa detalhada, descartaram-se as alternativas que, embora em sua maioria robustas, agregariam complexidade desnecessária ao projeto, que estava sendo desenvolvido por um único programador. A escolha final recaiu sobre a arquitetura MVC, que permite uma separação clara de responsabilidades e facilita o desenvolvimento e manutenção de aplicações interativas e escaláveis, mantendo o código organizado e modular, essencial para projetos que exigem flexibilidade para futuras atualizações.

Com a definição da arquitetura, iniciou-se a implementação do código. Para tanto, foi criado um ambiente virtual no IDE PyCharm, o que possibilitou a instalação e gestão de dependências isoladamente, garantindo que o projeto mantivesse seu próprio conjunto de pacotes e evitasse conflitos. Esse ambiente virtual foi nomeado com o projeto, proporcionando uma organização clara e independente das dependências necessárias para o desenvolvimento. Em seguida, foram configuradas as pastas e arquivos básicos para a estrutura do sistema, respeitando os princípios SOLID para garantir que o código fosse escalável, de fácil manutenção e leitura, características fundamentais para um *software* que deve evoluir com o tempo.

Durante a implementação das funcionalidades, cada recurso foi cuidadosamente projetado e adaptado à estrutura da arquitetura definida. O visualizador de áudio, por exemplo, foi construído a partir de um projeto *open source* com licença MIT disponível no GitHub, adaptado para integrar-se à interface da aplicação. A interface gráfica de usuário (GUI) foi construída utilizando uma biblioteca gráfica em Python; optou-se pela biblioteca PyQt5, que atende aos requisitos visuais e de interação da aplicação. A GUI foi estruturada de modo a incluir um terminal e uma aba de configurações, fornecendo controle e monitoramento de cada operação executada pelo sistema.

Para o reconhecimento de voz, foi implementada a API *speech-to-text* do Google, que proporciona alta precisão, especialmente em ambientes ruidosos, como especificado nos requisitos. O sintetizador de voz foi desenvolvido utilizando a API *text-to-speech* da ElevenLabs, que fornece uma qualidade de áudio natural e envolvente, aprimorando a experiência auditiva do usuário. APIs adicionais, como a do Gmail para envio de *e-mails*e a do Spotify para controle da reprodução de músicas, também foram integradas ao sistema, ampliando sua funcionalidade como assistente pessoal.

No que se refere à inteligência do sistema, a integração com agentes baseados no LLM da Groq permitiu que a aplicação analisasse e interpretasse as solicitações do usuário, oferecendo respostas contextualizadas e ações automatizadas. Essa inteligência permite que o sistema decida qual ação tomar com base nos comandos recebidos, incluindo a geração de texto e áudio.

Por fim, o sistema foi desenvolvido para oferecer uma experiência de usuário rica e interativa, garantindo que todos os requisitos funcionais e não funcionais fossem atendidos de forma integrada e fluida. Com uma interface intuitiva, funcionalidades robustas e código escalável, o *software* está preparado para futuras expansões e atualizações, consolidando-se como uma solução abrangente e inovadora na categoria de assistentes pessoais com inteligência artificial.

Capítulo 2

Revisão Teórica

2.1 História da IA

A busca pela compreensão e replicação da inteligência humana tem sido uma jornada fascinante ao longo da história IA. Desde o princípio, os pesquisadores têm procurado entender e emular as complexas capacidades do cérebro humano. Essa busca não é apenas um exercício intelectual, mas também uma tentativa de desvendar os mistérios do pensamento humano e criar sistemas de IA que possam rivalizar com a mente humana em termos de inteligência e adaptação.

2.1.1 O teste de Turing: 1940 - 1950

Por volta de 1950, o matemático inglês Alan Turing (1912-1954), considerado por muitos como o pai da computação, criou o famoso teste que leva seu nome: o teste de Turing. Esse teste compara a inteligência de uma máquina com a de um ser humano. Nele, um interrogador humano interage com um computador e um humano por meio de uma interface de texto. Se o interrogador não conseguir distinguir qual é o humano e qual é o computador com base nas respostas recebidas, o computador é considerado como tendo "passado" no teste (Turing, 1950). O teste de Turing teve um grande impacto no campo da inteligência artificial, apesar de ser criticado por alguns como uma medida limitada de inteligência. Embora Turing não tenha trabalhado diretamente no campo da IA como o entendemos hoje — pois o termo ainda não havia sido criado em sua época — suas ideias e conceitos, como o teste de Turing, foram fundamentais para o desenvolvimento inicial da IA como disciplina acadêmica e campo de pesquisa.

2.1.2 A criação do termo Inteligência Artificial: 1950 - 1960

Em 1955, John McCarthy propôs o termo "Inteligência Artificial" para descrever a área de estudo que se concentra na criação de sistemas que possam executar tarefas que normalmente requerem inteligência humana. Ele organizou a Conferência de Dartmouth

em 1956, juntamente com outros pioneiros da IA, como Marvin Minsky, Nathaniel Rochester e Claude Shannon. Esta conferência, que ocorreu no Dartmouth College, em Hanover, New Hampshire, Estados Unidos, é considerada o marco inicial da IA como disciplina acadêmica (Moor, 2006). O principal objetivo da conferência era reunir pesquisadores de diversas disciplinas, incluindo matemática, ciência da computação, psicologia e neurociência, para discutir a possibilidade de criar máquinas que pudessem exibir inteligência similar à humana. Após a conferência, o termo "Inteligencia artificial" proposto por McCharty em 1955 foi então aceito e formalizado.

Nesse período surgiram os primeiros programas de IA, como o *"Logic Theorist"* de Herbert Simon e Allen Newell, capaz de provar teoremas matemáticos. O *Logic Theorist* foi um programa de computador desenvolvido na Universidade Carnegie Mellon. Foi um dos primeiros programas de inteligência artificial e ficou conhecido por ser capaz de demonstrar teoremas na lógica proposicional. O *Logic Theorist* foi projetado para resolver problemas de lógica proposicional usando um método de busca heurística. Ele operava construindo árvores de prova para diferentes teoremas e aplicando regras de inferência para tentar encontrar uma prova para o teorema dado. O programa usava uma abordagem de força bruta, testando diferentes caminhos na árvore de prova até encontrar uma solução ou esgotar todas as possibilidades. O *Logic Theorist* foi notável por ser capaz de resolver alguns problemas de lógica complexos que até então só haviam sido resolvidos por humanos. Sua capacidade de demonstrar teoremas na lógica proposicional foi um marco importante no desenvolvimento da inteligência artificial e mostrou o potencial das máquinas para realizar tarefas intelectuais anteriormente consideradas exclusivas da mente humana. Embora o *Logic Theorist* tenha sido um marco significativo na história da inteligência artificial, ele tinha limitações óbvias em termos de eficiência e capacidade de lidar com problemas mais complexos. No entanto, seu sucesso demonstrou o potencial das abordagens computacionais para resolver problemas de lógica e abriu caminho para o desenvolvimento de sistemas mais avançados de inteligência artificial (Newell; Simon, 1956).

2.1.3 Investimentos em IA: 1960 - 1970

Em 1960 a IA começa a atrair investimentos significativos de agências governamentais e instituições de pesquisa. Com isso, surgem os primeiros sistemas de especialistas, como o "DENTRAL" (*Dendritic Algorithm*) para análise química e o "MYCIN" para diagnóstico médico. Também houve avanços em pesquisa de linguagem natural e visão computacional.

O DENDRAL foi desenvolvido na Universidade de Stanford por Edward Feigenbaum, Joshua Lederberg e seus colegas. Foi projetado para analisar espectros de massa e identificar estruturas moleculares de compostos químicos desconhecidos. O sistema era baseado em um conjunto de regras de inferência que permitiam analisar os dados dos espectros de massa e aplicar conhecimento sobre química orgânica para identificar as estruturas

moleculares correspondentes. O DENDRAL utilizava um modelo de árvore de decisão para representar o processo de inferência (Feigenbaum; Buchanan; Lederberg, 1970). O sucesso do DENDRAL demonstrou o potencial da inteligência artificial para lidar com problemas que requerem conhecimento especializado e raciocínio heurístico. Além disso, pavimentou o caminho para o desenvolvimento de outras aplicações de IA em ciências naturais e engenharia.

Já o MYCIN, também desenvolvido na Universidade de Stanford por Edward Shortliffe, foi projetado para diagnosticar doenças infecciosas bacterianas e recomendar tratamentos antibióticos com base em informações clínicas e laboratoriais fornecidas por médicos. O MYCIN funcionava como um sistema baseado em regras, onde as informações sobre o paciente e os sintomas eram inseridas no sistema, que então aplicava um conjunto de regras heurísticas para identificar a doença e sugerir tratamentos. O sistema foi projetado para imitar o raciocínio de especialistas humanos em doenças infecciosas. O MYCIN teve um grande impacto na pesquisa e no desenvolvimento de sistemas especialistas em medicina e em outros campos. Ele demonstrou o potencial da inteligência artificial para auxiliar profissionais de saúde no diagnóstico e tratamento de doenças, e inspirou o desenvolvimento de outros sistemas especialistas em áreas como radiologia, oncologia e dermatologia (Gulliford, 2015; Coates; Souhami; El Naqa, 2016).

Em 1966, foi criado um dos primeiros programas de processamento de linguagem natural (NLP), chamado ELiza, desenvolvido por Joseph Weizenbaum no MIT. Esse programa simulava uma conversa terapêutica com um psicoterapeuta, utilizando padrões simples de reconhecimento e substituição de texto. Embora fosse bastante rudimentar, ELiza foi uma das primeiras demonstrações práticas do processamento de linguagem natural e influenciou muitos sistemas posteriores (Weizenbaum, 1966; Weizenbaum, 1976).

Ainda na década de 1960, pesquisadores começaram a explorar a aplicação de técnicas computacionais para processar e analisar imagens. Isso incluiu experimentos com detecção de bordas, reconhecimento de padrões e extração de características. Houve avanços significativos na aplicação de técnicas de reconhecimento de padrões para a análise de imagens, incluindo sistemas que podiam identificar objetos simples em imagens, como letras e dígitos.

2.1.4 O Inverno da IA: 1970 - 1990

À partir de 1970, a IA enfrenta um período de desilusão conhecido como “inverno da IA”, devido a dificuldades em atingir as expectativas iniciais e a falta de avanços significativos. No entanto, nesse período, surgem técnicas como redes neurais artificiais e sistemas baseados em lógica difusa.

O “Inverno da IA” se refere a um período de desencanto e desinvestimento em IA que ocorreu nas décadas de 1970 e 1980, após um período inicial de grande entusiasmo e expectativas exageradas nas décadas de 1950 e 1960. Durante o início da história

da IA, na década de 1950 e início da década de 1960, houve um otimismo significativo sobre o potencial da IA para resolver uma variedade de problemas complexos e realizar tarefas anteriormente consideradas exclusivas da mente humana. No entanto, à medida que os pesquisadores começaram a enfrentar os desafios técnicos e práticos envolvidos na construção de sistemas de IA robustos e escaláveis, tornou-se claro que muitas das expectativas iniciais eram irrealistas (Lighthill, 1972).

Durante o "Inverno da IA", muitos laboratórios de pesquisa foram fechados, financiamentos foram cortados e o interesse público em IA diminuiu significativamente. No entanto, apesar desses desafios, a pesquisa em IA continuou avançando em áreas específicas e, eventualmente, ressurgiu com força nos anos 1990 e 2000, impulsionada por avanços significativos em algoritmos, computação e disponibilidade de dados. O "Inverno da IA" serve como um lembrete importante dos desafios e da natureza cíclica do progresso científico e tecnológico.

O surgimento de técnicas como redes neurais artificiais (RNAs) e sistemas baseados em lógica difusa foi uma resposta aos desafios enfrentados pela IA durante o "Inverno da IA" e um impulso para revitalizar o campo. As RNAs são modelos computacionais inspirados no funcionamento do cérebro humano, compostos por unidades de processamento chamadas neurônios, que estão conectadas por conexões ponderadas (Zou; Han; So, 2009). As RNAs mostraram-se eficazes em lidar com problemas complexos de aprendizado de máquina, reconhecimento de padrões, processamento de linguagem natural, visão computacional e muito mais. O ressurgimento do interesse em RNAs ajudou a impulsionar o renascimento da IA e contribuiu para o desenvolvimento de sistemas de IA mais sofisticados e capazes.

Os sistemas baseados em lógica difusa foram desenvolvidos para lidar com problemas de imprecisão e incerteza, que muitas vezes são encontrados em sistemas do mundo real. Os sistemas baseados em lógica difusa encontraram aplicações em uma ampla gama de domínios, incluindo controle de processos, sistemas de recomendação, tomada de decisão, diagnóstico médico e muito mais. A capacidade dos sistemas baseados em lógica difusa de lidar com a incerteza e a imprecisão os tornou uma ferramenta valiosa em situações onde as abordagens tradicionais de IA eram inadequadas.

O surgimento dessas técnicas representou uma mudança significativa no paradigma da IA, abrindo novas possibilidades para resolver problemas complexos e lidar com a incerteza inerente aos sistemas do mundo real. Eles contribuíram para revitalizar o campo da IA e impulsionar o progresso em direção a sistemas mais sofisticados e inteligentes.

2.1.5 Renascimento da IA: 2016 - Presente

Por volta do ano de 2016, a IA passa por um renascimento, impulsionado pelo aumento da capacidade computacional, grandes conjuntos de dados e avanços em algoritmos de aprendizado de máquina, especialmente redes neurais profundas. Aplicações de IA, como reconhecimento de voz, visão computacional, veículos autônomos e assistentes virtuais,

tornam-se cada vez mais comuns, e passam por um grande avanço com a criação da arquitetura *transformes* (Vaswani *et al.*, 2017).

O grande aumento do poder computacional, que subiu de alguns megabytes para terabytes, foi de vital importância para o renascimento da IA. Não apenas a capacidade de armazenamento de dados, mas também o poder de processamento e eficiência energética, aumentaram de forma similar. Este aumento do poder computacional foi de suma importância, tanto para treinar quanto para armazenar os atuais modelos de IA, tais como os *Large Language Models* (LLMs). Além disso a globalização, e popularização da internet impulsionaram a geração de grandes bancos de dados tornando possível o treinamento de modelos de ML mais robustos. Por fim, os avanços na criação de algoritmos de ML mais complexos juntamente com o avanço de poder computacional acompanhado com grandes quantidades de dados, permitiram grandes avanços na IA.

O aprendizado profundo, uma subárea de aprendizado de máquina que utiliza redes neurais profundas, tem sido responsável por muitos avanços em visão computacional, processamento de linguagem natural, reconhecimento de fala, jogos e muito mais. Arquiteturas de redes neurais profundas, como redes neurais convolucionais (CNNs) e redes neurais recorrentes (RNNs), têm alcançado desempenho humano e super-humano em uma variedade de tarefas, incluindo reconhecimento de imagem, tradução automática e geração de texto.

No NLP, modelos baseados em aprendizado profundo têm alcançado resultados impressionantes em tarefas como tradução automática, sumarização de texto, geração de texto, análise de sentimento, questionamento e resposta. Avanços em modelos de linguagem pré-treinados, como o *Generative Pre-trained Transformer* (GPT) e o *Bidirectional Encoder Representations from Transformers* (BERT), levaram a melhorias significativas na compreensão e geração de texto (Achiam *et al.*, 2023).

Avanços em visão computacional permitiram o reconhecimento de objetos em imagens e vídeos, detecção de rostos, segmentação semântica, rastreamento de objetos, e realidade aumentada. Técnicas como detecção de objetos em tempo real, reconhecimento facial em larga escala e sistemas de visão computacional aplicados a veículos autônomos e *drones* tornaram-se cada vez mais comuns.

Com o crescimento da aplicação de sistemas de IA em diversas áreas da vida cotidiana, cresceu também a preocupação com questões éticas, como viés algorítmico, privacidade, transparência, equidade e justiça (Tamkin *et al.*., 2021; Eloundou, 2023).

2.2 O processo de desenvolvimento de *software*

O desenvolvimento de *software* é uma disciplina complexa que envolve a criação de sistemas e aplicações computacionais para atender a diversas necessidades e solucionar problemas específicos. Este processo é fundamental para o avanço tecnológico e para

a inovação em múltiplos setores, desde a indústria e comércio até a educação e saúde (Pressman; Maxim, 2014). Para garantir a eficiência e a qualidade dos produtos de *software*, é essencial seguir metodologias bem-definidas e estruturadas.

O processo de desenvolvimento de *software* geralmente é dividido em várias fases distintas, conhecidas como o ciclo de vida do *software*. Essas fases incluem a concepção, o levantamento de requisitos, o *design*, a implementação, os testes, a implantação e a manutenção. Cada fase desempenha um papel crucial na entrega de um produto final funcional e confiável.

2.2.1 Concepção e Levantamento de Requisitos

A fase de Concepção e Levantamento de Requisitos é o alicerce de qualquer projeto de desenvolvimento de *software*. Esta fase é crucial para garantir que o *software* final atenderá às necessidades dos usuários e *stakeholders*, fornecendo uma base sólida para as fases subsequentes do desenvolvimento.

2.2.1.1 Concepção

A fase de Concepção é fundamental para definir a visão geral e os objetivos do projeto de *software*. O primeiro passo nesta fase é a identificação das necessidades, que envolve conversas iniciais com os *stakeholders* para compreender as necessidades de alto nível. Estas conversas podem incluir reuniões, entrevistas e *workshops*, onde são coletadas informações detalhadas sobre o problema que o *software* deve resolver (Pressman; Maxim, 2014).

Uma vez identificadas as necessidades, a definição do escopo do projeto torna-se essencial. Este passo determina o que estará incluído no projeto e, igualmente importante, o que estará fora dele. Um escopo bem definido ajuda a prevenir a expansão do projeto e garante que todos os envolvidos tenham expectativas claras e alinhadas.

Em seguida, é realizado um estudo de viabilidade. Este estudo avalia se o projeto é viável técnica, econômica e operacionalmente. A análise inclui a avaliação dos recursos disponíveis, a tecnologia necessária, o tempo e o orçamento necessários para completar o projeto. Esta avaliação é fundamental para assegurar que o projeto pode ser realizado dentro das restrições existentes.

Paralelamente, é desenvolvido um caso de negócio. Este documento justifica o investimento no projeto, apresentando os benefícios esperados, os custos envolvidos, os riscos associados e o retorno sobre o investimento. O caso de negócio serve como um argumento convincente para a aprovação e o financiamento do projeto, destacando como o *software* proposto agregará valor à organização.

Durante toda a fase de Concepção, a comunicação clara e contínua com os *stakeholders* é vital. Esta comunicação assegura que todas as partes interessadas tenham uma

compreensão comum do projeto, seus objetivos e as expectativas associadas. A fase de Concepção, portanto, estabelece uma base sólida para as etapas subsequentes do desenvolvimento de *software*, garantindo que o projeto se desenvolva de acordo com as necessidades e objetivos identificados.

2.2.1.2 Levantamento de requisitos

A fase de Levantamento de Requisitos é fundamental para definir com precisão o que o *software* deve fazer. O processo começa com a coleta de requisitos, onde se busca entender detalhadamente as necessidades e expectativas dos *stakeholders*. Essa coleta pode ser realizada através de entrevistas, questionários, *workshops* e análise de documentos existentes, visando captar tanto requisitos funcionais quanto não funcionais.

Os requisitos funcionais descrevem o que o sistema deve fazer. Eles incluem funcionalidades específicas, como operações, entradas, saídas e comportamentos. Por exemplo, um sistema de gerenciamento de biblioteca pode ter requisitos funcionais que detalhem o processo de empréstimo e devolução de livros, gerenciamento de catálogo e notificações de vencimento de prazos.

Já os requisitos não funcionais definem as qualidades que o sistema deve possuir. Eles abordam aspectos como desempenho, segurança, usabilidade e confiabilidade. Esses requisitos garantem que o sistema não apenas funcione corretamente, mas também atenda a critérios de qualidade importantes para os usuários e para o ambiente em que será utilizado.

Uma vez coletados, os requisitos precisam ser analisados e validados. A análise envolve a revisão detalhada dos requisitos para garantir que sejam completos, claros e livres de ambiguidades. Durante a validação, verifica-se se os requisitos realmente refletem as necessidades dos *stakeholders* e se são tecnicamente viáveis dentro das restrições do projeto.

Os requisitos, então, são documentados de forma estruturada. Esta documentação serve como referência durante todo o ciclo de vida do projeto, orientando a equipe de desenvolvimento e assegurando que todos os envolvidos tenham uma compreensão comum do que será desenvolvido. A documentação de requisitos pode incluir casos de uso, *user stories*, diagramas e especificações detalhadas.

É fundamental manter a comunicação contínua com os *stakeholders* durante todo o processo de levantamento de requisitos. Reuniões regulares e revisões de *feedback* ajudam a garantir que os requisitos estejam sempre alinhados com as expectativas e que qualquer mudança ou novo requisito seja identificado e tratado rapidamente.

A fase de Levantamento de Requisitos é, portanto, essencial para o sucesso do projeto, pois estabelece uma base clara e detalhada sobre o que deve ser desenvolvido, garantindo que a solução final atenda às necessidades dos usuários e esteja alinhada com os objetivos do projeto.

2.2.2 Design

A etapa de *Design* é importante para a transformação dos requisitos definidos na fase anterior em uma estrutura viável e detalhada para o desenvolvimento do *software*. Esta fase abrange a criação de uma arquitetura robusta e o planejamento da implementação das funcionalidades, garantindo que o sistema seja eficiente, escalável e fácil de manter. O *design* de um sistema começa com a definição da sua arquitetura, que serve como o esqueleto do *software*, estabelecendo a estrutura geral e os principais componentes. Existem várias abordagens arquitetônicas, incluindo a arquitetura em camadas, a arquitetura orientada a serviços (SOA) e a arquitetura de microsserviços. A escolha da arquitetura mais adequada depende dos requisitos do projeto, do escopo, da escalabilidade desejada e da equipe de desenvolvimento.

Após a definição da arquitetura, é feita a modelagem de dados, que envolve a elaboração de diagramas, como o Diagrama Entidade-Relacionamento (ER). Esse diagrama ilustra as entidades, seus atributos e os relacionamentos entre elas, sendo fundamental para a construção de um banco de dados eficiente, garantindo integridade e consistência das informações. Técnicas como normalização são aplicadas para reduzir redundâncias e melhorar a eficiência do armazenamento de dados.

Em seguida, o *design* de interfaces é uma etapa crítica, pois determina a interação dos usuários com o sistema. Esta fase inclui a criação de *wireframes*, *mockups* e protótipos de alta fidelidade, que devem ser intuitivos, responsivos e alinhados com a experiência do usuário (UX). Ferramentas como Sketch, Figma ou Adobe XD são frequentemente utilizadas para desenvolver esses protótipos. Além disso, são definidos os fluxos de navegação e as interações do usuário, garantindo uma experiência consistente e satisfatória.

O *design* de componentes envolve a especificação detalhada dos módulos e elementos do sistema. Cada componente é descrito em termos de suas responsabilidades, interfaces e interações com outros componentes. Para representar a estrutura e o comportamento do sistema, diagramas de classes, diagramas de sequência e diagramas de componentes são criados. Essa documentação é essencial para orientar a implementação e facilitar a compreensão do sistema pela equipe de desenvolvimento.

Durante a fase de *design*, também são escolhidos padrões de projeto, conhecidos como *design patterns*, para resolver problemas comuns de *design* de *software*. Padrões como *Singleton*, *Factory*, *Observer* e *Strategy* são aplicados para promover a reutilização de código, melhorar a manutenção e aumentar a escalabilidade do sistema. A escolha adequada desses padrões ajuda a evitar soluções ad-hoc, garantindo um *design* mais modular e flexível.

A prototipagem é uma prática comum nesta fase, permitindo validar ideias e conceitos antes da implementação. Testes de usabilidade são realizados com usuários reais ou representantes dos *stakeholders* para identificar possíveis melhorias na interface e na experiência do usuário (Nielsen, 1993). Os *feedbacks* coletados durante esses testes são

utilizados para refinar os protótipos, assegurando que o produto final seja fácil de usar e atenda às necessidades dos usuários.

Por fim, a documentação de *design* é elaborada de forma detalhada, incluindo descrições dos diagramas, especificações dos componentes, fluxos de dados e interfaces de usuário. Essa documentação serve como referência durante o desenvolvimento, testes e manutenção do sistema, além de facilitar a comunicação entre os membros da equipe e os *stakeholders*, garantindo que todos estejam alinhados com a visão e os detalhes do projeto.

2.2.3 Implementação

A fase de implementação é o momento em que o *design* previamente elaborado é transformado em código executável. Esta etapa é muito importante, pois representa a concretização dos planos e especificações definidos nas fases anteriores, resultando no desenvolvimento efetivo do *software*. Para que a implementação ocorra de forma eficiente, é essencial uma configuração adequada do ambiente de desenvolvimento. Isso envolve a instalação de ambientes de desenvolvimento integrados (IDEs) apropriados, como Visual Studio Code, IntelliJ IDEA ou PyCharm, dependendo da linguagem de programação escolhida, além da configuração de ferramentas de versionamento de código, como Git, que ajudam a rastrear alterações e facilitar a colaboração entre os desenvolvedores (Presmann; Maxim, 2014).

A implementação inicia-se com a codificação dos componentes definidos na fase de *design*, onde os desenvolvedores escrevem o código conforme as especificações detalhadas, utilizando a linguagem de programação selecionada. Durante essa fase, é fundamental seguir as práticas de Clean Code, assegurando que o código seja legível, modular e de fácil manutenção (Martin, 2008). É crucial garantir que cada componente atenda às suas responsabilidades e interaja corretamente com os demais.

Outro aspecto essencial durante a implementação é a Integração Contínua (CI), que envolve a integração frequente do código de todos os desenvolvedores em um repositório central. Ferramentas como Jenkins, Travis CI e GitLab CI são utilizadas para automatizar esse processo, executar testes automatizados e gerar *builds* do *software*. A prática de Integração Contínua facilita a detecção e correção rápida de erros, assegurando que o código integrado esteja sempre em um estado funcional (Fowler, 2006).

Os desenvolvedores também escrevem testes unitários para validar individualmente as unidades de código, como funções, métodos e classes. Para isso, utilizam ferramentas de teste como JUnit para Java, NUnit para C e pytest para Python. Além dos testes unitários, são elaborados testes automatizados que abrangem fluxos e cenários de uso mais amplos, garantindo a consistência e confiabilidade do *software* (Beck, 2003).

A revisão de código é uma prática fundamental nessa fase, pois assegura a qualidade do código e a conformidade com os padrões de desenvolvimento. Durante essa revisão, outros desenvolvedores examinam o código escrito para identificar possíveis melhorias,

bugs ou inconsistências. Ferramentas como GitHub, GitLab e Bitbucket facilitam esse processo, permitindo comentários e discussões sobre o código. Essa prática promove a colaboração e a disseminação de conhecimento dentro da equipe.

O gerenciamento de dependências é outra atividade importante que envolve a administração das bibliotecas e pacotes externos utilizados no projeto. Ferramentas como Maven e Gradle para Java, npm para JavaScript e pip para Python são empregadas para garantir que todas as bibliotecas necessárias estejam disponíveis e atualizadas, além de assegurar a compatibilidade das dependências e evitar vulnerabilidades de segurança (Jones, 2014).

As funcionalidades descritas nos requisitos são implementadas de forma incremental e iterativa, onde cada funcionalidade é desenvolvida, integrada e testada individualmente antes de ser combinada com outras. Essa abordagem permite um *feedback* contínuo e ajustes rápidos, garantindo que o *software* evolua conforme planejado e atenda às expectativas dos *stakeholders* (Basili; Rumbaugh, 1994).

Durante a implementação, é vital documentar o código adequadamente, incluindo comentários, documentação de métodos e classes, e guias de uso para facilitar a compreensão e a manutenção futura do *software*. Ferramentas como Javadoc para Java, Doxygen para C++ e Sphinx para Python são utilizadas para gerar documentação automática a partir do código comentado (Booch, 2007).

A gestão de configuração envolve o controle de diferentes versões e configurações do *software* durante a implementação. Ferramentas de controle de versão, como Git, são utilizadas para gerenciar ramificações, fusões e *releases* do *software*, garantindo que todas as mudanças sejam rastreadas e possibilitando a reversão a versões anteriores, se necessário (Chapman, 2010).

Para sistemas locais, a implementação também inclui a configuração e o *deploy* do *software* no ambiente onde será executado. Isso pode envolver a instalação de pacotes, configuração de bancos de dados locais e ajustes em servidores ou máquinas específicas. O *deploy* local deve ser testado para assegurar que o *software* funcione corretamente no ambiente de produção (Krebs, 2019).

Após a implementação inicial, o *software* é monitorado para identificar qualquer comportamento inesperado ou problemas de desempenho. Ferramentas de monitoramento podem ser utilizadas para coletar dados sobre o uso do software e identificar áreas que necessitam de melhorias. O *feedback* dos usuários e *stakeholders* é continuamente coletado e utilizado para guiar as futuras iterações de desenvolvimento (Lewin, 2011).

2.2.4 Testes

A fase de testes é essencial para assegurar que o *software* desenvolvido funcione conforme esperado, atenda aos requisitos especificados e esteja livre de erros. A realização de testes minuciosos permite identificar e corrigir defeitos, resultando em uma melhoria na qualidade e confiabilidade do produto final. O processo de testes inicia-se com o planeja-

mento, onde são definidos os objetivos, o escopo, as estratégias e os critérios de aceitação. Um plano de testes é elaborado, detalhando os tipos de testes a serem realizados, as ferramentas necessárias, os recursos envolvidos e o cronograma a ser seguido. Nesta etapa, também são identificados os casos de teste e preparados os dados de teste (Beck, 2003).

Diversos tipos de testes podem ser realizados durante essa fase. Os testes unitários validam o funcionamento de unidades individuais de código, como funções, métodos ou classes, geralmente sendo automatizados e escritos pelos desenvolvedores durante a implementação. Os testes de integração, por sua vez, verificam se diferentes módulos ou componentes do sistema interagem corretamente, assegurando que as interfaces funcionem como esperado (Myers, 2004). Os testes funcionais avaliam se o *software* atende aos requisitos funcionais especificados, baseando-se nos casos de uso para garantir que o sistema realize as tarefas conforme previsto (Graham *et al.*, 2009).

Os testes de sistema examinam o sistema como um todo, incluindo todos os módulos integrados, para verificar seu comportamento em um ambiente completo e realista. Já os testes de aceitação são realizados pelo cliente ou usuários finais, validando se o *software* atende às expectativas e requisitos estabelecidos, servindo como a última etapa antes da entrega (Dixon, 2010). Além disso, os testes de regressão garantem que mudanças ou adições de código não introduzam novos defeitos em partes já testadas do sistema e são repetidos regularmente durante o desenvolvimento (Rubin, 2011). Além disso, os testes de desempenho avaliam a performance do sistema sob diferentes condições de carga, incluindo testes de carga, estresse e escalabilidade para assegurar eficiência em situações de alta demanda (Jones, 2009). Por fim, os testes de segurança verificam a robustez do sistema contra ataques e vulnerabilidades, garantindo a proteção de dados sensíveis e resistência a tentativas de invasão.

A execução dos testes ocorre conforme o planejado, utilizando tanto testes automatizados quanto manuais. Ferramentas de automação, como Selenium, JUnit, TestNG e Pytest, são amplamente utilizadas para executar testes repetitivos de forma eficiente, enquanto testes manuais são realizados para cenários que exigem avaliação humana ou que são difíceis de automatizar (Bastide *et al.*, 2018). Durante essa fase, defeitos ou *bugs* encontrados são registrados em ferramentas de rastreamento, como Jira, Bugzilla ou Trello. Cada defeito é documentado com detalhes sobre o problema, os passos para reproduzi-lo, além de sua severidade e prioridade. Os desenvolvedores analisam e corrigem os defeitos, e os testes são reexecutados para verificar as correções (Nichols, 2010).

Após a execução dos testes, os resultados são revisados e analisados para avaliar a qualidade do *software*. Relatórios de testes são gerados para documentar os testes realizados, os defeitos encontrados e corrigidos, além do status geral do sistema (Rubin, 2011). A análise dos resultados permite identificar áreas problemáticas e oportunidades de melhoria. Os testes de aceitação do usuário (UAT) são conduzidos em colaboração com os usuários finais ou clientes, avaliando se o *software* atende às necessidades e expectativas

reais deles. O *feedback* obtido durante o UAT é crucial para garantir a satisfação do cliente e a usabilidade do sistema.

Após a conclusão bem-sucedida dos testes e a resolução de todos os defeitos críticos, o *software* é aprovado para lançamento. Realizam-se verificações finais para assegurar que todas as funcionalidades estejam implementadas corretamente e que o sistema esteja pronto para a produção.

2.2.5 Implantação

A fase de implantação é crucial para garantir que o *software* desenvolvido seja disponibilizado e operado em um ambiente de produção. Esta etapa envolve a preparação, instalação, configuração e a transferência do *software* para o ambiente onde ele será utilizado pelos usuários finais. Uma implantação bem-sucedida assegura que o *software* funcione conforme o esperado e que os usuários possam acessar e utilizar todas as suas funcionalidades.

Antes de iniciar a implantação, é essencial realizar um planejamento cuidadoso. Este planejamento inclui a definição das estratégias de implantação, a criação de um cronograma detalhado, a identificação dos recursos necessários e a preparação de um plano de contingência. Também é importante garantir que todos os componentes do *software* estejam prontos para a implantação, incluindo código, banco de dados, configurações e documentação (Peters, 2018).

A implantação geralmente envolve a movimentação do *software* através de vários ambientes, cada um com seu próprio propósito. O ambiente de desenvolvimento é onde os desenvolvedores escrevem e testam o código. Em seguida, o *software* passa para o ambiente de testes, onde é testado de forma mais rigorosa por uma equipe de garantia de qualidade (QA). Após isso, o *software* é validado no ambiente de homologação, que replica o ambiente de produção, sendo validado pelo cliente ou usuários finais antes de ir para produção (Smith, 2019). Finalmente, o ambiente de produção é onde o *software* está disponível para os usuários finais.

O processo de implantação pode variar dependendo das necessidades específicas do projeto, mas geralmente inclui algumas etapas principais. Primeiramente, é essencial realizar *backups* completos do sistema atual antes da implantação, incluindo dados, configurações e código, para garantir a possibilidade de reverter para o estado anterior em caso de problemas (Jones, 2020). Em seguida, é necessário configurar o ambiente de produção para garantir que ele suporte o novo *software*, o que pode incluir a instalação de dependências, configuração de servidores, redes e armazenamento.

A próxima etapa envolve a transferência de código e dados para o ambiente de produção. Isso pode ser feito manualmente ou usando ferramentas de automação de implantação, como Ansible, Chef ou Puppet (Anderson, 2021). Se houver alterações no esquema do banco de dados, elas devem ser aplicadas com cuidado, utilizando *scripts* de migração

para atualizar o banco de dados sem perda de dados. Após a transferência, é necessário configurar o *software* para operar no ambiente de produção, ajustando variáveis de ambiente, performance e configurações de segurança conforme necessário.

Após a implantação, é importante realizar verificações para garantir que o *software* está funcionando corretamente. Isso inclui testes funcionais, testes de desempenho e verificação de segurança (Williams, 2017). A automação da implantação é uma prática recomendada para tornar o processo mais eficiente, consistente e menos propenso a erros. Ferramentas como Jenkins, GitLab CI/CD, Docker e Kubernetes são amplamente usadas para automatizar a construção, teste e implantação do *software*. A automação permite a implantação contínua (CD), onde as mudanças no código são automaticamente implantadas em produção após passarem por testes automatizados (Taylor, 2018).

Após a implantação, é fundamental monitorar o *software* para garantir que ele esteja funcionando corretamente. Ferramentas de monitoramento, como Nagios, Prometheus, Grafana e Splunk, ajudam a identificar problemas de performance, erros e outras questões operacionais em tempo real (Brown, 2020). Além disso, é importante ter um plano de suporte para resolver rapidamente qualquer problema que possa surgir após a implantação.

Para garantir uma transição suave, é essencial fornecer treinamento aos usuários finais e à equipe de suporte técnico. A documentação detalhada do sistema, incluindo manuais de usuário, guias de instalação e *troubleshooting*, deve estar disponível para facilitar a utilização e manutenção do *software* (Wilson, 2019). A fase de implantação não termina com a disponibilização do *software*. É importante coletar *feedback* dos usuários finais para identificar áreas de melhoria e novas funcionalidades. A melhoria contínua, baseada no *feedback* dos usuários, garante que o *software* continue atendendo às necessidades e expectativas.

2.2.6 Manutenção

A fase de manutenção é uma etapa contínua e essencial no ciclo de vida do *software*, sendo responsável por garantir que ele permaneça funcional, eficiente e relevante após sua implantação inicial. Durante essa fase, o *software* é monitorado, atualizado e aprimorado para atender às necessidades dos usuários e às mudanças no ambiente operacional. A manutenção pode ser classificada em quatro categorias principais: corretiva, adaptativa, perfectiva e preventiva.

A manutenção corretiva é realizada para corrigir falhas que surgem após a implantação do *software*. Esses erros podem ser descobertos pelos usuários finais ou pela equipe de suporte técnico. A identificação rápida e a correção de falhas são essenciais para minimizar o impacto nos usuários e garantir a continuidade das operações. Essa manutenção envolve o diagnóstico do problema, a correção do código e a realização de testes para assegurar que a solução funcione corretamente e não introduza novos problemas (Johnson, 2020).

Com o passar do tempo, o ambiente em que o *software* opera pode mudar, exigindo adaptações para que ele continue a funcionar adequadamente. Isso caracteriza a manutenção adaptativa, que pode incluir atualizações de sistema operacional, alterações em outras aplicações com as quais o *software* interage ou modificações nos requisitos de hardware (Miller, 2019). Essa manutenção assegura que o *software* permaneça compatível com essas mudanças ambientais e tecnológicas, o que pode envolver a atualização de bibliotecas, a modificação de interfaces de integração ou a reconfiguração de componentes do sistema.

A manutenção perfectiva, por sua vez, é focada na melhoria do desempenho e na adição de novas funcionalidades ao *software*. Com base no *feedback* dos usuários e nas novas necessidades de negócios, a equipe de desenvolvimento pode implementar aprimoramentos para tornar o *software* mais eficiente, fácil de usar ou funcional (Brown, 2018). Essa manutenção pode incluir a otimização de algoritmos, a melhoria da interface do usuário ou a adição de novos módulos e funcionalidades.

A manutenção preventiva é realizada com o objetivo de evitar problemas futuros e prolongar a vida útil do *software*. Ela envolve a análise e a identificação de áreas que podem causar problemas no futuro, além da implementação de medidas para prevenir essas questões. Isso pode incluir a refatoração de código para melhorar sua qualidade e legibilidade, a atualização de componentes para versões mais seguras e estáveis, e a realização de testes de desempenho para identificar e resolver possíveis gargalos (Smith, 2021).

Durante a fase de manutenção, o monitoramento contínuo do *software* é fundamental. Ferramentas de monitoramento ajudam a identificar problemas de desempenho, segurança e disponibilidade em tempo real. Além disso, a coleta e análise de métricas de uso e desempenho fornecem *insights* valiosos para a melhoria contínua do *software* (Anderson, 2020). O gerenciamento de configurações e mudanças também é uma prática essencial nesta fase, já que todas as alterações devem ser documentadas e gerenciadas de forma controlada, garantindo que não haja impacto negativo na operação do sistema. Ferramentas de controle de versão, como Git, são utilizadas para rastrear mudanças no código e facilitar a colaboração entre a equipe de desenvolvimento.

Outro aspecto importante da fase de manutenção é o suporte ao usuário. A equipe de suporte técnico deve estar disponível para ajudar os usuários a resolver problemas, responder a perguntas e fornecer orientação sobre o uso do *software*. O *feedback* dos usuários é crucial para identificar áreas de melhoria e novas funcionalidades (Taylor, 2019).

2.3 Conceitos importantes

2.3.1 *Machine Learning*

Nos últimos anos, o *machine learning* (aprendizado de máquina) tem se destacado como uma das principais vertentes da IA, estando presente em uma ampla variedade de aplicações do cotidiano. Esse campo da ciência da computação possibilita que computadores realizem tarefas complexas sem a necessidade de serem explicitamente programados para cada ação. Em vez disso, os sistemas são treinados para identificar padrões em grandes conjuntos de dados, permitindo-lhes tomar decisões e realizar previsões com base nas informações recebidas (Jordan; Mitchell, 2015).

O aprendizado de máquina consiste em criar algoritmos capazes de aprender a partir de exemplos, adaptando-se conforme são expostos a novos dados. Esse processo é semelhante à forma como seres humanos aprendem a partir da experiência: ao observar um grande número de situações, é possível identificar padrões e tomar decisões mais acertadas em casos futuros (Bishop, 2006). Esse princípio permite que sistemas automatizados sejam cada vez mais eficazes em diversas áreas, como recomendações de conteúdo em plataformas de *streaming*, diagnósticos médicos, análise de risco financeiro e reconhecimento de voz.

Existem dois principais paradigmas no *machine learning*: o aprendizado supervisionado e o aprendizado não supervisionado. Cada um deles possui características específicas que os tornam adequados para diferentes tipos de problemas, dependendo da disponibilidade de dados e da natureza das informações a serem analisadas (NG, 2012).

2.3.1.1 Aprendizado Supervisionado

O aprendizado supervisionado é caracterizado pelo uso de conjuntos de dados previamente rotulados, nos quais as respostas corretas são fornecidas durante o treinamento do algoritmo. Nesse contexto, o sistema é apresentado a pares de entradas e saídas conhecidas, de modo que possa aprender a associar as características dos dados de entrada às respostas esperadas. O objetivo é que, após o treinamento, o algoritmo seja capaz de generalizar essas associações e fornecer respostas corretas quando exposto a novos dados (Goodfellow; Bengio; Courville, 2016).

Por exemplo, ao desenvolver um sistema capaz de identificar imagens de gatos e cachorros, um conjunto de treinamento contendo imagens rotuladas (cada uma identificada como sendo de um gato ou de um cachorro) é utilizado para ensinar o sistema a distinguir entre essas duas classes de animais. Durante o processo de aprendizado, o algoritmo identifica características específicas das imagens, como formas, cores e texturas, que ajudam a distinguir um gato de um cachorro. Após esse treinamento, o sistema é capaz de identificar corretamente novas imagens de animais que não foram apresentadas anteriormente.

Essa abordagem é amplamente utilizada em problemas em que se deseja prever uma variável específica a partir de características conhecidas, como na previsão de preços de

imóveis, classificação de *e-mails* como *spam* ou não *spam*, e até mesmo em diagnósticos de doenças a partir de imagens de exames médicos (Lecun; Bengio; Hinton, 2015).

2.3.1.2 Aprendizado Não Supervisionado

Diferentemente do aprendizado supervisionado, o aprendizado não supervisionado opera sem a necessidade de rótulos nos dados de entrada. Nesse caso, o algoritmo recebe um conjunto de dados não categorizados e busca identificar padrões ou estruturas ocultas nesses dados, sem que haja um conhecimento prévio sobre as respostas corretas. O objetivo é encontrar agrupamentos ou padrões que possam ser úteis para a análise dos dados (Bishop, 2006).

Um exemplo de aprendizado não supervisionado é a análise de um grande conjunto de imagens de diferentes frutas, sem que as frutas estejam identificadas por seus nomes. O sistema é capaz de identificar grupos de imagens que compartilham características semelhantes, como cor, formato e textura, formando agrupamentos de frutas que são parecidas entre si. O algoritmo pode identificar grupos que correspondem a maçãs, bananas ou laranjas, mesmo sem saber os nomes dessas frutas.

O aprendizado não supervisionado é frequentemente empregado em situações onde se deseja explorar e entender a estrutura de um conjunto de dados, como em análises de *clusters* de clientes em *marketing*, para identificar grupos de consumidores com comportamentos de compra semelhantes, ou na detecção de anomalias em sistemas de segurança, onde padrões de comportamento atípicos podem ser identificados (Goodfellow; Bengio; Courville, 2016).

2.3.1.3 Aplicações Práticas e Importância do *Machine Learning*

Tanto o aprendizado supervisionado quanto o não supervisionado são fundamentais para inúmeras aplicações que facilitam e aprimoram a vida moderna. O aprendizado supervisionado é frequentemente utilizado em sistemas de recomendação, que sugerem produtos ou conteúdos com base nas preferências anteriores de um usuário (NG, 2012). Já o aprendizado não supervisionado pode ser essencial na análise de grandes volumes de dados, ajudando a descobrir *insights* em áreas como a biomedicina, onde é utilizado para identificar novos grupos de pacientes com sintomas semelhantes (Lecun; Bengio; Hinton, 2015).

Além dessas aplicações, o *machine learning* tem um papel crucial no desenvolvimento de tecnologias emergentes, como carros autônomos, que combinam diversos algoritmos de aprendizado para detectar objetos e tomar decisões em tempo real (Jordan; Mitchell, 2015). A área de segurança também se beneficia, com sistemas capazes de detectar fraudes financeiras e identificar ameaças cibernéticas a partir da análise de grandes quantidades de dados em busca de comportamentos anômalos.

Neste contexto, ao empregar métodos supervisionados e não supervisionados, é possível abordar uma vasta gama de problemas, aprimorando a precisão e a eficiência de soluções em áreas tão diversas como saúde, finanças, entretenimento e transporte (Goodfellow; Bengio; Courville, 2016). Com o avanço contínuo da tecnologia, o aprendizado de máquina tende a desempenhar um papel ainda mais central na transformação digital e no desenvolvimento de novas soluções inovadoras.

2.3.2 *Large Language Models* (LLMs)

Os *Large Language Models* (Modelos de Linguagem de Grande Escala), são modelos de IA que foram treinados para compreender e gerar texto em linguagem natural de forma sofisticada. Esses modelos são uma evolução dos métodos tradicionais de processamento de linguagem natural (NLP, do inglês *Natural Language Processing*) e têm sido amplamente utilizados para diversas tarefas, como tradução automática, *chatbots*, resumo de textos, entre outras (Brown *et al.*, 2020).

Os LLMs são construídos utilizando arquiteturas de redes neurais profundas, especialmente os *Transformers*, que são estruturas que permitem que o modelo processe vastas quantidades de texto e aprenda padrões complexos de linguagem a partir de dados (Vaswani *et al.*, 2017). Ao serem expostos a bilhões de frases e documentos, os LLMs conseguem capturar as nuances do uso da linguagem, sendo capazes de produzir textos coerentes e com estilo próximo ao de um ser humano. Isso permite que esses modelos sejam aplicados em soluções que vão desde assistentes virtuais até a geração automática de conteúdos para sites e redes sociais.

O aprendizado desses modelos se dá através da análise de extensos conjuntos de dados textuais, que incluem livros, artigos e páginas da web. O modelo é treinado para prever a próxima palavra em uma sequência a partir de uma frase incompleta, baseando-se no conhecimento adquirido durante o treinamento. Esse método, conhecido como aprendizado não supervisionado, permite que os LLMs desenvolvam uma compreensão mais profunda das estruturas gramaticais e dos significados das palavras, além de contextos culturais (Goodfellow, Bengio e Courville, 2016). Por exemplo, ao receber a frase “O gato está em cima do. . .”, o LLM pode prever que a palavra mais provável para completar a frase seria “telhado” ou “sofá”, dependendo do contexto.

Os LLMs têm um impacto significativo em várias áreas, facilitando a interação entre humanos e computadores por meio da linguagem natural. Um exemplo prático é o uso de assistentes virtuais, como aqueles disponíveis em *smartphones* e dispositivos domésticos inteligentes, que conseguem responder a perguntas, realizar agendamentos e até contar piadas, aproveitando a capacidade dos LLMs de entender e gerar respostas apropriadas (Liu *et al.*, 2023). Além disso, esses modelos são amplamente utilizados em ferramentas de tradução automática, como o Google Translate, que se beneficiam da capacidade de compreender o contexto e adaptar as traduções entre diferentes idiomas, representando

um avanço em comparação aos métodos tradicionais que tratavam palavras de forma isolada (Vaswani *et al.*, 2017).

Outro campo transformado pelos LLMs é a análise de sentimentos e monitoramento de redes sociais, onde empresas utilizam esses modelos para examinar grandes volumes de comentários e postagens, identificando tendências e sentimentos em relação a produtos ou serviços. Essa análise proporciona uma tomada de decisão mais informada e uma resposta ágil a *feedbacks* de clientes (Radford *et al.*, 2019).

Apesar dos avanços, os LLMs enfrentam desafios significativos, especialmente no que diz respeito ao uso ético e à segurança. Treinados com grandes volumes de dados da internet, esses modelos podem reproduzir vieses presentes nos textos utilizados, o que implica que, sem supervisão adequada, podem gerar respostas que refletem preconceitos e informações incorretas (Bender *et al.*, 2021). Além disso, o treinamento de um LLM exige grandes recursos computacionais, como milhares de unidades de processamento gráfico (GPUs) e semanas de processamento contínuo, resultando em um alto consumo de energia e recursos financeiros (Strubell, Ganesh e McCracken, 2019). Isso levanta questões sobre a sustentabilidade e acessibilidade dessas tecnologias, visto que apenas grandes empresas e centros de pesquisa têm acesso a tais recursos.

O futuro dos LLMs aponta para um desenvolvimento ainda mais sofisticado, com métodos que possibilitem melhor controle sobre suas respostas e uma maior eficiência no uso de dados e recursos computacionais (Liu *et al.*, 2023). O objetivo é criar modelos não apenas poderosos, mas também mais seguros e éticos, reduzindo a disseminação de informações incorretas e a amplificação de vieses. Pesquisas em técnicas de *fine-tuning* (ajuste fino) e *prompt engineering* têm mostrado que é possível ajustar o comportamento dos LLMs para aplicações específicas, permitindo que um mesmo modelo seja adaptado para diferentes necessidades, desde atendimento ao cliente até suporte educacional (Brown *et al.*, 2020). Essa evolução pode representar um passo importante para democratizar o acesso a essas tecnologias, tornando-as úteis para pequenas empresas e iniciativas comunitárias. No entanto, é fundamental considerar os desafios relacionados a recursos e ética, garantindo que seu uso seja seguro e benéfico para a sociedade (Bender *et al.*, 2021).

2.3.3 Agentes

Nos últimos anos, a integração LLMs em sistemas de agentes inteligentes tem revolucionado a maneira como as máquinas interagem com os humanos. Agentes são programas de *software* que executam tarefas de forma autônoma, interagindo com o ambiente e tomando decisões com base nas informações recebidas (Russell e Norvig, 2020). Ao serem combinados com LLMs, esses agentes adquirem a capacidade de processar e gerar linguagem natural com um nível de sofisticação sem precedentes, tornando-se especialmente úteis em aplicações como assistentes virtuais, atendimento ao cliente e sistemas de suporte em diversas áreas (Brown *et al.* 2020).

A precisão com que os LLMs compreendem e geram texto permite que os agentes respondam a perguntas, entendam comandos complexos e realizem tarefas que anteriormente exigiam intervenção humana. Isso é possível devido à habilidade dos LLMs de captar o contexto e as nuances da linguagem, proporcionando respostas relevantes e coerentes em cada interação (Liu *et al.* 2023). Dessa forma, esses agentes se tornam intermediários poderosos entre os usuários e os sistemas computacionais.

Os agentes que utilizam LLMs combinam técnicas de processamento de linguagem natural (NLP) com habilidades de tomada de decisão e aprendizado. O LLM atua como o "cérebro linguístico" do agente, sendo responsável por interpretar as entradas dos usuários e formular respostas adequadas. Além disso, esses agentes podem ser integrados a outras fontes de informação, como bancos de dados, APIs de serviços externos e sensores de dispositivos físicos. Isso possibilita que eles realizem ações práticas, como reservar um voo, encontrar informações em tempo real ou controlar dispositivos em uma casa inteligente (Goodfellow, Bengio e Courville, 2016).

Por exemplo, um assistente virtual que utiliza um LLM pode receber um comando como: "Qual é a previsão do tempo para amanhã na minha cidade?". O LLM interpreta a solicitação, identifica que a tarefa envolve a consulta de dados meteorológicos e, então, acessa uma API de previsão do tempo para obter a resposta, que é apresentada ao usuário em linguagem natural (Russell e Norvig, 2020). A capacidade do agente de entender o contexto e realizar ações autônomas torna a interação com máquinas mais fluida e intuitiva.

A utilização de agentes equipados com LLMs está transformando diversos setores, sendo o atendimento ao cliente automatizado um dos exemplos mais notáveis. Empresas podem empregar agentes para responder perguntas frequentes, orientar clientes em processos de suporte e até mesmo realizar vendas de forma proativa, tudo por meio de conversas que parecem naturais e personalizadas. Essa abordagem não apenas melhora a experiência do usuário, mas também permite que as empresas escalem seus serviços de atendimento sem um aumento proporcional de custos (Liu *et al.*, 2023).

Na área de educação, os agentes com LLMs são utilizados para criar tutores virtuais capazes de explicar conceitos complexos aos estudantes, responder dúvidas e oferecer *feedback* sobre o desempenho em exercícios. Essas ferramentas se mostram particularmente úteis em ambientes de ensino à distância, onde a interação personalizada com um instrutor humano pode ser limitada (Brown *et al.*, 2020). Outro campo de aplicação relevante é a automação de tarefas em ambientes empresariais. Agentes que utilizam LLMs podem realizar análises de documentos, sintetizar relatórios, organizar reuniões e até mesmo gerar rascunhos de contratos legais, agilizando processos que antes eram demorados e burocráticos. Essa automação de tarefas administrativas libera os funcionários para atividades mais criativas e estratégicas, aumentando a produtividade geral das organizações, como discutido por Goodfellow, Bengio e Courville (2016).

Apesar das diversas vantagens oferecidas pelos agentes baseados em LLMs, eles também apresentam desafios significativos. Um dos principais problemas é a possibilidade de esses agentes reproduzirem vieses presentes nos dados de treinamento dos LLMs. Como os LLMs são treinados com grandes volumes de texto da internet, é possível que aprendam preconceitos e estereótipos, resultando em respostas inadequadas ou discriminatórias, conforme alertado por Bender *et al.* (2021). Dessa forma, é fundamental que as empresas que utilizam esses agentes implementem mecanismos de monitoramento e ajustes contínuos para mitigar esses problemas.

Outro desafio importante é garantir que os agentes sejam transparentes em suas operações, assegurando que os usuários saibam quando estão interagindo com um sistema automatizado, em vez de um ser humano. Essa transparência é especialmente relevante em contextos sensíveis, como na área da saúde, onde é imprescindível que o usuário compreenda que está interagindo com uma máquina e não com um profissional de saúde (Strubell, Ganesh e McCracken, 2019). A transparência é essencial para manter a confiança dos usuários e garantir que os agentes sejam utilizados de maneira ética.

O futuro dos agentes baseados em LLMs promete novas oportunidades de automação e interação em diversos contextos. A pesquisa atual se concentra em melhorar a eficiência energética dos LLMs, permitindo que modelos menores e mais acessíveis sejam utilizados em dispositivos móveis, o que poderia expandir ainda mais a aplicação desses agentes em situações do dia a dia, conforme apontado por Liu *et al.*(2023). Além disso, avanços em técnicas de *few-shot learning* (aprendizado com poucos exemplos) possibilitam que os agentes sejam rapidamente adaptados a novas tarefas e contextos, sem a necessidade de grandes volumes de dados de treinamento (Brown *et al.* 2020).

Com o desenvolvimento contínuo de tecnologias que aumentam a capacidade dos LLMs de interagir com diferentes tipos de sistemas e dispositivos, os agentes inteligentes têm o potencial de transformar setores inteiros, oferecendo suporte mais personalizado e eficiente a usuários em todo o mundo, como destacado por Goodfellow, Bengio e Courville (2016).

Neste sentido, agentes que utilizam LLMs representam um avanço significativo na inteligência artificial, possibilitando interações naturais entre humanos e máquinas e automatizando tarefas complexas. Com sua capacidade de compreender e gerar texto de forma autônoma, esses agentes estão mudando a maneira como os serviços são prestados e como as pessoas interagem com a tecnologia. No entanto, é essencial que seu desenvolvimento seja acompanhado de uma reflexão ética sobre os desafios e impactos sociais de sua implementação, conforme destacado por Bender *et al.* (2021).

Na Figura 1 é mostrada a estrutura básica de um agente, incluindo a capacidade de usar ferramentas e planejamento.

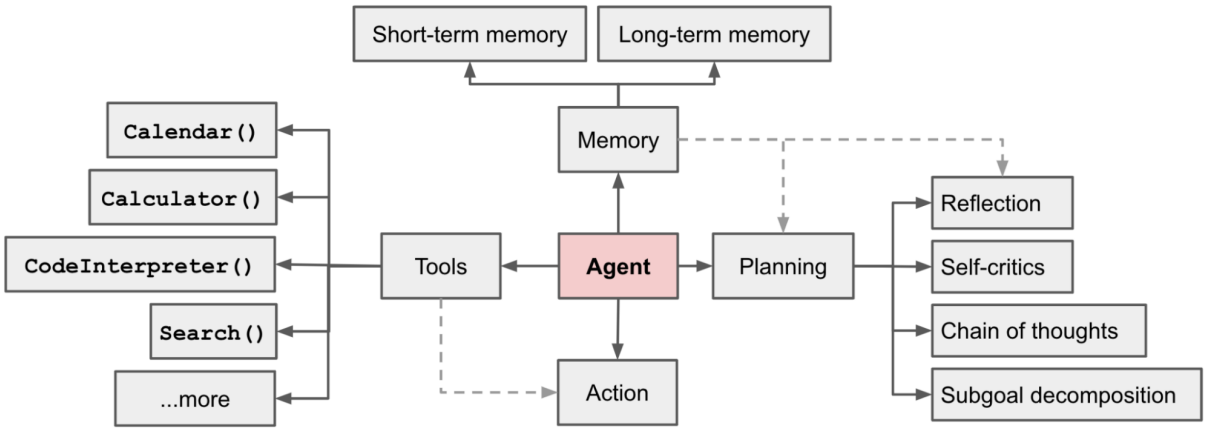

Figura 1 – Agentes

2.3.4 SOLID: Cinco Princípios para Melhorar a Qualidade do Código

Os princípios SOLID formam um conjunto de diretrizes voltadas ao desenvolvimento de *software* modular, escalável e fácil de manter, amplamente aplicados em programação orientada a objetos. O acrônimo SOLID foi introduzido por Michael Feathers como referência aos cinco princípios propostos por Robert C. Martin no final dos anos 1990, cujo objetivo era ajudar desenvolvedores a criar códigos mais organizados, reutilizáveis e menos propensos a erros e acoplamentos excessivos (Martin, 2003). Os cinco princípios formam as bases da boa prática na arquitetura de *software* e contribuem para a construção de sistemas robustos e de fácil manutenção.

O princípio da responsabilidade única (*Single Responsibility Principle* (SRP)), define que uma classe deve ter apenas uma razão para mudar, ou seja, deve possuir apenas uma responsabilidade central. A ideia é que cada classe ou módulo seja responsável por uma única funcionalidade, o que facilita a manutenção e reduz a complexidade do código (Martin, 2003). O SRP ajuda a evitar a duplicação de código e torna o sistema mais adaptável a futuras mudanças, uma vez que qualquer modificação em uma responsabilidade específica da aplicação se concentrará em uma única classe.

O princípio aberto/fechado (*Open/Closed Principle* (OCP)) estabelece que uma entidade de *software* (como uma classe, módulo ou função) deve estar aberta para extensão, mas fechada para modificação. Isso significa que o comportamento de uma classe pode ser estendido sem que seja necessário alterar seu código fonte, geralmente por meio do uso de herança ou implementação de interfaces (Martin, 2003). A adoção do OCP facilita a adição de novas funcionalidades sem comprometer o código existente, o que contribui para a estabilidade e flexibilidade do sistema.

Já o Princípio da Substituição de Liskov (*Liskov Substitution Principle* (LSP)), desen-

volvido pela cientista da computação Barbara Liskov, orienta que, se uma classe derivada substitui uma classe base, ela deve poder ser usada de forma intercambiável com a classe base, sem alterar a funcionalidade do programa (Liskov; Wing, 1994). Em outras palavras, classes filhas devem respeitar o comportamento da classe pai. Esse princípio é importante para garantir a corretude do sistema e a previsibilidade do código ao utilizar herança, ajudando a evitar erros decorrentes do uso inadequado de subclasses.

O Princípio da Segregação de Interfaces (*Interface Segregation Principle* (ISP)) sugere que uma classe não deve ser obrigada a implementar interfaces que não utiliza. Em vez de criar interfaces amplas e abrangentes, o princípio orienta o desenvolvimento de interfaces específicas para cada tipo de cliente, promovendo uma estrutura modular e flexível (Martin, 2003). A segregação de interfaces permite que classes dependam apenas dos métodos que realmente utilizam, reduzindo acoplamentos desnecessários e facilitando a manutenção do código.

Por fim, o princípio da inversão de dependência (*Dependency Inversion Principle* (DIP)), recomenda que classes de alto nível não dependam de classes de baixo nível, mas que ambas dependam de abstrações. Além disso, abstrações não devem depender de detalhes; em vez disso, os detalhes devem depender de abstrações (Martin, 2003). Esse princípio permite que módulos sejam desenvolvidos de maneira independente e reduz a dependência entre componentes, o que facilita a implementação de testes e a reutilização de código (Freeman; Pryce, 2009).

2.4 Arquiteturas de *software*

Diversas arquiteturas de software são discutidas neste projeto para que no final uma seja escolhida. Entre as opções avaliadas, destacam-se a arquitetura monolítica, a arquitetura de microsserviços, a arquitetura MVC e a arquitetura de plugins. Nos próximos parágrafos, descreveremos essas arquiteturas e as razões que levaram a sua consideração para este projeto.

2.4.1 Arquitetura Monolítica

A arquitetura monolítica é uma abordagem de desenvolvimento de software onde todos os componentes de uma aplicação são integrados e executados como uma única unidade. Esse estilo de arquitetura é amplamente utilizado, especialmente em sistemas legados ou projetos de menor porte, onde a simplicidade de gestão do código é uma vantagem importante (Fowler, 2002; Drucker *et al.*, 2019). Sua facilidade de implantação e a simplicidade no processo inicial de desenvolvimento fazem com que seja uma escolha comum em pequenas empresas ou projetos com escopo limitado (Richardson, 2018).

Na arquitetura monolítica, os módulos principais de uma aplicação, como a interface de usuário (UI), a lógica de negócio e o acesso a dados, estão todos interconectados em um

único bloco de código. Isso significa que a aplicação é compilada e implantada como uma única entidade, facilitando a implantação centralizada. Dessa forma, qualquer atualização exige recompilar e reinserir toda a aplicação, mesmo que a modificação seja mínima. Esse tipo de arquitetura permite comunicação interna direta entre as partes da aplicação, por meio de chamadas de métodos ou funções, o que proporciona uma interação mais rápida entre módulos, dispensando protocolos de rede e contribuindo para um desempenho mais eficiente em muitos casos. Em termos de escalabilidade, uma aplicação monolítica geralmente recorre à escalabilidade vertical, aumentando os recursos do servidor em que está hospedada, como memória ou CPU. No entanto, há limites para o quanto esse tipo de escalabilidade pode crescer.

A simplicidade no desenvolvimento é uma das principais vantagens da arquitetura monolítica, pois todas as partes do sistema estão centralizadas, o que facilita o desenvolvimento, teste e depuração. A criação de ambientes de teste é mais simples e o processo de implementação de ferramentas de integração e entrega contínuas (CI/CD) é menos complexo. Além disso, como a comunicação entre módulos ocorre internamente, a aplicação não sofre com a sobrecarga de comunicação em rede, o que se traduz em um desempenho inicial satisfatório.

Por outro lado, a manutenção e a evolução da aplicação podem se tornar complexas conforme o sistema cresce em tamanho e complexidade, já que mudanças em uma parte do código podem gerar impactos imprevistos em outras partes, aumentando o risco de *bugs*. A escalabilidade também é um problema, pois é difícil escalar partes específicas da aplicação de forma independente. Caso apenas um módulo exija mais recursos, todos os outros módulos também terão que ser escalados, gerando um uso ineficiente dos recursos. Em ambientes grandes, a implantação de sistemas monolíticos pode ser complexa, especialmente para equipes grandes, onde a coordenação de mudanças na mesma base de código pode levar a conflitos e atrasos. Outro ponto negativo é o longo tempo de inicialização em sistemas complexos, já que compilar, testar e implantar uma aplicação monolítica pode ser demorado e impactar a produtividade da equipe.

2.4.2 Arquitetura de microsserviços

A arquitetura de microsserviços é um estilo de desenvolvimento de *software* em que uma aplicação é composta por pequenos serviços independentes, cada um executando uma função específica. Esses serviços são desenvolvidos, implantados e escalados de forma independente, permitindo maior flexibilidade e escalabilidade em comparação com a arquitetura monolítica (Fowler; Lewis, 2014; Newman, 2015). A abordagem de microsserviços é especialmente vantajosa em projetos de grande escala, onde a divisão de responsabilidades facilita a manutenção e o desenvolvimento contínuo (Thönes, 2015).

As características da arquitetura de microsserviços incluem a criação de serviços independentes, onde cada um é responsável por uma tarefa específica, como gerenciamento de

usuários ou processamento de pagamentos. A comunicação entre esses serviços geralmente se dá por meio de APIs, que podem ser REST ou gRPC, ou por mensagens assíncronas através de filas, como RabbitMQ ou Kafka. Essa arquitetura permite a escalabilidade granular, onde apenas os serviços que precisam de mais recursos são escalados individualmente, ao contrário da escalabilidade global encontrada na arquitetura monolítica. Além disso, os microsserviços podem ser desenvolvidos e implantados de maneira independente, permitindo que equipes diferentes trabalhem em serviços separados e utilizem tecnologias variadas conforme necessário. Um dos benefícios notáveis dessa arquitetura é a tolerância a falhas: se um serviço falhar, isso não impede o funcionamento dos outros serviços, aumentando a resiliência do sistema.

As vantagens da arquitetura de microsserviços incluem uma escalabilidade flexível, onde componentes individuais podem ser ajustados conforme a demanda, e um desenvolvimento modular que facilita a divisão de tarefas entre várias equipes. Isso aumenta a produtividade, pois as equipes podem trabalhar em diferentes serviços sem interferências. A facilidade de adaptação a novas tecnologias é outro ponto positivo, já que cada microsserviço pode empregar a tecnologia mais adequada para resolver um problema específico. Além disso, a resiliência à falha é um destaque, permitindo que a aplicação continue operando mesmo que um dos microsserviços apresente problemas. A capacidade de desenvolver e implantar cada serviço de forma independente também resulta em ciclos de desenvolvimento e *deploy* mais rápidos, possibilitando entregas contínuas de novas funcionalidades.

Entretanto, a arquitetura de microsserviços não é isenta de desvantagens. A complexidade de gerenciamento de múltiplos serviços independentes pode ser um desafio, pois cada um precisa ser monitorado e mantido separadamente, o que pode resultar em sobrecarga operacional. As dependências de rede e a latência na comunicação entre serviços também podem ser problemáticas, especialmente para aplicações que demandam alto desempenho. Além disso, o gerenciamento de dados distribuídos se torna mais complicado, pois cada serviço pode ter seu próprio banco de dados, dificultando a consistência de dados e as transações distribuídas. O monitoramento e o d*debugging* de sistemas compostos por vários serviços exigem ferramentas específicas para rastrear falhas e garantir o funcionamento adequado. Por fim, a orquestração e o gerenciamento de dependências entre serviços em sistemas grandes podem ser desafiadores, exigindo uma coordenação cuidadosa.

Na implementação da arquitetura de microsserviços, alguns padrões e ferramentas são comumente utilizados. O uso de um API Gateway é uma prática comum, funcionando como um ponto de entrada para todas as requisições de clientes externos, que roteia chamadas para os serviços adequados, além de oferecer autenticação, autorização e balanceamento de carga. Serviços de mensageria, como RabbitMQ e Kafka, são frequentemente utilizados para comunicação assíncrona, permitindo que um serviço publique eventos para que outros possam consumi-los. Contêineres, como os oferecidos pelo Docker, são ampla-

mente utilizados para gerenciar a implantação e execução dos microsserviços, enquanto o Kubernetes atua como uma plataforma de orquestração que lida com escalabilidade e resiliência. Ferramentas como Istio ou Linkerd também são utilizadas para gerenciar a comunicação entre microsserviços, proporcionando recursos adicionais como monitoramento de tráfego e segurança.

Grandes empresas de tecnologia têm adotado a arquitetura de microsserviços para lidar com a escalabilidade e complexidade de seus sistemas. A Netflix, por exemplo, utiliza microsserviços para gerenciar sua plataforma de streaming de vídeo e autenticação de usuários, permitindo a escalabilidade de serviços individuais. A Amazon segmentou seu extenso sistema de *e-commerce* em microsserviços gerenciáveis, como catálogo de produtos e pagamentos. O Uber também emprega essa arquitetura para gerenciar diversas funcionalidades, como rastreamento de motoristas e gerenciamento de pagamentos.

2.4.3 Arquitetura MVC

A arquitetura MVC (*Model-View-Controller*) é um padrão de design amplamente utilizado no desenvolvimento de *software*, especialmente em aplicações *web*. O objetivo do MVC é separar a aplicação em três componentes principais: *Model*, *View* e *Controller*. Essa separação facilita o desenvolvimento, a manutenção e a escalabilidade da aplicação, permitindo que cada componente tenha uma responsabilidade bem definida (Buschmann *et al.*, 1996; Reenskog, 2014). A utilização do MVC é uma prática comum em frameworks de desenvolvimento web, como Django e Ruby on Rails, devido à sua capacidade de organizar o código de forma clara e modular (Freeman; Robson, 2014).

A arquitetura MVC é composta por três componentes principais: o Modelo, a Visão e o Controlador. O Modelo é responsável por gerenciar a lógica de dados da aplicação, representando os dados e suas regras de negócio. Ele interage com o banco de dados para realizar operações como consultas, atualizações e exclusões de dados, além de definir a estrutura dos dados que o sistema utilizará, como atributos e métodos de cada entidade. O Modelo também notifica as Visões sobre quaisquer mudanças nos dados, permitindo que elas sejam atualizadas conforme necessário.

A Visão, por sua vez, tem a responsabilidade de apresentar os dados ao usuário. Esta camada é a interface que o usuário vê e com a qual interage. O principal objetivo da Visão é exibir as informações de maneira clara e intuitiva, formatando os dados fornecidos pelo Modelo para que sejam apresentados corretamente. Importante destacar que a Visão não contém lógica de negócios nem interage diretamente com o banco de dados; sua função é meramente renderizar as informações recebidas do Modelo.

Por último, o Controlador atua como intermediário entre o Modelo e a Visão. Ele processa as requisições do usuário, invoca as ações necessárias no Modelo e decide qual Visão deve ser renderizada em resposta. O Controlador gerencia o fluxo de dados entre o

Modelo e a Visão, assegurando que as alterações nos dados sejam refletidas nas interfaces da aplicação.

O funcionamento básico da arquitetura MVC, conforme ilustrado na Figura 2, se dá através de um ciclo de interação entre o usuário e a aplicação. Primeiramente, o usuário interage com a Visão, como ao clicar em um botão ou enviar um formulário. O Controlador recebe essa requisição, processa a entrada do usuário e decide quais ações tomar, podendo envolver consultas ou atualizações de dados. O Controlador então solicita ao Modelo que realize as operações de lógica de negócios necessárias. Uma vez que o Modelo retorna os dados processados, o Controlador seleciona a Visão adequada para apresentar esses dados ao usuário. Finalmente, a Visão exibe as informações de forma adequada, completando o ciclo, que se repete à medida que novas requisições são feitas.

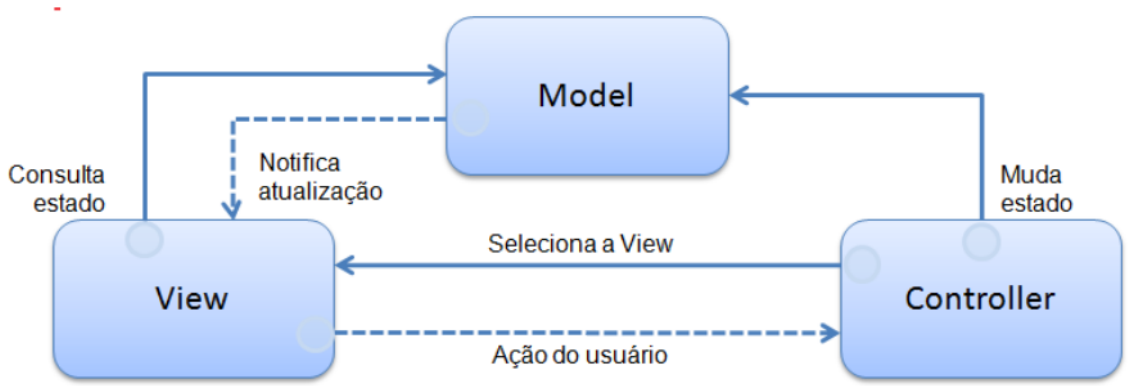

Figura 2 – Arquitetura MVC

Entre as vantagens da arquitetura MVC, destaca-se a separação de responsabilidades, que facilita a organização do código, já que o Modelo lida com os dados e a lógica de negócios, a Visão se encarrega da interface com o usuário e o Controlador coordena a interação entre ambos. Essa separação torna o código mais modular e compreensível. Além disso, a estrutura modular permite fácil manutenção e adição de novas funcionalidades, como um designer modificando a aparência da Visão sem impactar a lógica do Modelo ou do Controlador. A reutilização de código também é uma vantagem, visto que a lógica de negócios está centralizada no Modelo e a lógica de apresentação na Visão. Isso promove a utilização dessas partes em diferentes contextos da aplicação. Outro ponto favorável é a testabilidade, uma vez que a arquitetura MVC facilita a criação de testes unitários e de integração, especialmente para a camada de lógica de negócios.

Entretanto, a arquitetura MVC não é isenta de desvantagens. Para aplicações simples, sua adoção pode introduzir uma complexidade inicial desnecessária, já que a separação de componentes pode ser vista como um overhead em projetos muito pequenos. Além disso, o número de arquivos tende a aumentar, uma vez que cada Modelo, Visão e Controlador pode estar em arquivos separados, o que pode dificultar a gestão em projetos maiores se

não houver uma organização adequada. A comunicação entre os componentes também pode se tornar complexa em aplicações de grande escala, demandando uma orquestração cuidadosa do fluxo de dados para garantir que as alterações no Modelo sejam corretamente refletidas nas Visões.

2.4.4 Arquitetura de *pluguins*

A arquitetura de *plugins* é um padrão de design de *software* que permite que uma aplicação principal seja estendida por meio de módulos independentes chamados *plugins*. Esses *plugins* podem ser adicionados, removidos ou atualizados sem que seja necessário modificar o núcleo da aplicação (Gamma *et al.*, 1994; Martin, 2006). Essa abordagem oferece uma forma flexível de adicionar novas funcionalidades a um sistema, permitindo que a aplicação evolua sem que sua estrutura principal seja alterada (Fowler, 2012). A arquitetura de *plugins* é amplamente utilizada em sistemas que precisam acomodar uma ampla variedade de funcionalidades sem comprometer a estabilidade do núcleo do *software*.

A arquitetura de *plugins* é caracterizada pela extensibilidade, permitindo que funcionalidades sejam adicionadas de forma modular à aplicação. Essa abordagem possibilita a integração de novos recursos simplesmente através da adição de novos *plugins*, sem a necessidade de alterar o código-base. Outra característica importante é a modularidade; cada *plugin* funciona como um módulo independente, implementando funcionalidades específicas e interagindo com o núcleo da aplicação por meio de uma interface definida. Isso assegura que os *plugins* possam ser desenvolvidos e testados de forma isolada.

Além disso, a arquitetura de *plugins* apresenta um alto grau de desacoplamento, onde a lógica principal da aplicação é separada dos módulos de extensão. Isso significa que o núcleo da aplicação não precisa conhecer os detalhes de cada *plugin*, apenas como integrá-los através de uma API ou interface de *plugins*. O dinamismo também é uma característica fundamental; em muitas implementações, os *plugins* podem ser carregados ou descarregados em tempo de execução, permitindo que funcionalidades sejam ativadas ou desativadas conforme a necessidade do usuário, sem que a aplicação precise ser reiniciada.

A arquitetura de *plugins* é composta por três componentes principais. O primeiro deles é o Núcleo da Aplicação, que serve como a base do sistema, fornecendo as funcionalidades essenciais e definindo as interfaces ou pontos de extensão que os *plugins* devem seguir. Esse núcleo é responsável por gerenciar o ciclo de vida dos *plugins*, incluindo seu carregamento, inicialização e descarregamento. Além disso, ele oferece APIs para que os *plugins* interajam com o núcleo e entre si, geralmente incluindo um sistema de gerenciamento de *plugins* que localiza, carrega e registra os *plugins* disponíveis.

Os *plugins* são os módulos que estendem a aplicação, implementando funcionalidades específicas. Eles seguem uma interface padrão definida pelo núcleo, garantindo uma integração consistente. Cada *plugins* normalmente possui sua própria lógica de negócios e pode incluir desde extensões simples, como um novo formato de arquivo para exportação,

até funcionalidades mais complexas, como a integração com serviços externos. Os *plugins* podem ser desenvolvidos tanto pela equipe interna quanto por terceiros, permitindo que a comunidade ou parceiros contribuam para a evolução da aplicação.

Por fim, as Interfaces de *plugins* facilitam a comunicação entre o núcleo da aplicação e os *plugins*, definindo como estes devem ser estruturados e quais métodos ou APIs precisam implementar. Essas interfaces asseguram que todos os *plugins* sejam compatíveis com o núcleo e possam ser carregados de forma previsível.

A principal vantagem da arquitetura de *plugins* é a facilidade com que novas funcionalidades podem ser adicionadas à aplicação. Essa extensibilidade torna o sistema altamente adaptável a novas demandas dos usuários. Além disso, equipes de desenvolvimento podem trabalhar em diferentes *plugins* de maneira independente, sem interferir na lógica do núcleo, o que acelera o processo de desenvolvimento. Os *plugins* podem ser desenvolvidos, testados e implantados separadamente.

Outro ponto positivo é a customização, pois a arquitetura de *plugins* permite que usuários ou administradores ajustem as funcionalidades de acordo com suas necessidades, ativando apenas os *plugins* que são relevantes para eles. A manutenção do sistema também se torna mais simples, pois se um *plugin* apresentar problemas, ele pode ser desativado ou atualizado sem impactar o funcionamento do núcleo. Além disso, em muitos casos, a arquitetura de *plugins* permite que desenvolvedores externos contribuam com novos *plugins*, criando um ecossistema de funcionalidades que enriquece a aplicação original, como é comum em plataformas de software populares.

Apesar das vantagens, a arquitetura de *plugins* possui desvantagens. Uma delas é a complexidade na gerência de dependências, que cresce à medida que o número de *plugins* aumenta. Algumas extensões podem depender de outras para funcionar corretamente, resultando em problemas de compatibilidade. Também há a possibilidade de conflitos, uma vez que cada *plugin* é desenvolvido de forma independente, o que pode levar a erros ou comportamentos inesperados, especialmente se vários *plugins* tentarem modificar o mesmo comportamento ou recurso da aplicação.

Garantir a qualidade em um sistema que utiliza muitos *plugins* pode ser desafiador, já que cada um deles precisa ser testado em diversos cenários e combinações. Isso pode aumentar o esforço necessário para realizar testes abrangentes na aplicação. Além disso, a arquitetura de *plugins* pode introduzir uma sobrecarga de performance, principalmente se muitos *plugins* forem carregados simultaneamente ou se a integração entre eles não for otimizada, impactando a eficiência do sistema como um todo.

A arquitetura de *plugins* é amplamente utilizada em diversas áreas de software onde a extensibilidade é um requisito importante. Um exemplo notável são os Sistemas de Gerenciamento de Conteúdo (CMS), como WordPress, Joomla e Drupal, que permitem que desenvolvedores criem *plugins* para adicionar novas funcionalidades aos sites, como galerias de imagens e sistemas de e-commerce.

Outros exemplos incluem IDEs (Ambientes de Desenvolvimento Integrado), como Eclipse, Visual Studio Code e IntelliJ IDEA, que utilizam essa arquitetura para permitir que desenvolvedores personalizem e estendam a funcionalidade do ambiente de desenvolvimento. Os navegadores web, como Google Chrome e Mozilla Firefox, também permitem a instalação de extensões (ou *plugins*) que adicionam funcionalidades extras, como bloqueadores de anúncios e ferramentas de desenvolvimento. Além disso, alguns sistemas operacionais permitem a adição de *plugins* para funcionalidades específicas, como *codecs* de áudio e vídeo ou *drivers* de dispositivos.

Capítulo 3

Implementação

3.1 Concepção e Requisitos

Durante a fase de concepção deste *software*, idealizou-se um programa que integrasse múltiplas inteligências artificiais em uma única plataforma, projetada para atuar como um assistente pessoal abrangente. A proposta incluiu a implementação de funcionalidades inovadoras, inexistentes em tecnologias atuais, como o ChatGPT na versão *web*. Entre essas funcionalidades, destacam-se o reconhecimento de voz, o acesso à internet e uma interface de usuário amigável e intuitiva, todas pensadas para oferecer ao usuário uma experiência de interação mais prática e completa.

Na fase de levantamento de requisitos, após o processo de concepção e a definição inicial do projeto, foram identificados tanto requisitos funcionais quanto não funcionais para garantir a eficiência do sistema e uma boa experiência ao usuário.

Os requisitos funcionais englobam funcionalidades específicas para o sistema. Em primeiro lugar, ele deve dispor de um reconhecimento de voz eficaz, que seja capaz de funcionar corretamente mesmo em ambientes com ruído, compreendendo com precisão o que o usuário fala. Além disso, deve oferecer um sintetizador de voz que soe natural, proporcionando uma conversa mais realista e menos mecânica. O sistema precisa também ser inteligente, ou seja, apto a analisar e interpretar com precisão as intenções do usuário, respondendo de forma apropriada aos seus pedidos. Para aumentar sua utilidade, deve operar em múltiplos idiomas, incluindo inicialmente português, inglês, espanhol e alemão, com a possibilidade de expansão para outras línguas. Entre as outras funcionalidades destacadas estão a capacidade de enviar *e-mails* para contatos individuais ou múltiplos destinatários simultaneamente; a reprodução de músicas no Spotify com base em nome do artista ou título da faixa; a criação, edição e leitura de anotações; a consulta de valores de criptomoedas, como Bitcoin e Ethereum; e a exibição da data e hora atuais. Além disso, o sistema deve apresentar uma interface gráfica (*Graphical User Interface*) atraente e amigável, incluindo um visualizador de áudio, um terminal que exiba os resultados das operações realizadas e um botão central para iniciar a interação por voz.

Quanto aos requisitos não funcionais, foi estabelecido que o código do sistema deve ser limpo e fácil de entender, facilitando a manutenção e futuras modificações. A escalabilidade também é uma prioridade, sendo importante que o sistema permita a adição de novas funcionalidades de forma simples, o que requer uma arquitetura adequada e a aplicação dos princípios SOLID. Finalmente, o sistema deve respeitar a privacidade do usuário, garantindo que não haja margem para violações, preservando a segurança dos dados dos usuários.

Essas definições na concepção e levantamento de requisitos forneceram uma estrutura sólida para o desenvolvimento de um *software* funcional e robusto, capaz de atender às necessidades dos usuários e de se expandir para novas funcionalidades conforme necessário.

3.2 Arquitetura escolhida

Embora a arquitetura monolítica apresente benefícios, como simplicidade e rapidez no início do desenvolvimento, sua estrutura única pode resultar em desafios de manutenção e escalabilidade com o tempo. Dado que a escalabilidade é um requisito essencial para o projeto, a arquitetura monolítica foi descartada como opção.

A arquitetura de microsserviços oferece flexibilidade e escalabilidade, além de permitir que equipes de desenvolvimento trabalhem de maneira independente. No entanto, essa abordagem também traz desafios, como a complexidade operacional e a necessidade de ferramentas sofisticadas para monitoramento e gerenciamento. Por conta disso, a decisão de adotar essa arquitetura para o projeto foi descartada, uma vez que a equipe de desenvolvimento é composta por apenas um programador, o que tornaria implementação excessivamente complexa.

A arquitetura de *plugins* é uma abordagem poderosa para criar sistemas flexíveis e extensíveis, permitindo a adição de novas funcionalidades sem impactar a base do sistema. Ela é amplamente adotada em aplicações que exigem alto grau de personalização e adaptabilidade, como CMS, IDEs e navegadores. No entanto, a gestão da complexidade e da compatibilidade entre *plugins* é crucial para garantir a estabilidade e eficiência do sistema. Por essa razão, essa arquitetura não foi escolhida para o projeto em questão, considerando as particularidades e necessidades específicas do desenvolvimento em andamento.

Já a arquitetura MVC é um padrão de *design* robusto que promove a separação de responsabilidades, facilitando o desenvolvimento e a manutenção de aplicações, especialmente aquelas que envolvem interfaces com o usuário e interações complexas com dados, como as aplicações web. No entanto, é fundamental avaliar a complexidade do projeto antes de optar pelo MVC, pois a simplicidade é preferível. A escolha por esse padrão pode resultar em aplicações mais organizadas e preparadas para evoluções futuras, assegurando que o sistema permaneça modular e adaptável.

Por esses motivos, a arquitetura MVC foi escolhida para o projeto em questão, uma

vez que os requisitos não funcionais incluem simplicidade e facilidade de entendimento do software. Assim, a opção por arquiteturas de microserviços ou de *plugins* não seria adequada. Além disso, a arquitetura MVC é extremamente popular atualmente, o que a torna mais compreensível para a maioria dos programadores, fazendo dela a escolha ideal. Devido a grande quantidade de tecnologias diferentes envolvendo APIs, e diversas ferramentas, bem como a facilidade de uso e expertise do programador, a linguagem escolhida para desenvolver este *software* foi Python.

3.3 Configuração do ambiente de desenvolvimento

O ambiente integrado de desenvolvimento ou no caso IDE escolhido foi o Pycharm, pois para o programador é mais fácil de criar um ambiente virtual nesta IDE e também existem outras facilidades e familiaridades do programador com este programa. A versão utilizada do Pycharm foi a 2023.3.3, e a versão do interpretador Python escolhida foi a 3.12. Foi criado um novo projeto chamado *My_Assistant* e com ele foi gerado um ambiente virtual novo, para armazenar todas as bibliotecas utilizadas.

3.4 Implementação de funcionalidades

As funcionalidades descritas nos requisitos foram implementadas da seguinte forma:

- **Sistema de reconhecimento de voz:** Para implementar essa funcionalidade, optou-se pela biblioteca SpeechRecognition em conjunto com a API do Google. Essa escolha foi baseada na popularidade da biblioteca, que oferece alta precisão e qualidade no reconhecimento, além de não ter limites de uso. Um diferencial importante é sua capacidade de se ajustar a ruídos de fundo, atendendo assim ao requisito funcional de operar em ambientes ruidosos. Essa adaptabilidade torna o sistema mais robusto e eficiente em diversas situações.

- **Sintetizador de voz natural:** Durante o desenvolvimento, foram testados dois sintetizadores de voz: o da Google e o da ElevenLabs. Embora o sintetizador da Google ofereça um limite de uso maior, a qualidade da voz gerada não é tão natural quanto a da ElevenLabs. Para atender ao requisito de uma geração de voz mais natural e realista, foi escolhida a API da ElevenLabs, que proporciona uma experiência auditiva mais agradável e envolvente.

- **Sistema inteligente:** Inicialmente, várias abordagens para a arquitetura do sistema foram consideradas, visando a interpretação das solicitações do usuário. No entanto, ao longo do projeto, descobrimos a tecnologia de Agentes, que permite a integração de diversas IAs em um único ambiente. Para isso, optou-se por utilizar

o LLM da Groq, que possibilita uma reflexão e interpretação mais precisas das solicitações dos usuários, resultando em um sistema inteligente e responsivo. **Suporte a múltiplos idiomas:** Os LLMs contemporâneos oferecem suporte a diversos idiomas, incluindo os quatro selecionados nos requisitos. Contudo, é importante ressaltar que a mudança do interpretador de voz deve ser feita manualmente, uma vez que a API do Google para reconhecimento de voz está configurada para o idioma português por padrão. Essa flexibilidade permite atender a um público diversificado.

- **Envio de *E-mails*:** Para implementar a funcionalidade de envio de *e-mails*, foram escolhidas as bibliotecas smtplib e *e-mail*. Essas ferramentas proporcionam uma maneira eficaz de criar e enviar *e-mails* diretamente pelo agente, integrando a comunicação por *e-mail* ao sistema de maneira fluida.

- **Tocar músicas no Spotify:** A biblioteca Spotipy foi selecionada para desenvolver a funcionalidade de manipulação do Spotify. Isso permite ao agente acessar e controlar a reprodução de músicas, criando uma experiência musical interativa para o usuário.

- **Criação de anotações:** Para implementar a funcionalidade de criar e ler anotações, utilizou-se a biblioteca OS, que facilita o gerenciamento de arquivos. Essa ferramenta permite que o agente registre informações importantes e as recupere quando necessário.

- **Consulta de valores de criptomoedas:** Para a funcionalidade que obtém os valores de criptomoedas, foi escolhida a biblioteca e API coinmarketcapapi. Isso possibilita que o sistema forneça informações atualizadas sobre as flutuações do mercado de criptomoedas.

- **Data e hora atual:** Para implementar a funcionalidade que fornece a data e hora atuais, utilizou-se a biblioteca datetime. Essa ferramenta simples, mas eficaz, permite que o agente forneça informações temporais precisas aos usuários.

- **Interface Gráfica do Usuário:** Após avaliar várias bibliotecas gráficas, optou-se pela PyQt5 para o desenvolvimento da GUI principal do projeto. Para a geração de visualizações de áudio, foi utilizada a biblioteca Matplotlib, que oferece recursos robustos para a criação de gráficos e visualizações dinâmicas.

Essas implementações visam não apenas atender aos requisitos funcionais, mas também proporcionar uma experiência de usuário mais rica e interativa, garantindo a eficácia do *software*.

Capítulo 4

Resultados e Discussões

Neste capítulo, são apresentados os resultados obtidos com a aplicação desenvolvida, bem como uma análise sobre seu desempenho e os desafios enfrentados. A interface foi projetada para ser intuitiva, facilitando a interação do usuário com elementos como um visualizador de áudio, um terminal de *feedback* e um botão central para ativação por voz. Esses componentes permitem ao usuário um controle direto e transparente sobre o sistema, além de tornar a experiência de uso mais acessível e interativa. Ao longo da seção, são discutidas as funcionalidades implementadas, a usabilidade da interface, e a eficácia do assistente em interpretar e responder aos comandos. Adicionalmente, são abordados os principais obstáculos superados durante o desenvolvimento, com reflexões sobre o impacto social e ético dessa tecnologia.

4.1 Resultados

Conforme decidido na análise de requisitos, a aplicação apresenta uma interface de usuário intuitiva e amigável, projetada para facilitar a interação. A interface inclui um visualizador de áudio na parte superior e, logo abaixo, um terminal que exibe os resultados e o *feedback* da aplicação. Na parte inferior central da tela, há um botão grande que, ao ser clicado, ativa a função de escuta do programa, permitindo que o usuário inicie sua interação verbal. A Figura 3 ilustra a aplicação já finalizada, demonstrando o *layout* e a disposição dos elementos na tela.

O *output* do terminal pode variar amplamente. Por exemplo, ele pode exibir a mensagem "Fale agora", sinalizando ao usuário que é o momento de iniciar a sua fala. Além disso, o terminal também exibe "pensamentos"do agente inteligente, proporcionando uma melhor compreensão do que está ocorrendo em segundo plano durante a interação. Na Figura 4 é mostrado uma imagem de um exemplo de *output* do terminal.

Na Figura 4, é possível observar que a primeira mensagem exibida no terminal é "Fale agora", indicando ao usuário que ele pode começar a falar. Em seguida, o terminal registra a entrada do usuário com a frase *"Recognized text*: Diga-me quem foi Albert Einstein em

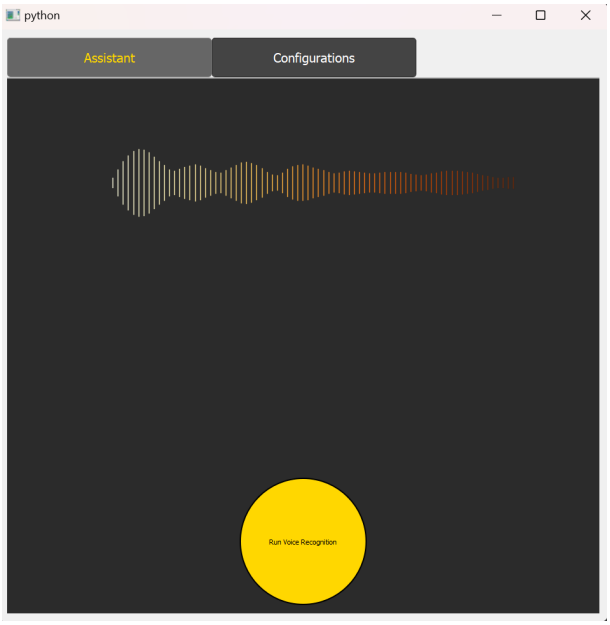

Figura 3 – Aplicação pronta

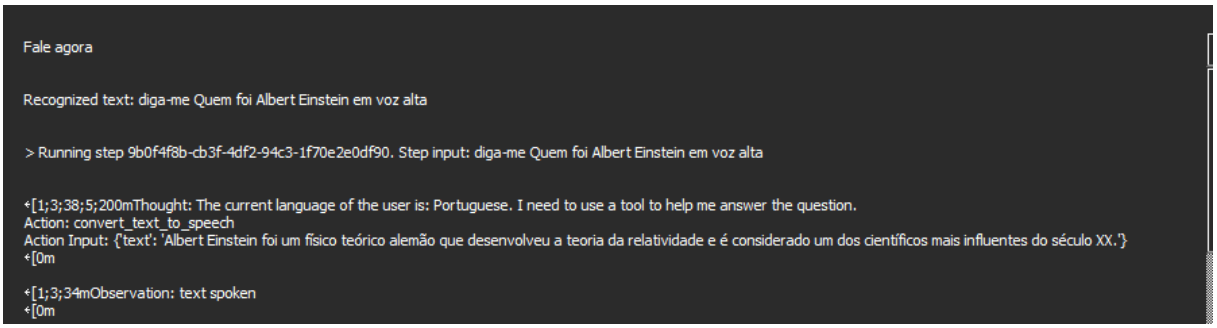

Figura 4 – Terminal

voz alta", que representa o pedido feito pelo usuário nessa interação. Após reconhecer corretamente o que foi dito, o agente processa a informação e gera um "pensamento"interno: "A linguagem do texto é português, e eu preciso de uma ferramenta para responder a esta pergunta."Como o usuário solicitou que o agente falasse em voz alta sobre Albert Einstein, o sistema compreende que deve utilizar a ferramenta *convert_text_to_speech*. Em resposta, o agente invoca essa ferramenta para converter a descrição sobre Albert Einstein em áudio. Uma vez que a conversão de texto em voz é concluída, o agente retorna uma mensagem de *feedback* ao terminal: *"text spoken"*. Com isso, a interação é finalizada de forma eficiente, demonstrando a funcionalidade integrada do sistema e sua capacidade de realizar tarefas complexas de maneira fluida e responsiva.

Na figura 5, é possível observar uma segunda interação com o agente, onde é perguntado de forma sequencial que horas são, que dia é hoje, qual o valor atual da Bitcoin, e por fim é solicitado que envie por *email* as informações obtidas. Como é possível observar na imagem o assistente pessoal realiza todas as solicitações com sucesso.

Figura 5 – Terminal segunda interação

Esses resultados destacam a eficácia da aplicação em atender aos requisitos funcionais, oferecendo uma experiência de usuário satisfatória e interativa.

4.1.1 Arquitetura implementada

Para o desenvolvimento do assistente virtual proposto, foi adotada uma arquitetura baseada no padrão MVC, juntamente com os princípios de design SOLID, com o objetivo de garantir modularidade, manutenibilidade e escalabilidade do sistema. Essa abordagem permite a fácil adição de novas funcionalidades no futuro, mantendo o sistema organizado e eficiente.

A arquitetura MVC divide o projeto em três componentes principais: *Models* (Modelos), *Views* (Visualizações) e *Controllers* (Controladores), com o objetivo de desacoplar a lógica de aplicação da interface com o usuário e do gerenciamento de dados. No contexto deste projeto, os *Models* são responsáveis pelo armazenamento e gerenciamento de dados. A pasta data contém as anotações feitas pelos agentes durante as interações com os usuários, incluindo histórico de conversas, preferências e outros dados persistentes que podem ser utilizados para melhorar futuras interações.

A interface do sistema é tratada pelas *Views*, responsáveis por apresentar informações ao usuário e captar entradas. O módulo UI gerencia a interface gráfica, que inclui um visualizador de áudio, um terminal para visualização de *output's* e uma aba de configurações onde os usuários podem selecionar o idioma e inserir chaves de API. Essa interface permite que o usuário interaja de forma intuitiva, enquanto o módulo *main* recebe as entradas da ui e direciona as ações para os controladores apropriados.

A Figura 6 mostra um diagrama ilustrando a interação entre o usuário e a aplicação bem como o fluxo de informação entre os módulos.

Os *Controllers* gerenciam a lógica central do sistema. O módulo *main* atua como o orquestrador que conecta todos os outros módulos, processando entradas da interface e coordenando a execução de funcionalidades.

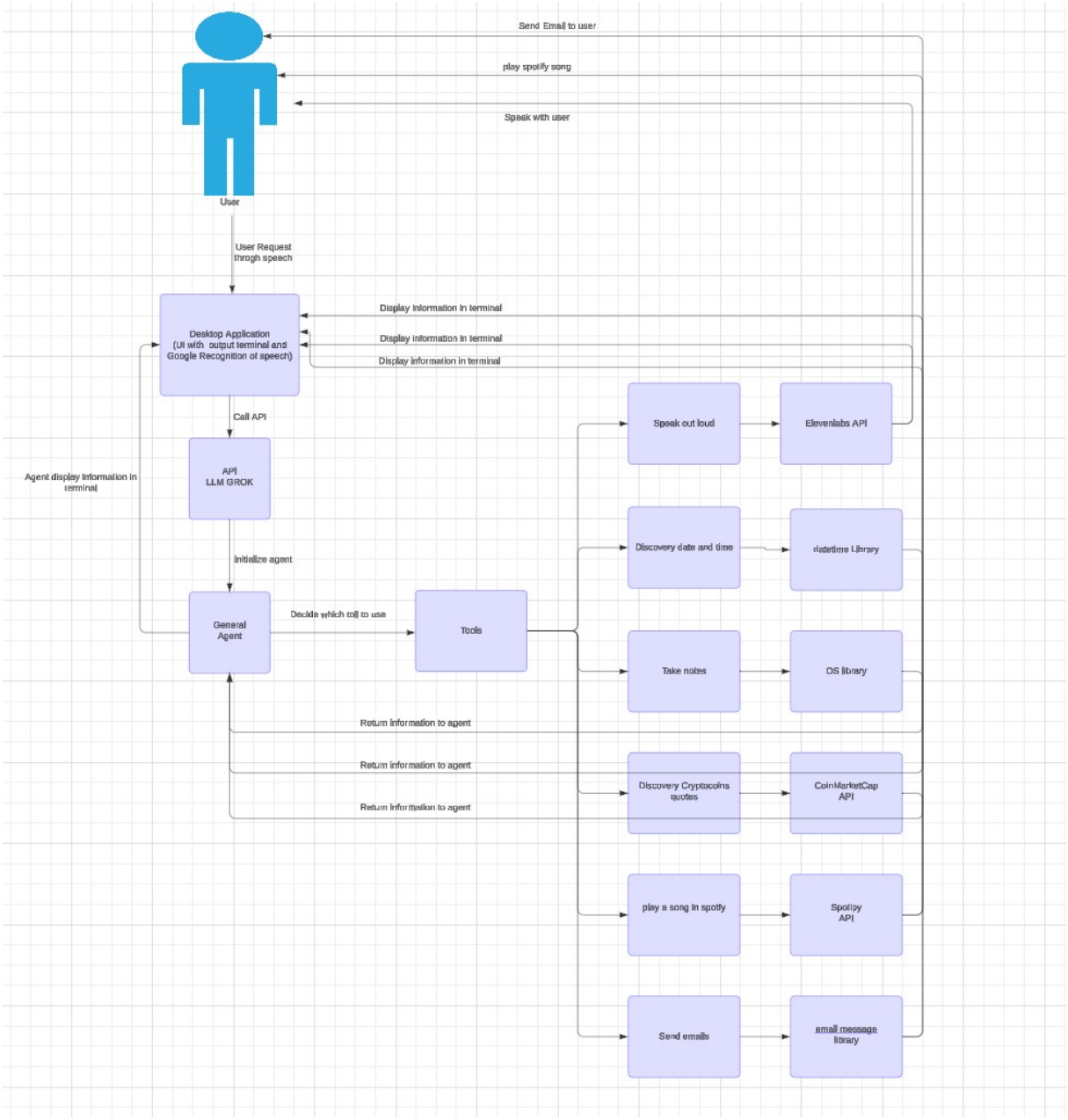

Figura 6 – Diagrama de interação entre o usuário e a aplicação

O módulo *agents* gerencia os agentes responsáveis por interagir com o usuário e delegar tarefas aos outros módulos, como *features*, *recognition_of_speech* e *speak*. O módulo *features* oferece funcionalidades adicionais, como envio de *e-mails*, obtenção de cotações de moedas e reprodução de músicas no Spotify. Já o módulo *recognition_of_speech* é responsável por converter comandos de voz em texto utilizando a API do Google *Speech-to-Text*, enquanto o módulo *speak* realiza a síntese de voz para respostas audíveis, usando a API da ElevenLabs.

O projeto segue a estrutura MVC onde o *main* funciona como ponto central de orquestração, conectando a interface (UI) e os módulos de controle *(agents, features, recog-*

nition_of_speech, speak) aos dados armazenados na pasta data. Essa arquitetura proporciona flexibilidade e organização, permitindo futuras expansões e melhorias no assistente virtual, mantendo o sistema eficiente e escalável.

A Figura 7 mostra um diagrama ilustrando a interação entre os módulos.

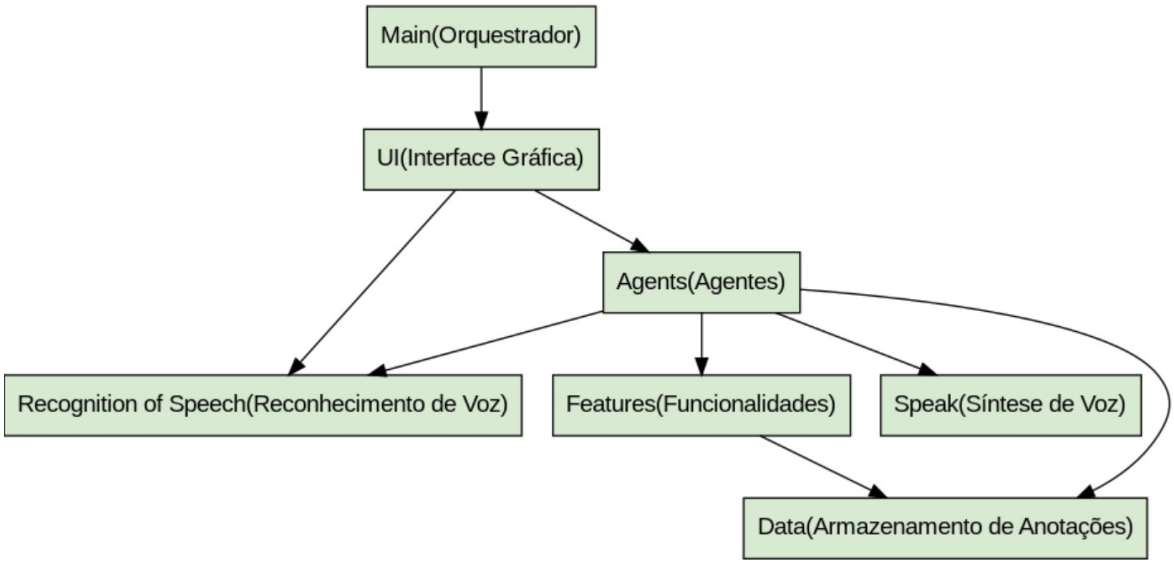

Figura 7 – Interação entre módulos

4.2 Desafios enfrentados

Um dos maiores desafios enfrentados neste projeto foi a necessidade de desenvolver toda a aplicação, incluindo o *front-end* (interface gráfica do usuário), o *back-end* (toda a lógica subjacente), os testes e o ciclo completo de desenvolvimento de *software*, sendo apenas um programador.

Essa tarefa exigiu que o código fosse mantido limpo e escalável, em conformidade com os princípios SOLID. Em ambientes corporativos uma aplicação dessa magnitude geralmente é desenvolvida por equipes de programadores ou, pelo menos, por um programador em cada área, como na equipe responsável pela interface gráfica.

No entanto, utilizando códigos *open source* e empregando LLMs e agentes para auxiliar no desenvolvimento, foi possível realizar todo o trabalho de forma independente.

Outra dificuldade enfrentada foi lidar com a imprevisibilidade dos LLMs, pois em algumas interações com a aplicação o sistema demonstrou um comportamento imprevisível nos seus *output's*. Em algumas ocasiões, a resposta do agente vinha incompleta, como se ele esquecesse de alguma das solicitações, outras vezes a ordem das respostas das solicitações eram diferentes. Esta imprevisibilidade é inerente dos LLMs, portanto é um problema difícil de resolver.

Embora já tenha sido mencionada a dificuldade em garantir que tudo estivesse alinhado aos princípios SOLID, é importante reiterar que, mesmo os maiores LLMs possuem um número limitado de *tokens* que podem ser gerados. Portanto, apesar da significativa assistência dos LLMs, muitas partes do código precisaram ser elaboradas manualmente para que atendessem aos requisitos, incluindo a conformidade com os princípios SOLID.

Outro desafio foi a criatividade na definição das funcionalidades do assistente. Como a perspectiva de um único autor é naturalmente limitada, tornou-se crucial garantir que o código fosse de fácil leitura e que possibilitasse a implementação de novas funcionalidades. Dessa forma, qualquer pessoa que tenha acesso ao código *open source* pode contribuir, implementando funcionalidades que atendam a suas necessidades específicas.

O tempo também se mostrou um desafio, uma vez que, além da elaboração do código, foi necessário desenvolver este trabalho e a documentação do projeto, o que demandou mais tempo do que a própria codificação. Além disso, o único programador envolvido neste projeto teve que conciliar suas atividades com um trabalho e outras obrigações não relacionadas ao desenvolvimento da aplicação.

Por fim, o conhecimento técnico também representou um desafio, uma vez que graduandos normalmente não possuem a experiência e o conhecimento necessários para desenvolver uma aplicação completa de forma autônoma. Muitos conhecimentos técnicos, como o conhecimento de Docker e Kubernetes, não eram dominados pelo programador. Assim, se o programador tivesse um conhecimento técnico mais abrangente, um maior número de funcionalidades poderia ter sido implementado.

4.3 Impacto social e ético

A adoção de assistentes pessoais que utilizam LLMs pode gerar impactos éticos e sociais variados, com potencial de influenciar a sociedade tanto positivamente quanto negativamente. À medida que esses modelos se tornam mais avançados no processamento e geração de linguagem natural, torna-se fundamental avaliar as repercussões dessa tecnologia, tanto no nível individual quanto no coletivo.

Uma das preocupações centrais está na privacidade dos usuários, já que assistentes pessoais precisam coletar e processar grandes volumes de dados para personalizar suas respostas. Esse processo pode implicar em riscos de segurança, especialmente em casos de vazamento de informações ou uso indevido dos dados coletados, destacando a necessidade de práticas robustas de proteção e de transparência sobre o tratamento das informações (Sharma *et al.*, 2020).

Além da privacidade, outra questão ética é a possibilidade de reprodução de vieses e estereótipos pela tecnologia. Como os LLMs são treinados em grandes bases de dados obtidas da internet, eles podem acabar incorporando preconceitos sociais presentes nesses dados e, com isso, refletir e até amplificar tais vieses nas interações com os usuários. Isso

evidencia a necessidade de investir em métodos de mitigação de vieses, permitindo que os modelos promovam uma interação mais inclusiva e equitativa (Bender *et al.*, 2021).

O impacto no mercado de trabalho também é significativo. A automação de tarefas, proporcionada pela IA em assistentes pessoais, tende a transformar o cenário profissional, aumentando a eficiência e a produtividade, mas também levando à substituição de funções tradicionais por sistemas automatizados. Embora isso possa provocar o desemprego em certos setores, é possível que novas funções e oportunidades surjam, especialmente nas áreas de manutenção e desenvolvimento tecnológico, exigindo que os profissionais se qualifiquem para atender à demanda emergente (Krafft *et al.*, 2020).

Por outro lado, os assistentes pessoais baseados em LLMs podem favorecer a acessibilidade e a inclusão de diferentes grupos sociais. Para pessoas com deficiência ou mobilidade reduzida, esses assistentes podem facilitar atividades cotidianas, como agendar compromissos ou controlar dispositivos domésticos por meio de comandos de voz. Contudo, é essencial que essa tecnologia seja desenvolvida para atender às necessidades de usuários com variadas capacidades e perfis socioeconômicos (Alam *et al.*, 2021).

Outro ponto crucial é a confiança e transparência, aspectos fundamentais para a aceitação da tecnologia. A maneira como os assistentes pessoais interagem e compartilham informações influencia diretamente a percepção pública sobre a inteligência artificial. Práticas transparentes e a comunicação clara sobre as capacidades e limitações dos modelos de linguagem são importantes para fortalecer a confiança e evitar expectativas irreais, promovendo uma aceitação mais consciente e informada (Walker *et al.*, 2022).

Os impactos sociais e éticos da implementação de assistentes pessoais baseados em LLM são amplos e complexos. Para que a tecnologia avance de forma responsável, é essencial que desenvolvedores e empresas considerem as implicações relacionadas à privacidade, equidade, empregabilidade e inclusão. Enfrentar esses desafios de maneira proativa não apenas beneficia os usuários, mas também contribui para o desenvolvimento de uma inteligência artificial mais ética e sustentável.

4.4 Sugestões para Trabalhos Futuros

Embora a aplicação desenvolvida já apresente uma série de funcionalidades, existem diversas possibilidades de aprimoramentos que podem ser implementadas para melhorar ainda mais a eficiência, a escalabilidade e a experiência do usuário. Essas sugestões visam tanto a evolução técnica quanto a ampliação do escopo da aplicação, tornando-a ainda mais adaptada às necessidades dos usuários e às demandas tecnológicas atuais.

4.4.1 Implementação de Docker e Kubernetes

Uma melhoria significativa seria a utilização do Docker para a criação de ambientes isolados e seguros para a execução da aplicação. O Docker permitiria encapsular todas as

dependências e configurações em contêineres, facilitando a replicação do ambiente de desenvolvimento e a portabilidade do software entre diferentes sistemas operacionais. Além disso, em uma futura versão *web* da aplicação, o uso de Kubernetes para a orquestração dos contêineres poderia otimizar a gestão de recursos, garantindo uma maior escalabilidade e resiliência da aplicação em ambientes de produção.

4.4.2 Desenvolvimento de uma Versão Web

A criação de uma versão web da aplicação utilizando *frameworks* como Flask ou Django, juntamente com tecnologias de *frontend* como HTML, CSS e JavaScript, poderia ampliar consideravelmente o acesso e a usabilidade do assistente pessoal. Uma versão *web* permitiria que usuários interagissem com a aplicação diretamente por meio de navegadores, sem a necessidade de instalação de software em seus dispositivos. Isso também possibilitaria a integração com outras APIs e serviços online, expandindo o leque de funcionalidades oferecidas.

4.4.3 Aprimoramento da Experiência do Usuário (UX)

A experiência do usuário é um aspecto crucial em assistentes pessoais, já que ela determina a satisfação e a facilidade de uso. Melhorias na interface gráfica GUI e na interação por voz poderiam tornar a aplicação ainda mais intuitiva e acessível. A adoção de técnicas mais avançadas de design de UX, como testes de usabilidade e análises de *feedback* de usuários, ajudaria a identificar pontos de melhoria na interface, proporcionando uma interação mais fluida e agradável.

4.4.4 Integração com APIs de Serviços Adicionais

Expandir as integrações com outras APIs de serviços populares poderia aumentar a versatilidade da aplicação. Por exemplo, a integração com sistemas de controle de dispositivos inteligentes (como assistentes domésticos e dispositivos IoT) poderia transformar o assistente em um verdadeiro *hub* de automação residencial. Além disso, adicionar a capacidade de acessar mais fontes de informação, como APIs de notícias, clima e sistemas financeiros, poderia enriquecer as respostas fornecidas aos usuários.

4.4.5 Implementação de Aprendizado Contínuo

A aplicação poderia se beneficiar da implementação de um sistema de aprendizado contínuo, que permitisse ao assistente adaptar-se progressivamente aos padrões e preferências dos usuários ao longo do tempo. Esse tipo de abordagem poderia melhorar a personalização das respostas e o atendimento às necessidades específicas de cada usuário.

No entanto, é importante considerar os desafios éticos e técnicos dessa implementação, especialmente no que diz respeito à privacidade e segurança dos dados.

4.4.6 Realização de Testes de Aceitação e Validação com Usuários Finais

Para validar a eficácia do assistente e identificar oportunidades de aprimoramento, uma sugestão importante é realizar testes de aceitação com usuários finais. Essa etapa permitiria colher *feedback* detalhado sobre o desempenho da aplicação em diferentes cenários de uso e sobre a satisfação dos usuários com as funcionalidades oferecidas. Além de fornecer informações valiosas para ajustes e melhorias, os testes de aceitação ajudam a garantir que a aplicação esteja alinhada às expectativas e necessidades de seus usuários.

4.4.7 Suporte a Mais Idiomas e Dialetos

Embora a aplicação inicial suporte múltiplos idiomas, é possível expandir esse suporte para incluir mais línguas e dialetos regionais. Isso aumentaria a acessibilidade da aplicação para um público ainda mais amplo, permitindo que pessoas de diferentes regiões e contextos culturais possam interagir de forma natural com o assistente. A integração de técnicas de processamento de linguagem natural (NLP) voltadas para idiomas de baixa representatividade também poderia ser uma contribuição significativa para a inclusão digital.

Capítulo 5

Conclusão

A conclusão deste projeto sintetiza o desenvolvimento e as contribuições de um assistente pessoal inteligente, baseado em modelos de linguagem de larga escala (LLMs) e estruturado em uma arquitetura MVC. O trabalho resultou em uma aplicação prática e funcional, integrando reconhecimento e síntese de voz, interface gráfica amigável e diversas funcionalidades automatizadas, como o envio de e-mails e busca de informações online. Utilizando princípios de design modular, como a arquitetura MVC e as práticas SOLID, o projeto mostrou-se escalável, de fácil manutenção e alinhado às demandas de segurança e privacidade dos usuários, fundamentais para sistemas que manipulam dados sensíveis.

Entre as contribuições mais relevantes, destaca-se a integração de LLMs para interpretação de linguagem natural, trazendo uma interação mais intuitiva entre humanos e máquinas e estabelecendo uma base para o aprimoramento contínuo dos assistentes pessoais. A implementação de uma estrutura modular e open source viabiliza a colaboração da comunidade e promove a democratização no desenvolvimento de assistentes, permitindo a expansão do projeto e o surgimento de novas funcionalidades com o apoio de outros desenvolvedores.

Em termos de futuro desenvolvimento, foram sugeridas melhorias que podem expandir a aplicação, incluindo a implementação de contêineres com Docker e Kubernetes, o desenvolvimento de uma versão web, a integração com dispositivos IoT e serviços adicionais, além do suporte a um número maior de idiomas. Essas sugestões não só ampliam o escopo de uso da aplicação, mas também reforçam seu compromisso com a usabilidade e acessibilidade, favorecendo a inclusão digital. Conclui-se que o projeto fornece uma plataforma robusta e adaptável, com grande potencial para evoluções futuras na área de assistentes pessoais inteligentes, fortalecendo o uso de IA na criação de interfaces acessíveis e eficientes.

Referências

ACHIAM, J. et al. Gpt-4 technical report. arXiv preprint arXiv:2303.08774, 2023.

ALAM, S. et al. Human-centered AI for people with disabilities: Principles and practices. AI & Society, v. 36, p. 195-207, 2021.

ANDERSON, J. Automating software deployment with Ansible. New York: O'Reilly Media, 2021.

ANDERSON, P. Monitoring software performance: tools and techniques. London: Springer, 2020.

BASILI, V. R.; RUMBAUGH, J. The role of software engineering in the evolution of software systems. In: Proceedings of the IEEE International Conference on Software Engineering. IEEE, 1994.

BASTIDE, R.; MARQUES, F.; PEREIRA, A. Testing software systems: a practical guide. 2. ed. New York: Wiley, 2018.

BISHOP, C. M. Pattern recognition and machine learning. New York: Springer, 2006.

BUTLER, D. (2016). Principles of Object-Oriented Design: SOLID Principles. Addison-Wesley.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. Pattern-oriented software architecture: A system of patterns. John Wiley & Sons, 1996.

BECK, K. Test-driven development: by example. Boston: Addison-Wesley, 2003.

BENDER, E. M.; GEBRU, T.; MCMILLAN-MAJOR, A.; MITCHELL, M. On the dangers of stochastic parrots: Can language models be too big? In: Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency (FAccT). New York: ACM, 2021.

BOOCH, G. Object-oriented analysis and design with applications. 3. ed. Boston: Addison-Wesley, 2007.

BROWN, L. Monitoring and maintaining software systems. London: Springer, 2020.

BROWN, R. Enhancing software functionality through maintenance. New York: O'Reilly Media, 2018.

CHAPMAN, A. Managing version control for software projects. 2. ed. London: Springer, 2010.

DIXON, J. Software testing and continuous quality improvement. New York: Auerbach Publications, 2010.

DRUCKER, P.; SMITH, J.; TAYLOR, R. Software Architecture Fundamentals. IEEE Press, 2019.

ELOUNDOU, T. et al. GPTs are GPTs: An early look at the labor market impact potential of large language models. arXiv preprint arXiv:2303.10130, 2023.

EL NAQA, I.; MURPHY, M. J. Image-guided radiation therapy and co-registration. CRC Press, 2015.

FEIGENBAUM, E. A.; BUCHANAN, B. G.; LEDERBERG, J. On generality and problem solving: A case study using the DENDRAL program. NASA-CR-123182, 1970.

FOWLER, M. Patterns of enterprise application architecture. Addison-Wesley, 2002.

FOWLER, M. Continuous delivery: reliable software releases through build, test, and deployment automation. Boston: Addison-Wesley, 2006.

FOWLER, M. Patterns of enterprise application architecture. Addison-Wesley, 2012.

FOWLER, M.; LEWIS, J. Microservices: a definition of this new architectural term. ThoughtWorks, 2014. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 20 out. 2024.

FREEMAN, E.; ROBSON, E. Head First Design Patterns: A Brain-Friendly Guide. O'Reilly Media, 2014.

FREEMAN, S.; PRYCE, N. Growing Object-Oriented Software, Guided by Tests. Addison-Wesley, 2009.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. Deep Learning. Cambridge: MIT Press, 2016.

GRAHAM, D.; DORLING, A.; HAYES, B. FOUNDATIONS OF SOFTWARE TESTING: ISTQB CERTIFICATION. 4. ed. LONDON: CENGAGE LEARNING, 2009.

GULLIFORD, S. L. Modelling of Normal Tissue Complication Probabilities (NTCP): Review of application of machine learning in predicting NTCP. In: Machine Learning in Radiation Oncology. Cham: Springer, 2015. p. 277-310.

HE, Q.; LI, X.; CAI, B. Graph neural network recommendation algorithm based on improved dual tower model. Journal of AI Research, v. 14, n. 1, p. 3853, 2024.

JOHNSON, L. Software maintenance and support. San Francisco: Pearson, 2020.

JORDAN, M. I.; MITCHELL, T. M. Machine learning: Trends, perspectives, and prospects. Science, v. 349, n. 6245, p. 255-260, 2015.

JONES, C. Software assessments, benchmarks, and best practices. 2. ed. New York: Addison-Wesley, 2009.

JONES, C. Software assessments, benchmarks, and best practices: using benchmark metrics to improve project performance. New York: Addison-Wesley, 2014.

JONES, R. Backup strategies for software deployment. Boston: Pearson, 2020.

KREBS, B. The security and privacy implications of continuous deployment. In: Proceedings of the Conference on Software Engineering, 2019.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. Nature, v. 521, n. 7553, p. 436-444, 2015.

LEWIN, K. Action research and minority problems. Journal of Social Issues, v. 2, n. 4, p. 34-46, 2011.

LIGHTHILL, J. Artificial intelligence: a general survey. Artificial Intelligence: a paper symposium. Science Research Council, 1972.

LISKOV, B.; WING, J. M. A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS), v. 16, n. 6, p. 1811–1841, 1994.

LIU, P.; YANG, W.; ZHANG, Y. A survey on large language models. Journal of AI Research, v. 45, n. 2, p. 102-115, 2023.

MARTIN, R. C. Agile software development: Principles, patterns, and practices. Prentice Hall, 2003.

MCCARTHY, J. WHAT IS ARTIFICIAL INTELLIGENCE? Stanford University, 2007.

MENG, X. Cross-domain information fusion and personalized recommendation in artificial intelligence recommendation system based on mathematical matrix decomposition. Scientific Reports, v. 14, n. 1, p. 7816, 2024.

MILLER, D. ADAPTIVE MAINTENANCE IN SOFTWARE ENGINEERING. BOSTON: CRC PRESS, 2019.

MOOR, J. The Dartmouth College Artificial Intelligence Conference: The next fifty years. AI Magazine, v. 27, n. 4, p. 87–89, 2006.

NEEDHAM, By Dr RM. Lighthill report: Artificial intelligence: a paper symposium.

NEWELL, A.; SIMON, H. A. The logic theory machine: a complex information processing system. IRE Transactions on Information Theory, v. IT-2, n. 3, p. 61-79, 1956.

NEWMAN, S. Building microservices: designing fine-grained systems. O'Reilly Media, 2015.

NG, Andrew. Machine Learning. Stanford University, 2012. Disponível em: https://www.coursera.org/learn/machine-learning. Acesso em: 25 out. 2024.

NICHOLS, M. Software testing: A craftsman's approach. 4. ed. New York: Auerbach Publications, 2010.

NIELSEN, J. Usability engineering. San Francisco: Morgan Kaufmann, 1993.

OWLER, M. Patterns of enterprise application architecture. Addison-Wesley Professional, 2002.

PETERS, A. Deployment planning for software projects. New York: CRC Press, 2018.

PRESSMAN, R. S.; MAXIM, B. R. Software engineering: a practitioner's approach. 9. ed. New York: McGraw-Hill Education, 2014.

RADFORD, A.; et al. Language models are unsupervised multitask learners. OpenAI, 2019.

REENSKOG, M. Model-View-Controller (MVC) architecture pattern for JavaScript frameworks. IEEE Software, v. 31, n. 1, p. 97-100, 2014. Disponível em: https://ieeexplore.ieee.org/document/6740859. Acesso em: 20 out. 2024.

RICHARDSON, C. Microservices patterns: with examples in Java. Manning Publications, 2018.

RUBIN, K. S. Essential Scrum: A Practical Guide to the Most Popular Agile Process. 2. ed. Upper Saddle River: Addison-Wesley, 2011.

RUSSELL, S.; NORVIG, P. Artificial Intelligence: A Modern Approach. 4. ed. Upper Saddle River: Pearson, 2020.

SHARMA, G. et al. Privacy and data protection in the age of AI. Computer Law & Security Review, v. 36, n. 5, p. 101485, 2020.

SMITH, A. PREVENTIVE MAINTENANCE STRATEGIES FOR SOFTWARE SYSTEMS. CHICAGO: WILEY, 2021.

SMITH, D. Software testing and validation. Chicago: McGraw-Hill, 2019.

SOMMERVILLE, I. Software engineering. 10. ed. Boston: Addison-Wesley, 2010.

STRUBELL, E.; GANESH, A.; MCCRACKEN, E. Energy and policy considerations for deep learning in NLP. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL). Florence: ACL, 2019.

TAMKIN, A. et al. Understanding the capabilities, limitations, and societal impact of large language models. arXiv preprint arXiv:2102.02503, 2021.

TAYLOR, M. CONTINUOUS DEPLOYMENT IN AGILE ENVIRONMENTS. SAN FRANCISCO: JOSSEY-BASS, 2018.

TAYLOR, J. User support in software maintenance. Los Angeles: Sage Publications, 2019.

THÖNES, J. Microservices. IEEE Software, v. 32, n. 1, p. 116-116, 2015. Disponível em: https://ieeexplore.ieee.org/document/7030215. Acesso em: 20 out. 2024.

TURING, A. Computing machinery and intelligence. Mind, 1950.

VASWANI, A. et al. Attention is all you need. In: Advances in Neural Information Processing Systems, v. 30, p. 5998-6008, 2017.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, Ł.; POLOSUKHIN, I. Attention is All You Need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17), Long Beach, CA, USA, 2017. Disponível em: https://arxiv.org/abs/1706.03762. Acesso

em: 14 nov. 2024.

WEIZENBAUM, J. Computer Power and Human Reason: From Judgment to Calculation. New York: W. H. Freeman and Co, 1976.

WEIZENBAUM, J. ELIZA - A computer program for the study of natural language communication between man and machine. Communications of the Association for Computing Machinery, v. 9, n. 1, p. 36-45, 1966.

WILSON, T. User training and documentation strategies. Los Angeles: Sage Publications, 2019.

YANG, H.; CHEN, F.; ALIYU, S. Modern software cybernetics: New trends. Journal of Systems and Software, v. 124, p. 169-186, 2017.

ZOU, J.; HAN, Y.; SO, S. Overview of artificial neural networks. In: Artificial neural networks: methods and applications. p. 14-22, 2009.