MINICURSOS

DO I CONGRESSO DE DESENVOLVIMENTO DE SISTEMAS E COMPUTAÇÃO - CODEC

FAPEPI
FUNDAÇÃO DE AMPARO À PESQUISA DO ESTADO DO PIAUÍ

LAPEC
{ Laboratório de Pesquisas e Estudos em Computação }

INSTITUTO FEDERAL
Piauí

MINICURSOS DO I CONGRESSO DE DESENVOLVIMENTO DE SISTEMAS E COMPUTAÇÃO (CODEC 2025)

ORGANIZAÇÃO

IALLEN GÁBIO DE SOUSA SANTOS

RICARDO MOURA SEKEFF BUDARUICHE

MAYLLON VERAS DA SILVA

WANDERSON DE VASCONCELOS RODRIGUES DA SILVA

JONATHAS JIVAGO DE ALMEIDA CRUZ

MAYKOL LIVIO SAMPAIO VIEIRA SANTOS

JEFERSON DO NASCIMENTO SOARES

MARCOS RAMON PAULINO RESENDE

TAMIRES ALMEIDA CARVALHO

Porto Alegre

Sociedade Brasileira de Computação – SBC

2025

Dados Internacionais de Catalogação na Publicação (CIP)

C749 Congresso de Desenvolvimento de Sistemas e Computação (1. : 22 – 23 setembro 2025 : Piripiri)

Minicursos do CODEC 2025 [recurso eletrônico]. Organização: Iallen Gábio de Sousa Santos (et al.) – Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2025.

109 p. : il. : PDF; 10 MB

Modo de acesso: World Wide Web.
Inclui bibliografia
ISBN 978-85-7669-661-2 (e-book)

1. Computação – Brasil – Evento. 2. Sistemas computacionais. 3. Desenvolvimento de sistemas. I. Santos, Iallen Gábio de Sousa Santos. II. Sociedade Brasileira de Computação. VI. Título.

CDU 004(063)

Ficha catalográfica elaborada por Annie Casali – CRB-10/2339

Biblioteca Digital da SBC – SBC OpenLib

Sociedade Brasileira de Computação
Av. Bento Gonçalves, 9500
Setor 4 | Prédio 43.412 | Sala 219 | Bairro
Agronomia Caixa Postal 15012 | CEP 91501-970
Porto Alegre - RS
Fone: (51) 992526018
sbc@sbc.org.br

SINOPSE EM PORTUGUÊS

O Livro de Minicursos do Congresso de Desenvolvimento de Sistemas e Computação – CODEC 2025, realizado em Piripiri-PI e sediado pelo Instituto Federal do Piauí (IFPI), reúne conteúdos derivados dos minicursos apresentados durante o evento, refletindo temas atuais e práticos das áreas de tecnologia e desenvolvimento de sistemas. O primeiro capítulo, “Construindo APIs Escaláveis com Flask, Firestore e Render”, aborda a criação e implantação de APIs RESTful com Python, destacando o uso de arquiteturas modulares, bancos NoSQL e deploy automatizado na nuvem. Em “Desenvolvendo um Sistema Multi-Agentes para Pesquisa Científica com LangGraph”, o segundo capítulo explora o uso de agentes inteligentes e frameworks de IA para automatizar processos de busca, validação e análise de dados em pesquisas científicas. No terceiro capítulo, “Explorando Testes End-to-End com Playwright”, são apresentados conceitos e práticas de automação de testes de software, evidenciando a importância da qualidade e da confiabilidade no desenvolvimento de aplicações web. O quarto capítulo, “Sistemas Embarcados: Uma Abordagem Prática com BitDogLab”, propõe uma introdução experimental ao universo dos sistemas embarcados, unindo fundamentos de eletrônica e programação em atividades práticas. Por fim, “Introdução ao Git e GitHub: Controle de Versão na Prática” oferece uma visão acessível das ferramentas essenciais para o versionamento de código e colaboração em projetos de software. Com temas que vão da programação em nuvem à automação e ao hardware educacional, esta coletânea traduz o espírito do CODEC 2025: promover a integração entre ensino, pesquisa e prática profissional, estimulando o aprendizado ativo e a inovação tecnológica.

SINOPSE EM INGLÊS

The Book of Short Courses of the Congress on Systems and Computing Development – CODEC 2025, held in Piripiri, Piauí, and organized by the Federal Institute of Piauí (IFPI), compiles the extended versions of the short courses presented during the event. The volume encompasses contemporary and practice-oriented topics that reflect the current challenges and innovations in the fields of computing, software engineering, and systems development. The opening chapter, “Building Scalable APIs with Flask, Firestore, and Render”, presents a comprehensive approach to the design and deployment of RESTful APIs using Python, emphasizing modular architectures, NoSQL data persistence, and continuous cloud deployment practices. The second chapter, “Developing a Multi-Agent System for Scientific Research with LangGraph”, discusses the application of artificial intelligence through intelligent agents and multi-agent frameworks to automate scientific data retrieval, validation, and analysis processes. The third chapter, “Exploring End-to-End Testing with Playwright”, addresses the theoretical and practical foundations of software testing, highlighting automation strategies that ensure quality assurance and reliability in web applications. The fourth chapter, “Embedded Systems: A Practical Approach with BitDogLab”, introduces the fundamentals of embedded systems through experimental activities that integrate hardware and software concepts in an accessible educational context. Finally, “Introduction to Git and GitHub: Version Control in Practice” provides a structured overview of version control principles and collaborative development workflows, focusing on the practical use of Git and GitHub in academic and professional environments. Collectively, the chapters illustrate the interdisciplinary nature of computing and software development. The CODEC 2025 short course collection embodies the event’s mission to promote the integration of teaching, research, and professional practice, fostering active learning, collaboration, and technological innovation.

Sumário

Capítulo 1

Construindo APIs escaláveis com Flask, Firestore e Render

Antônio Felipe Passos da Silva e Maykol Livio Sampaio Vieira Santos

Abstract

This short course teaches how to develop and deploy RESTful APIs using Python, Flask, Google Firestore, and Render. Participants will learn to implement a modular architecture (MVC and a Service Layer) to create scalable and maintainable code. The content covers everything from fundamental API concepts to data persistence with NoSQL and continuous deployment to the cloud, all based on a hands-on project building example API.

Resumo

Este minicurso ensina a desenvolver e implantar APIs RESTful utilizando Python, Flask, Google Firestore e Render. Os participantes aprenderão a implementar uma arquitetura modular (MVC e Camada de Serviços) para criar código escalável e de fácil manutenção. O conteúdo cobre desde os conceitos fundamentais de APIs até a persistência de dados com NoSQL e o deploy contínuo na nuvem, tudo baseado em um projeto prático de uma API exemplo.

1.1. Contexto e Relevância

No atual ecossistema de desenvolvimento de software, as Interfaces de Programação de Aplicações (APIs) são componentes arquitetônicos cruciais, atuando como a fundação para a construção de aplicações web e móveis modernas, bem como sistemas distribuídos [Andrade 2024]. Elas funcionam como "pontes"que permitem que diferentes softwares se comuniquem e troquem informações de maneira padronizada [Brito 2020]. A proficiência na construção de APIs robustas, escaláveis e bem estruturadas é uma competência altamente valorizada e demandada no mercado de tecnologia [Rodrigues 2025].

No entanto, desenvolvedores iniciantes frequentemente enfrentam desafios na transição do conhecimento teórico para a aplicação prática em projetos de software, especialmente aqueles que exigem organização e preparação para produção. Este minicurso tem

como objetivo preencher essa lacuna, oferecendo um roteiro pragmático que abrange todo o ciclo de vida de uma API, desde a configuração inicial do ambiente até a implantação na nuvem. A abordagem é fundamentada na Aprendizagem Baseada em Projetos (ABP), utilizando como guia uma API real desenvolvida em um contexto de extensão acadêmica, o que confere ao conteúdo um caráter prático e aplicado, essencial para alinhar o aprendizado às demandas autênticas do desenvolvimento de software.

1.2. Justificativa e Tecnologias Escolhidas

A escolha das tecnologias para este projeto é estratégica e alinhada com as necessidades do mercado. O Flask (disponível em https://flask.palletsprojects.com/), um microframework Python, foi selecionado por sua leveza e flexibilidade [Silva 2019], permitindo uma demonstração clara de padrões de arquitetura sem a complexidade de um framework completo. O padrão MVC promove a separação de responsabilidades, um princípio que facilita a manutenção e a testabilidade do código [Valente 2020]. A adição de uma camada de Serviços, por sua vez, resulta em um projeto com maior desacoplamento e facilidade de testes.

Para a persistência de dados, foi escolhido o Google Firestore (disponível em https://cloud.google.com/firestore), um banco de dados NoSQL gerenciado que oferece flexibilidade e escalabilidade para aplicações web e móveis [Google s.d.-a; Towards Data Science 2022]. Por fim, a plataforma Render (disponível em https://render.com/) simplifica o processo de implantação contínua (CI/CD), um conceito crucial para a entrega ágil e segura de software em ambientes de produção [Render s.d.; Neupane 2024; ProgrammingKnowledge 2025; TestDriven.io 2022].

Este capítulo, que serve como fundamentação teórica para o minicurso, está organizado em três módulos principais, refletindo a progressão do aprendizado: (1) Fundamentos das APIs REST e do Framework Flask, (2) Padrões de Arquitetura e Persistência de Dados, e (3) Preparação para Produção e Implantação Contínua. O objetivo é fornecer um embasamento robusto para que os participantes não apenas sigam o projeto prático, mas compreendam os princípios subjacentes a cada decisão técnica.

Este capítulo integra o material de apoio de um minicurso prático a ser ministrado no evento científico CODEC-PI, com o objetivo de aproximar a fundamentação teórica da prática de desenvolvimento de APIs. Para complementar a leitura, disponibilizamos um repositório público no GitHub contendo os códigos e exemplos utilizados durante o minicurso: `https://github.com/Felipe-Silva7/culinary_api_flask`.

1.3. A Arquitetura REST (Representational State Transfer)

Antes do estabelecimento de padrões como o REST, a comunicação entre sistemas distribuídos era frequentemente dominada por arquiteturas mais rígidas e complexas, como RPC (Remote Procedure Call) e SOAP (Simple Object Access Protocol). Em resposta a essa complexidade, a arquitetura REST emergiu como um estilo arquitetural mais leve e padronizado, aplicando os princípios de sucesso da própria web para a comunicação entre aplicações.

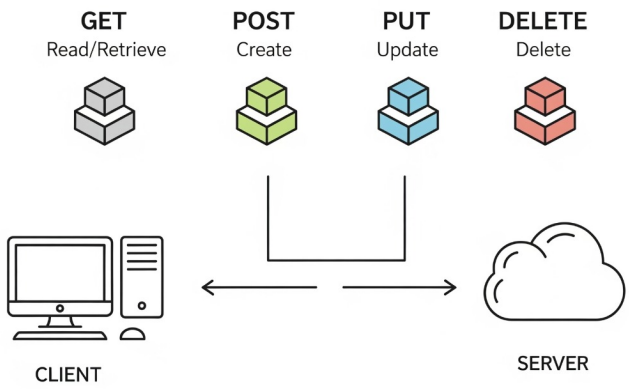


Figura 1.1. Ilustração do modelo Cliente-Servidor e a utilização dos verbos HTTP para manipulação de recursos na arquitetura REST.

A Figura 1.1 demonstra visualmente dois dos pilares do REST: a clara separação entre o cliente (Client) e o servidor (Server) e a forma como a comunicação é padronizada através dos verbos do protocolo HTTP para manipular recursos. Essa abordagem, concebida por Roy Fielding em sua tese de doutorado [Gran Cursos Online 2022], não define um protocolo, mas sim um conjunto de restrições de arquitetura que visam a escalabilidade, simplicidade e confiabilidade [Red Hat 2023]. Em sua essência, uma API RESTful trata qualquer entidade como um **recurso** (resource) que pode ser criado, lido, atualizado ou excluído por meio de uma interface uniforme [Gran Cursos Online 2022].

Para que um sistema seja considerado RESTful, ele deve aderir a um conjunto de restrições arquiteturais, das quais as principais são:

- **Cliente-Servidor:** Separação total das responsabilidades da interface do usuário (cliente) e do armazenamento de dados (servidor) [Gran Cursos Online 2022]. Essa separação permite que ambos evoluam de forma independente.
- **Comunicação Sem Estado (Stateless):** Cada requisição do cliente para o servidor deve conter toda a informação necessária para ser processada. O servidor não armazena o estado da sessão do cliente, o que simplifica o design e melhora a escalabilidade horizontal [Gran Cursos Online 2022].

- **Cacheability:** As respostas do servidor podem ser marcadas como "cacheáveis"ou "não cacheáveis". Isso permite que clientes e intermediários reutilizem respostas, melhorando significativamente a performance e a eficiência da rede.
- **Interface Uniforme:** A manipulação dos recursos ocorre por meio de um conjunto padronizado e limitado de operações, que no contexto da web são os verbos do protocolo HTTP (GET, POST, PUT, DELETE, etc.) [Gran Cursos Online 2022]. Essa uniformidade é o que garante o desacoplamento entre cliente e servidor. A Tabela 1.1 demonstra a correspondência direta entre os verbos HTTP e as operações CRUD (Create, Read, Update, Delete), que formam a base da maioria das interações em APIs.

Tabela 1.1. Mapeamento de Verbos HTTP para Operações CRUD.

Verbo HTTP	Ação Semântica	Operação CRUD
GET	Recuperar a representação de um recurso	Read (Ler)
POST	Criar um novo recurso	Create (Criar)
PUT	Atualizar um recurso existente ou substituí-lo	Update (Atualizar)
DELETE	Remover um recurso	Delete (Remover)

Além desses princípios, um conceito avançado e poderoso na arquitetura REST é o **HATEOAS** (Hypermedia as the Engine of Application State). O HATEOAS transforma a API de um conjunto estático de endpoints em um sistema dinâmico e "auto-descritivo". Isso é alcançado incluindo links na própria resposta da API, informando ao cliente quais ações subsequentes podem ser realizadas em um determinado recurso [Telles 2018]. Por exemplo, uma resposta que representa um pedido pode conter links para `/pedidos/123/cancelar` ou `/pedidos/123/ver-detalhes`. O cliente, em vez de ter conhecimento prévio das URIs, apenas precisa seguir esses links. Esse mecanismo reduz drasticamente o acoplamento entre cliente e servidor, tornando a API mais flexível e resiliente a mudanças [Telles 2018].

1.4. O Micro-framework Flask

A escolha do Flask como a tecnologia central para a construção da API neste minicurso é uma decisão de design estratégica, alinhada com os princípios de simplicidade, controle e flexibilidade. O Flask é um micro-framework WSGI (Web Server Gateway Interface) para Python, conhecido por sua abordagem minimalista [The Pallets Projects s.d.]. Sua filosofia central é fornecer um núcleo sólido e extensível para o desenvolvimento web, sem impor ferramentas ou padrões específicos ao desenvolvedor.

1.4.1. Uma Aplicação Mínima: "Olá, Mundo!"

Para ilustrar a simplicidade e a elegância do Flask, nada é mais eficaz do que um exemplo prático. O código a seguir representa a aplicação Flask mais simples possível, um "Olá, Mundo!"funcional que pode ser salvo em um arquivo como `app.py` e executado diretamente.

```
# 1. Importa a classe Flask do pacote flask
```

```
from flask import Flask

# 2. Cria uma instância da aplicação
app = Flask(__name__)

# 3. Define uma rota usando um decorator
@app.route('/')
def hello_world():
    # 4. A função que executa e retorna a resposta
    return 'Olá, Mundo!'

# 5. Bloco para rodar o servidor de desenvolvimento
if __name__ == '__main__':
    app.run(debug=True)
```

Este pequeno bloco de código já contém todos os elementos centrais de uma aplicação Flask:

- **Linha 1:** A importação da classe `Flask`, que é o núcleo de qualquer aplicação.
- **Linha 2:** A criação do **objeto de aplicação explícito**, que chamamos de `app`. A variável `__name__` ajuda o Flask a localizar recursos como templates e arquivos estáticos.
- **Linha 3:** Este é um **decorator de rota**. Ele diz ao Flask: "Quando alguém acessar a URL raiz ('/') do site, execute a função que está logo abaixo". É assim que o Flask conecta URLs a código Python.
- **Linha 4:** Esta é a **função de visualização** (view function). É o código que é executado para tratar a requisição do usuário e gerar uma resposta. Neste caso, a resposta é a string 'Olá, Mundo!'.
- **Linha 5:** Este é um bloco padrão em Python para garantir que o servidor de desenvolvimento só seja executado quando o script é chamado diretamente. O comando `app.run(debug=True)` inicia um servidor local simples, que recarrega automaticamente a cada alteração no código, facilitando muito o desenvolvimento.

Com apenas algumas linhas de código, temos um servidor web completo e funcional. Essa simplicidade é o ponto de partida que nos permite, como veremos a seguir, adicionar camadas de complexidade de forma controlada e organizada.

A principal diferença entre um micro-framework como o Flask e um framework *full-stack* como o Django reside em sua abrangência e "opinião"[Splunk 2023]. Enquanto frameworks *full-stack* vêm com um ecossistema robusto e integrado de ferramentas (ORM, painel de administração, sistema de autenticação), o Flask oferece apenas o essencial: roteamento de requisições e um motor de templates (Jinja). Essa característica o torna ideal para APIs REST, protótipos e aplicações onde o desenvolvedor deseja ter controle total sobre a arquitetura e as bibliotecas utilizadas [Quora 2018].

A filosofia de design do Flask se manifesta em várias vantagens práticas, especialmente através do uso de um objeto de aplicação explícito (`app = Flask(__name__)`). Em contraste com frameworks que gerenciam esse objeto de forma implícita, a abordagem do Flask oferece benefícios claros [The Pallets Projects s.d.]:

- **Testabilidade Aprimorada:** Facilita a criação de múltiplas instâncias da aplicação para testes unitários, permitindo isolar e verificar comportamentos específicos de forma controlada.
- **Extensibilidade Direta:** Permite a subclassificação da classe `Flask` para customizar comportamentos internos do framework, oferecendo um nível avançado de personalização.
- **Modularidade e Confiabilidade:** Garante que recursos (como arquivos estáticos e templates) sejam carregados de forma previsível em relação ao módulo da aplicação. Isso evita problemas de caminhos relativos ao diretório de trabalho, que são uma fonte comum de erros em ambientes de produção [The Pallets Projects s.d.].

O Flask não impõe um padrão de arquitetura rígido, o que concede ao desenvolvedor a liberdade de estruturar o projeto como preferir. Essa flexibilidade, no entanto, exige disciplina para não resultar em código desorganizado. É exatamente neste ponto que o minicurso se aprofunda, demonstrando como aplicar padrões de design maduros — como a arquitetura em camadas e o Service Layer — a um framework "não opinativo"para construir uma aplicação robusta e escalável. A Tabela 1.2 resume o contraste entre as duas filosofias de frameworks.

Tabela 1.2. Comparativo: Micro-framework vs. Full-Stack Framework.

Característica	**Micro-framework (Ex: Flask)**	**Full-Stack Framework (Ex: Django)**
Tamanho	Pequeno, núcleo enxuto	Grande, com diversos módulos
Curva de Aprendizagem	Rápida e direta	Mais complexa, exige mais tempo
Ferramentas Inclusas	Roteamento, Jinja (templating)	Roteamento, ORM, segurança, admin
Casos de Uso	APIs REST, protótipos, microsserviços	Aplicações web complexas e monolíticas
Paradigma	Flexível, “não opinativo”	Estruturado por convenções, “opinativo”

1.5. Arquitetura da Aplicação e Padrões de Projeto

A construção de software escalável e de fácil manutenção não é resultado do acaso, mas sim da adoção de padrões arquiteturais sólidos e princípios de design bem estabelecidos. Esta seção aprofunda os padrões e as decisões de arquitetura que estruturam o projeto do minicurso, garantindo que a aplicação seja modular, testável e pronta para crescer.

1.5.1. O Princípio da Separação de Responsabilidades

O fundamento de uma boa arquitetura é o princípio da **Separação de Responsabilidades** (*Separation of Concerns - SoC*). A abordagem mais comum para aplicar este princípio é

o desenvolvimento em **camadas**, que divide o sistema em componentes lógicos distintos [UDS Tecnologia 2021]. Cada camada possui um propósito específico — como apresentação, lógica de negócios ou acesso a dados — e se comunica com as camadas adjacentes por meio de interfaces bem definidas [Telles 2018].

Essa estrutura é crucial para a manutenção e a escalabilidade. Uma alteração na camada de persistência (como a migração de banco de dados) não deve impactar a lógica de negócios, desde que o "contrato"de comunicação entre as camadas seja mantido. Isso reduz o risco de regressões, simplifica a depuração e permite que equipes diferentes trabalhem em partes distintas do sistema simultaneamente [UDS Tecnologia 2021].

1.5.2. Padrões para a Estrutura do Projeto em Flask

Com a flexibilidade do Flask, a aplicação de padrões que organizam a estrutura do código é essencial. Adotamos dois padrões principais para garantir a modularidade.

1.5.2.1. Organizando o Código com Blueprints

À medida que uma aplicação Flask cresce, manter todas as rotas em um único arquivo se torna impraticável. Os **Blueprints** são a solução do Flask para este problema. Eles permitem agrupar um conjunto de rotas, templates e arquivos estáticos relacionados, funcionando como "mini-aplicações"dentro do projeto principal. No nosso caso, poderíamos ter um Blueprint para autenticação e outro para as receitas, por exemplo. Isso torna o código mais limpo, organizado e reutilizável.

1.5.2.2. O Padrão Factory para Criação da Aplicação

O padrão **Application Factory** (Fábrica de Aplicações) consiste em encapsular a criação da instância da aplicação Flask dentro de uma função (comumente chamada `create_app()`). Essa abordagem evita problemas de importação circular e é fundamental para a testabilidade, pois permite criar múltiplas instâncias da aplicação com diferentes configurações — uma para desenvolvimento, outra para produção e quantas forem necessárias para os testes.

1.5.3. Padrões para a Lógica de Negócios: MVC + Services

O padrão Model-View-Controller (MVC) é uma das abordagens mais consagradas para organizar a lógica interna de uma aplicação. Ele divide o código em três papéis interligados:

- **Modelo (Model):** Representa os dados e as regras de negócio associadas a eles.
- **Visão (View):** A camada de apresentação (no caso de uma API, a representação dos dados, como JSON).
- **Controlador (Controller):** Intermediário que recebe as requisições, interage com o Modelo e seleciona a Visão para a resposta.

Embora o Flask não force o uso do MVC, ele se adapta perfeitamente ao padrão [Stack Overflow 2012]. No entanto, em uma arquitetura MVC pura, a lógica de negócios pode sobrecarregar os controladores, gerando os temidos *"Fat Controllers"*. Para resolver isso, introduzimos uma camada adicional, a **Camada de Serviços**, ilustrada na Figura 1.2.

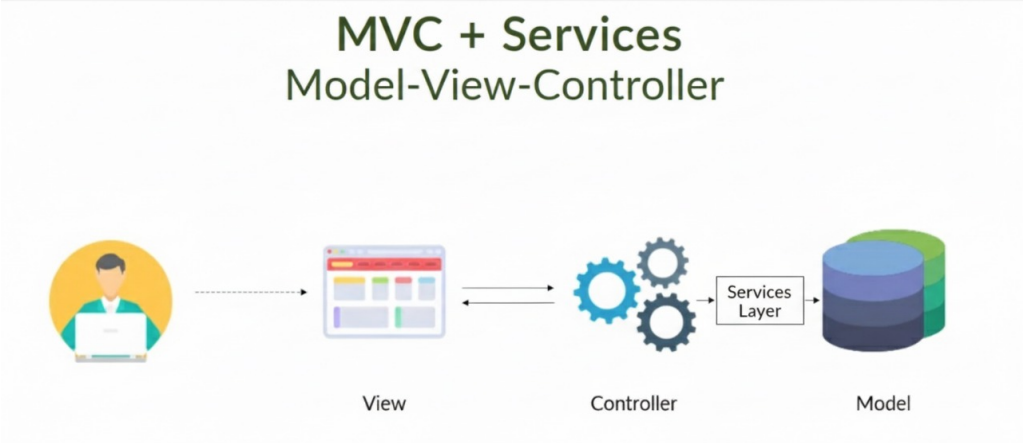


Figura 1.2. Arquitetura MVC estendida com uma Camada de Serviços (Service Layer).

A **Camada de Serviços** (Service Layer) atua como um intermediário entre os controladores e os modelos, com a responsabilidade exclusiva de centralizar e encapsular a lógica de negócios da aplicação [AppMaster s.d.]. Ao extrair essa lógica para uma camada dedicada, o Controlador se torna um *"Thin Controller"*, cuja única função é gerenciar o fluxo da requisição HTTP, delegando todas as operações de negócio para o serviço correspondente. Essa abordagem resulta em um código mais coeso, reutilizável e, crucialmente, muito mais fácil de testar de forma isolada [UDS Tecnologia 2021; LBODEV 2024].

1.5.4. Persistência de Dados com Google Firestore

A escolha do banco de dados recaiu sobre o **Google Firestore**, um banco de dados NoSQL gerenciado, flexível e escalável, ideal para aplicações modernas [Firebase s.d.]. Diferente do modelo relacional de tabelas e linhas, o Firestore é orientado a documentos. Seus dados são organizados em uma hierarquia simples, porém poderosa, conforme mostra a Figura 1.3.

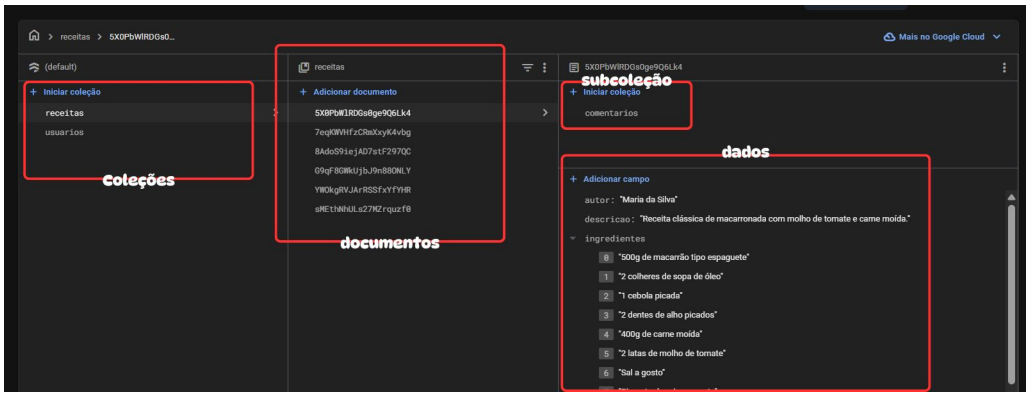


Figura 1.3. Modelo de dados hierárquico do Firestore, baseado em coleções e documentos.

Como a imagem demonstra, a estrutura do Firestore é composta por **coleções**, que são contêineres para **documentos**. Cada documento, por sua vez, armazena os dados em formato de pares chave-valor e pode conter estruturas aninhadas ou até mesmo **subcoleções** [Firebase s.d.; AppMaster 2023]. Essa estrutura permite modelar dados complexos de forma intuitiva, como uma coleção de "receitas"onde cada documento de receita pode conter uma subcoleção de "ingredientes".

Uma das vantagens mais notáveis do Firestore é que o desempenho das consultas é proporcional ao tamanho do *conjunto de resultados*, e não ao tamanho total da coleção [STEM hash 2024]. Isso significa que, mesmo com milhões de documentos, uma consulta que retorna 10 resultados será extremamente rápida.

1.5.5. Princípios para um Código de Qualidade: S.O.L.I.D.

Além dos padrões de arquitetura, a qualidade do código é guiada pelos princípios **S.O.L.I.D.**, um acrônimo para cinco pilares do design orientado a objetos que promovem a escrita de software mais compreensível, flexível e de fácil manutenção. São eles:

- **S** - Single Responsibility Principle (Princípio da Responsabilidade Única)
- **O** - Open/Closed Principle (Princípio Aberto/Fechado)
- **L** - Liskov Substitution Principle (Princípio da Substituição de Liskov)
- **I** - Interface Segregation Principle (Princípio da Segregação de Interface)
- **D** - Dependency Inversion Principle (Princípio da Inversão de Dependência)

Neste minicurso, a aplicação da Camada de Serviços é um exemplo claro do Princípio da Responsabilidade Única, pois remove a lógica de negócio dos controladores, dando a cada componente um único motivo para mudar.

1.6. Engenharia de Software para Produção

A transição de um ambiente de desenvolvimento local para um ambiente de produção real é um dos momentos mais críticos no ciclo de vida de um software. Essa etapa exige a adoção de práticas de engenharia que garantam a robustez, segurança e, acima de tudo, a consistência da aplicação. Esta seção aborda o pilar fundamental para alcançar esses objetivos: o gerenciamento de dependências.

1.6.1. Gerenciamento de Dependências: Consistência e Segurança

O gerenciamento de dependências é o processo que assegura que um projeto de software funcione de maneira previsível em qualquer ambiente, seja na máquina de outro desenvolvedor ou no servidor de produção [GitHub s.d.]. A base dessa prática em Python é o uso de **ambientes virtuais** (com ferramentas como `venv`), que isolam as bibliotecas de um projeto e evitam conflitos de versão com outras aplicações do sistema [DEV Community 2024].

Para garantir a reprodutibilidade, é essencial o uso de um arquivo de dependências, como o `requirements.txt`. Este arquivo funciona como uma "receita"do ambiente, listando as bibliotecas e suas versões exatas, permitindo que o ambiente de produção seja recriado com precisão. As boas práticas de gerenciamento incluem [GitHub s.d.]:

- **Fixação de Versões (Pinning):** Utilizar a saída do comando `pip freeze > requirements.txt` para "travar"as versões exatas de todas as dependências e sub-dependências, garantindo que o mesmo código funcione sobre a mesma base de bibliotecas.

- **Auditorias de Vulnerabilidades:** Integrar ao fluxo de trabalho a verificação regular de dependências com vulnerabilidades de segurança conhecidas, utilizando ferramentas que analisam o arquivo de requisitos.

- **Automação de Atualizações:** Configurar ferramentas para monitorar e aplicar patches de segurança de forma automatizada, mantendo o projeto protegido contra ameaças emergentes.

Ferramentas modernas como o **UV** vêm para otimizar ainda mais esse fluxo, combinando a criação de ambientes virtuais e a instalação de pacotes de forma extremamente rápida, simplificando a vida do desenvolvedor e a automação em pipelines [DEV Community 2024].

1.7. DevOps na Prática: Deploy Contínuo com Render

Uma vez que a aplicação está bem estruturada e suas dependências são gerenciadas, o próximo desafio é entregá-la ao usuário final de forma eficiente e segura. O paradigma **DevOps** surge para quebrar as barreiras entre desenvolvimento e operações, utilizando a automação como ponte [Red Hat 2024]. A espinha dorsal dessa automação é o pipeline de CI/CD.

1.7.1. O Paradigma da Integração e Implantação Contínua (CI/CD)

Tradicionalmente, a implantação de software era um processo manual, lento e propenso a erros humanos, resultando em entregas demoradas e instabilidade em produção [Inedo 2017]. O **CI/CD** (*Continuous Integration* e *Continuous Delivery/Deployment*) resolve isso ao criar um pipeline automatizado que transforma o código-fonte em uma aplicação funcional na nuvem, como ilustrado na Figura 1.4.

Figura 1.4. Fluxo visual de um pipeline de CI/CD, desde o commit até a implantação.

O fluxo demonstrado na Figura 1.4 representa um caminho automatizado onde cada alteração no código passa por uma série de validações antes de chegar ao usuário. Esse pipeline é composto por três práticas principais [Red Hat 2023; Red Hat 2024]:

- **Integração Contínua (CI):** A prática de mesclar frequentemente as alterações de código de vários desenvolvedores em um repositório central. A cada envio, tes-

tes automatizados são executados para validar a nova versão e fornecer feedback rápido, garantindo a saúde do código-base.

- **Entrega Contínua (CD - Delivery):** A automação da preparação do código para o lançamento. Após passar nos testes da CI, o código é empacotado e enviado para um ambiente de "preparação"(*staging*), aguardando apenas uma aprovação manual para ser implantado em produção.
- **Implantação Contínua (CD - Deployment):** O passo final da automação. Se o código passar por todas as etapas anteriores, ele é implantado automaticamente em produção, sem qualquer intervenção humana.

A adoção de um pipeline de CI/CD, mesmo que simplificado, traz benefícios imensos para qualquer projeto de software, conforme detalhado na Tabela 1.3.

Tabela 1.3. Benefícios de um Pipeline de CI/CD.

Benefício	Descrição Detalhada
Agilidade na Entrega	Acelera a cadência de releases, permitindo que novas funcionalidades cheguem ao usuário de forma contínua e mais rápida.
Redução de Erros	Automatiza testes e validações, minimizando o erro humano e capturando bugs antes que cheguem à produção.
Custo-Benefício	Reduz o tempo e o esforço manual, liberando os desenvolvedores para se concentrarem em agregar valor ao produto.
Qualidade do Código	O feedback imediato dos testes e a integração em pequenos lotes incentivam um código de maior qualidade e melhor colaboração.

1.7.2. Render como Solução Prática para Deploy Contínuo

A plataforma **Render** exemplifica a implementação prática e acessível dos princípios de CI/CD. Trata-se de uma solução de hospedagem na nuvem que simplifica radicalmente o processo de implantação, graças à sua integração nativa com repositórios Git como o GitHub [Render s.d.].

Ao configurar um serviço no Render e vinculá-lo a um *branch* específico, o processo de implantação se torna totalmente automatizado. A cada `push` para esse *branch*, o Render detecta a alteração e dispara um processo de **auto-deploy**. Esse processo executa os comandos definidos pelo desenvolvedor, como `pip install -r requirements.txt` e `gunicorn app:app`, garantindo que a versão mais recente e validada da aplicação esteja sempre em produção, com *zero-downtime* (sem interrupção do serviço) [Render s.d.].

Dessa forma, o Render democratiza o acesso a práticas avançadas de DevOps. Ele permite que os participantes do minicurso vivenciem na prática os benefícios da automação, transformando os conceitos de CI/CD em uma realidade tangível e de fácil implementação.

1.8. Conclusão

Este capítulo forneceu o embasamento teórico para o minicurso, conectando os conceitos de engenharia de software com as tecnologias abordadas. A jornada iniciou-se contextu-

alizando o papel crucial das APIs no desenvolvimento moderno e focou na **arquitetura REST** como o padrão de design que se consolidou por sua simplicidade e alinhamento com os princípios da web [Kong Inc. 2021]. A escolha do **Flask** foi justificada por sua natureza de micro-framework, que oferece a flexibilidade ideal para a implementação de padrões de arquitetura sofisticados [Quora 2018].

O núcleo do capítulo aprofundou-se na arquitetura da aplicação, apresentando um conjunto de padrões para garantir um projeto profissional. Foram introduzidos padrões para a organização estrutural do código, como **Blueprints** para modularidade e o padrão **Factory** para a criação de instâncias testáveis. Em seguida, detalhou-se a arquitetura da lógica de negócios com o padrão **MVC + Services**, que promove controladores coesos e de fácil manutenção. Como alicerce para um código de alta qualidade, foram apresentados os princípios **S.O.L.I.D.**. A persistência de dados com o **Google Firestore** ilustrou a aplicação de um banco de dados NoSQL moderno e escalável [Firebase s.d.].

Por fim, o capítulo dedicou suas duas últimas seções à preparação para o ambiente de produção. A primeira abordou a **engenharia de software local**, destacando o **gerenciamento de dependências** como prática essencial para a segurança e consistência do projeto [GitHub s.d.]. A segunda explorou o **paradigma de DevOps na prática**, explicando o pipeline de **CI/CD** e demonstrando como a **plataforma Render** serve como um exemplo prático e acessível da automação do processo de deploy [Red Hat 2024].

Em suma, a fundamentação teórica aqui presente eleva o minicurso de um simples tutorial para uma experiência de aprendizado que enfatiza os princípios de engenharia por trás das ferramentas. Ao compreender esses conceitos, os participantes estarão mais bem preparados para desenvolver APIs robustas, escaláveis e alinhadas com as melhores práticas do mercado, transpondo com sucesso o conhecimento teórico para a aplicação em projetos de software reais.

Referências

Andrade, M. (2024) “Boas práticas no Design e Desenvolvimento de APIs”, DIO.

AppMaster (2023) “Firestore: um mergulho profundo no banco de dados NoSQL do Firebase”, Disponível em: `https://appmaster.io/pt/blog/banco-de-dados-nosql-firestore`. Acessado em: 31 de agosto de 2025.

AppMaster (s.d.) “Camada de serviço API”, Disponível em: `https://appmaster.io/pt/glossary/camada-de-servico-api`. Acessado em: 31 de agosto de 2025.

AWS (s.d.) “O que é SOA? – Explicação sobre arquitetura orientada a serviços”, Disponível em: `https://aws.amazon.com/pt/what-is/service-oriented-architecture/`. Acessado em: 31 de agosto de 2025.

Brito, B. (2020) “API RESTful Boas práticas”, Medium.

DEV Community (2024) “UV - A Ferramenta que Simplifica o Gerenciamento de Ambientes e Dependências no Python”,Disponível em: https://dev.to/kevin_ff4e10b8c916155f9d4/uv-a-ferramenta-que-simplifica-o-gerenciamento-de-ambientes-e-depender

Firebase (s.d.) “Documentação do Firestore”, Disponível em: https://firebase.google.com/docs/firestore?hl=pt-br. Acessado em: 31 de agosto de 2025.

GitHub (s.d.) “Melhores práticas de manutenção de dependências”, Disponível em: https://docs.github.com/pt/enterprise-server@3.17/code-security/dependa Acessado em: 31 de agosto de 2025.

Google Cloud (s.d.) “Firestore”, Disponível em: https://cloud.google.com/products/firestore. Acessado em: 31 de agosto de 2025.

Gran Cursos Online (2022) “Conceituação e principais pontos sobre a arquitetura REST”, Disponível em: https://blog.grancursosonline.com.br/conceituacao-e-principais-pontos-s Acessado em: 31 de agosto de 2025.

Inedo (2017) “Manual Deployment Disasters”, Disponível em: https://blog.inedo.com/devops/manual-deployment-disasters/. Acessado em: 31 de agosto de 2025.

Kong Inc. (2021) “The Evolution of APIs: From RPC to SOAP and XML”, Disponível em: https://konghq.com/blog/enterprise/evolution-apis-rpc-soap-xml. Acessado em: 31 de agosto de 2025.

LBODEV (2024) “O Que é Service Layer: Entenda Sua Importância”, Disponível em: https://lbodev.com.br/glossario/o-que-e-service-layer/. Acessado em: 31 de agosto de 2025.

Macoratti .net (2019) “.NET - Apresentando HATEOAS”, Disponível em: https://www.macoratti.net/19/05/net_hateoas1.htm. Acessado em: 31 de agosto de 2025.

Mufid, I. and Basofi, A. (2019) “Design an MVC Model using Python for Flask Framework Development”, Semantic Scholar.

MuleSoft (s.d.) “Camada de API”, Disponível em: https://www.mulesoft.com/pt/api/rest/api-layer. Acessado em: 31 de agosto de 2025.

Neupane, A. (2024) “How to deploy a Flask app to Render” [Vídeo], YouTube.

Opsera (s.d.) “12 Business Benefits of CI/CD | A CI/CD Overview”, Disponível em: https://www.opsera.io/blog/ci-cd-business-benefits. Acessado em: 31 de agosto de 2025.

ProgrammingKnowledge (2025) “How to Deploy Python Flask App on Render” [Vídeo], YouTube.

Quora (2018) “What are the differences between a micro-framework and a full-stack framework?”, Disponível em: https://www.quora.com/What-are-the-differences-between-a-micro-framewo Acessado em: 31 de agosto de 2025.

Red Hat (2023) “API REST - O que é API REST?”, Disponível em: https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api. Acessado em: 31 de agosto de 2025.

Red Hat (2024) “What is CI/CD?”, Disponível em: https://www.redhat.com/en/topics/devops/what-is-ci-cd. Acessado em: 31 de agosto de 2025.

Render (s.d.) “Deploy a Flask App on Render”, Disponível em: https://render.com/docs/deploy-flask. Acessado em: 31 de agosto de 2025.

Rodrigues, D. (2025) “Aplicações modernas com Python: Desenvolvimento Web com Flask e FastAPI”, Google Books.

Sarro, T. (s.d.) “Good Practices on Manual Deployment”, Medium. Disponível em: https://medium.com/@tadeusarro/good-practices-on-manual-deployment-ed2 Acessado em: 31 de agosto de 2025.

Silva, T. (2019) “Flask: Crie aplicações web robustas com Python”, Casa do Código.

Splunk (2023) “Intro to Python Frameworks Part 2 – Full-Stack vs. Micro Framework”, Disponível em: https://www.splunk.com/en_us/blog/observability/intro-to-python-framew Acessado em: 31 de agosto de 2025.

Stack Overflow (2012) “Flask-framework: MVC pattern”, Disponível em: https://stackoverflow.com/questions/12547206/flask-framework-mvc-patte Acessado em: 31 de agosto de 2025.

STEM hash (2024) “Google’s Cloud Firestore Database: Leveraging The Binary Search Algorithm”, Disponível em: https://stemhash.com/google-cloud-firestore-database/. Acessado em: 31 de agosto de 2025.

Telles, D. (2018) “Princípios de uma API REST”, Medium. Disponível em: https://unicorncoder.medium.com/princ%C3%ADpios-de-uma-api-rest-c8e08c2 Acessado em: 31 de agosto de 2025.

TestDriven.io (2022) “Deploying a Flask App to Render”.

The Pallets Projects (s.d.) “Flask Documentation”, Disponível em: https://flask.palletsprojects.com/. Acessado em: 31 de agosto de 2025.

Towards Data Science (2022) “Essentials for Working With Firestore in Python”.

UDS Tecnologia (2021) “Desenvolvimento de software em camadas - funcionamento e vantagens”, Disponível em: https://uds.com.br/blog/desenvolvimento-software-camadas/. Acessado em: 31 de agosto de 2025.

Unicamp (s.d.) “Uma proposta de arquitetura para o protocolo NETCONF sobre SOAP”, Repositório Unicamp. Disponível em: https://repositorio.unicamp.br/Busca/Download?codigoArquivo=468663. Acessado em: 31 de agosto de 2025.

Valente, M. T. (2020) “Engenharia de software moderna. Princípios e práticas para desenvolvimento de software com produtividade”.

Chapter 2

Desenvolvendo um Sistema Multi-Agentes para Pesquisa Científica com LangGraph

Larissa Souza do Nascimento e Ricardo Moura Sekeff Budaruiche

Abstract

In recent years, Artificial Intelligence (AI) has evolved rapidly, becoming a strategic technology across many areas of knowledge. In the field of scientific research, AI plays a crucial role in automating tasks, processing large volumes of data, and generating relevant insights. This chapter presents a practical approach to building a multi-agent system for supporting scientific research using the ***LangGraph*** *framework. The system enables the orchestration of autonomous agents specialized in search, validation, and data analysis, integrating tools such as Tavily AI, arXiv API, and ChromaDB. We discuss the architecture, implementation process, and ethical considerations related to AI usage in science. The goal is to equip researchers and developers with both conceptual understanding and practical skills to apply multi-agent systems in their research workflows, promoting efficiency, transparency, and reproducibility in scientific investigation.*

Resumo

Nos últimos anos, a Inteligência Artificial (IA) evoluiu rapidamente, consolidando-se como uma tecnologia estratégica em diversas áreas do conhecimento. No campo da pesquisa científica, a IA exerce papel essencial na automação de tarefas, no processamento de grandes volumes de dados e na geração de insights relevantes. Este capítulo apresenta uma abordagem prática para a construção de um sistema multi-agentes de apoio à pesquisa científica utilizando o framework LangGraph. O sistema permite a orquestração de agentes autônomos especializados em busca, validação e análise de dados, integrando ferramentas como Tavily AI, arXiv API e ChromaDB. São discutidos a arquitetura, o processo de implementação e as considerações éticas envolvidas no uso da IA na ciência. O objetivo é capacitar pesquisadores e desenvolvedores com conhecimentos conceituais e práticos para empregar sistemas multi-agentes em fluxos de pesquisa, promovendo eficiência, transparência e reprodutibilidade na investigação científica.

2.1. Introdução

Nas últimas décadas, o avanço dos modelos de linguagem de larga escala (*Large Language Models* – LLMs) tem transformado de maneira profunda a forma como sistemas computacionais interagem com o conhecimento humano. Esses modelos demonstram capacidade crescente de compreender, sintetizar e gerar linguagem natural com um nível de fluidez sem precedentes, o que tem impulsionado novas aplicações em diversas áreas, da educação à pesquisa científica (Jordan; Mitchell, 2015).

Com o crescimento exponencial do volume de informações disponíveis em bases acadêmicas e na *web*, tornou-se inviável para pesquisadores acompanhar manualmente todas as publicações e inovações relevantes em seus respectivos campos de estudo. Nesse contexto, a automação de processos cognitivos, como busca, filtragem e análise de dados científicos, emerge como uma necessidade estratégica.

A Inteligência Artificial (IA), especialmente em sua vertente de Aprendizagem de Máquina e dos LLMs, vem desempenhando papel central nesse movimento de transformação. Ferramentas baseadas em IA são atualmente capazes de coletar, validar e integrar conhecimento proveniente de múltiplas fontes de forma contextualizada, apoiando o pesquisador não apenas como instrumento de busca, mas como um parceiro cognitivo (Meloni *et al.*, 2023). Essa nova geração de sistemas inteligentes está alinhada com as tendências discutidas em (Bender *et al.*, 2021), que ressaltam a importância de equilibrar o poder dos modelos de linguagem com mecanismos de controle, explicabilidade e responsabilidade ética, aspectos fundamentais para sua aplicação em ambientes científicos.

Inserido nesse cenário, o presente capítulo apresenta o desenvolvimento do ***SAPIEN – Sistema de Agentes para Pesquisa e Ensino***, uma solução computacional baseada em um sistema **multiagente** e construída sobre o *framework LangGraph*. O *SAPIEN* foi concebido para automatizar etapas do ciclo de pesquisa científica, incluindo a coleta de artigos em fontes como *arXiv* e *Tavily AI*, o processamento linguístico e semântico dos textos, a validação contextual por meio de *embeddings* e o armazenamento vetorial otimizado com *ChromaDB*. O sistema é resultado de esforços acadêmicos originalmente apresentados em (Nascimento; Budaruiche, 2025), sendo aqui expandido com aprofundamento teórico, descrição técnica e análise arquitetural.

A escolha do **LangGraph** como núcleo do sistema fundamenta-se em suas propriedades de modularidade e persistência de estado. Diferentemente de *frameworks* lineares, como o *LangChain*, o *LangGraph* permite representar fluxos de execução em forma de grafo direcionado, no qual cada nó corresponde a um agente e as arestas controlam o fluxo condicional de dados. Essa característica possibilita a criação de ciclos de retroalimentação, em que os agentes podem reavaliar resultados anteriores e aprimorar suas decisões (Chase, 2024).

Além disso, a natureza *stateful* do *LangGraph* permite que o sistema mantenha memória contextual ao longo das interações, assegurando continuidade no raciocínio e coerência nas respostas, aspecto essencial em domínios científicos complexos. Essa arquitetura mostra-se particularmente adequada a cenários nos quais múltiplos agentes, como buscadores, validadores e analisadores semânticos, devem cooperar para atingir um objetivo comum: transformar grandes volumes de dados em conhecimento estruturado e

relevante.

Dessa forma, este capítulo tem como objetivo apresentar, de maneira detalhada, os fundamentos teóricos, metodológicos e tecnológicos do **SAPIEN**. São discutidos o processo de configuração e desenvolvimento do sistema, sua arquitetura modular, a integração entre agentes e ferramentas de IA, bem como as implicações éticas e técnicas associadas à utilização de modelos de linguagem na ciência contemporânea. Ao final, busca-se demonstrar que sistemas multiagentes baseados em LLMs, como o *SAPIEN*, constituem uma abordagem promissora para a modernização dos métodos de pesquisa científica, promovendo maior eficiência, reprodutibilidade e transparência no processo de descoberta do conhecimento.

2.2. Fundamentação Teórica

2.2.1. Agentes Inteligentes

O conceito de agente inteligente constitui um dos alicerces da pesquisa em IA e em sistemas multiagente. Tradicionalmente, define-se um agente como uma entidade capaz de perceber o ambiente por meio de sensores, processar essas percepções e agir sobre o ambiente por meio de atuadores, de modo a perseguir objetivos explícitos ou implícitos de forma autônoma. Essa definição enfatiza três aspectos fundamentais, percepção, raciocínio/decisão e ação, que conferem ao agente um comportamento orientado por propósito.

Em uma perspectiva contemporânea, os agentes inteligentes, ou *AI agents*, são sistemas de *software* altamente autônomos, capazes de perseguir metas em nome de usuários ou outras entidades, demonstrando raciocínio, planejamento e memória ao longo da execução de tarefas compostas. Essas capacidades incluem: (i) a decomposição de objetivos em subtarefas; (ii) o uso de modelos de raciocínio simbólico, baseados em regras ou estatísticos; (iii) a interação com ferramentas externas, como APIs, bases de dados ou modelos de linguagem; e (iv) a adaptação contínua mediante aprendizagem ou *feedback*. Em suma, tais agentes não apenas reagem ao ambiente, mas também são capazes de antecipar estados futuros, planejar ações e cooperar com outros agentes ou sistemas.

Com o surgimento dos LLMs e das arquiteturas *agentic* (multiagente), observa-se o aparecimento de uma nova categoria de agentes, que incorporam funcionalidades como memória persistente, colaboração entre agentes e decomposição dinâmica de tarefas complexas. Esse avanço amplia a definição clássica ao introduzir um nível superior de autonomia, característico dos chamados *agentic AI systems*, nos quais os agentes são capazes de gerar problemas, formular subobjetivos, delegar atividades a outros agentes e evoluir ao longo do tempo.

No contexto da pesquisa científica, os agentes inteligentes emergem como componentes críticos em sistemas voltados à automação de tarefas de alto volume, como rastreamento de literatura, extração de metadados, análise semântica e validação de resultados. Por meio de arquiteturas multiagente, cada agente assume uma especialização funcional, por exemplo, busca, processamento de linguagem natural (NLP) ou armazenamento vetorial, enquanto um agente supervisor orquestra o fluxo de execução e a comunicação entre eles. Esse tipo de organização favorece a escalabilidade, a modularidade e a manutenção

de um estado contextualizado entre interações.

Além disso, os agentes inteligentes possuem características centrais que os distinguem de simples sistemas automatizados:

- **Autonomia**: capacidade de agir sem intervenção humana contínua, controlando suas próprias ações e decisões;
- **Reatividade**: habilidade de perceber mudanças no ambiente em tempo real e responder de maneira apropriada;
- **Proatividade**: competência para tomar a iniciativa e buscar objetivos futuros, em vez de apenas reagir a estímulos;
- **Sociabilidade**: capacidade de interagir e comunicar-se com outros agentes, humanos ou sistemas, cooperando em prol de metas comuns.

Esses atributos são especialmente relevantes para o sistema aqui descrito, uma vez que os agentes devem operar de forma colaborativa, hierarquizada e persistente, mantendo o contexto, compartilhando conhecimento e refinando sucessivamente suas respostas. Em particular, ao integrar LLMs como componentes de raciocínio, por exemplo, na interpretação de *prompts* ou na análise semântica, é necessário que o agente não apenas execute etapas isoladas, mas também encadeie múltiplas operações, como planejamento, chamada de ferramentas, verificação de consistência e arquivamento de resultados. Assim, configura-se um ciclo contínuo de percepção, decisão e ação.

Por fim, é importante observar que o uso de agentes inteligentes em domínios críticos, como a pesquisa científica automatizada, envolve desafios técnicos e éticos significativos, entre eles a manutenção de memória útil e relevante, a mitigação de alucinações cognitivas de modelos de linguagem, a rastreabilidade das decisões e a modularização da arquitetura para fins de auditoria e supervisão. Esses aspectos reforçam a necessidade de projetar agentes inteligentes não apenas sob a ótica da funcionalidade, mas também da confiabilidade, explicabilidade e adaptabilidade ao longo do tempo.

2.2.2. Sistemas Multiagente e LLMs

Os sistemas multiagente (SMAs) consistem em coleções de agentes autônomos que interagem, cooperam ou competem entre si para resolver problemas distribuídos e, frequentemente, complexos. Em sua formulação clássica, cada agente possui seus próprios sensores, atuadores e objetivos, podendo coordenar suas ações com os demais a fim de alcançar metas compartilhadas (Luo *et al.*, 2019).

Com o advento dos LLMs, esse paradigma foi revitalizado: agentes capazes de empregar raciocínio em linguagem natural, acionar ferramentas externas, comunicar-se entre si e manter estado ao longo do tempo passaram a compor sistemas de apoio em domínios como pesquisa científica, revisão de literatura e descoberta de conhecimento.

No contexto da biblioteca *LangGraph*, os SMAs são formalizados como grafos de execução, nos quais cada nó representa um agente (ou ferramenta) e as arestas definem o fluxo de controle e a atualização de estado. A arquitetura denominada *Supervisor (tool-calling)* foi adotada pelo sistema proposto, o *SAPIEN*, por diversos motivos:

- **Modularidade e especialização**: cada agente, por exemplo, busca *web*, extração de metadados ou validação vetorial, é implementado como uma ferramenta especializada, o que facilita testes, manutenção e evolução individual.
- **Orquestração centralizada**: o agente supervisor atua como ponto de controle lógico, invocando as ferramentas apropriadas conforme o contexto, levando em conta a complexidade da tarefa e a necessidade de conhecimento específico de domínio.
- **Persistência de estado e memória contextual**: por meio do grafo de execução do *LangGraph*, o sistema mantém e transfere estado, como histórico de buscas, *embeddings* gerados e resultados validados, reforçando a continuidade das interações e aprimorando a coerência das respostas.

A adoção dessa arquitetura permite que o *SAPIEN* realize *handoffs* explícitos entre agentes, isto é, que o supervisor delegue a execução a outro agente (ou ferramenta) e receba o resultado por meio de um controle estruturado de grafo, conforme exemplificado na documentação oficial. Essa decomposição do *workflow* em agentes-ferramenta sob controle de um supervisor favorece a escalabilidade do sistema, especialmente quando novos domínios ou fontes de dados são incorporados.

Além disso, a literatura destaca benefícios significativos dos sistemas multiagente em comparação a *pipelines* lineares compostos por agentes isolados:

- **Separação de responsabilidades**: possibilita a criação de agentes especialistas em tarefas específicas, resultando em desempenho superior em relação a agentes genéricos que precisam selecionar entre múltiplas ferramentas ou estratégias;
- **Modularidade e extensibilidade**: facilita a experimentação e a melhoria incremental de cada agente, permitindo atualizações isoladas sem comprometer a integridade do sistema como um todo;
- **Controle de fluxo baseado em grafos**: substitui sequências rígidas de execução por arquiteturas flexíveis, que viabilizam lógica condicional, laços de retroalimentação e interações dinâmicas entre agentes.

Em síntese, ao combinar um agente supervisor controlador com agentes especializados implementados como ferramentas, o *SAPIEN* adota uma arquitetura robusta e escalável para lidar com tarefas típicas da pesquisa científica: coleta de dados provenientes da *web* e de repositórios, processamento semântico, validação cruzada e armazenamento vetorial, todos integrados em um ciclo contínuo de raciocínio e execução coordenada.

2.2.3. Frameworks para Agentes Baseados em LLMs

Entre as principais ferramentas para o desenvolvimento de agentes baseados em modelos de linguagem destacam-se o **LangChain**, o **AutoGen**, o **CrewAI** e o **LangGraph**. Enquanto o *LangChain* se sobressai na definição de fluxos lineares de tarefas, o *LangGraph* possibilita a construção de fluxos complexos e cíclicos, com controle detalhado de

estado e persistência (Barbarroxa; Gomes; Vale, 2024). Essa característica o torna particularmente adequado para aplicações científicas que demandam rastreabilidade, coerência contextual e continuidade de processamento.

Estudos como o de (Khandare *et al.*, 2023) evidenciam o papel central das bibliotecas em *Python* e dos *frameworks* de Inteligência Artificial no ecossistema computacional contemporâneo, ressaltando a importância da modularidade, reusabilidade e extensibilidade na concepção de agentes inteligentes. Nesse contexto, ferramentas como o *LangGraph* representam um avanço significativo, ao proporcionar infraestrutura flexível para orquestração de agentes em ambientes que exigem alta integração entre raciocínio simbólico, aprendizado estatístico e persistência de memória.

2.2.4. Tecnologias Utilizadas

Para o desenvolvimento do sistema multiagente voltado à realização de pesquisas científicas e buscas na *web*, foram empregadas tecnologias modernas e adequadas ao contexto de aplicações baseadas em modelos de linguagem. A seguir, apresentam-se as principais ferramentas utilizadas:

- ***Python***: utilizada como linguagem de programação principal devido à sua ampla adoção na comunidade de inteligência artificial, simplicidade sintática, vasta gama de bibliotecas voltadas ao desenvolvimento de agentes e suporte nativo a paradigmas como orientação a objetos e programação funcional. Além disso, o *Python* apresenta excelente integração com *frameworks* de IA, como *TensorFlow*, *PyTorch*, *LangChain* e *LangGraph*, facilitando tanto a prototipagem quanto o escalonamento de soluções complexas (Khandare *et al.*, 2023).

- ***LangGraph***: *framework* de orquestração baseado em grafos direcionados, utilizado para a construção do sistema multiagente. Permite a criação de fluxos dinâmicos e cíclicos entre agentes, com controle refinado sobre os estados e transições, sendo ideal para aplicações que requerem tomada de decisão condicionada e persistência de estado ao longo de múltiplas interações.

- ***TavilySearch***: ferramenta de busca na *web* integrada via API, utilizada para fornecer respostas baseadas em fontes abertas e atualizadas. Sua principal função no sistema é atender às demandas informacionais gerais que extrapolam o escopo de bases científicas especializadas.

- ***ChromaDB***: banco de dados vetorial responsável pelo armazenamento de *embeddings* extraídos de documentos e artigos científicos. Distingue-se por sua eficiência, interface intuitiva, modelo de código aberto e desempenho otimizado para conjuntos de dados de pequena e média escala. Essas características tornam o *ChromaDB* particularmente apropriado para aplicações acadêmicas e sistemas multiagentes em fase de prototipagem. Além disso, apresenta elevada performance em tarefas de recuperação semântica, como demonstrado em estudos recentes (Mathur; Chhabra, 2024).

- **arXiv API**: fonte de dados especializada, utilizada para acesso estruturado a artigos científicos em formato XML. Permite consultas precisas por metadados (au-

tores, palavras-chave, datas e resumos), sendo essencial para a operação do agente responsável por pesquisas bibliográficas.

- ***Anthropic Claude 3.5***: LLM de última geração utilizado para geração e interpretação de respostas dentro do sistema. Sua alta capacidade de raciocínio, compreensão de contexto extenso e desempenho superior em tarefas de múltiplos turnos o tornam adequado para funções de supervisão e coordenação de subagentes.

2.2.5. Configuração do Ambiente

A configuração do ambiente de desenvolvimento do *SAPIEN* segue um fluxo padronizado, projetado para assegurar a reprodutibilidade e a consistência na execução do sistema. O processo utiliza o interpretador *Python 3.10+* e adota o gerenciamento de dependências por meio de ambiente virtual. Essa abordagem isola bibliotecas e versões, evitando conflitos e facilitando a portabilidade entre diferentes sistemas operacionais (Khandare *et al.*, 2023).

O primeiro passo consiste em clonar o repositório do projeto a partir do GitHub, conforme ilustrado no trecho de código a seguir:

```
git clone https://github.com/larissaNa/SapienAgent.git

cd SapienAgent
```

Em seguida, cria-se um ambiente virtual e ativa-se o contexto de execução:

```
python -m venv venv
#Linux/macOS
source venv/bin/activate
#Windows
.\venv\Scripts\activate
```

A instalação das dependências é realizada a partir do arquivo `requirements.txt`, o qual lista todas as bibliotecas necessárias ao funcionamento do sistema, incluindo **Flask**, **LangGraph**, **LangChain**, **ChromaDB**, **APScheduler**, **SentenceTransformers** e **arXiv API**. Essa etapa é executada com o comando:

```
pip install -r requirements.txt
```

Posteriormente, configuram-se as variáveis de ambiente responsáveis pela autenticação com APIs externas:

```
TAVILY_API_KEY="sua_chave"
ANTHROPIC_API_KEY="sua_chave"
```

Essas chaves são indispensáveis ao funcionamento dos agentes de busca e interpretação, garantindo acesso às ferramentas **Tavily AI** (Tavily, 2024) e **Claude (Anthropic)** (Anthropic, 2024).

Concluída a configuração, a execução do sistema é realizada por meio do comando:

```
python run.py
```

Após a inicialização, a interface web torna-se acessível em `http://127.0.0.1:5000/`, permitindo ao usuário realizar consultas científicas em linguagem natural. A Figura 2.1 apresenta, de forma esquemática, as etapas de preparação e execução do ambiente de desenvolvimento.

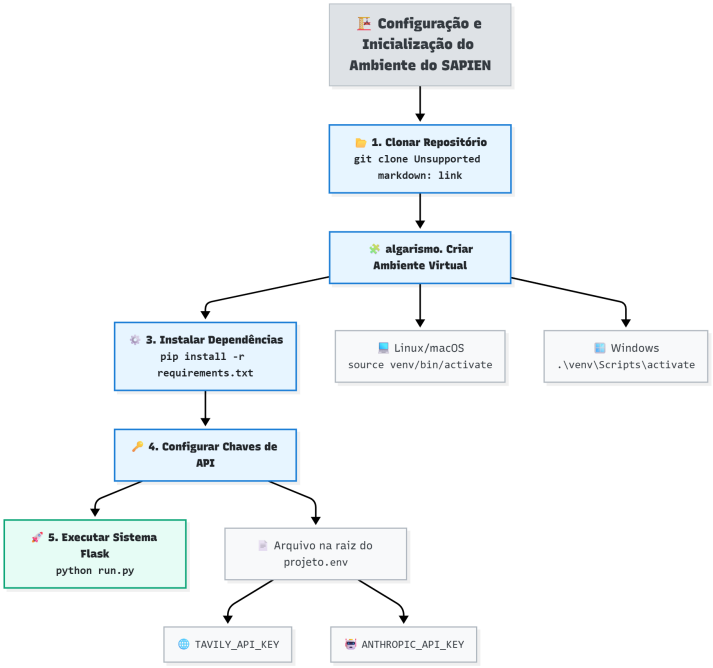


Figura 2.1: Etapas de preparação e execução do ambiente do *SAPIEN*

2.3. Estrutura e Componentes do Projeto

A arquitetura de *software* do *SAPIEN* foi concebida com base em princípios de modularidade e separação de responsabilidades, assegurando manutenção simplificada e extensibilidade para novos agentes ou serviços (Nascimento; Budaruiche, 2025). A estrutura de diretórios segue o padrão adotado em sistemas baseados no *Flask*, organizando os componentes principais conforme descrito na Tabela 2.1.

Além da estrutura de diretórios, o sistema é organizado em módulos funcionais interdependentes. A camada **core** concentra o processamento lógico dos agentes e define o grafo *LangGraph*, responsável por conectar as operações em sequência e ciclos de retroalimentação. O módulo **services.py** gerencia a comunicação assíncrona entre agentes, utilizando *threads* para execução paralela e controle de fluxo de mensagens.

A Figura 2.2 ilustra a estrutura em grafo utilizada para orquestrar os agentes, permitindo controle flexível do fluxo de dados e das decisões dentro do sistema.

Os agentes especializados foram projetados com foco em eficiência e especialização de tarefas.

Tabela 2.1: Estrutura de diretórios e descrição dos principais componentes do *SAPIEN*.

Diretório / Arquivo	**Descrição**
`app/`	Contém o código principal da aplicação Flask e o núcleo multiagente.
`app/core/`	Núcleo do sistema. Implementa o grafo de execução, os agentes e a configuração central.
`app/core/agents.py`	Define os agentes especializados: *tavily_agent*, *arxiv_agent*, *nlp_agent*, *validation_agent*, *chromadb_agent* e *sched_agent*.
`app/core/services.py`	Implementa o fluxo de processamento via LangGraph, gerenciando a troca de mensagens e o raciocínio dos agentes.
`app/core/config.py`	Arquivo de configuração global, responsável pela inicialização de modelos de linguagem, embeddings e banco vetorial.
`app/static/`	Diretório de arquivos estáticos da interface web, incluindo folhas de estilo, scripts JavaScript e imagens.
`app/templates/`	Armazena os templates HTML utilizados na renderização da interface interativa de chat.
`chroma_db/`	Base local de dados vetoriais utilizada para armazenamento semântico de documentos.
`requirements.txt`	Lista de dependências necessárias à execução do projeto.
`run.py`	Ponto de entrada principal da aplicação. Inicializa o servidor Flask e o grafo de agentes.

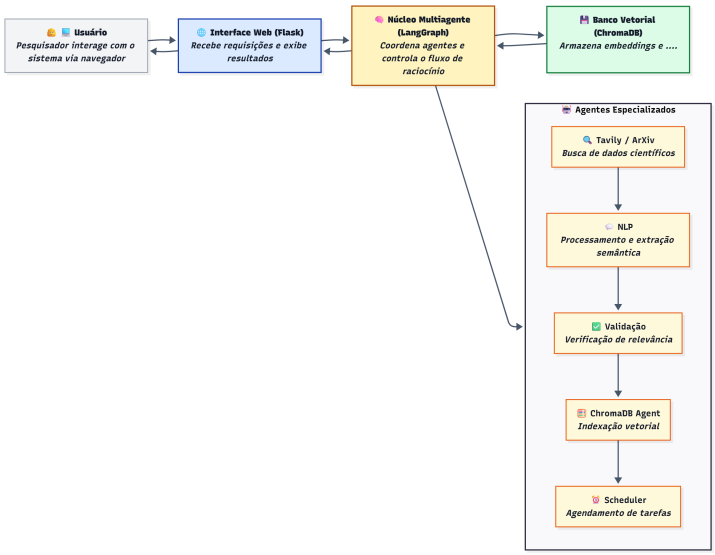


Figura 2.2: Estrutura em grafo utilizada para a orquestração dos agentes no *SAPIEN*

Por exemplo, o **tavily_agent** realiza buscas em fontes abertas, o **nlp_agent** aplica técnicas de processamento linguístico e o **validation_agent** emprega embeddings do

modelo *all-MiniLM-L6-v2* para avaliar similaridade semântica. A Figura 2.3 apresenta o fluxo interno de processamento de um agente, evidenciando as etapas de entrada, transformação e resposta.

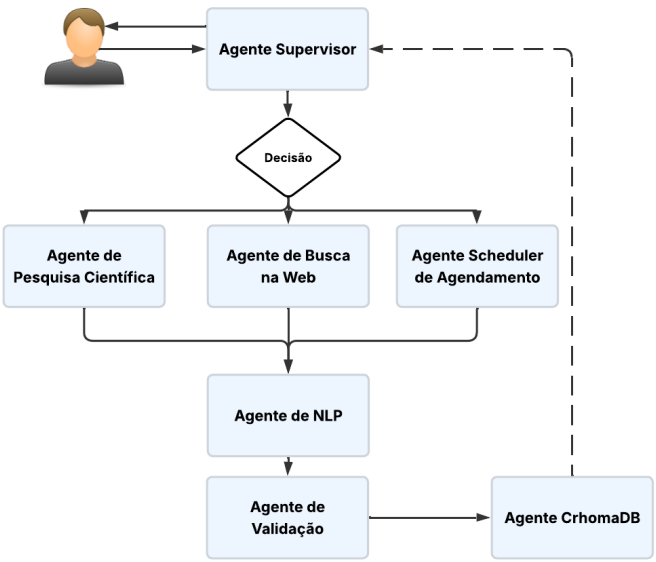


Figura 2.3: Fluxo interno de processamento de um agente especializado no *SAPIEN*

O design modular e desacoplado permite que novos agentes ou ferramentas sejam adicionados sem comprometer a arquitetura existente, promovendo escalabilidade e reutilização de código. Essa organização não apenas aprimora a manutenibilidade e a rastreabilidade do sistema, mas também o torna um recurso didático eficaz para o ensino de arquiteturas baseadas em agentes. A divisão lógica entre camadas possibilita que estudantes e pesquisadores explorem, modifiquem e estendam o *SAPIEN*, favorecendo experimentação e aprendizado em Inteligência Artificial aplicada à pesquisa científica.

2.3.1. Implementação do Sistema

A implementação do *SAPIEN* foi realizada utilizando o *framework* **LangGraph** para modelar o ciclo de execução e interação entre agentes. Cada nó do grafo representa um agente especializado, enquanto as arestas definem a ordem, as dependências e as condições de transição entre etapas de processamento.

O uso de grafos cíclicos no *LangGraph* permite a reavaliação contínua das respostas e o refinamento iterativo dos resultados, promovendo maior consistência semântica e precisão contextual. Esse mecanismo de retroalimentação possibilita que o sistema aprimore progressivamente a qualidade das inferências, ajustando suas decisões com base em verificações sucessivas e validações semânticas realizadas pelos agentes especializados.

2.4. Demonstração Prática e Casos de Uso

A fase de demonstração prática teve como objetivo validar o funcionamento integrado dos agentes que compõem o sistema *SAPIEN*, bem como avaliar sua aplicabilidade em tarefas reais de pesquisa científica. O experimento consistiu na execução de consultas automatizadas a bases de conhecimento acadêmico, com destaque para o repositório *arXiv* e para o mecanismo de busca científica na *web* fornecido pela ferramenta **Tavily AI**.

Durante a demonstração, o sistema foi executado em ambiente local, configurado conforme as etapas descritas na Seção 2.1. O acesso se deu por meio da interface *web* desenvolvida com o *framework* **Flask**, que possibilita a interação direta entre o usuário e o agente supervisor. As consultas foram formuladas em linguagem natural, e o agente supervisor interpretou os comandos, determinando o fluxo de execução e acionando os agentes especializados de acordo com o tipo, o domínio e a complexidade da solicitação.

2.4.1. Execução das Consultas

Foram realizadas diferentes categorias de consultas, com o intuito de demonstrar o potencial do sistema *SAPIEN* no apoio a atividades de pesquisa científica. A seguir, apresentam-se alguns exemplos de comandos executados durante a fase experimental:

- "Busque artigos sobre *transformers* em IA nos últimos dois anos."
- "Agende buscas semanais sobre IA aplicada à educação."
- "Cancele todas as buscas agendadas."

Ao receber uma solicitação, o **Agente Supervisor** inicia o processo iterativo de orquestração das tarefas. Inicialmente, o **Tavily Agent** realiza a busca em fontes abertas de pesquisa, enquanto o **arXiv Agent** coleta publicações recentes em repositórios científicos. Os resultados obtidos são encaminhados ao **NLP Agent**, responsável por extrair informações relevantes, normalizar metadados e executar análises linguísticas sobre os textos recuperados.

Na sequência, o **Validator Agent** aplica filtros semânticos baseados em *embeddings*, assegurando a relevância dos resultados e eliminando duplicidades ou publicações fora do escopo da consulta. Por fim, o **ChromaDB Agent** armazena os documentos processados no banco vetorial local, permitindo consultas futuras baseadas em similaridade semântica e facilitando a reutilização do conhecimento previamente adquirido.

A Figura 2.4 ilustra o protótipo prático da interface entre o usuário e o agente de pesquisa, evidenciando o fluxo de consulta, análise e resposta conduzido pelo sistema.

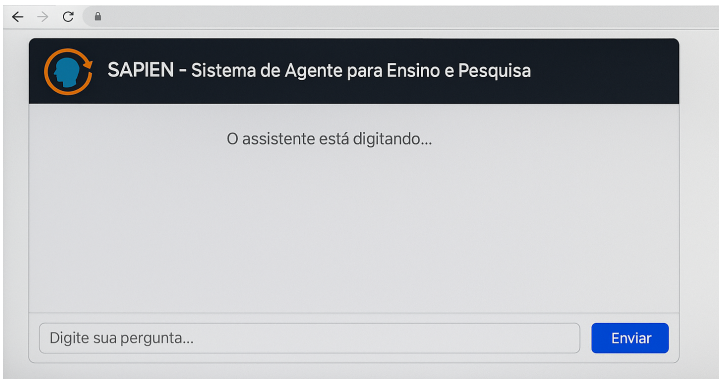


Figura 2.4: Interface de usuário

2.4.2. Casos de Uso Representativos

Entre os cenários avaliados, destacam-se três casos de uso práticos que ilustram o potencial do *SAPIEN* em apoiar pesquisadores e instituições acadêmicas:

1. **Monitoramento temático contínuo:** o sistema possibilita o agendamento de buscas recorrentes sobre tópicos específicos, como "redes neurais convolucionais" ou "aprendizagem federada", executadas em intervalos regulares. Os resultados são atualizados automaticamente, reduzindo a necessidade de acompanhamento manual e garantindo atualizações constantes da base de conhecimento.

2. **Exploração semântica de artigos:** graças à integração com o **ChromaDB**, o *SAPIEN* permite consultas baseadas em similaridade vetorial, possibilitando identificar publicações conceitualmente relacionadas, mesmo quando empregam terminologias distintas. Essa funcionalidade contribui para revisões de literatura mais amplas e interdisciplinares.

3. **Validação automatizada de relevância:** o agente de validação semântica compara os resumos recuperados com o contexto da consulta original, atribuindo pesos de relevância de acordo com a proximidade semântica. Essa etapa reduz significativamente ruídos e elimina resultados pouco pertinentes ao tema em análise.

2.5. Descrição do Código e Implementação dos Módulos

O sistema *SAPIEN* foi desenvolvido segundo princípios de modularidade, clareza estrutural e extensibilidade, favorecendo a integração eficiente entre agentes especializados. Sua implementação segue os fundamentos de engenharia de *software* aplicados a sistemas multiagente, em conformidade com o paradigma de orquestração proposto pelo *framework* **LangGraph**.

Nas subseções seguintes, são apresentados os principais arquivos e trechos de código que compõem o núcleo funcional do sistema, destacando-se o papel de cada módulo no fluxo geral de execução e as interações entre os componentes que formam a arquitetura multiagente do *SAPIEN*.

2.5.1. Roteamento e Inicialização da Aplicação

O ponto de entrada do sistema é o arquivo `app/routes.py`, responsável por inicializar o servidor *Flask* e definir as rotas de comunicação entre a interface web e o núcleo multiagente. A aplicação é instanciada por meio da função `create_app()`, que encapsula a configuração e segue o padrão de projeto *factory*, permitindo modularidade e escalabilidade na implantação em ambientes de produção.

```
from app import create_app
app = create_app()

if name == "main":
app.run(debug=True)
```

Esse design orientado à fábrica (*factory pattern*) favorece a separação de dependências e simplifica o processo de manutenção e expansão do sistema. Durante o desenvolvimento, o servidor é executado em modo de depuração (`debug=True`), possibilitando inspeção em tempo real. Para ambientes de produção, a configuração pode ser facilmente adaptada para execução em contêineres *Docker* ou plataformas em nuvem, garantindo portabilidade e segurança operacional.

2.5.2. Camada de Configuração e Recursos Compartilhados

O módulo `app/core/config.py` centraliza a configuração global do sistema, são carregadas as chaves de autenticação das APIs externas (**Anthropic** e **Tavily**) e instanciados os modelos correspondentes:

```
load_dotenv()

def _set_env(var: str):
    value = os.getenv(var)
    if not value:
        raise EnvironmentError(f"Vari vel de ambiente {var} n o
            definida.")
    return value

_set_env("TAVILY_API_KEY")
_set_env("ANTHROPIC_API_KEY")

....

# caches globais
adicionados_arxiv_links = set()
processed_content_hashes = set()
current_processed_data = None

# scheduler compartilhado
from apscheduler.schedulers.background import BackgroundScheduler
scheduler = BackgroundScheduler()
scheduler.start()
active_jobs = {}
```

Essa camada também gerencia o **BackgroundScheduler**, proveniente da biblioteca *APScheduler*, responsável pela execução automatizada de tarefas em segundo

plano. Por meio desse agendador, o sistema realiza buscas periódicas e atualizações contínuas na base de dados científica, permitindo o monitoramento automatizado de temas de interesse.

2.5.3. Gerenciamento de Estado Compartilhado

O módulo `shared_state.py` implementa um mecanismo leve de controle e compartilhamento de estado entre os agentes durante o ciclo de execução do sistema. Esse estado é mantido em memória e registra informações sobre conteúdos processados, *hashes* de validação e resultados de tarefas agendadas.

```
_current_processed_data: Optional[Dict[str, Any]] = None
_processed_content_hashes: Set[str] = set()
_scheduler_results: list = []
```

Funções auxiliares são responsáveis por definir, recuperar e limpar esses dados de forma controlada, assegurando coerência entre os componentes do sistema e evitando condições de corrida em execuções concorrentes. Esse design fornece uma camada de persistência temporária entre chamadas assíncronas, reduz redundâncias durante execuções iterativas e mantém a consistência do fluxo de informações entre os agentes.

2.5.4. Definição dos Agentes e do Supervisor

O módulo `agents.py` define os agentes especializados, cada um associado a uma função específica no pipeline de pesquisa científica. A criação de cada agente é realizada por meio do método `create_react_agent()` do *framework* **LangGraph**, que associa um modelo de linguagem a ferramentas e comportamentos personalizados.

Os principais agentes definidos são:

- **Tavily Agent**: executa buscas em fontes abertas e bases científicas na web;
- **arXiv Agent**: coleta publicações diretamente do repositório *arXiv*;
- **NLP Agent**: realiza o processamento e a normalização de conteúdo textual;
- **Validation Agent**: aplica filtragem semântica baseada em *embeddings*;
- **ChromaDB Agent**: armazena conteúdos vetorizados para consultas futuras;
- **Scheduler Agent**: gerencia tarefas recorrentes de coleta e atualização.

A coordenação entre os agentes é realizada pelo **Agente Supervisor**, configurado com base na arquitetura *Supervisor Tool-Calling*, conforme descrita na documentação oficial do **LangGraph** (Chase, 2024). Essa abordagem permite que o supervisor atue como controlador dinâmico, decidindo, em tempo de execução, qual agente deve ser invocado e quais parâmetros devem ser transmitidos em cada etapa do processo.

```
supervisor_graph = create_supervisor(
model=llm,
agents=[tavily_agent, arxiv_agent, sched_agent,
```

```
nlp_agent, validation_agent, chromadb_agent],
prompt=("Voc      um supervisor que coordena um sistema "
"multiagente de pesquisas cient ficas.")
)
compiled_supervisor = supervisor_graph.compile()
```

2.5.5. Construção do Grafo e Execução do Pipeline

O módulo `service.py` é responsável pela compilação do grafo principal de execução e pela definição da lógica de fluxo de mensagens entre os agentes. Ele utiliza a classe `StateGraph` do **LangGraph**, na qual cada nó representa um agente e cada aresta define a sequência de comunicação e dependência entre tarefas.

```
graph = StateGraph(StateSchema)
graph.add_node("supervisor", compiled_supervisor)
graph.add_edge(START, "supervisor")
compiled = graph.compile(checkpointer=MemorySaver())
```

A função `run()` inicia o processo a partir da entrada textual fornecida pelo usuário, criando um identificador único de *thread* para rastrear o contexto da sessão. Durante a execução, as mensagens são processadas de forma iterativa, e as respostas são filtradas para capturar apenas as geradas pelos agentes de alto nível, eliminando ruídos intermediários de comunicação. Essa estrutura de grafo permite ciclos de retroalimentação entre agentes, possibilitando reavaliações sucessivas de resultados e refinamento dinâmico das respostas, um diferencial importante frente a *pipelines* lineares tradicionais.

2.5.6. Camada Vetorial e Persistência

O módulo `vectorstore.py` concentra a criação e o gerenciamento do banco vetorial local, implementado com a biblioteca **ChromaDB**. Essa camada é fundamental para o armazenamento semântico e a recuperação eficiente de textos processados, viabilizando consultas baseadas em similaridade vetorial.

```
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/
    all-MiniLM-L6-v2")
vectorstore = Chroma(
collection_name="artigos_cientificos",
embedding_function=embeddings,
persist_directory="./chroma_db"
)
```

A combinação entre *embeddings* densos e armazenamento vetorial permite ao *SAPIEN* executar buscas semânticas de alta precisão, superando as limitações de consultas puramente lexicais. Esse mecanismo garante que documentos semanticamente relacionados possam ser recuperados mesmo quando não compartilham termos idênticos, favorecendo a descoberta de conhecimento e a ampliação de revisões bibliográficas automatizadas.

2.5.7. Ferramentas e Fluxo de Processamento

A camada de ferramentas (`app/core/tools`) é responsável pela implementação das funcionalidades específicas utilizadas pelos agentes do sistema. Cada ferramenta é encap-

sulada em uma função decorada com o modificador `@tool`, fornecido pelo *LangChain*, o que permite sua integração direta ao grafo de execução do *LangGraph*. Essas ferramentas representam as "ações atômicas" do sistema multiagente, seguindo o fluxo lógico de coleta, processamento, validação e armazenamento de dados científicos.

2.5.7.1. Processamento Linguístico (NLP Agent)

O módulo `nlp_process.py` implementa o agente responsável pela normalização linguística, enriquecimento de metadados e preparação de conteúdo para indexação vetorial. Durante a execução, o texto bruto é higienizado, os contadores de palavras e caracteres são extraídos, e um *hash* MD5 é gerado para rastreamento e prevenção de duplicatas. Esses dados são posteriormente armazenados no estado compartilhado do sistema, tornando-se disponíveis para uso pelos demais agentes.

```
normalized_content = re.sub(r'\s+', ' ', raw_content.strip())
content_hash = hashlib.md5(normalized_content.encode()).hexdigest()
set_current_processed_data(processed_data)
```

O agente também realiza a detecção automática do idioma predominante e registra a data de processamento, assegurando a rastreabilidade e a integridade dos documentos analisados. Essa etapa de pré-processamento é essencial para garantir consistência linguística e otimizar o desempenho das consultas semânticas posteriores.

2.5.7.2. Busca e Coleta de Dados Científicos

A etapa de coleta de informação é implementada por dois módulos complementares: `web_search_with_flow.py` e `simple_arxiv_search.py`. O primeiro utiliza a API do **Tavily AI** para efetuar buscas em fontes científicas reconhecidas — como *Nature*, *Science* e *IEEE*, enquanto o segundo acessa diretamente o repositório *arXiv* por meio de requisições HTTP.

```
url = f"http://export.arxiv.org/api/query?search_query=all:{query}
"
r = requests.get(url)
```

Os resultados obtidos são encaminhados ao pipeline completo de processamento (**NLP → Validação → ChromaDB**), assegurando consistência semântica e eliminando duplicidades de artigos já indexados. O agente de coleta foi projetado para lidar com diferentes formatos de entrada, *string*, dicionário ou instância *Pydantic*, o que amplia sua robustez e compatibilidade com o sistema supervisor. Essa flexibilidade reforça a capacidade adaptativa do *SAPIEN*, permitindo sua integração com múltiplas fontes de dados e fluxos de pesquisa automatizada.

2.5.7.3. Validação Semântica

O módulo `validate_content.py` é responsável pela etapa de verificação de relevância semântica, na qual o conteúdo processado é comparado com o tópico de pesquisa orig-

inal. Para isso, utiliza-se o modelo de embeddings *SentenceTransformer all-MiniLM-L6-v2*, com cálculo da similaridade de cosseno entre os vetores de contexto correspondentes.

```
similarity = cosine_similarity(text_embedding, topic_embedding)[0][0]
is_relevant = similarity >= threshold
```

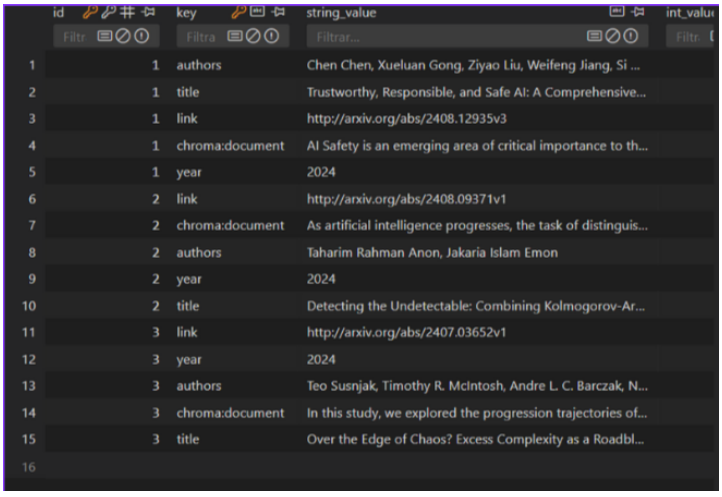
A função `validate_content()` também evita o reprocessamento de textos já avaliados, verificando previamente o estado global de *hashes* armazenados. Quando a similaridade ultrapassa o limiar configurado (geralmente 0.6), o documento é considerado relevante e encaminhado para o módulo de armazenamento vetorial. Esse procedimento assegura que apenas conteúdos semanticamente coerentes com o tema sejam indexados, reduzindo ruído e otimizando a precisão do sistema.

2.5.7.4. Armazenamento Vetorial no ChromaDB

O módulo `store_in_chromadb.py` finaliza o pipeline de processamento, transformando os documentos validados em representações vetoriais e persistindo-os na base *ChromaDB*. Para otimizar o desempenho das consultas por similaridade, o conteúdo textual é segmentado em blocos menores (*chunks*) por meio do *RecursiveCharacterTextSplitter*, o que garante granularidade adequada durante a vetorização.

```
splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
    chunk_overlap=200)
docs = splitter.split_documents([document])
vectorstore.add_documents(docs)
vectorstore.persist()
```

Após a persistência, o estado compartilhado é limpo para evitar redundâncias em execuções subsequentes. Esse mecanismo mantém a base vetorial organizada, consistente e preparada para consultas semânticas de alta eficiência, permitindo que o sistema realize buscas contextuais com grande precisão. A Figura 2.5 demonstra o armazenamento vetorial no banco de dados ChromaDB, sendo armazenado dados como autor, título, link, resumo e ano de publicação.



	id	key	string_value	int_valu
	Filtr	Filtra	Filtrar...	Filtr
1	1	authors	Chen Chen, Xueluan Gong, Ziyao Liu, Weifeng Jiang, Si ...	
2	1	title	Trustworthy, Responsible, and Safe AI: A Comprehensive...	
3	1	link	http://arxiv.org/abs/2408.12935v3	
4	1	chroma:document	AI Safety is an emerging area of critical importance to th...	
5	1	year	2024	
6	2	link	http://arxiv.org/abs/2408.09371v1	
7	2	chroma:document	As artificial intelligence progresses, the task of distinguis...	
8	2	authors	Taharim Rahman Anon, Jakaria Islam Emon	
9	2	year	2024	
10	2	title	Detecting the Undetectable: Combining Kolmogorov-Ar...	
11	3	link	http://arxiv.org/abs/2407.03652v1	
12	3	year	2024	
13	3	authors	Teo Susnjak, Timothy R. McIntosh, Andre L. C. Barczak, N...	
14	3	chroma:document	In this study, we explored the progression trajectories of...	
15	3	title	Over the Edge of Chaos? Excess Complexity as a Roadbl...	
16				

Figura 2.5: Armazenamento Semântico

2.5.7.5. Agendamento de Pesquisas Automáticas

O módulo `scheduler_tools.py` implementa as funções que permitem ao sistema executar pesquisas automáticas de forma periódica, a partir de instruções expressas em linguagem natural. Por exemplo, comandos como:

```
"pesquise sobre agentes inteligentes durante 2 minutos a cada 30
   segundos"
```

são interpretados por meio de expressões regulares, das quais são extraídos os parâmetros de tempo e de tema. As tarefas são então gerenciadas pelo *APScheduler*, que executa as rotinas em segundo plano sem interromper o fluxo principal da aplicação. O agente também oferece suporte para cancelamento de tarefas ativas e consulta de resultados concluídos.

```
scheduler.add_job(tarefa, 'interval', seconds=int_seg, id=job_id)
add_scheduler_result(f"[{tema}] {resultado}")
```

Essa funcionalidade permite que o *SAPIEN* opere de forma contínua e autônoma, realizando o monitoramento temático de publicações científicas sem necessidade de supervisão manual constante. Com isso, o sistema se aproxima de um modelo de *assistente de pesquisa persistente*, capaz de acompanhar tendências e atualizar periodicamente a base de conhecimento.

2.5.8. Síntese da Camada de Ferramentas

A camada de ferramentas constitui o elo entre a orquestração lógica, realizada pelo *LangGraph*, e a execução concreta das tarefas de pesquisa. Cada módulo é projetado para desempenhar uma função autônoma, porém cooperativa, permitindo que o **Agente Supervisor** componha fluxos de execução complexos de forma dinâmica e adaptativa.

Essa abordagem modular, baseada em *tool-calling*, segue o paradigma descrito na documentação do *LangGraph* (Chase, 2024), no qual o supervisor atua como um nó decisório que invoca agentes conforme a necessidade, utilizando chamadas de ferramenta parametrizadas. O resultado é uma arquitetura altamente extensível, na qual novos agentes ou ferramentas podem ser integrados ao pipeline com impacto mínimo na estrutura existente.

A integração entre esses módulos resulta em um sistema robusto e escalável, no qual cada componente desempenha um papel claramente definido dentro de uma arquitetura cooperativa de agentes. O uso do *LangGraph* viabiliza a modelagem declarativa do fluxo de execução, o *ChromaDB* assegura consultas semânticas de alta eficiência, e o *Flask* fornece uma interface web acessível para interação em linguagem natural. Essa combinação tecnológica oferece uma base sólida para expansões futuras, como a inclusão de agentes especializados em sumarização automática, tradução científica ou geração de relatórios temáticos.

2.5.9. Resultados e Benefícios Observados

Os resultados obtidos durante a fase de demonstração confirmaram que o sistema é capaz de executar fluxos multiagentes de forma coordenada, preservando o estado global e garantindo coerência entre as etapas do pipeline. Observou-se que, em consultas envolvendo múltiplas fontes de informação, o tempo médio de resposta foi significativamente reduzido em relação à pesquisa manual tradicional, especialmente em virtude da integração direta com APIs científicas, como *Tavily AI* e *arXiv*.

Além do ganho de eficiência, destaca-se a principal contribuição prática do sistema: a **redução da sobrecarga cognitiva** dos pesquisadores. O *SAPIEN* atua como um assistente inteligente capaz de compreender intenções em linguagem natural, sintetizar informações e apresentar resultados prontos para análise, diminuindo o esforço necessário para coleta e triagem de literatura científica.

Essa abordagem mostra-se especialmente promissora em contextos educacionais e corporativos, nos quais agentes especializados podem ser configurados para coletar, resumir e validar informações em domínios específicos. O uso combinado de *LLMs* e arquiteturas multiagentes, como a implementada no *SAPIEN*, amplia o potencial da automação científica, promovendo maior produtividade, reprodutibilidade e confiabilidade no ciclo de pesquisa.

2.6. Mitigação de Alucinações e Otimização

Para mitigar o fenômeno das alucinações, respostas incorretas ou infundadas geradas por modelos de linguagem, foram aplicadas estratégias de *re-ranking* e *cross-checking*, nas quais os resultados produzidos por diferentes agentes são comparados entre si para validação de consistência (Almeida da Silva *et al.*, 2024). Esse mecanismo permite priorizar respostas convergentes e reduzir a influência de ruído semântico proveniente de fontes isoladas.

A abordagem adotada segue práticas recentes de mitigação propostas em (Perov; Perova, 2024), complementando recomendações de transparência e rastreabilidade em sistemas de IA científica. Além disso, o design modular do *SAPIEN* possibilita incorporar verificadores externos ou heurísticas de controle de confiança (*confidence scoring*), aprimorando o controle de qualidade das respostas e a confiabilidade global do sistema.

2.7. Discussão e Considerações Éticas

A incorporação de agentes autônomos em processos de pesquisa científica representa um avanço expressivo, mas também levanta desafios éticos e epistemológicos relevantes. Modelos de linguagem podem refletir vieses inerentes aos dados de treinamento ou gerar respostas inconsistentes com o estado da arte (Bender *et al.*, 2021). Nesse contexto, a transparência, a responsabilidade e a explicabilidade tornam-se princípios fundamentais para o uso responsável de sistemas de IA (Meloni *et al.*, 2023).

O *SAPIEN* adota práticas de auditoria de histórico e monitoramento contínuo das interações entre agentes, garantindo rastreabilidade das decisões e permitindo a revisão humana dos resultados. Esses mecanismos reduzem o risco de propagação de erros e aumentam a confiabilidade científica, alinhando-se a princípios de governança e ética em

IA.

2.8. Conclusão

Este capítulo apresentou o desenvolvimento do sistema multiagente **SAPIEN**, uma plataforma baseada no *LangGraph* voltada à automação e otimização do processo de pesquisa científica. A arquitetura proposta combina agentes especializados em busca, processamento e validação, integrando múltiplas fontes de informação e mantendo um controle persistente de estado e contexto.

Os experimentos realizados demonstraram o potencial do uso combinado de *LLMs* e sistemas multiagentes para ampliar a eficiência, a qualidade e a rastreabilidade da produção científica. A solução proposta contribui não apenas para reduzir o esforço cognitivo de pesquisadores, mas também para estabelecer um modelo de orquestração modular e auditável, compatível com os princípios de transparência e reprodutibilidade da ciência moderna.

Como trabalhos futuros, pretende-se incorporar suporte a dados multimodais (texto, imagem e código-fonte), explorar métricas de confiança baseadas em evidência cruzada e desenvolver mecanismos colaborativos que permitam interação entre múltiplos usuários ou equipes de pesquisa em tempo real.

References

ALMEIDA DA SILVA, Wildemarkes de *et al.* Mitigation of Hallucinations in Language Models in Education: A New Approach of Comparative and Cross-Verification. *In:* 2024 IEEE International Conference on Advanced Learning Technologies (ICALT). [*S. l.: s. n.*], 2024. p. 207–209. DOI: 10.1109/ICALT61570.2024.00066.

ANTHROPIC. **Anthropic key**. [*S. l.: s. n.*], 2024. Acessado em 20 de março de 2025. Available from: https://www.anthropic.com/.

BARBARROXA, Rafael; GOMES, Luis; VALE, Zita. Benchmarking Large Language Models for Multi-agent Systems: A Comparative Analysis of AutoGen, CrewAI, and TaskWeaver. *In:* MATHIEU, Philippe; DE LA PRIETA, Fernando (eds.). **Advances in Practical Applications of Agents, Multi-Agent Systems, and Digital Twins: The PAAMS Collection**. Cham: Springer Nature Switzerland, 2024. p. 39–48. ISBN 978-3-031-70415-4.

BENDER, Emily M. *et al.* On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? *In:* PROCEEDINGS of the 2021 ACM Conference on Fairness, Accountability, and Transparency. Virtual Event, Canada: Association for Computing Machinery, 2021. (FAccT '21), p. 610–623. ISBN 9781450383097. DOI: 10.1145/3442188.3445922. Available from: https://doi.org/10.1145/3442188.3445922.

CHASE, Harrison. **LangGraph Documentation**. [*S. l.: s. n.*], 2024. Acessado em 14 de março de 2025. Available from: https://langchain-ai.github.io/langgraph/.

JORDAN, M. I.; MITCHELL, T. M. Machine learning: Trends, perspectives, and prospects. **Science**, v. 349, n. 6245, p. 255–260, 2015. DOI: 10.1126/science.aaa8415. eprint: https://www.science.org/doi/pdf/10.1126/science.aaa8415. Available from: https://www.science.org/doi/abs/10.1126/science.aaa8415.

KHANDARE, Anand *et al.* Analysis of Python Libraries for Artificial Intelligence. *In:* BALAS, Valentina Emilia; SEMWAL, Vijay Bhaskar; KHANDARE, Anand (eds.). **Intelligent Computing and Networking**. Singapore: Springer Nature Singapore, 2023. p. 157–177. ISBN 978-981-99-0071-8.

LUO, Tianze *et al.* Multi-Agent Collaborative Exploration through Graph-based Deep Reinforcement Learning. *In:* 2019 IEEE International Conference on Agents (ICA). [*S. l.: s. n.*], 2019. p. 2–7. DOI: 10.1109/AGENTS.2019.8929168.

MATHUR, Srushti; CHHABRA, Aayush. Vector Search Algorithms: A Brief Survey. *In:* 2024 4th International Conference on Ubiquitous Computing and Intelligent Information Systems (ICUIS). [*S. l.: s. n.*], 2024. p. 365–371. DOI: 10.1109/ICUIS64676.2024.10866377.

MELONI, Antonello *et al.* Integrating Conversational Agents and Knowledge Graphs Within the Scholarly Domain. **IEEE Access**, v. 11, p. 22468–22489, 2023. DOI: 10.1109/ACCESS.2023.3253388.

NASCIMENTO, Larissa Souza do; BUDARUICHE, Ricardo Moura Sekeff. Como Criar seu Próprio Assistente de Pesquisa Científica com LangGraph. *In:* ANAIS do ENUCOMPI 2025. [*S. l.*]: Sociedade Brasileira de Computação, 2025. DOI: 10.5753/sbc.16935.3.3. Available from: https://doi.org/10.5753/sbc.16935.3.3.

PEROV, Vadim; PEROVA, Nina. AI Hallucinations: Is “Artificial Evil” Possible? *In:* 2024 IEEE Ural-Siberian Conference on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT). [*S. l.: s. n.*], 2024. p. 114–117. DOI: 10.1109/USBEREIT61901.2024.10584048.

TAVILY. **Tavily key**. [*S. l.: s. n.*], 2024. Acessado em 20 de março de 2025. Available from: https://www.tavily.com/.

Chapter 3

Explorando Testes End-to-End com Playwright: Um Convite à Automação de Qualidade

Matusalen Costa Alves, Iallen Gábio de Santos Sousa, Mayllon Veras da Silva

Abstract

This chapter explores the discipline of software testing and the importance of End-to-End testing in ensuring the quality of modern web applications. The study introduces the Playwright framework, which provides a modern and robust solution for creating reliable automated tests. It includes a guide to setting up the environment, writing tests using actions and assertions, and analyzing execution reports. Furthermore, it demonstrates the practical application of these concepts through the automation of a "ToDo List" application, highlighting the use of the Page Object Model pattern, code generation with CodeGen, and debugging techniques with the Trace Viewer.

Resumo

Este capítulo explora a disciplina de testes de software e a importância dos testes End-to-End na garantia da qualidade de aplicações web modernas. O estudo apresenta o framework Playwright, que oferece uma solução moderna e robusta para a criação de testes automatizados confiáveis. Inclui um guia para a configuração do ambiente, a escrita de testes com o uso de ações e asserções, e a análise de relatórios de execução. Além disso, demonstra a aplicação prática desses conceitos por meio da automação de uma aplicação "ToDo List", evidenciando o uso do padrão Page Object Model, geração de código com o CodeGen e de técnicas de depuração com o Trace Viewer.

3.1. Introdução

No cenário contemporâneo do desenvolvimento de software, a entrega de produtos digitais de alta qualidade transcendeu o status de diferencial competitivo para se tornar um requisito fundamental para a relevância e o sucesso de qualquer projeto. A complexidade crescente das aplicações, caracterizadas por arquiteturas distribuídas, interfaces interativas e a necessidade de compatibilidade com uma vasta gama de dispositivos e navegadores

impõe desafios às equipes de desenvolvimento. Nesse contexto, a garantia da qualidade deixa de ser uma fase isolada no final do ciclo de vida para se consolidar como uma tarefa contínua e integrada a todas as etapas da produção [Sommerville 2019].

No âmbito dos testes, um teste automatizado é um processo de verificação e validação de software que utiliza ferramentas e scripts para executar rotinas de checagem sem intervenção humana [Sommerville 2019]. Esta abordagem substitui tarefas repetitivas e manuais, o que permite a execução de um grande volume de testes de forma rápida e consistente.

A qualidade é um pilar em Engenharia de Software. Conforme preconiza Robert C. Martin, a conduta de um programador profissional exige a certeza de que o código entregue funciona como esperado, e o conjunto de testes (também conhecido como suíte de testes) automatizados é um dos principais mecanismo para prover essa garantia. A ausência de testes compromete a funcionalidade do produto e introduz o conhecido "débito técnico" que dificulta a manutenção e a evolução do sistema a longo prazo [Martin 2012].

A prática da automação de testes é, portanto, a fundação sobre a qual a agilidade e a sustentabilidade de projetos modernos estão calcadas. Martin Fowler argumenta que a capacidade de refatorar o código (aperfeiçoar seu design interno sem alterar seu comportamento externo) é diretamente dependente da existência de uma rede de testes confiáveis fornecida pela suíte de testes automatizados [Fowler 2020]. Sem essa rede, qualquer alteração se torna arriscada. Essa condição dificulta a melhoria contínua e a capacidade da equipe de responder rapidamente a novas demandas.

A relevância dessa disciplina é ainda mais acentuada pelo advento de novas tecnologias, como os Modelos de Linguagem de Grande Escala (LLMs), que têm sido cada vez mais utilizados para a geração automática de código. Embora essas ferramentas possam acelerar o desenvolvimento, elas também introduzem a necessidade de uma verificação rigorosa, visto que a revisão manual de todo o código gerado é, muitas vezes, impraticável [Vaithilingam et al. 2022]. Nesse novo paradigma, os testes automatizados tornam-se essenciais para validar o software.

Para organizar as estratégias de validação, a indústria adota modelos como a pirâmide de testes. Esta é organizada em uma base larga de testes unitários, uma camada intermediária de testes de integração e, no topo, uma camada mais seleta de testes End-to-End. Estes últimos são cruciais por simularem a jornada completa do usuário; eles validam fluxos do início ao fim e garantem que todos os componentes do sistema funcionem de maneira coesa.

Neste minicurso, materializado na forma deste capítulo, nos concentraremos na criação de testes End-to-End com o uso do Playwright, uma ferramenta moderna mantida pela Microsoft. O Playwright se destaca por oferecer uma solução eficiente, rápida e confiável para a automação de interações em navegadores, o que o torna uma escolha adequada para enfrentar os desafios discutidos.

O restante deste capítulo está organizado da seguinte maneira: a Seção 3.2 aprofunda os conceitos da disciplina de testes de software; a Seção 3.3 apresenta a arquitetura e os diferenciais do ecossistema Playwright; a Seção 3.4 detalha os passos práticos para a configuração e utilização da ferramenta; a Seção 3.5 demonstra, através de um estudo de

caso, a automação de uma aplicação real; e, por fim, a Seção 3.6 conclui o trabalho com uma síntese dos aprendizados e sugestões para estudos futuros.

3.2. A Disciplina de Testes de Software

A verificação e validação são disciplinas fundamentais da Engenharia de Software, responsáveis por assegurar que um sistema computacional atenda às suas especificações e satisfaça as necessidades dos seus usuários. Dentro deste escopo, a prática de testes de software se estabelece como o principal mecanismo para identificar defeitos e avaliar a qualidade de um produto. Esta seção explora os conceitos essenciais desta disciplina, a começar por uma análise da evolução histórica dos testes automatizados. Em seguida, apresentaremos a Pirâmide de Testes, o modelo estratégico mais influente para a organização de suítes de testes, e, por fim, detalharemos o papel crítico dos testes End-to-End, que são o foco deste capítulo.

3.2.1. História e Evolução dos Testes Automatizados

A prática de automatizar testes de software evoluiu em paralelo com as próprias metodologias de desenvolvimento. Nas abordagens mais tradicionais, como o modelo em cascata, os testes eram frequentemente relegados a uma fase final e executados de forma predominantemente manual, um processo lento, repetitivo e suscetível a falhas humanas. A necessidade de otimizar essa etapa impulsionou o surgimento das primeiras ferramentas de automação, muitas baseadas em scripts simples ou em técnicas de captura e repetição de interações do usuário.

A Figura 3.1 ilustra os marcos dessa progressão. A linha do tempo demonstra a transição de um modelo sequencial e manual, associado ao modelo cascata, para ciclos iterativos e ágeis, que culminaram nas práticas de testes contínuos e na exploração de testes autônomos com o avanço da inteligência artificial.

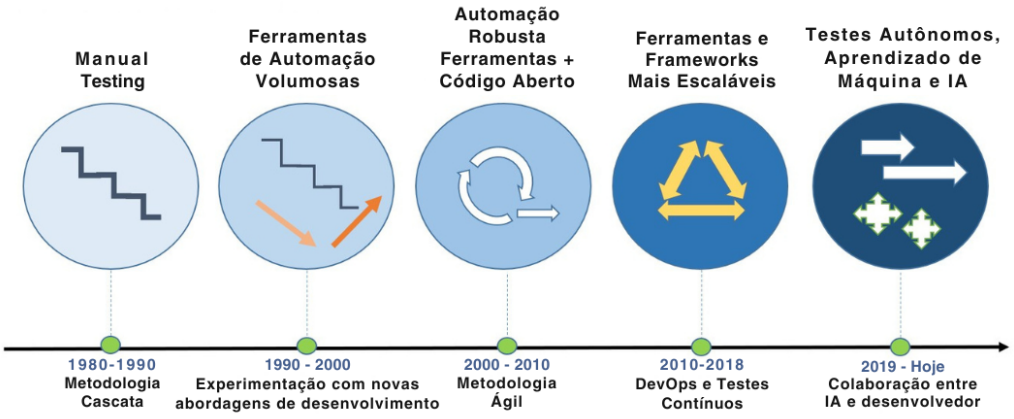


Figure 3.1. Marcos da evolução da automação de testes, da metodologia cascata aos testes contínuos e autônomos.

Uma mudança de paradigma ocorreu com a ascensão das metodologias ágeis no final da década de 1990. A partir de então, os testes passaram a ser vistos não apenas

como uma atividade de verificação final, mas como uma parte intrínseca do processo de desenvolvimento e design. A criação de frameworks de teste, como os da família xUnit, foi fundamental para essa transformação, pois forneceu aos desenvolvedores as ferramentas para escrever testes de forma sistemática e integrar a automação ao processo de codificação.

Essa evolução foi acelerada pela consolidação da cultura DevOps e das esteiras de Integração e Entrega Contínua (CI/CD), que tornaram a automação de testes um requisito essencial. Em um ciclo de vida onde novas versões do software são liberadas com alta frequência, a execução manual de testes de regressão se torna impraticável. A automação passou a ser, portanto, o pilar que garante a segurança e a agilidade necessárias para a inovação contínua [Sommerville 2019].

3.2.2. A Pirâmide de Testes: Estratégias e Níveis

Com a proliferação dos testes automatizados, tornou-se necessária a criação de um modelo estratégico para orientar sua implementação de forma eficiente. O modelo mais amplamente adotado pela indústria é a Pirâmide de Testes, um conceito originalmente proposto por Mike Cohn [Cohn 2009]. A pirâmide é uma heurística visual que descreve a proporção ideal entre diferentes tipos de testes em uma suíte de automação.

A Figura 3.2 representa visualmente este modelo. A largura de cada camada sugere o volume ideal de testes, enquanto os ícones laterais ilustram as características de cada nível: a base é a mais rápida e de menor custo, enquanto o topo é o mais lento e de maior custo.

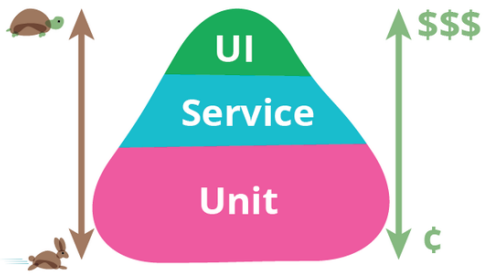


Figure 3.2. Representação da Pirâmide de Testes e suas características de velocidade e custo.

A base da pirâmide é composta pelos Testes de Unidade (*Unit Tests*). Estes testes verificam os menores componentes do sistema, como uma função ou uma classe, de forma isolada. São caracterizados por sua alta velocidade de execução e baixo custo de manutenção. Por testarem a lógica de negócio em seu nível mais granular, eles devem constituir a maior parte da suíte de testes.

A camada intermediária é composta pelos Testes de Integração, por vezes chamados de Testes de Serviço (*Service Tests*). O objetivo destes testes é verificar a interação entre dois ou mais componentes do sistema, como a comunicação entre um serviço de aplicação e o banco de dados. São mais lentos e complexos que os testes de unidade, pois envolvem múltiplos componentes, e, por isso, devem existir em menor número.

No topo da pirâmide, encontram-se os Testes End-to-End. Estes testes validam um fluxo completo do sistema sob a perspectiva do usuário final, o que geralmente envolve a interação com a interface gráfica. Conforme detalhado por Martin Fowler, embora os testes End-to-End ofereçam a maior confiança sobre o funcionamento do sistema, eles também são os mais lentos, frágeis e caros para manter. Portanto, a estratégia da pirâmide recomenda que eles sejam utilizados de forma seletiva, focados nos fluxos mais críticos do negócio [Fowler 2012].

3.2.3. O Papel Crítico dos Testes End-to-End

Um teste End-to-End (E2E) é uma técnica de teste que simula um cenário de usuário real do início ao fim. Diferentemente dos testes de unidade e integração, que operam em camadas mais baixas e com partes isoladas do código, um teste E2E interage com o sistema através da sua interface de usuário, da mesma forma que um cliente faria. O seu escopo abrange todas as camadas da arquitetura da aplicação, desde a interface gráfica no navegador até os serviços de backend e o banco de dados.

A Figura 3.3 ilustra de forma esquemática este processo. A interação do usuário ocorre na camada mais externa, a interface (representada pelo formulário), e desencadeia uma série de operações que atravessam os diversos componentes da arquitetura do sistema (representados pelos cubos), como serviços de aplicação e bancos de dados. Um teste E2E bem-sucedido valida a integridade de todo esse percurso.

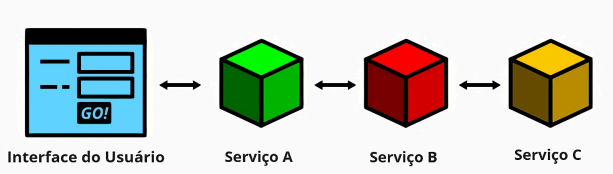


Figure 3.3. Exemplo do fluxo de um teste End-to-End, da interface aos componentes do sistema.

O principal valor dos testes E2E reside na sua capacidade de fornecer um alto grau de confiança de que a aplicação, como um todo, está funcionando corretamente e atendendo aos requisitos de negócio. Ao validar fluxos completos, como um processo de cadastro de usuário ou a finalização de uma compra em um e-commerce, os testes E2E garantem que a integração entre os diversos componentes do sistema está operando de forma coesa.

Contudo, a implementação de testes E2E apresenta desafios. A sua natureza integrada os torna inerentemente mais lentos, pois dependem de operações de rede, renderização de interface e acesso a banco de dados. Eles também são mais frágeis, ou seja, podem falhar devido a pequenas alterações na interface do usuário que não necessariamente representam um defeito na lógica de negócio. Por essa razão, a sua criação e manutenção exigem um planejamento cuidadoso e a aplicação de padrões de projeto específicos, como discute Gerard Meszaros em sua obra sobre padrões de teste [Meszaros 2007]. Apesar desses desafios, os testes E2E são uma camada indispensável em uma estratégia de qualidade, pois estão entre os poucos capazes de validar a experiência completa do usuário.

3.3. O Ecossistema Playwright

Após a fundamentação teórica sobre a disciplina de testes de software, esta seção direciona o foco para a ferramenta central deste capítulo: o Playwright. O objetivo é realizar uma imersão técnica em seu ecossistema, a fim de demonstrar por que ele se estabelece como uma solução moderna e eficaz para os desafios da automação de testes E2E.

Iniciaremos com uma análise detalhada do conceito e da arquitetura que garantem a velocidade e a confiabilidade da ferramenta. Em seguida, exploraremos suas funcionalidades-chave e os diferenciais que otimizam a experiência de desenvolvimento. Por fim, faremos uma análise comparativa que posiciona o Playwright em relação a outras ferramentas consolidadas no mercado.

3.3.1. Conceito e Arquitetura

O Playwright é um framework de automação de código aberto, mantido pela Microsoft, projetado para atender às demandas do desenvolvimento de aplicações web modernas. Seu objetivo é fornecer uma API única, coesa e poderosa para a automação dos três principais motores de renderização de navegadores: Chromium (utilizado por Google Chrome e Microsoft Edge), WebKit (utilizado pelo Apple Safari) e Firefox. A filosofia do projeto se concentra em três pilares: velocidade, capacidade e, principalmente, confiabilidade, para eliminar a instabilidade que historicamente afeta os testes de interface de usuário.

A Figura 3.4 ilustra a arquitetura da ferrameta. No lado do cliente, os testes podem ser escritos em diversas linguagens, como TypeScript, JavaScript e Python. Esses scripts enviam instruções ao servidor Playwright, que as traduz em comandos específicos para o protocolo de depuração do navegador. A conexão WebSocket garante uma comunicação bidirecional e eficiente, permitindo que o Playwright controle o navegador com precisão e receba eventos em tempo real. Essa estrutura é a base para muitas das funcionalidades avançadas da ferramenta, como a capacidade de interceptar requisições de rede e a execução de testes em múltiplos contextos de forma isolada.

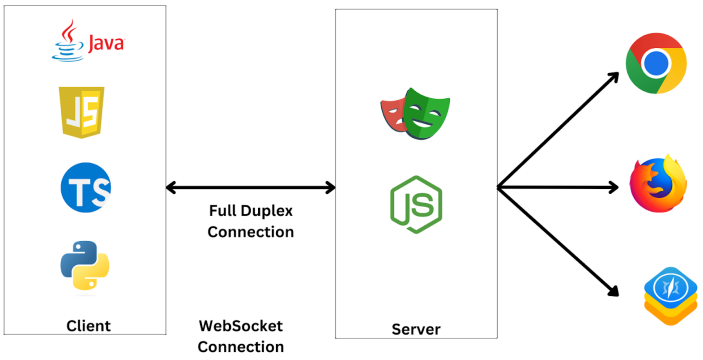


Figure 3.4. Visão geral da arquitetura de comunicação do Playwright.

O principal diferencial técnico do Playwright reside em sua arquitetura. Diferentemente de soluções mais antigas que dependem de protocolos baseados em HTTP para a

comunicação entre o script de teste e o navegador, o Playwright opera em um modelo fora do processo. Nele, o script de teste se comunica com um servidor Node.js que, por sua vez, envia comandos aos navegadores por meio de uma conexão WebSocket persistente. Essa comunicação direta e de baixa latência evita pontos de falha e gargalos de desempenho, o que resulta em uma execução de testes mais rápida e estável [Microsoft 2025].

3.3.2. Funcionalidades-Chave e Diferenciais

O Playwright se distingue por um conjunto de funcionalidades nativas projetadas para otimizar a experiência de desenvolvimento e aumentar a confiabilidade dos testes. Estes recursos abordam desafios comuns na automação de testes, como a instabilidade, a dificuldade de depuração e a complexidade na criação de novos scripts. A seguir, detalharemos as ferramentas que compõem esses diferenciais.

3.3.2.1. Trace Viewer: Depuração de Viagem no Tempo

Um dos maiores desafios dos testes E2E é a depuração. Testes que falham em um ambiente de integração contínua podem ser difíceis de diagnosticar, pois o desenvolvedor não tem acesso ao estado do navegador no momento da falha. O Playwright soluciona este problema com o Trace Viewer, uma de suas ferramentas mais poderosas.

Ao final de uma execução de testes, o Playwright gera um relatório em HTML, como o apresentado na Figura 3.5, que exibe o resultado de cada teste executado em diferentes navegadores. Este relatório centraliza os resultados e serve como ponto de partida para a análise.

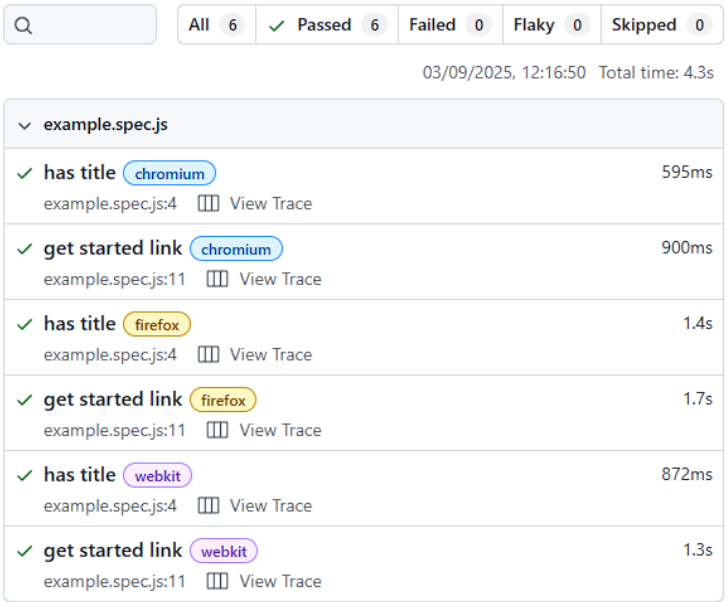


Figure 3.5. Relatório de testes do Playwright com a lista de execuções.

A partir do relatório, é possível inspecionar cada teste individualmente para visualizar os passos executados, como ganchos (**hooks**), ações e asserções, conforme ilustrado na Figura 3.6. Para uma análise mais profunda, o relatório oferece acesso ao histórico completo da execução.

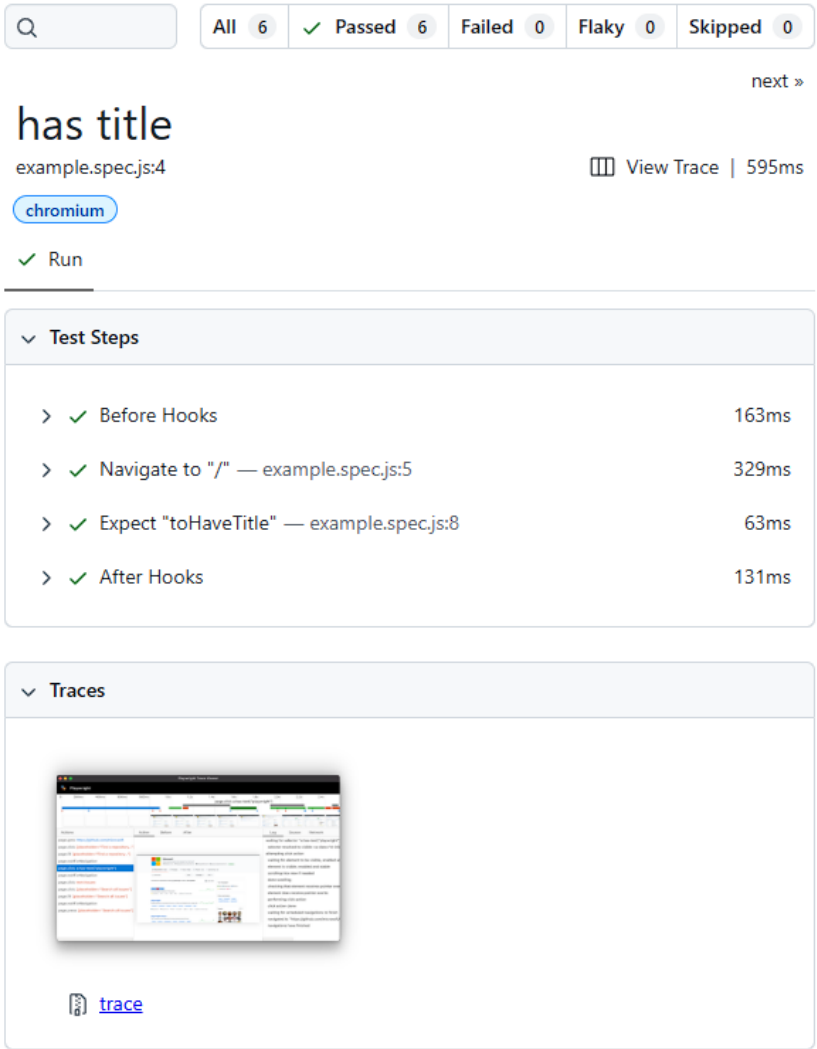


Figure 3.6. Visualização detalhada de um teste específico no relatório.

A interface do Trace Viewer, exibida na Figura 3.7, proporciona uma experiência de depuração de "viagem no tempo". Ela captura um traço completo da execução do teste, permitindo que o desenvolvedor navegue pela linha do tempo e inspecione o estado da aplicação em cada momento. A ferramenta exibe a lista de ações, o snapshot do DOM antes e depois de cada ação, logs do console e requisições de rede. Esse nível de detalhe contextualizado reduz drasticamente o tempo necessário para identificar a causa raiz de uma falha [Microsoft 2025].

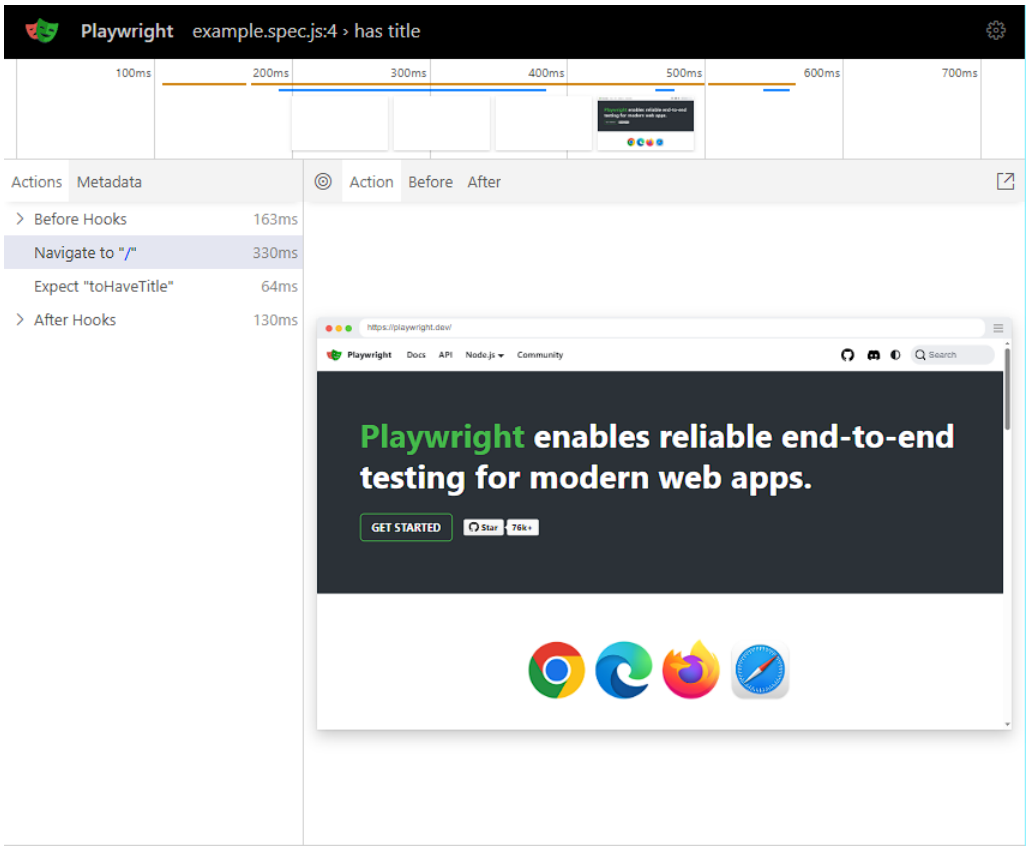


Figure 3.7. Interface principal do Trace Viewer com a linha do tempo, ações e o snapshot do DOM.

3.3.2.2. Auto-waits: O Fim da Instabilidade

Um dos problemas mais comuns na automação de testes é a instabilidade (flakiness), onde testes falham de forma intermitente sem uma causa aparente. Frequentemente, a raiz do problema está em condições de corrida (race conditions), nas quais o script de teste tenta interagir com um elemento da interface antes que ele esteja totalmente carregado ou pronto para receber uma ação. A solução tradicional para isso envolvia a inserção de pausas fixas ou esperas explícitas no código, o que o torna mais lento e complexo.

O Playwright resolve este problema de forma nativa com seu mecanismo de esperas automáticas, conhecido como auto-wait. Antes de executar qualquer ação, como um clique ou o preenchimento de um campo, a ferramenta realiza uma série de verificações para garantir que o elemento alvo esteja pronto para a interação. Essas checagens incluem verificar se o elemento está visível, estável (sem animações em andamento) e habilitado para receber eventos.

Essa inteligência nativa simplifica enormemente o código do teste. A Figura 3.8 demonstra a clareza de um script de teste em Playwright, onde os comandos são diretos e focados na intenção do usuário. Por trás dessa simplicidade, a ferramenta gerencia a

sincronização.

Locator Call Log Errors Console Network 31 Source Attachments

example.spec.js

```
1 // @ts-check
2 import { test, expect } from '@playwright/test';
3
4 test('has title', async ({ page }) => {
5   await page.goto('https://playwright.dev/');
6
7   // Expect a title "to contain" a substring.
8   await expect(page).toHaveTitle(/Playwright/);
9 });
10
11 test('get started link', async ({ page }) => {
12   await page.goto('https://playwright.dev/');
13
14   // Click the get started link.
15   await page.getByRole('link', { name: 'Get started' }).click();
16
17   // Expects page to have a heading with the name of Installation.
18   await expect(page.getByRole('heading', { name: 'Installation' })).toBeVisible();
19 });
20
```

Figure 3.8. Exemplo de código de teste em Playwright, com comandos diretos.

Na Figura 3.9, extraída do log de uma execução, mostra que, mesmo em um comando como ‘page.goto‘, o Playwright aguarda ativamente por eventos específicos da página, como o evento "load". Esse mecanismo de espera automática é um dos principais motivos da alta confiabilidade dos testes escritos com Playwright [Microsoft 2025].

Locator Call Log Errors Console Network 31 Source Attachments

navigating to "https://playwright.dev/", waiting until "load" 317ms

Figure 3.9. Log de execução de um comando, que evidencia a espera por um evento específico da página.

3.3.2.3. Codegen: Geração de Testes Acelerada

Para acelerar a criação de novos testes e diminuir a barreira de entrada para novos usuários, o Playwright inclui a ferramenta Codegen. Ao ser iniciada, ela abre uma janela de navegador junto com uma janela "Playwright Inspector". À medida que o usuário interage com a aplicação no navegador (clica em botões, preenche formulários, navega entre páginas), o Codegen grava essas ações e as traduz em tempo real para código Playwright.

A Figura 3.10 demonstra a ferramenta em ação. Na parte superior, o navegador exibe a aplicação web sendo testada, enquanto na parte inferior, o "Playwright Inspector" exibe o código gerado a partir das interações. Esta funcionalidade é extremamente útil tanto para aprender a sintaxe da API do Playwright quanto para criar rapidamente o

esqueleto de um novo teste, que pode então ser refinado e adaptado pelo desenvolvedor [Microsoft 2025].

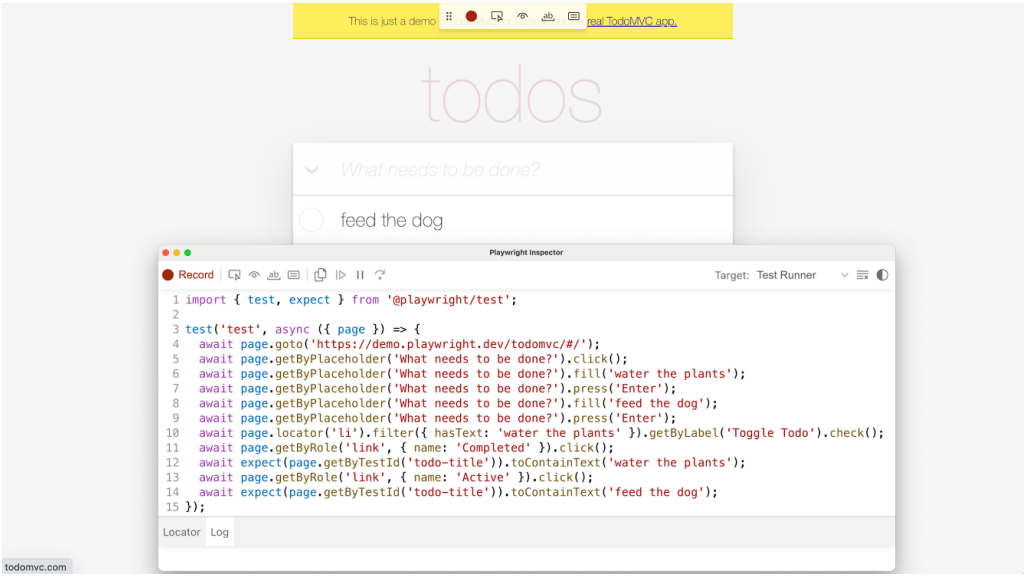


Figure 3.10. Ferramenta Codegen em execução, com o navegador e a janela do inspetor que gera o código em tempo real.

3.3.2.4. Suporte Multi-Navegador e Execução Paralela

Finalmente, um dos maiores diferenciais do Playwright é seu suporte nativo e de primeira classe aos três principais motores de renderização de navegadores: Chromium, Firefox e WebKit. Isso permite que as equipes de desenvolvimento validem o comportamento de suas aplicações em um ambiente verdadeiramente multiplataforma, o que garante uma experiência de usuário consistente.

A capacidade de executar o mesmo conjunto de testes em diferentes navegadores é evidenciada no relatório de testes, como já demonstrado na Figura 3.5, onde cada teste foi validado nos três motores. Adicionalmente, o Playwright foi projetado para executar testes em paralelo por padrão, distribuindo-os entre múltiplos processos de trabalho. Essa abordagem reduz drasticamente o tempo total de execução da suíte de testes, um fator crucial em ambientes de integração contínua.

3.3.3. Análise Comparativa

Para compreender o valor e os diferenciais do Playwright, é útil posicioná-lo no ecossistema de ferramentas de automação de testes E2E. Embora existam diversas soluções, o mercado foi historicamente dominado pelo Selenium, com o Cypress emergindo como uma alternativa moderna popular. Cada uma dessas ferramentas possui uma arquitetura e uma filosofia distintas, que resultam em diferentes vantagens e desvantagens. As informações apresentadas na Tabela 3.1 foram compiladas a partir da documentação oficial de cada ferramenta [Microsoft 2025, Selenium 2025, Cypress 2025].

Table 3.1. Análise Comparativa de Frameworks de Testes E2E.

Critério	Playwright	Selenium	Cypress
Suporte a Linguagens	JavaScript, Java, Python, .NET	JavaScript, Java, C#, Python, etc.	JavaScript
Driver do Navegador	Não requer driver	Requer um driver para cada navegador	Não requer driver
Relatórios Nativos	Sim	Não	Sim
Recursos de Depuração	Ferramentas nativas e depuração "time-traveling"	Não possui ferramentas de depuração nativas	Ferramentas nativas e depuração "time-traveling"
Esperas Automáticas	Sim	Não	Sim

A tabela destaca as diferenças na experiência do desenvolvedor. O Selenium, com seu vasto suporte a linguagens, oferece grande flexibilidade, mas exige uma configuração mais complexa, como a gestão de drivers específicos para cada navegador. Por outro lado, Cypress e Playwright proporcionam uma experiência mais integrada, com relatórios nativos, esperas automáticas e ferramentas de depuração avançadas que simplificam a escrita e a manutenção dos testes.

Já o Playwright, diferente das duas outras alternativas, se posiciona de forma vantajosa ao combinar o amplo suporte a linguagens, similar ao do Selenium, com as conveniências modernas encontradas no Cypress, estabelecendo-se como uma solução que equilibra poder e facilidade de uso.

3.4. Instalando o Playwright no Projeto

Após a apresentação conceitual do ecossistema Playwright, esta seção inicia a jornada prática. O objetivo é guiar o leitor através dos passos fundamentais para a criação e execução de um projeto de testes do início ao fim.

Iniciaremos com as instruções para a instalação e configuração completa do ambiente de desenvolvimento.

Em seguida, faremos uma análise detalhada da anatomia de um arquivo de teste, explicando seus componentes essenciais, como comandos e asserções.

Por fim, demonstraremos como executar a suíte de testes e analisar os relatórios de resultados gerados pela ferramenta.

3.4.1. Configuração do Ambiente

O primeiro passo para utilizar o Playwright é a preparação do ambiente de desenvolvimento. O principal pré-requisito é ter o Node.js instalado, que inclui o gerenciador de pacotes npm. Com o ambiente Node.js pronto, a instalação do Playwright é realizada por meio de um único comando que inicia um assistente de configuração interativo.

Para iniciar um novo projeto ou adicionar o Playwright a um projeto existente, o comando a ser executado no terminal é o apresentado no Código 3.1.

```
npm init playwright@latest
```

Listing 3.1. Comando de inicialização do Playwright via npm.

Ao executar este comando, o assistente de configuração fará algumas perguntas para personalizar o projeto, como:

- A escolha entre TypeScript ou JavaScript para a escrita dos testes.
- O nome do diretório onde os testes serão armazenados (o padrão é *tests*).
- A adição de um fluxo de trabalho para o GitHub Actions, útil para a integração contínua.
- A confirmação para instalar os navegadores necessários (Chromium, Firefox e WebKit).

Após a conclusão do assistente, o Playwright criará uma estrutura de arquivos e diretórios no projeto. Essa estrutura inicial, conforme detalhado na documentação e demonstrado no Código 3.2, organiza os arquivos de forma lógica e inclui exemplos para facilitar os primeiros passos do desenvolvedor [Microsoft 2025].

```
.
|-- playwright.config.ts
|-- package.json
|-- tests/
|   |-- example.spec.ts
|-- tests-examples/
    |-- demo-todo-app.spec.ts
```

Listing 3.2. Estrutura de arquivos gerada pela instalação do Playwright.

O arquivo *playwright.config.ts*, presente na estrutura do Código 3.2, é o centro de controle do projeto, onde são definidas configurações como os navegadores alvo, tempos de espera e formatos de relatório. O diretório *tests/* é o local padrão para os testes do projeto, enquanto o *tests-examples/* contém exemplos mais elaborados que demonstram diferentes funcionalidades da ferramenta.

3.4.2. Anatomia de um Teste: Comandos e Asserções

Um teste em Playwright, em sua essência, é uma sequência de duas operações fundamentais: a execução de ações para simular a interação do usuário com a página e a verificação do estado da aplicação por meio de asserções. A estrutura de um teste é declarada dentro de uma função *test()*, que recebe como argumento a fixture *page*, um objeto que representa uma única aba no navegador e serve como a principal interface para a automação.

O Código 3.3 apresenta um arquivo de teste completo, *example.spec.ts*, que contém dois cenários de teste distintos. O primeiro verifica se a página possui o título esperado, e o segundo valida a navegação ao clicar em um link e checar o conteúdo da nova página.

```
import { test, expect } from '@playwright/test';

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
```

```
5   // Espera que o título da página contenha o texto "Playwright
        ".
6   await expect(page).toHaveTitle(/Playwright/);
7 });
8
9 test('get started link', async ({ page }) => {
10  await page.goto('https://playwright.dev/');
11  // Clica no link com o nome "Get started".
12  await page.getByRole('link', { name: 'Get started' }).click();
13  // Espera que a página possua um cabeçalho com o nome "
        Installation".
14  await expect(page.getByRole('heading', { name: 'Installation'
        })).toBeVisible();
15 });
```

Listing 3.3. Exemplo de um arquivo de teste completo em Playwright.

As ações em um teste começam, geralmente, com a navegação para uma URL, como visto na linha *await page.goto('https://playwright.dev/')*. Após o carregamento da página, o teste interage com seus elementos. Para encontrar esses elementos, o Playwright utiliza a API de **Locators**, que são objetos que representam uma forma de encontrar um ou mais elementos na página a qualquer momento. No exemplo do Código 3.3, *page.getByRole('link', name: 'Get started')* é um locator que encontra um link com o texto específico. Uma vez que o elemento é localizado, ações como *.click()* podem ser executadas.

As asserções são utilizadas para verificar se a aplicação se comporta como o esperado após uma série de ações. O Playwright utiliza a função *expect()*, que, combinada com seus validadores assíncronos, aguarda até que uma condição seja atendida ou um tempo limite seja atingido. Essa característica torna os testes mais resilientes e menos suscetíveis a falhas por tempo. Nos exemplos, *expect(page).toHaveTitle(/Playwright/)* e *expect(locator).toBeVisible()* são asserções que pausam a execução do teste até que a condição de validação seja verdadeira.

É importante notar que o Playwright garante o isolamento total entre os testes. Cada função *test* recebe uma instância de *page* que pertence a um *BrowserContext* exclusivo, o que é equivalente a um perfil de navegador completamente novo. Isso significa que cookies, armazenamento local e sessões não são compartilhados entre os testes, o que garante que a execução de um não possa interferir no resultado do outro.

3.4.3. Executando Testes e Analisando Resultados

Uma vez que os testes são escritos, o próximo passo é executá-los para verificar o comportamento da aplicação. O Playwright oferece uma interface de linha de comando (CLI) para gerenciar a execução dos testes de forma flexível e poderosa.

O comando fundamental para executar toda a suíte de testes é apresentado no Código 3.4. Por padrão, este comando executa todos os arquivos de teste encontrados no projeto, em paralelo e em modo *headless* (sem abrir uma interface gráfica do navegador) para todos os navegadores configurados no arquivo *playwright.config.ts*.

```
npx playwright test
```

Listing 3.4. Comando para executar a suíte de testes.

Após a execução, os resultados são exibidos diretamente no terminal, como mostra o Código 3.5, com um resumo de quantos testes passaram ou falharam em cada navegador.

```
Running 6 tests using 3 workers
  6 passed (4.3s)
```

Listing 3.5. Exemplo de saída do terminal após a execução dos testes.

A execução pode ser personalizada com diversos parâmetros. Por exemplo, a flag *–headed* executa os testes em um navegador com interface gráfica visível, enquanto a flag *–project* permite especificar um único navegador, como em *npx playwright test –project chromium*.

O principal artefato para a análise dos resultados é o Relatório HTML. Ele fornece um painel interativo para explorar os resultados de cada teste. O relatório é aberto automaticamente quando há falhas, mas pode ser acessado a qualquer momento com o comando do Código 3.6.

```
npx playwright show-report
```

Listing 3.6. Comando para visualizar o relatório HTML.

A interface principal do relatório, já apresentada na Figura 3.5, permite uma análise visual completa dos resultados. A partir dela, é possível filtrar testes por status (aprovado, falhou, etc.) e por navegador, além de inspecionar os passos de cada teste e acessar o traço completo da execução para uma depuração aprofundada, como será detalhado na próxima seção.

Após a exploração dos conceitos teóricos e das ferramentas práticas, esta seção consolida o aprendizado através da aplicação completa dos conhecimentos em um projeto real. O objetivo é guiar o leitor na construção de uma suíte de testes E2E para uma aplicação web do tipo "ToDo List", um exemplo clássico utilizado para a demonstração de tecnologias de frontend. Seguiremos um fluxo de trabalho estruturado: primeiramente, definiremos o escopo e os cenários de teste; em seguida, implementaremos os testes utilizando o padrão de projeto Page Object Model; e, por fim, demonstraremos como depurar e analisar os resultados.

3.5. Estudo de Caso: Todo List com Definição do Escopo e Cenários de Teste

A aplicação "ToDo List" a ser testada possui uma interface simples para o gerenciamento de tarefas. O usuário pode adicionar, editar, marcar como concluída e excluir tarefas. Nosso objetivo é criar um conjunto de testes automatizados que valide estas funcionalidades críticas, garantindo a integridade da experiência do usuário.

Para este capítulo, implementaremos um conjunto representativo dos cenários de teste mais importantes. A suíte de testes completa, com validações adicionais, estará disponível no repositório do projeto para consulta. Os cenários que abordaremos são:

- **Carregamento da Página:** Verificar se a aplicação carrega corretamente e exibe seus elementos principais.
- **Adição de Tarefa:** Validar a criação de uma ou mais tarefas.
- **Edição de Tarefa:** Assegurar que uma tarefa existente pode ser editada.
- **Exclusão de Tarefa:** Verificar se uma tarefa pode ser removida da lista.
- **Marcar Tarefa como Concluída:** Validar a funcionalidade de marcar e desmarcar uma tarefa.
- **Validação de Tarefa Duplicada:** Garantir que a aplicação lida corretamente com a tentativa de adicionar uma tarefa com o mesmo texto de uma já existente.
- **Validação de Tarefa Vazia:** Verificar se o sistema impede a adição de uma tarefa sem texto.

A Figura 3.11 exibe a interface principal da aplicação que será o objeto de nosso estudo de caso.

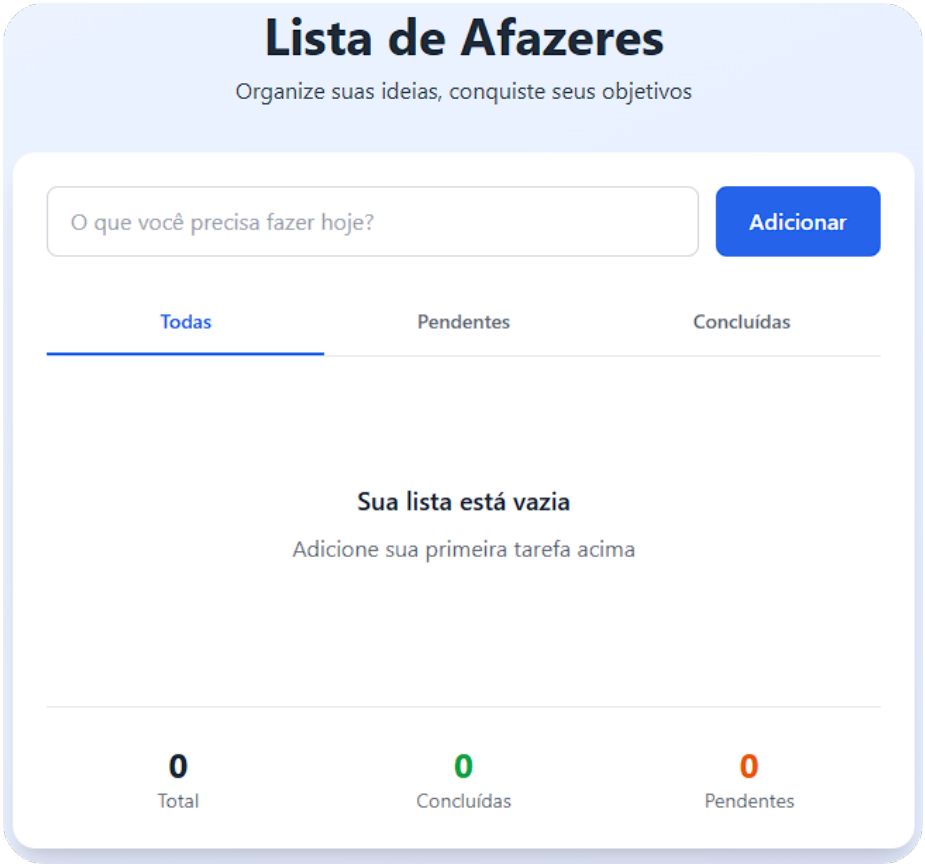


Figure 3.11. Interface da aplicação "ToDo List" utilizada no estudo de caso.

3.5.1. Implementação Prática com Page Object Model

Para implementar os cenários de teste de forma organizada e de fácil manutenção, utilizaremos o padrão de projeto Page Object Model (POM). O POM é uma técnica de design que consiste em criar uma classe para cada página da aplicação, encapsulando os detalhes da interface do usuário. O principal benefício desta abordagem é a separação de responsabilidades: a classe Page Object lida com a complexidade de encontrar e interagir com os elementos da página, enquanto o arquivo de teste se concentra apenas na lógica e nas asserções do cenário [Microsoft 2025].

A Figura 3.12 ilustra este padrão. As páginas da aplicação web são mapeadas para classes Page Object, que por sua vez são utilizadas pelos scripts de teste. Essa camada de abstração desacopla os testes da estrutura interna da interface, tornando-os mais resilientes a mudanças no HTML.

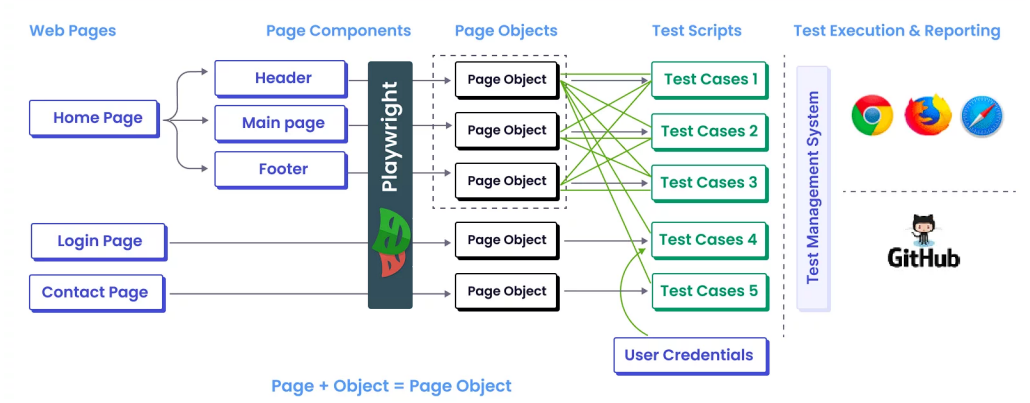


Figure 3.12. Diagrama do fluxo de trabalho com o padrão Page Object Model.

Para a nossa aplicação, criamos a classe *TodoPage*. O código completo desta classe está disponível no repositório do projeto para consulta [Alves 2025]; aqui, destacaremos seus componentes essenciais. Primeiramente, a classe centraliza todos os localizadores de elementos em seu construtor, como demonstrado no Código 3.7. Se um seletor na interface do usuário mudar no futuro, só precisaremos atualizá-lo neste único local.

```
// Trecho de TodoPage.js
constructor(page) {
  this.page = page;

  // Localizadores dos elementos principais
  this.taskInput = page.locator('#taskInput');
  this.addButton = page.locator('#addButton');
  this.taskList = page.locator('#taskList');
  // ... outros localizadores disponiveis no repositorio
}
```

Listing 3.7. Trecho do construtor da classe TodoPage com os localizadores.

Em seguida, a classe expõe métodos de alto nível que representam as ações do usuário. O Código 3.8 mostra o método *addTask*, que esconde os detalhes de implementação (preencher o campo e clicar no botão) por trás de uma única ação com um nome claro e intuitivo.

```
// Trecho de TodoPage.js
async addTask(taskText) {
  await this.taskInput.fill(taskText);
  await this.addButton.click();
}
```

Listing 3.8. Exemplo de um método de ação na classe TodoPage.

Com a classe *TodoPage* pronta, o arquivo de teste (*todo.spec.js*), apresentado no Código 3.9, se torna extremamente limpo e legível. Ele se concentra no fluxo do cenário e nas validações, utilizando a instância do Page Object para orquestrar as interações com o navegador. Note como o teste lê quase como um roteiro de caso de uso, sem a desordem de seletores de CSS.

```
import { test, expect } from '@playwright/test';
import { TodoPage } from './TodoPage'; // Importa a classe

test.describe('Gerenciamento de Tarefas', () => {
  let todoPage;

  test.beforeEach(async ({ page }) => {
    todoPage = new TodoPage(page);
    await todoPage.goto();
  });

  test('deve adicionar uma nova tarefa', async () => {
    const taskText = 'Comprar pao';
    await todoPage.addTask(taskText);
    await expect(todoPage.taskList).toContainText(taskText);
  });

  // ... outros cenarios de teste disponiveis no repositorio
});
```

Listing 3.9. Arquivo de teste que utiliza a classe TodoPage para implementar os cenarios.

3.5.2. Depuração e Análise Avançada com o Trace Viewer

Um dos aspectos mais desafiadores na manutenção de uma suíte de testes é a investigação de falhas. Um teste pode falhar por diversos motivos, desde um defeito real na aplicação até um problema no próprio script de teste. Para demonstrar como diagnosticar um problema de forma eficiente, simularemos um cenário comum: um teste que falha devido a um localizador incorreto.

O Código 3.10 apresenta uma variação do nosso teste de adicionar tarefa. Nele,

introduzimos um erro proposital na linha 22: o seletor para o botão de adicionar (‘[data-testid="adicionar-button"]‘) está incorreto; o correto seria ‘[data-testid="add-button"]‘.

```
1  test('Teste de Falha: Seletor incorreto do botao adicionar',
       async () => {
2      const taskText = 'Minha primeira tarefa';
3      await todoPage.locators.taskInput.fill(taskText);
4
5      // ERRO PROPOSITAL: Usa seletor incorreto para o botao
6      // Original: [data-testid="add-button"]
7      // Incorreto: [data-testid="adicionar-button"]
8      await todoPage.page.locator('[data-testid="adicionar-button
           "]').click();
9
10     // Estas verificacoes vao falhar porque a tarefa nao foi
           adicionada
11     await expect(todoPage.getTaskCount()).toBe(1);
12 });
```

Listing 3.10. Exemplo de um teste com uma falha proposital (seletor incorreto).

Ao executar este teste, o Playwright irá falhar. O primeiro passo da nossa análise é inspecionar o Relatório HTML. A Figura 3.13 exibe a tela de resultado para o teste que falhou. O relatório imediatamente nos informa o tipo de erro (*TimeoutError*, indicando que a ferramenta esperou por um elemento que nunca apareceu) e aponta para a linha exata do código que causou a falha.

Para uma análise mais profunda, clicamos em "View Trace". Dentro do Trace Viewer, a aba "Actions", mostrada na Figura 3.14, destaca em vermelho a ação exata que falhou. Neste caso, o comando *.click()*. Isso permite que o desenvolvedor isole imediatamente o ponto problemático da execução.

Para entender o motivo da falha, a aba "Errors", na Figura 3.15, fornece o log detalhado. A mensagem "*TimeoutError: waiting for locator('[data-testid="adicionar-button"]')*" confirma que o Playwright esgotou o tempo de espera porque não conseguiu encontrar o elemento com o seletor especificado. Através deste processo de análise, do geral para o específico, o desenvolvedor pode diagnosticar rapidamente que o problema é um erro no seletor, em vez de um defeito na funcionalidade da aplicação.

3.6. Conclusão

Ao longo deste capítulo, realizamos uma jornada completa pelo universo dos testes E2E, desde seus fundamentos teóricos até a sua aplicação prática com o framework Playwright. Iniciamos por estabelecer a disciplina de testes de software como um pilar essencial da engenharia de software moderna, explorando sua evolução histórica e o modelo estratégico da Pirâmide de Testes. Com essa base, aprofundamos no ecossistema Playwright, analisando sua arquitetura, suas funcionalidades-chave e seu posicionamento em relação a outras ferramentas do mercado.

A parte prática do capítulo guiou o leitor desde a configuração inicial de um pro-

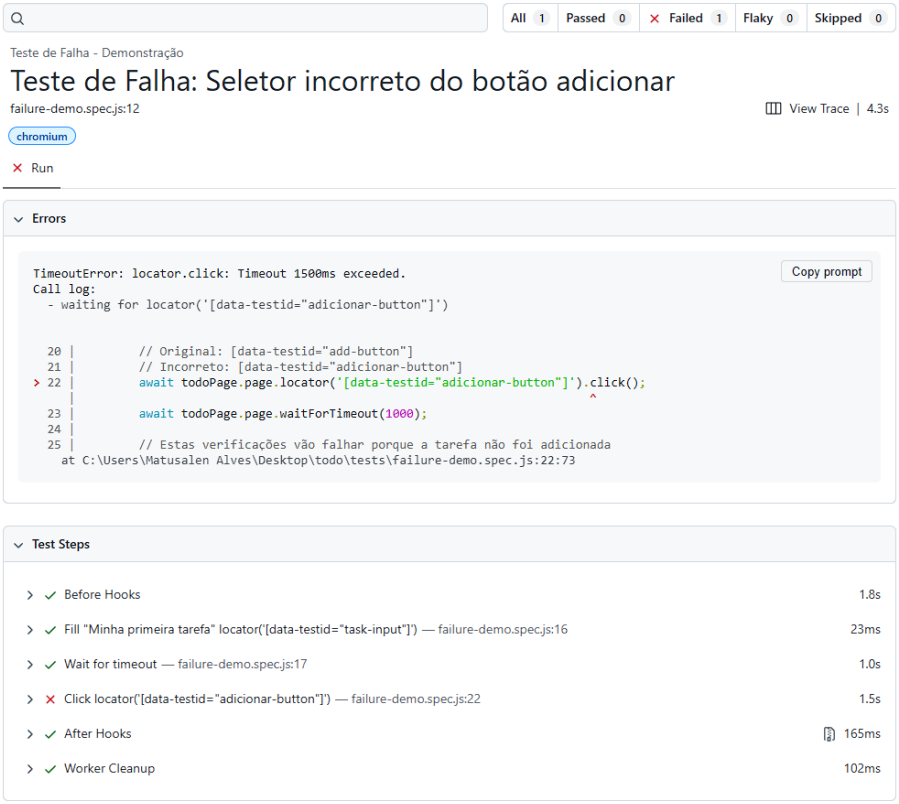


Figure 3.13. Relatório HTML exibindo o teste que falhou e o erro correspondente.

jeto até a escrita e execução de testes, culminando em um estudo de caso detalhado. Na implementação para a aplicação "ToDo List", demonstramos como escrever os testes e como estruturá-los de forma fácil, aplicando o padrão de projeto POM. Além disso, abordamos um aspecto crucial do dia a dia do desenvolvimento: a depuração de testes, mostrando como o Trace Viewer acelera a identificação e a correção de falhas.

A principal mensagem deste capítulo é que a automação de testes E2E, com o auxílio de ferramentas modernas como o Playwright, é uma prática acessível e de alto impacto para qualquer equipe de desenvolvimento. Recursos como as esperas automáticas e as ferramentas de depuração visual não são apenas conveniências, mas soluções diretas para os desafios de instabilidade e complexidade que historicamente tornaram os testes de UI um processo custoso.

Para os leitores que desejam aprofundar seus conhecimentos, o próximo passo natural é explorar as outras capacidades que o ecossistema Playwright oferece, como o suporte nativo para testes de API, a implementação de testes de regressão visual e a funcionalidade de testes de componentes. Além disso, a integração da suíte de testes a um pipeline de Integração e Entrega Contínua (CI/CD), como o GitHub Actions, é uma etapa fundamental para automatizar completamente o processo de garantia de qualidade. A documentação oficial do Playwright permanece como o recurso mais completo para o

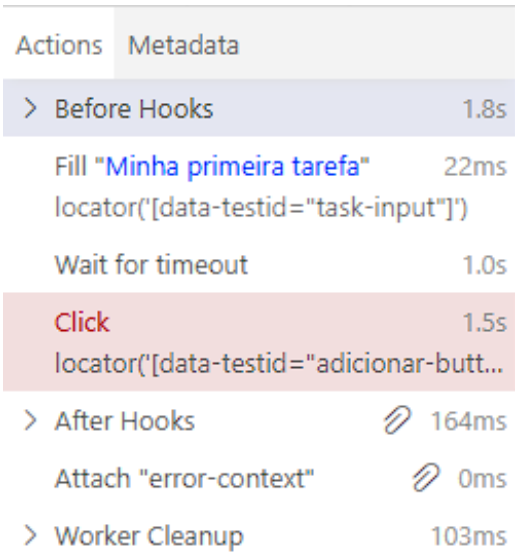


Figure 3.14. Aba "Actions" do Trace Viewer, destacando a etapa que falhou.

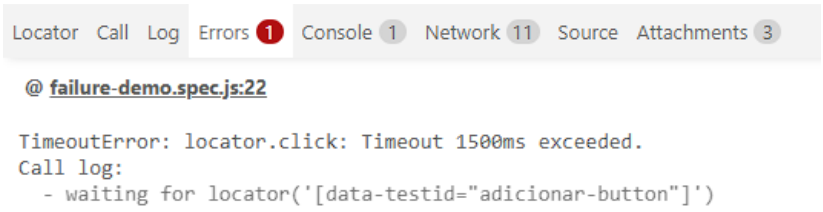


Figure 3.15. Aba "Errors" do Trace Viewer com o log detalhado da falha.

estudo contínuo, e o repositório do nosso estudo de caso serve como um exemplo prático e funcional de referência.

References

[Alves 2025] Alves, M. C. (2025). todo-codec: Aplicação para o minicurso de testes e2e com playwright. `https://github.com/watusalen/todo-codec`. GitHub. Acesso em: 2025-10-20.

[Cohn 2009] Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional.

[Cypress 2025] Cypress (2025). Cypress documentation. `https://docs.cypress.io/`. Acesso em: 2025-10-20.

[Fowler 2012] Fowler, M. (2012). The test pyramid. `https://martinfowler.com/bliki/TestPyramid.html`. Acesso em: 2025-10-20.

[Fowler 2020] Fowler, M. (2020). *Refatoração: Aperfeiçoando o design de códigos existentes*. Novatec Editora.

[Martin 2012] Martin, R. C. (2012). *O codificador limpo: um código de conduta para programadores profissionais*. Alta Books.

[Meszaros 2007] Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional.

[Microsoft 2025] Microsoft (2025). Playwright documentation. `https://playwright.dev/docs/intro`. Acesso em: 2025-10-20.

[Selenium 2025] Selenium (2025). Selenium documentation. `https://www.selenium.dev/documentation/`. Acesso em: 2025-10-20.

[Sommerville 2019] Sommerville, I. (2019). *Engenharia de Software*. Pearson Brasil.

[Vaithilingam et al. 2022] Vaithilingam, P., Zhang, T., and Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. ACM.

Chapter 4

Sistemas Embarcados: Uma Abordagem Prática com BitDogLab

Cairon Ferreira Prado, Darlys Ferreira Neris de Aguiar, Fabrício de Carvalho Mota, Jonathas Jivago de Almeida Cruz, Matusalen Costa Alves, Pedro Henrique Valentino

Abstract

This minicourse presents a practical introduction to embedded systems using the BitDogLab development board. Targeted at beginners with basic programming skills, the minicourse introduces essential hardware and software concepts through hands-on activities. Participants will work with digital I/O, analog sensors (ADC), pull-up/down resistors, and develop a simple interactive game. The proposal aims to foster interest in embedded systems through accessible and engaging experimentation.

Resumo

Este minicurso apresenta uma introdução prática aos sistemas embarcados utilizando a placa de desenvolvimento BitDogLab. Destinado a iniciantes com conhecimentos básicos em programação, aborda conceitos fundamentais de hardware e software por meio de atividades interativas. Os participantes trabalharão com entradas e saídas digitais, sensores analógicos (ADC), resistores pull-up/down e desenvolverão um jogo interativo simples. A proposta busca despertar o interesse por sistemas embarcados de forma acessível e didática.

4.1. Introdução

A educação em sistemas embarcados tornou-se um pilar para a formação de profissionais capazes de desenvolver tecnologias que interagem diretamente com o mundo físico. A onipresença desses sistemas — que permeiam veículos autônomos, eletrodomésticos inteligentes, dispositivos médicos e redes de manufatura — projeta um mercado que reforça a urgência de formar engenheiros com um domínio versátil e integrado de hardware e software [Pasricha 2022].

Historicamente, o ensino dessa área enfrenta desafios significativos. Currículos tradicionais de engenharia, muitas vezes compartimentados em domínios como Ciência da Computação e Engenharia Elétrica, dificultam a formação interdisciplinar que é essencial para o desenvolvimento de sistemas embarcados [Sztipanovits et al. 2005]. Como resposta a essa lacuna, os Computadores de Placa Única (SBCs), como Raspberry Pi e Arduino, foram amplamente adotados na educação, com estudos sistemáticos demonstrando um aumento no engajamento e na motivação dos alunos por meio de uma abordagem mais prática [Ariza and Baez 2021].

Nesse contexto, plataformas educacionais focadas na experimentação surgem como ferramentas poderosas de aprendizagem ativa. Frameworks pedagógicos recentes enfatizam a importância de um currículo que aborde desde os fundamentos de hardware e software até a integração de sensores, com foco em projetos práticos para capacitar profissionais para áreas como robótica e automação [Benyeogor et al. 2024].

A placa BitDogLab, objeto central deste minicurso, insere-se nesse movimento. Desenvolvida com base no Raspberry Pi Pico, ela apoia atividades STEAM (Ciência, Tecnologia, Engenharia, Artes e Matemática) e busca democratizar o acesso ao ensino de sistemas embarcados com uma baixa barreira de entrada. Ao combinar hardware aberto e suporte educativo, a plataforma permite que estudantes avancem de conceitos teóricos a projetos práticos de forma integrada [Fruett et al. 2024].

O restante deste capítulo está organizado da seguinte maneira: a Seção 4.2 explora os conceitos teóricos de sistemas embarcados; a Seção 4.3 apresenta em detalhes a placa BitDogLab; a Seção 4.4 detalha os fundamentos de programação e eletrônica necessários para a prática; a Seção 4.5 consolida o aprendizado com a construção de um jogo interativo; e, por fim, a Seção 4.6 conclui o trabalho.

4.2. Fundamentos de Sistemas Embarcados

Para explorar de forma prática o desenvolvimento de projetos com a placa BitDogLab, é essencial, primeiramente, estabelecer uma base conceitual sólida. A Engenharia de Sistemas Embarcados é um campo vasto e interdisciplinar, que combina conhecimentos de eletrônica, ciência da computação e engenharia de controle. Esta seção introduz os conceitos fundamentais da área, começando pela definição formal de um sistema embarcado e suas classificações. Em seguida, serão destacadas as principais diferenças que o distinguem da computação de propósito geral e, por fim, será apresentada a arquitetura típica que caracteriza esses sistemas, detalhando seus componentes de hardware e software.

4.2.1. O que é um Sistema Embarcado?

Um sistema embarcado é, em sua essência, um sistema computacional projetado para ser o "cérebro" oculto dentro de um dispositivo maior, com a missão de executar uma ou poucas funções de forma dedicada. Diferente de um computador de propósito geral (como um notebook, que pode rodar inúmeros programas diferentes), um sistema embarcado é uma combinação otimizada de hardware e software, desenvolvida sob medida para uma aplicação específica. Seus componentes — um processador, memória, interfaces de entrada e saída (I/O) e o software especializado conhecido como firmware — são dimensionados para operar sob restrições rigorosas de tempo de resposta, consumo de energia, custo

e tamanho físico. Essa natureza dedicada significa que o hardware e o software são intrinsecamente acoplados, priorizando a eficiência e a interação com o mundo físico em vez da versatilidade genérica [Wolf 2001, Micco et al. 2018, Kopetz 2022].

Para compreender a diversidade desses sistemas, eles podem ser classificados com base em suas demandas operacionais. A seguir, apresentamos uma taxonomia com as principais categorias [Akdur et al. 2018, Kopetz 2022, Pereira et al. 2017]:

1. **Tempo Real Estrito (Hard Real-Time):** São sistemas em que uma falha no cumprimento de um prazo pode ter consequências catastróficas. A correção do sistema depende criticamente do tempo. Exemplos clássicos incluem o sistema de freios ABS de um carro, um marca-passo cardíaco ou o piloto automático de uma aeronave.
2. **Tempo Real Suave (Soft Real-Time):** Nesses sistemas, a falha em cumprir um prazo resulta em uma degradação da qualidade ou do desempenho, mas não em uma falha crítica do sistema. Um exemplo é a transmissão de vídeo em uma smart TV, onde um pequeno atraso pode causar um travamento momentâneo na imagem.
3. **Missão Crítica (Safety-Critical):** Esta classificação abrange sistemas cuja falha pode resultar em danos significativos, ferimentos ou morte. Eles exigem processos de desenvolvimento e certificações de segurança rigorosos e são comuns em setores como o automotivo, médico e aeroespacial.
4. **IoT e Borda (Edge):** Focados em conectividade, esses sistemas operam na "borda" da rede, coletando e, muitas vezes, pré-processando dados localmente antes de enviá-los para a nuvem. O baixo consumo de energia é um requisito fundamental para esses dispositivos, que frequentemente operam com baterias.
5. **Sistemas Ciber-Físicos (CPS):** Representam uma integração profunda entre computação, rede e processos físicos. Eles operam em um ciclo de realimentação contínuo (malha fechada), onde sensores monitoram o ambiente e atuadores o modificam, como em robôs industriais ou redes elétricas inteligentes.

Uma característica fundamental que define o desenvolvimento de sistemas embarcados é a enorme importância dos requisitos não-funcionais. Estes requisitos descrevem como o sistema deve operar, em vez de o que ele deve fazer. Propriedades como o tempo máximo de resposta a um evento, o consumo de energia por operação, a confiabilidade ao longo de anos de uso contínuo e a tolerância a falhas são, muitas vezes, mais importantes do que a própria lógica da aplicação.

Devido a essa criticidade, o desenvolvimento de sistemas embarcados em setores regulados é guiado por normas de segurança e padrões formais rigorosos, como a ISO 26262 para a indústria automotiva, a IEC 62304 para software de dispositivos médicos e a DO-178C para a aviação. Essas normas impõem processos estritos de gerenciamento de requisitos, rastreabilidade e testes. Mesmo em um contexto educacional, a introdução a noções básicas de verificação e validação (como testes de software-in-the-loop e hardware-in-the-loop) prepara o estudante para as exigências do mercado, que enfrenta desafios constantes na especificação e validação desses requisitos não-funcionais complexos [Kopetz 2022, Pereira et al. 2017, Garousi et al. 2018].

4.2.2. Diferenças para a Computação de Propósito Geral

Sistemas embarcados distinguem-se fundamentalmente da computação de propósito geral — como desktops, laptops e servidores — pelo seu foco em tarefas específicas e pela sua profunda integração com o mundo físico [Elsevier / ScienceDirect 2025]. Enquanto um computador tradicional pode ser visto como uma ferramenta universal, projetada para executar múltiplas e variadas aplicações sob sistemas operacionais complexos, um sistema embarcado é uma ferramenta especialista, otimizada para executar sua função de forma autônoma e eficiente, muitas vezes sem intervenção humana constante [Wolf 2001, Kopetz 2022].

As diferenças se manifestam de forma concreta tanto no hardware quanto no software. Computadores de propósito geral são projetados para alto desempenho, equipados com processadores de múltiplos núcleos, gigabytes de memória RAM e sistemas de armazenamento massivo. Em contraste, o hardware de um sistema embarcado é minimalista e otimizado. Ele geralmente utiliza microcontroladores (MCUs) ou Sistemas em um Chip (SoCs), que integram processador, memória e periféricos (como portas de comunicação) em um único componente. A memória é limitada a kilobytes ou poucos megabytes, e o software, conhecido como firmware, é altamente especializado. Em muitos casos, em vez de um sistema operacional completo como Windows ou Linux, o sistema pode rodar diretamente sobre o hardware ou utilizar um Sistema Operacional de Tempo Real (RTOS), que é um software minimalista projetado para garantir a execução de tarefas dentro de prazos rigorosos [Kopetz 2022, Akesson et al. 2020].

Uma das distinções mais críticas reside na operação em tempo real. A maioria dos sistemas de propósito geral opera com base no desempenho médio; não há problema se um programa levar alguns milissegundos a mais para abrir. Em muitos sistemas embarcados, no entanto, a previsibilidade é essencial. Eles demandam respostas determinísticas a eventos externos, o que significa que uma ação deve ser concluída dentro de um prazo máximo garantido. Para assegurar essa previsibilidade, os engenheiros analisam métricas como o Pior Caso de Tempo de Execução (WCET). O sistema de airbag de um veículo, por exemplo, não pode depender de um "tempo médio" de resposta; ele deve acionar em milissegundos, sempre.

Além disso, a interação com o mundo físico é a principal razão de ser de um sistema embarcado. Ele utiliza sensores para perceber o ambiente (medindo temperatura, pressão, movimento) e atuadores para agir sobre ele (acionando motores, luzes, válvulas). Esse acoplamento direto com o hardware exige um ciclo de desenvolvimento e verificação distinto, que frequentemente inclui técnicas como testes de Hardware-in-the-Loop (HIL), onde o sistema real é testado em um ambiente que simula suas interações físicas. Essa abordagem contrasta com a computação tradicional, que foca em interfaces mais abstratas entre o usuário e o software [Akesson et al. 2020, Pereira et al. 2017, Garousi et al. 2018].

Em suma, a identidade dos sistemas embarcados deriva de seu caráter dedicado, que impõe um forte vínculo entre hardware e software. As restrições de custo, energia e tamanho, a necessidade de determinismo temporal e a integração direta com o ambiente físico orientam todas as escolhas de arquitetura, as práticas de desenvolvimento e as técnicas de validação, distinguindo-os fundamentalmente dos computadores de uso geral.

4.2.3. A Arquitetura Típica: Hardware e Software

A arquitetura de um sistema embarcado é caracterizada por uma integração eficiente entre seus componentes de hardware e software, projetada para otimizar a função específica do dispositivo. Embora as implementações variem enormemente, uma estrutura fundamental pode ser identificada na maioria dos projetos.

O núcleo de processamento do sistema é geralmente um microcontrolador (MCU) ou um System-on-Chip (SoC). O MCU pode ser entendido como o "cérebro" do dispositivo, pois integra em um único circuito a Unidade Central de Processamento (CPU), a memória (tanto a volátil, RAM, quanto a não-volátil, Flash) e uma variedade de periféricos. Esses periféricos incluem timers para o controle de tempo, portas de Entrada/Saída de Propósito Geral (GPIO), conversores analógico-digitais (ADC) e controladores para protocolos de comunicação como I²C e SPI, que permitem ao MCU interagir com outros componentes eletrônicos.

Para perceber o ambiente, o sistema utiliza sensores. Eles funcionam como os "sentidos" do dispositivo, capturando dados do mundo físico e convertendo-os em sinais elétricos que o microcontrolador pode processar. Exemplos incluem acelerômetros que medem movimento, microfones que captam som ou termistores que medem a temperatura. Para agir sobre o ambiente, o sistema emprega atuadores, que podem ser vistos como os "músculos". Componentes como motores, LEDs, telas ou relés recebem comandos elétricos do microcontrolador e os convertem em ações físicas, como movimento, luz ou som. Juntos, sensores e atuadores formam um ciclo de controle que pode ser de malha aberta (apenas executa uma ação) ou de malha fechada (reage continuamente às mudanças percebidas pelos sensores) [Kopetz 2022, Akdur et al. 2018].

No âmbito do software, o firmware é o programa especializado que orquestra todos esses componentes de hardware. Frequentemente escrito em linguagens de baixo nível como C ou C++, que oferecem controle direto sobre o hardware, o firmware é responsável por inicializar os periféricos, ler os dados dos sensores, executar a lógica da aplicação e enviar os comandos apropriados para os atuadores. Para sistemas que precisam gerenciar múltiplas tarefas com garantias de tempo, pode-se utilizar um (RTOS), como o FreeRTOS. Um RTOS é um sistema operacional minimalista cujo principal objetivo é garantir o determinismo temporal, ou seja, a capacidade de executar tarefas dentro de prazos rigorosamente definidos.

Essa arquitetura evidencia um conceito central da área: o co-design de hardware e software. As decisões tomadas em uma área impactam diretamente a outra. Por exemplo, a escolha de um sensor de baixo custo e menor precisão pode exigir algoritmos de filtragem mais complexos no software para compensar o ruído. Inversamente, um hardware mais poderoso pode simplificar o firmware. A análise desses trade-offs entre latência, consumo de energia e complexidade é uma habilidade crítica no desenvolvimento de sistemas embarcados e um aspecto fundamental para ilustrar a otimização integrada em projetos educacionais [Akdur et al. 2018, Micco et al. 2018].

4.3. Apresentando a Placa de Desenvolvimento BitDogLab

Após a exploração dos fundamentos teóricos dos sistemas embarcados, esta seção direciona o foco para a ferramenta de hardware que servirá como nosso laboratório prático: a placa de desenvolvimento BitDogLab. Criada para ser uma porta de entrada acessível ao universo da eletrônica e da programação de baixo nível, a BitDogLab materializa os conceitos de hardware e software em uma plataforma interativa. Iniciaremos com uma visão geral de sua concepção e filosofia educacional e, em seguida, faremos uma análise detalhada dos periféricos integrados que utilizaremos em nosso estudo de caso.

4.3.1. Visão Geral da Placa

A BitDogLab é uma placa educacional versátil, construída sobre o Raspberry Pi Pico, que visa democratizar o ensino de sistemas embarcados por meio de um ecossistema open-source. Seu objetivo principal é fomentar o aprendizado progressivo em programação, eletrônica e sistemas ciber-físicos, ao proporcionar um ambiente sinestésico que integra elementos visuais, auditivos e interativos. A plataforma incentiva a modificação colaborativa, permitindo que usuários copiem, fabriquem e aprimorem o design, o que a torna ideal para projetos educacionais que enfatizam a inovação e a colaboração [Fruett et al. 2024, Community 2025].

A versatilidade para a prototipagem rápida é um dos pilares do projeto. A BitDogLab oferece suporte a versões de montagem manual (through-hole) e de montagem em superfície (SMD), com todos os arquivos de design, incluindo esquemáticos e layouts, disponíveis no formato KiCad. Isso facilita a fabricação personalizada e o estudo aprofundado de seu circuito. A placa é programada principalmente em MicroPython, com firmwares específicos que já incluem bibliotecas para o controle de todos os periféricos integrados. O núcleo da placa, o microcontrolador RP2040, com seus dois núcleos ARM Cortex-M0+, 264 kB de SRAM e o subsistema de I/O Programável (PIO), habilita a experimentação com processamento paralelo e a criação de protocolos de comunicação customizados, permitindo que os projetos evoluam de simples jogos educativos a sistemas de sensoriamento ambiental de forma ágil [Foundation 2021, Community 2025, Industries 2025].

4.3.2. Periféricos Integrados: Visão Geral dos Componentes

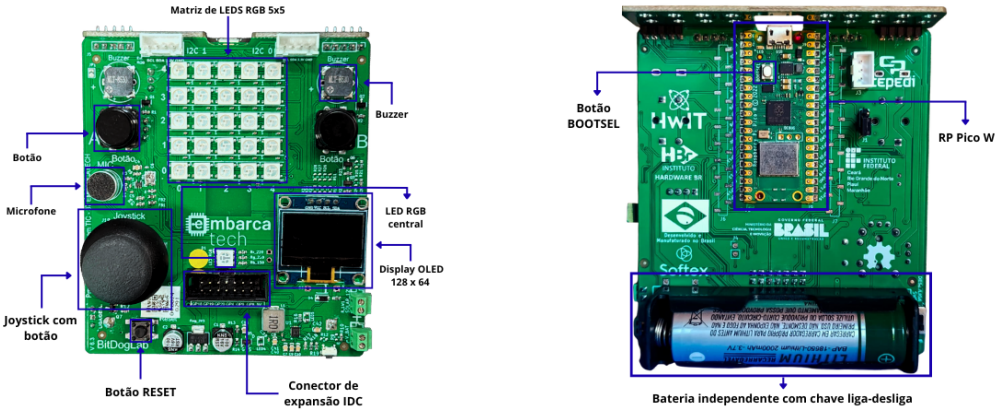
A Figura 4.1 apresenta as duas faces da placa BitDogLab, destacando a riqueza de componentes que a tornam um laboratório portátil completo para experimentação prática. Ao integrar sensores e atuadores diretamente no circuito, a placa elimina a necessidade de montagens complexas em protoboards, permitindo que o estudante foque na lógica de programação e na interação com o hardware desde o início.

A vista frontal (Subfigura 4.1a) concentra os componentes de interação com o usuário:

- **Display OLED:** Uma tela gráfica para exibir informações, menus e animações.
- **Matriz de LEDs:** Um conjunto de LEDs RGB endereçáveis (WS2812B) para feedback visual dinâmico.
- **Joystick Analógico:** Permite a entrada de dados em dois eixos (X e Y), ideal para

controle de movimento.

- **Botões de Usuário:** Entradas digitais para comandos discretos.
- **Microfone MEMS:** Um sensor para capturar som ambiente e criar aplicações reativas a ruídos.
- **Buzzer Piezoelétrico:** Um atuador para gerar feedback sonoro e melodias simples.



(a) Placa BitDogLab (vista frontal)

(b) Placa BitDogLab (vista reversa)

Figure 4.1: Visões da Placa BitDogLab

O verso da placa (Subfigura 4.1b) abriga os circuitos de suporte, como o sistema de alimentação, o módulo de carregamento para bateria de Li-ion e o botão *bootsel*, que facilita a gravação de um novo firmware através de uma simples interface de arrastar e soltar.

4.3.3. Periféricos Integrados: Análise Técnica

A Figura 4.1, apresentada anteriormente, serve como base para esta análise detalhada dos periféricos integrados da BitDogLab. A seguir, descrevem-se os principais componentes com foco em suas características técnicas — como protocolos de comunicação, resoluções e consumos de energia — e sua relevância para projetos educacionais em sistemas embarcados, promovendo a compreensão prática de conceitos como ADC, PWM e I²C.

- **Joystick Analógico:** Este componente serve como uma entrada de controle multidirecional. Ele está conectado a canais do Conversor Analógico-Digital (ADC) do RP2040, que possui uma resolução de 12 bits. Isso significa que a posição do joystick em cada eixo pode ser lida como um valor entre 0 e 4095, permitindo uma detecção de movimento precisa e gradual, ideal para interfaces interativas como jogos ou o controle de robôs [Foundation 2021, Community 2025].

- **Botões de Usuário:** Dois botões físicos oferecem entradas digitais robustas para interações simples, como selecionar menus ou disparar eventos. Eles são conectados a pinos GPIO configurados com resistores de *pull-down*, o que garante um estado lógico estável. O firmware pode ser programado para detectar o acionamento desses botões por meio de checagens contínuas ou, de forma mais eficiente, através de interrupções de hardware, um conceito fundamental em sistemas embarcados [Foundation 2021, Community 2025].

- **Buzzer Piezoelétrico:** Mapeado para um pino GPIO específico, este atuador sonoro permite a geração de tons e melodias simples. Seu funcionamento é controlado por modulação por largura de pulso (PWM), uma técnica que o RP2040 suporta nativamente. Ao variar a frequência e a largura do pulso, é possível controlar a nota e o volume do som, o que exemplifica a aplicação de conceitos de geração de sinais para fornecer feedback auditivo em projetos [Foundation 2021, Community 2025].

- **Microfone MEMS:** A placa integra um microfone digital (MP34DT01-M) que utiliza uma interface PDM (*Pulse Density Modulation*) para se comunicar com o microcontrolador. Com alta sensibilidade e baixo consumo de energia, este sensor é adequado para aplicações de sensoriamento de áudio, como o reconhecimento de comandos de voz simples ou a detecção de eventos sonoros (como uma palma) para acionar uma ação no sistema [STMicroelectronics 2015, Foundation 2021].

- **Tela OLED:** Para a saída de informações visuais, a BitDogLab conta com uma tela de matriz de pontos com resolução de 128x64 pixels, controlada pelo driver SSD1306. A comunicação com o microcontrolador é feita via protocolo I²C, um barramento serial comum em sistemas embarcados. A tela possui um buffer de memória interno, o que permite atualizações eficientes, e é perfeita para exibir menus, status de sensores ou interfaces gráficas simples, tornando os projetos mais interativos e informativos [Solomon Systech / Adafruit (mirror) 2010, Community 2025].

- **LEDs RGB Endereçáveis:** Um conjunto de LEDs RGB (baseados no chip WS2812B) oferece um recurso de feedback visual altamente dinâmico. Cada LED pode ser controlado individualmente para exibir qualquer uma das 16 milhões de cores, através de um protocolo serial de um fio. No RP2040, o subsistema de I/O Programável (PIO) é ideal para controlar esses LEDs, pois pode gerar os sinais de temporização precisos que o protocolo exige sem sobrecarregar a CPU principal [WORLDSEMI 2013, Foundation 2021, Fruett et al. 2024].

4.4. Conceitos Essenciais de Programação e Eletrônica

Com a familiaridade estabelecida com os componentes físicos da placa BitDogLab, esta seção aprofunda-se nos conceitos de programação e eletrônica fundamentais para dar vida a esses periféricos. Para construir aplicações interativas e que percebem o ambiente, é necessário compreender como o microcontrolador lê informações de sensores e como ele comanda os atuadores. Ao integrar noções de interfaces digitais e analógicas e gerenciamento de sinais, estes tópicos preparam o terreno para o desenvolvimento do nosso estudo de caso. Utilizando o microcontrolador RP2040 como referência, exploraremos as técnicas

para leitura e controle de periféricos, facilitando a transição de conceitos teóricos para protótipos funcionais [Foundation 2021, Fruett et al. 2024].

4.4.1. Entradas e Saídas Digitais (GPIO)

Os pinos de Entrada/Saída de Propósito Geral, conhecidos pela sigla GPIO (*General-Purpose Input/Output*), constituem a interface mais fundamental entre o microcontrolador e o mundo físico. Eles são canais de comunicação versáteis que podem ser configurados via software para operar de duas maneiras principais: como uma **entrada**, para ler informações do ambiente, ou como uma **saída**, para controlar dispositivos externos. O microcontrolador RP2040, presente na BitDogLab, oferece 30 desses pinos multifuncionais, que são a base para a interação com sensores e atuadores [Foundation 2021, Community 2025].

No modo digital, um pino GPIO opera com apenas dois estados lógicos: ALTO (*HIGH*) ou BAIXO (*LOW*). O estado ALTO corresponde a uma tensão elétrica específica (3.3V no RP2040), representando o valor binário "1", enquanto o estado BAIXO corresponde a 0V, representando o "0".

- **Configuração como Saída (Output):** Quando um pino é configurado como saída, o firmware pode controlar seu estado, "escrevendo" um valor ALTO ou BAIXO nele. Essa é a base para controlar atuadores digitais. Por exemplo, para acender um LED ou acionar o buzzer na BitDogLab, o firmware instrui o microcontrolador a colocar o pino correspondente em estado ALTO, fornecendo a tensão necessária para a ação. É crucial, no design de circuitos, respeitar a corrente máxima que cada pino pode fornecer (cerca de 12 mA no RP2040) para evitar danos tanto ao microcontrolador quanto ao componente externo.
- **Configuração como Entrada (Input):** Quando configurado como entrada, o pino "escuta" o nível de tensão presente nele, permitindo que o microcontrolador leia o estado de sensores digitais, como os botões da BitDogLab. Para detectar o pressionamento de um botão, o firmware monitora o estado do pino associado. A detecção pode ser feita de duas formas: por *polling*, onde o software verifica o estado do pino repetidamente em um laço, ou por *interrupções*, uma técnica mais eficiente onde o próprio hardware notifica a CPU quando uma mudança de estado (como uma borda de subida ou descida do sinal) ocorre.

O diagrama de pinagem, ou *pinout*, é a ferramenta de referência essencial para o desenvolvedor de sistemas embarcados. A Figura 4.2 exibe o pinout do Raspberry Pi Pico W, a base da nossa BitDogLab. Ele mapeia cada pino físico a sua numeração de GPIO (ex: GP0, GP1) e suas funções alternativas, como os canais de comunicação I²C, SPI e os canais do conversor ADC. A consulta a este diagrama é o primeiro passo para conectar e programar qualquer periférico externo.

A programação de GPIOs em sistemas como o RP2040 é robusta, com suporte a operações atômicas que garantem o acesso seguro aos pinos mesmo em aplicações com múltiplas tarefas concorrentes. Em projetos educacionais com a BitDogLab, a manipulação de GPIOs é o primeiro passo para a criação de aplicações interativas, como o nosso jogo

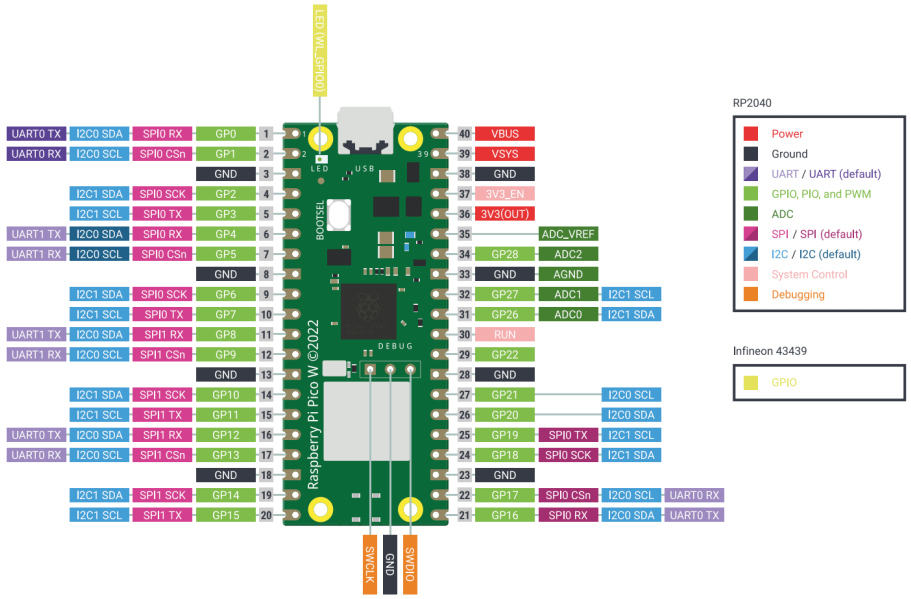


Figure 4.2: Diagrama de pinagem (pinout) do Raspberry Pi Pico W, base da BitDogLab.

de reação, e para o controle de periféricos mais complexos, como os LEDs RGB, que utilizam o subsistema de I/O Programável (PIO) para gerar os protocolos de comunicação necessários [Foundation 2021, WORLDSEMI 2013, Fruett et al. 2024].

4.4.2. Garantindo Sinais Estáveis: Resistores Pull-up e Pull-down

Ao trabalhar com entradas digitais, como botões, um dos desafios mais comuns é garantir que o microcontrolador realize uma leitura de sinal estável e previsível. Um pino GPIO configurado como entrada, quando não está conectado a um nível de tensão definido (nem 3.3V nem 0V), entra em um estado de "flutuação". Neste estado, o pino se comporta como uma pequena antena, suscetível a captar ruídos elétricos do ambiente, o que pode levar o microcontrolador a interpretar valores lógicos aleatórios (0s e 1s), resultando em um comportamento errático da aplicação [Instruments 2021, Foundation 2021].

Para resolver este problema, utilizamos resistores de *pull-up* ou *pull-down*. Esses componentes são essenciais para "ancorar" o pino de entrada a um estado lógico padrão quando o circuito está aberto (por exemplo, quando um botão não está pressionado).

- **Resistor de Pull-up:** Um resistor de *pull-up* conecta o pino GPIO a uma fonte de tensão positiva (VCC, ou 3.3V na BitDogLab). Com essa configuração, quando o botão está solto, o pino lê um estado lógico ALTO por padrão. Ao pressionar o botão, o circuito se fecha para o terra (GND), a corrente flui para o caminho de menor resistência, e o pino passa a ler um estado lógico BAIXO.

- **Resistor de Pull-down:** De forma análoga, um resistor de *pull-down* conecta o pino ao terra (GND). Neste caso, o estado padrão do pino quando o botão está solto é BAIXO. Ao pressionar o botão, o circuito se fecha para a fonte de tensão (VCC), e o

pino passa a ler um estado lógico ALTO.

Para simplificar o design de circuitos, microcontroladores modernos como o RP2040 incluem resistores de *pull-up* e *pull-down* internos, que podem ser habilitados via software para cada pino GPIO. Na BitDogLab, os botões já são projetados com essa estabilização. Compreender este conceito é vital em projetos educacionais, pois ensina sobre a importância da estabilidade de sinais e como prevenir falhas de leitura, uma das fontes mais comuns de bugs em sistemas embarcados [Instruments 2021, Community 2025, Foundation 2021].

4.4.3. Lendo o Mundo Analógico: O Conversor ADC

Enquanto os pinos GPIO em modo digital são perfeitos para ler estados discretos (ligado/desligado, pressionado/solto), o mundo físico é, em sua maior parte, analógico. Grandezas como temperatura, intensidade de luz e a posição de um joystick não variam em saltos, mas sim de forma contínua. Para que um microcontrolador, que opera no domínio digital, possa interpretar esses sinais, ele precisa de um tradutor: o **Conversor Analógico-Digital**, ou ADC (*Analog-to-Digital Converter*).

O ADC é um periférico fundamental que mede uma tensão analógica contínua em um pino e a converte em um valor numérico digital que o software pode processar. A precisão dessa conversão é definida pela **resolução** do ADC, medida em bits. O RP2040, por exemplo, possui um ADC de 12 bits, o que significa que ele pode representar a faixa de tensão de entrada (de 0 a 3.3V) em 2^{12}, ou 4096, níveis distintos. Na prática, isso permite leituras muito precisas de sensores como o joystick da BitDogLab, onde um pequeno movimento pode ser detectado como uma mudança sutil no valor digital lido [Foundation 2021, Kester 2009].

Outro parâmetro crucial é a **taxa de amostragem**, que define quantas vezes por segundo o ADC realiza uma conversão. Para reconstruir um sinal analógico de forma fiel no domínio digital, a teoria da comunicação estabelece que a taxa de amostragem deve ser pelo menos o dobro da frequência máxima do sinal que se deseja medir, um princípio conhecido como Teorema de Nyquist-Shannon [Shannon 1948].

No contexto da BitDogLab, o ADC é a ponte que permite ao firmware ler a posição exata do joystick e a intensidade do som captado pelo microfone. No entanto, a aquisição de dados analógicos no mundo real apresenta desafios, como o ruído elétrico, que pode causar flutuações nas leituras. Para mitigar isso, técnicas de software, como a aplicação de médias móveis (*moving average*) ou *oversampling*, são frequentemente utilizadas para filtrar o ruído e obter um valor mais estável e preciso. A compreensão desses conceitos e técnicas é essencial para o desenvolvimento de sistemas embarcados que interagem de forma confiável com o ambiente [STMicroelectronics 2009, Lee and Levin 2025, Fruett et al. 2024].

4.5. Estudo de Caso: Construindo um Jogo de Velocidade

Este estudo de caso aplica os conceitos teóricos e práticos discutidos anteriormente em um projeto interativo e educativo, utilizando a placa BitDogLab para desenvolver o jogo "Ligeirinho" — uma aplicação de medição de tempo de reação que integra programação embarcada, interfaces digitais e analógicas, e periféricos sensoriais. Baseado no microcon-

trolador RP2040, o projeto exemplifica o co-design hardware-software, com foco em entradas/saídas GPIO, gerenciamento de sinais, temporização e feedback multimodal (visual, auditivo e tátil). Essa implementação prática reforça princípios de sistemas embarcados, como determinismo temporal e eficiência energética, enquanto promove experimentação em contextos STEAM, alinhando-se a abordagens educacionais que incentivam a prototipagem rápida e a depuração iterativa [Fruett et al. 2024, Alves 2025, Foundation 2021].

4.5.1. Definição do Escopo e Lógica do Jogo

O fluxo lógico segue uma estrutura sequencial com estados bem definidos: (i) inicialização e exibição de mensagem de prontidão no display OLED; (ii) detecção de pressão no botão A para iniciar; (iii) fase de preparação com LED verde aceso e atraso randômico (1-5 segundos); (iv) ativação da fase de reação com LED vermelho, buzzer e início de temporizador; (v) captura de tempo ao pressionar B, com cálculo de latência em milissegundos; (vi) exibição do resultado no OLED e reset para nova rodada. Essa lógica incorpora mecanismos de debouncing para estabilidade de sinais e interrupções para respostas determinísticas, destacando restrições de tempo real soft (onde atrasos degradam a experiência, mas não causam falhas críticas). O escopo enfatiza acessibilidade, utilizando firmware em C com SDK do RP2040, e incentiva extensões como multiplayer ou integração com sensores adicionais para explorar trade-offs em complexidade e consumo [Alves 2025, Foundation 2021, Kopetz 2022].

4.5.2. Implementação Prática do Firmware

A implementação do firmware para o "Ligeirinho" é realizada em linguagem C utilizando o SDK 2.1.0 do Raspberry Pi Pico, com configuração via CMakeLists.txt para compilar o executável e vincular bibliotecas essenciais (pico_stdlib, hardware_timer, hardware_pwm, hardware_clocks, hardware_i2c). Esse arquivo gerencia o processo de build de forma cross-platform, definindo padrões de C/C++ (C11 e C++17), importando o SDK via pico_sdk_import.cmake e inicializando-o com pico_sdk_init(). Ele também especifica o board como pico_w (para suporte Wi-Fi, embora não usado aqui, permitindo extensões futuras), ativa saídas de depuração via USB/UART (pico_enable_stdio_usb e pico_enable_stdio_uart) e gera outputs extras como .uf2 para upload bootloader [Pi 2025, Foundation 2021, Alves 2025].

Um trecho chave extraído do CMakeLists.txt ilustra a adição do executável e vinculação de bibliotecas, destacando a integração com periféricos do RP2040 (timers, PWM, clocks e I²C para OLED):

```
# Adiciona o arquivo-fonte correto
add_executable(Ligeirinho Ligeirinho.c inc/ssd1306_i2c.c)

# Define o nome e a versão do programa
pico_set_program_name(Ligeirinho "Ligeirinho")
pico_set_program_version(Ligeirinho "0.1")

# Ativa saída USB para depuração
pico_enable_stdio_uart(Ligeirinho 0)
pico_enable_stdio_usb(Ligeirinho 1)

# Adiciona bibliotecas necessárias
```

```
target_link_libraries(Ligeirinho pico_stdlib hardware_timer
    hardware_pwm hardware_clocks hardware_i2c)

# Inclui diretórios do projeto
target_include_directories(Ligeirinho PRIVATE ${CMAKE_CURRENT_LIST_DIR
    })

# Gera arquivos adicionais necessários para o Pico
pico_add_extra_outputs(Ligeirinho)
```

Listing 4.1: Configuração do ambiente de compilação para o projeto Ligeirinho com o SDK do Raspberry Pi Pico

Aqui, add_executable compila os fontes principais (Ligeirinho.c e a biblioteca OLED ssd1306_i2c.c), enquanto target_link_libraries vincula módulos hardware para suporte a timers (medição de reação), PWM (controle de LEDs e buzzer), clocks (gerenciamento de frequência) e I²C (display). A ativação de stdio USB facilita depuração via serial, crucial para testes educativos, e pico_add_extra_outputs gera binários como .uf2 para upload direto, promovendo iterações rápidas sem ferramentas externas. Essa configuração exemplifica boas práticas em co-design, onde o build reflete restrições embarcadas como memória limitada (via stdlib minimalista) e determinismo temporal (via hardware_timer para WCET baixo) [Pi 2025, Akdur et al. 2018].

O código integra conceitos da Seção 1.4, como GPIOs para entradas/saídas digitais (botões e LEDs), resistores pull-up internos para estabilidade de sinais, e PWM para controle de brilho e áudio, evitando estados de flutuação e garantindo respostas precisas. A seguir, apresentam-se trechos comentados do arquivo Ligeirinho.c, destacando a aplicação prática na BitDogLab [Alves 2025, Foundation 2021, Community 2025].

Primeiro, a inicialização de hardware e periféricos demonstra o uso de GPIOs como entradas com pull-up (para botões, prevenindo flutuação conforme Seção 1.4.2) e saídas PWM para LEDs e buzzer (controlando duty cycle para eficiência energética):

```
// Inicializa a interface I2C para o display OLED
i2c_init(i2c1, ssd1306_i2c_clock * 1000);
gpio_set_function(I2C_SDA, GPIO_FUNC_I2C);
gpio_set_function(I2C_SCL, GPIO_FUNC_I2C);
gpio_pull_up(I2C_SDA);
gpio_pull_up(I2C_SCL);

// Inicializa o display OLED e exibe mensagem inicial
ssd1306_init();
display_text("PRESSIONE A PARA COMECAR!");

//Configura os botões como entradas com pull-up interno
gpio_init(BUTTON_START);
gpio_init(BUTTON_STOP);
gpio_set_dir(BUTTON_START, GPIO_IN);
gpio_set_dir(BUTTON_STOP, GPIO_IN);
gpio_pull_up(BUTTON_START);
gpio_pull_up(BUTTON_STOP);

// Inicializa os LEDs para PWM
```

```
pwm_init_led(LED_GREEN);
pwm_init_led(LED_RED);
// Inicialmente, ambos os LEDs estão desligados
pwm_set_gpio_level(LED_GREEN, 0);
pwm_set_gpio_level(LED_RED, 0);

// Inicializa o buzzer com PWM
pwm_init_buzzer(BUZZER);
```

Listing 4.2: Inicialização de periféricos da BitDogLab para o jogo Ligeirinho

Aqui, gpio_pull_up aplica resistores internos (aprox. 50-80kΩ) para estabilidade, enquanto pwm_init_led/buzzer configura slices PWM com wrap fixo (1000 para LEDs, ajustável para buzzer), ilustrando controle de saídas digitais moduladas para feedback sensorial [Instruments 2021, Foundation 2021].

Em seguida, a lógica de debouncing e interrupções para botões integra detecção de bordas e temporização, evitando ruídos em entradas digitais:

```
bool debounce_button(uint gpio)
{
    static uint32_t last_time = 0;
    uint32_t current_time = to_ms_since_boot(get_absolute_time());

    if (current_time - last_time < 50)
    {
        return false;
    }

    last_time = current_time;
    return gpio_get(gpio) == 0;
}

// Configura a interrupção para o botão B (BUTTON_STOP)
gpio_set_irq_enabled_with_callback(BUTTON_STOP, GPIO_IRQ_EDGE_FALL,
   true, &gpio_callback);
```

Listing 4.3: Implementação de debouncing e interrupção para botões no jogo Ligeirinho

Essa função debounce_button filtra pressões rápidas (<50ms), complementando pull-ups para sinais estáveis, enquanto interrupções (EDGE_FALL) garantem respostas em tempo real sem polling constante, otimizando consumo [Foundation 2021, Instruments 2021].

A fase de jogo utiliza timers absolutos para medição precisa e rand() para atraso aleatório, aplicando conceitos de tempo real:

```
void start_game()
{
    // ... (preparação com LED verde via PWM)
    pwm_set_gpio_level(LED_GREEN, LED_ON);

    uint delay_ms = 1000 + (rand() % 4000);
    for (uint i = 0; i < delay_ms / 10; i++)
    {
        sleep_ms(10);
```

```
        if (gpio_get(BUTTON_STOP) == 0)
        {
            false_start_detected = true;
            break;
        }
    }

    // ... (ativação de LED vermelho, buzzer e timer)
    pwm_set_gpio_level(LED_GREEN, 0);
    pwm_set_gpio_level(LED_RED, LED_ON);
    buzzer_beep(3000, 300);
    start_timer();
    reaction_phase = true;
}

// Callback de interrupção para botão B
void gpio_callback(uint gpio, uint32_t events)
{
    if (gpio == BUTTON_STOP && game_running && reaction_phase)
    {
        reaction_time = get_absolute_time();
        button_b_pressed = true;
    }
}
```

Listing 4.4: Lógica principal e controle do jogo Ligeirinho

Finalmente, o loop principal e cálculo de tempo unem esses elementos, com exibição via OLED (usando I²C, embora não haja ADC explícito aqui, o framework suporta extensões analógicas como joystick para variações):

```
while (true)
{
    if (debounce_button(BUTTON_START))
    {
        if (!game_running)
        {
            start_game();
        }
        sleep_ms(300);
    }

    if (game_running && reaction_phase && button_b_pressed)
    {
        uint32_t elapsed_time = get_elapsed_time();
        pwm_set_gpio_level(LED_RED, 0);
        stop_buzzer(0, NULL);

        char buffer[20];
        sprintf(buffer, "Tempo: %.1f ms", (float)elapsed_time);
        display_text(buffer);

        sleep_ms(5000);
        // Reset estados
        game_running = false;
```

```
        reaction_phase = false;
        // ...
        display_text("PRESSIONE A PARA COMECAR!");
    }
}
```

Listing 4.5: Loop principal do jogo Ligeirinho para controle de botões e exibição de resultados

Essa estrutura aplica GPIOs, pull-ups e timers da Seção 1.4, com PWM para atuadores, promovendo depuração via hardware-in-the-loop [Alves 2025, Garousi et al. 2018].

4.5.3. Análise e Execução do Projeto

Ao executar o firmware na BitDogLab (compilado e carregado via USB em modo bootloader), o comportamento esperado inicia com a mensagem "PRESSIONE "A" PARA COMECAR!" no OLED. Pressionar o botão A acende o LED verde (brilho 50% via PWM), seguido de atraso aleatório; em seguida, o LED vermelho acende com beep no buzzer (3000Hz por 300ms), iniciando o timer. Pressionar B captura o tempo (ex.: "Tempo: 250.0 ms" exibido), desliga LEDs e reseta após 5s. Queimas de largada piscam o LED vermelho três vezes com mensagem "MUITO CEDO!". Essa execução destaca estabilidade de sinais (sem flutuações graças a pull-ups e debouncing) e determinismo temporal (latências <1ms via interrupções), com consumo eficiente (50mA em operação). Incentiva-se experimentação: ajuste delays para simular hard real-time, integre joystick via ADC para controles analógicos, ou adicione métricas WCET para análise de desempenho, fomentando iterações educativas e depuração de falhas como ruídos em botões [Alves 2025, Foundation 2021, Kopetz 2022, Garousi et al. 2018].

4.6. Conclusão

Este capítulo apresentou uma jornada introdutória ao universo dos sistemas embarcados, partindo dos pilares teóricos até a aplicação prática dos conceitos na placa de desenvolvimento BitDogLab. Iniciamos com a definição de sistemas embarcados como soluções computacionais dedicadas, otimizadas para operar sob restrições de tempo real, energia e custo, em contraste com a flexibilidade da computação de propósito geral [Wolf 2001, Kopetz 2022]. A análise da arquitetura típica destacou o acoplamento intrínseco entre hardware — microcontroladores, sensores e atuadores — e software — o firmware —, enfatizando o co-design como uma estratégia essencial para balancear requisitos funcionais e não-funcionais [Akdur et al. 2018, Micco et al. 2018].

A apresentação da BitDogLab como uma ferramenta educacional open-source demonstrou sua versatilidade para a prototipagem rápida. A exploração de seus periféricos integrados permitiu a aplicação prática de conceitos fundamentais, como o uso de pinos GPIO para o controle de entradas e saídas digitais, a aplicação de resistores de pull-up/down para garantir a estabilidade de sinais e a utilização do conversor ADC para a leitura de sensores analógicos [Foundation 2021, Fruett et al. 2024, Instruments 2021]. O estudo de caso com o jogo "Ligeirinho" consolidou esses aprendizados, ao exigir a implementação de uma lógica com temporização precisa, feedback multimodal (visual e sonoro) e um ciclo de depuração interativo, ilustrando a integração hardware-software em um contexto

STEAM acessível [Alves 2025, Garousi et al. 2018].

A integração entre hardware e software emerge, portanto, como o cerne do desenvolvimento de sistemas embarcados. O projeto "Ligeirinho" exemplifica essa sinergia, onde a escolha de usar PWM para controlar a intensidade dos LEDs e os tons do buzzer, ou o protocolo I²C para se comunicar com a tela OLED, são decisões coordenadas que influenciam diretamente o firmware. Essa experiência prática prepara os aprendizes para os desafios da indústria, que incluem a conformidade com normas de segurança (como a ISO 26262) e a análise de falhas, além de fomentar a criatividade para a prototipagem de novos sistemas ciber-físicos [Pereira et al. 2017, Pasricha 2022]. A BitDogLab, com seu design colaborativo, oferece um ponto de partida ideal para a experimentação, incentivando o aprendizado contínuo em áreas como robótica, automação e dispositivos interativos, alinhando-se às demandas de um campo em constante evolução [Fruett et al. 2024, Industries 2025, Sztipanovits et al. 2005].

References

[Akdur et al. 2018] Akdur, D., Comer, R., and Magedanz, T. (2018). Co-design and co-simulation of heterogeneous embedded systems: A survey. *Simulation Modelling Practice and Theory*, 89:276–292.

[Akesson et al. 2020] Akesson, B., Nasri, M., Nelissen, G., Altmeyer, S., and Davis, R. I. (2020). An empirical survey into industry practice in real-time embedded systems. *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

[Alves 2025] Alves, C. M. (2025). Ligeirinho — jogo de reflexo com bitdoglab. GitHub Repository. Jogo de tempo de reação que utiliza LEDs, botões, buzzer e display OLED em C com SDK do Raspberry Pi Pico (RP2040). Acesso em: 6 de setembro de 2025.

[Ariza and Baez 2021] Ariza, J. A. and Baez, H. (2021). Understanding the role of single-board computers in engineering and computer science education: A systematic literature review. *Computer Applications in Engineering Education*.

[Benyeogor et al. 2024] Benyeogor, M. S., Benyeogor, A., Olaiya, K. A., and Agumey, P. (2024). Advancing embedded systems education: A pedagogical programming framework for smart system and control applications. *TechRxiv / arXiv preprint*.

[Community 2025] Community, B. (2025). Bitdoglab — open-hardware educational platform (readme and hcd). `https://github.com/BitDogLab/BitDogLab`. Acesso em: 1 de setembro de 2025.

[Elsevier / ScienceDirect 2025] Elsevier / ScienceDirect (2025). Embedded computer — sciencedirect topics. `https://www.sciencedirect.com/topics/computer-science/embedded-computer`. Acesso em: 1 de setembro de 2025.

[Foundation 2021] Foundation, R. P. (2021). Rp2040 microcontroller datasheet. `https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf`. Acesso em: 1 de setembro de 2025.

[Fruett et al. 2024] Fruett, F., Barbosa, F. P., Fraga, S. C. Z., and ao Guimarães, P. I. A. (2024). Empowering steam activities with artificial intelligence and open hardware: The bitdoglab. *IEEE Transactions on Education*, 67(3):462–471.

[Garousi et al. 2018] Garousi, V., Felderer, M., and Ribeiro, L. (2018). Testing embedded software: A survey of the literature. *Information and Software Technology*, 95:123–147.

[Industries 2025] Industries, A. (2025). Bitdoglab is a raspberry pi pico learning lab which encourages ai-assisted programming. `https://blog.adafruit.com/2025/02/03/bitdoglab-is-a-raspberry-pi-pico-learning-lab-which-encourages-ai-assisted-programming`. Acesso em: 1 de setembro de 2025.

[Instruments 2021] Instruments, T. (2021). Implications of slow or floating cmos inputs (application report scba004e). Texas Instruments Application Report SCBA004E, July 1994 – Rev. E (Reviewed July 2021). `https://www.ti.com/lit/an/scba004e/scba004e.pdf`. Acesso em: 1 de setembro de 2025.

[Kester 2009] Kester, W. (2009). Adc architectures ii: Successive approximation adcs (mt-021 tutorial). Analog Devices Tutorial. `https://www.analog.com/media/en/training-seminars/tutorials/mt-021.pdf`. Acesso em: 1 de setembro de 2025.

[Kopetz 2022] Kopetz, H. (2022). *Real–Time Systems: Design Principles for Distributed Embedded Applications*. Springer. Referência moderna que cobre requisitos temporais em sistemas embarcados.

[Lee and Levin 2025] Lee, M. and Levin, M. (2025). Anti-alias filter design for a 100 khz bandwidth data acquisition signal chain. Texas Instruments Application Note SBAA655A. `https://www.ti.com/lit/pdf/sbaa655a/sbaa655a.pdf`. Acesso em: 1 de setembro de 2025.

[Micco et al. 2018] Micco, L. D., Vargas, F. L., and Fierens, P. I. (2018). A literature review on embedded systems. *IEEE Latin America Transactions*. Survey com definição e panorama de sistemas embarcados.

[Pasricha 2022] Pasricha, S. (2022). Embedded systems education in the 2020s: Challenges, reflections, and future directions. *arXiv preprint*, arXiv:2206.03263.

[Pereira et al. 2017] Pereira, T., Albuquerque, D., Sousa, A., Alencar, F., and Castro, J. (2017). Retrospective and trends in requirements engineering for embedded systems: A systematic literature review. *Proceedings of the 20th Workshop on Requirements Engineering (WER 2017)*. Revisão sistemática sobre engenharia de requisitos em sistemas embarcados.

[Pi 2025] Pi, R. (2025). raspberrypi/pico-sdk. GitHub repository. SDK oficial para desenvolvimento em C/C++ para microcontroladores RP-series (RP2040 / Pico), com APIs para GPIO, PWM, I²C, timers, PIO e mais. Acesso em: 1 de setembro de 2025.

[Shannon 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27(3–4):379–423, 623–656.

[Solomon Systech / Adafruit (mirror) 2010] Solomon Systech / Adafruit (mirror) (2010). Ssd1306 oled driver – datasheet. `https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf`. Acesso em: 1 de setembro de 2025.

[STMicroelectronics 2009] STMicroelectronics (2009). Understanding and minimising adc conversion errors: Application note an1636. `https://www.st.com/resource/en/application_note/an1636-understanding-and-minimising-adc-conversion-errors-stmicroelectronics.pdf`. Acesso em: 1 de setembro de 2025.

[STMicroelectronics 2015] STMicroelectronics (2015). Mp34dt01-m – mems digital microphone datasheet. `https://www.st.com/resource/en/datasheet/mp34dt01-m.pdf`. Acesso em: 1 de setembro de 2025.

[Sztipanovits et al. 2005] Sztipanovits, J., Biswas, G., Frampton, K., Gokhale, A., et al. (2005). Introducing embedded software and systems education and advanced learning technology in an engineering curriculum. *ACM Transactions on Embedded Computing Systems*, 4(3):549–568.

[Wolf 2001] Wolf, W. (2001). *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann. Livro-texto clássico sobre sistemas embarcados.

[WORLDSEMI 2013] WORLDSEMI (2013). Ws2812b intelligent control led – datasheet. `https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf`. Acesso em: 1 de setembro de 2025.

Capítulo 5

Introdução ao Git e GitHub: Controle de Versão na Prática

Deyvison Samuel Gomes do Nascimento, Maria Vitória da Silva Araújo, Maria Yasmin Oliveira Mélo, M.e Maykol Lívio Sampaio Vieira Santos

Resumo

O minicurso "Introdução ao Git e GitHub"tem como objetivo apresentar, de forma acessível e prática, os fundamentos do controle de versão por meio das ferramentas Git, GitHub e GitHub Desktop. Destinado a estudantes iniciantes na área de tecnologia, o curso visa capacitar os participantes a utilizarem essas ferramentas essenciais tanto no contexto acadêmico quanto no mercado de trabalho. Com carga horária total de três horas, o conteúdo abordará desde a criação de uma conta no GitHub, a utilização dos principais comandos e fluxos de trabalho, além do envio e atualização de projetos em repositórios remotos. A metodologia adotada combina exposições teóricas com atividades práticas em laboratório, permitindo que os participantes acompanhem passo a passo o funcionamento das ferramentas e desenvolvam autonomia na utilização do versionamento de código. Ao final, espera-se que os alunos estejam aptos a criar e gerenciar seus próprios repositórios, colaborar em projetos em equipe e compreender a lógica do controle de versões distribuído, consolidando uma base sólida para práticas modernas de desenvolvimento.

Palavras-chave: *Git, GitHub, Versionamento de Código, GitHub Desktop*

Abstract

The short course "Introduction to Git and GitHub"aims to present, in an accessible and practical way, the fundamentals of version control through the tools Git, GitHub, and GitHub Desktop. Designed for beginner students in the field of technology, the course seeks to enable participants to use these essential tools both in academic settings and in

Deyvison Samuel Gomes do Nascimento (apresentador) é estudante do curso de Tecnologia em Análise e Desenvolvimento de Sistemas pelo Instituto Federal do Piauí (IFPI) campus Piripiri. 1
Maria Vitória da Silva Araújo é estudante do curso de Tecnologia em Análise e Desenvolvimento de Sistemas pelo IFPI campus Piripiri.
Maria Yasmin Oliveira Mélo (apresentadora) é estudante do curso de Tecnologia em Análise e Desenvolvimento de Sistemas pelo IFPI campus Piripiri.
Maykol Lívio Sampaio Vieira Santos (orientador) é professor de Informática no IFPI campus Piripiri e Mestre em Tecnologia e Gestão em EAD pela UFRPE (UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO).

the job market. With a total duration of three hours, the content will cover everything from creating a GitHub account, using the main commands and workflows, to sending and updating projects in remote repositories. The methodology combines theoretical explanations with hands-on lab activities, allowing participants to follow the step-by-step operation of the tools and develop autonomy in using code versioning. By the end of the course, students are expected to be able to create and manage their own repositories, collaborate on team projects, and understand the logic of distributed version control, establishing a solid foundation for modern development practices.
Keywords: *Git, GitHub, Code Versioning, GitHub Desktop*

5.1. Introdução

O Git surgiu em 2005, criado por Linus Torvalds após a ruptura com o BitKeeper como um sistema de controle de versão distribuído, rápido, seguro e adequado a fluxos de trabalho colaborativos. Sua eficiência, confiabilidade e suporte a branches tornaram-no padrão mundial no desenvolvimento de software. Já o GitHub, lançado em 2008, expandiu o uso do Git ao oferecer uma interface web intuitiva e recursos sociais, como *forks*, *pull requests* e *issues*, transformando-se em um espaço central de hospedagem e colaboração de código. Com a aquisição pela Microsoft em 2018, a plataforma ganhou ainda mais infraestrutura e integração com serviços em nuvem, consolidando-se como a maior comunidade de desenvolvedores do mundo. Hoje, Git e GitHub são essenciais para a organização, segurança e produtividade no desenvolvimento de software, além de impulsionarem práticas modernas de colaboração, DevOps e aprendizado em programação [Git-SCM, 2025; Microsoft, 2018].

5.1.1. História do Git

Antes do Git, alguns sistemas já buscavam controlar versões de código, como o *Source Code Control System* (SCCS), criado em 1972, que registrava alterações de forma sequencial, mas tinha limitações e funcionava apenas em Unix. Mais tarde, em 1986, o *Concurrent Versions System (CVS)*, trouxe o conceito de repositório compartilhado e permitiu colaboraço entre vários desenvolvedores, embora apresentasse falhas em operaçes de merge e dificuldades no gerenciamento de arquivos [Rochkind, 1975; Spinellis, 2005; Grune, 1986; Free Software Foundation, 1993].

Apesar dos avanços, esses sistemas eram centralizados, ou seja, dependiam de um servidor único para armazenar o código. Esse modelo gerava problemas como a dependência total do servidor — que, em caso de falha, interrompia o trabalho — e a limitação na colaboração em larga escala, mostrando a necessidade de soluções mais flexíveis e robustas [Loeliger & McCullough, 2012; Tech-in-Japan, 2021].

5.1.2. Linux e BitKeeper

Nos anos 1990, o Linux Kernel já era um dos maiores projetos colaborativos de software, mas até 2002 as contribuições eram integradas manualmente, com *patches* enviados por e-

mail, em um processo lento e sujeito a erros. Para resolver essas limitações, a comunidade passou a utilizar o BitKeeper, um sistema distribuído que permitia a cada desenvolvedor manter uma cópia completa do repositório, facilitando o trabalho offline e a integração em larga escala. Essa adoção representou um grande avanço em relação a modelos centralizados como o Subversion (SVN) [Pro Git — git-scm.com].

No entanto, o BitKeeper era proprietário e, em 2005, o acordo que permitia seu uso gratuito pela comunidade do Linux Kernel foi encerrado. Isso gerou um impasse, já que depender de uma tecnologia fechada colocava em risco a continuidade do projeto. Foi nesse cenário que Linus Torvalds decidiu criar uma nova ferramenta, livre e distribuída, capaz de atender às necessidades do Linux Kernel: o Git [Pro Git, s.d.; LWN.net, s.d.].

5.1.3. Criação do Git

Quando perdeu o acesso ao BitKeeper, Linus Torvalds decidiu criar seu próprio sistema de controle de versão distribuído. Para isso, estabeleceu alguns requisitos fundamentais que orientariam o desenvolvimento da nova ferramenta: precisava ser rápido, superando os sistemas existentes; deveria ser distribuído, garantindo que cada desenvolvedor tivesse uma cópia completa do repositório em sua máquina; tinha que ser seguro, assegurando a integridade dos dados; e, por fim, precisava oferecer bom suporte a fluxos de trabalho não lineares, lidando de forma eficiente com branches e merges [i-Programmer].

Pouco tempo depois, Linus transferiu a manutenção do projeto para Junio C. Hamano, que rapidamente se destacou na comunidade e até hoje atua como mantenedor principal do Git, liderando seu desenvolvimento contínuo e garantindo sua evolução ao longo dos anos [Pro Git, s.d.].

5.1.4. Evolução

Após sua criação em 2005, o Git evoluiu rapidamente, recebendo melhorias constantes e conquistando uma comunidade cada vez maior. Em 2007, ele começou a se popularizar fora do desenvolvimento do kernel Linux, chamando a atenção de outros projetos de software livre. No ano seguinte, em 2008, surgiu o GitHub, uma plataforma que revolucionou a forma de usar o Git ao oferecer uma interface web simples e recursos como *issues*, *pull requests* e colaboração social. Essa combinação foi decisiva para a popularização do Git em escala global [Pro Git, s.d.; GitHub, 2025].

A partir de 2010, grandes empresas de tecnologia, como Google, Microsoft, Facebook e Twitter, passaram a adotar o Git como ferramenta padrão em seus fluxos de desenvolvimento, o que consolidou sua posição como principal sistema de controle de versão distribuído. Em 2018, a Microsoft adquiriu o GitHub, fortalecendo ainda mais o ecossistema em torno do Git e demonstrando a importância estratégica dessa ferramenta para o desenvolvimento de software moderno [Microsoft, 2018; Wired, 2018].

5.1.5. Por que o Git se tornou o padrão?

O Git se consolidou como padrão no desenvolvimento de software por adotar um modelo distribuído, no qual cada cópia do repositório funciona como um *backup* completo e independente, permitindo trabalho mesmo sem conexão a um servidor central. Sua performance também é um destaque, pois operações locais como *commit, branch e merge* são executadas de forma muito rápida em comparação com sistemas anteriores [i-Programmer; Pro Git — git-scm.com].

Outro diferencial é o suporte a *branches* leves, que facilita fluxos paralelos e colaborativos, além da forte ênfase em segurança, com uso de algoritmos de hash como SHA-1 e SHA-256 para proteger a integridade do histórico. O crescimento de plataformas como GitHub, GitLab e Bitbucket criou um ecossistema robusto de colaboração, o que consolidou o Git como o sistema de controle de versão mais utilizado no mundo [GitLab, 2025; Git, 2025; Rewind, 2024].

5.2. História do GitHub

Em 2008, surgiu o GitHub, fundado por Tom Preston-Werner, Chris Wanstrath, PJ Hyett e, posteriormente, Scott Chacon, que entrou para contribuir com a documentação. O objetivo inicial da plataforma era combinar o poder do Git com funcionalidades sociais, criando um espaço que facilitasse o processo de hospedar, revisar e colaborar em projetos, tanto de código aberto quanto privados [Founding; PSL Models; Medium; WIRED].

O grande diferencial do GitHub estava em sua interface web amigável, que permitia navegar pelos repositórios de forma simples e intuitiva. Além disso, introduziu recursos que se tornaram padrão na colaboração de software, como *forks, pull requests e issues*, transformando o fluxo de trabalho dos desenvolvedores. Outro aspecto inovador foi a criação de perfis e métricas para programadores, funcionando como uma espécie de "rede social para desenvolvedores", o que estimulou ainda mais a colaboração e a visibilidade dentro da comunidade de software [Timeline; WIRED].

5.2.1. Crescimento inicial

A partir de 2010, o GitHub deixou de ser apenas um espaço para projetos de código aberto e passou a atrair empresas e grandes corporações, que enxergaram na plataforma uma forma eficiente de organizar e gerenciar fluxos de desenvolvimento. Nesse contexto, a plataforma evoluiu para hospedar não apenas repositórios públicos, mas também projetos privados, o que ampliou significativamente seu alcance no mercado corporativo [Encyclopedia Britannica; GitHub, 2014].

O crescimento foi exponencial, com milhões de desenvolvedores migrando para o GitHub, que se tornou o padrão de facto em hospedagem e colaboração em projetos de software. O modelo baseado em *forks, pull requests e issues* consolidou-se como referência global, influenciando inclusive plataformas concorrentes. Esse movimento marcou a transição do GitHub de uma ferramenta voltada majoritariamente para a comuni-

dade de código aberto para uma infraestrutura essencial no desenvolvimento de software em escala global, utilizada tanto por programadores independentes quanto por grandes empresas de tecnologia [Wired, 2012; Wired, 2013].

Em 2018, o GitHub foi adquirido pela Microsoft por US$ 7,5 bilhões, em uma das maiores negociações do setor de tecnologia daquele período. A compra gerou desconfiança inicial na comunidade de código aberto, que historicamente via a Microsoft com certo receio [Microsoft, 2018; TechCrunch, 2018]. Sob a gestão da Microsoft, a plataforma recebeu novos investimentos, expandiu sua infraestrutura e fortaleceu a integração com serviços como o Azure e outras ferramentas do ecossistema Microsoft. Longe de perder relevância, o GitHub continuou crescendo e consolidou-se ainda mais como o principal espaço de colaboração em software no mundo, reunindo milhões de desenvolvedores e empresas em torno do desenvolvimento aberto e compartilhado [TechCrunch, 2018; Wired, 2018].

5.2.2. Status atual e impacto

Atualmente, o GitHub é a maior plataforma de hospedagem de código do mundo, reunindo mais de 100 milhões de repositórios e cerca de 90 milhões de desenvolvedores registrados. A plataforma atende tanto projetos de código aberto, que impulsionam a inovação coletiva, quanto empresas privadas, que utilizam seus recursos para gerenciar fluxos de desenvolvimento em larga escala [Encyclopedia Britannica].

O GitHub tornou-se peça central na cultura DevOps e nos processos de Integração Contínua e Entrega/Deploy Contínuo(CI/CD), permitindo integração contínua, automação e colaboração global de software. Além disso, consolidou-se também como um hub educacional, oferecendo iniciativas como o GitHub *Student Developer Pack*, cursos e ações voltadas ao incentivo do aprendizado de programação, contribuindo para a formação de novas gerações de desenvolvedores [GitHub, 2025; GitHub, 2025].

5.3. Importância do Git/GitHub hoje

O Git é uma das ferramentas mais importantes no desenvolvimento de software atualmente, pois funciona como um sistema de controle de versão distribuído. Ele registra todo o histórico de modificações feitas em um projeto, permitindo que desenvolvedores acompanhem cada alteração no código, retornem a versões anteriores quando necessário e entendam como o sistema evoluiu ao longo do tempo. Esse recurso evita perdas de trabalho e garante maior segurança no processo de desenvolvimento, além de permitir que cada programador mantenha em sua máquina uma cópia completa do repositório com todo o histórico do projeto [Pro Git Book – Chacon & Straub, Apress, 2014; Atlassian – What is Git?].

Outro ponto central é a colaboração. O Git facilita o trabalho em equipe, permitindo que vários desenvolvedores atuem simultaneamente em um mesmo projeto sem que suas mudanças interfiram umas nas outras. Essa característica torna o fluxo de trabalho

mais ágil, estruturado e produtivo. Em resumo, o Git não é apenas uma ferramenta para salvar versões de código, mas um verdadeiro pilar do desenvolvimento moderno, garantindo organização, segurança e eficiência no ciclo de criação de software [HostRagons, 2025; Pro Git Book – Chacon & Straub, Apress, 2014].

O GitHub é uma das plataformas mais relevantes do ecossistema tecnológico, funcionando como um espaço central para hospedagem, colaboração e compartilhamento de código. Baseado no Git, ele amplia suas funcionalidades ao oferecer uma interface prática na nuvem e ferramentas que apoiam tanto projetos individuais quanto grandes iniciativas globais. Um dos seus principais diferenciais é o incentivo à colaboração: milhões de desenvolvedores e organizações utilizam recursos como *pull requests, issues e discussions* para propor melhorias, revisar alterações e resolver problemas em conjunto. Esse ambiente participativo transformou o GitHub em um ponto de encontro para projetos de impacto mundial, como o Linux, o React e o Visual Studio Code [GitHub Docs – About GitHub; Coursera – What is GitHub?].

Além disso, a plataforma integra práticas modernas de DevOps e CI/CD, permitindo configurar fluxos automáticos de testes, validação e implantação, o que aumenta a eficiência e a qualidade das entregas. O GitHub também tem grande importância na educação, com iniciativas como o *Student Developer Pack*, que oferece ferramentas profissionais gratuitas e incentiva estudantes a aprenderem de forma prática como funciona o desenvolvimento colaborativo. Assim, o GitHub consolidou-se como mais que um repositório de código: é um ecossistema essencial para colaboração, inovação e aprendizado em escala global [DEV Community – How GitHub Improves Security and CI/CD Workflows, 2024; GitHub, 2025].

5.4. Conceitos Iniciais

Os conceitos básicos de Git e GitHub giram em torno do controle de versão e da colaboração em projetos de software. O Git é um sistema que registra todas as alterações feitas em arquivos, permitindo acompanhar o histórico, restaurar versões anteriores e trabalhar com ramificações para testar novas ideias sem comprometer a versão principal. Já o GitHub é uma plataforma online que utiliza o Git como base, mas adiciona ferramentas de colaboração, como revisão de código, gerenciamento de tarefas e integração com automações. No uso prático, o fluxo básico envolve criar ou clonar um repositório, registrar mudanças com *commits*, enviar e receber atualizações de um repositório remoto e gerenciar branches para desenvolvimento paralelo. Assim, Git e GitHub juntos oferecem organização, segurança e eficiência no desenvolvimento individual ou em equipe[Chacon & Straub, 2014; GitHub Docs, 2025; GeeksforGeeks, 2023].

5.4.1. O que é controle de versão?

O controle de versão é um sistema que registra e gerencia todas as alterações feitas em arquivos de um projeto ao longo do tempo, permitindo retornar a versões anteriores,

acompanhar a evolução e desfazer erros. Além de organização, ele facilita o trabalho em equipe, já que vários desenvolvedores podem atuar de forma simultânea sem sobrescrever o trabalho uns dos outros. Com o uso de ramificações (*branches*), cada membro pode testar soluções em separado e depois integrá-las de forma segura ao projeto principal [Earth Data Science – Version Control Introduction; Atlassian – Git branching explained].

O GitHub, por sua vez, leva os benefícios do Git para a nuvem, oferecendo hospedagem de repositórios e ampliando as possibilidades de colaboração. Além de compartilhar código, a plataforma fornece recursos como *pull requests*, gerenciamento de tarefas, permissões de acesso, integração com ferramentas de automação e suporte a projetos de qualquer escala. Dessa forma, Git e GitHub tornaram-se pilares do desenvolvimento moderno, garantindo organização, eficiência e qualidade em equipes de diferentes tamanhos [GitHub Docs – About GitHub; Everhour Blog – Why GitHub is important in modern development (2025)].

5.4.2. Diferença entre Git e GitHub

O Git é um sistema de controle de versão distribuído que funciona localmente, permitindo registrar todas as alterações de um projeto, acompanhar o histórico completo de versões, restaurar estados anteriores e criar ramificações (*branches*) sem comprometer a versão principal. Cada alteração inclui informações sobre autor, data e modificações realizadas, garantindo rastreabilidade e organização do projeto. Por ser distribuído, cada colaborador possui uma cópia completa do repositório, podendo trabalhar offline e sincronizar alterações apenas quando necessário [Stack Overflow – Git is a revision control system; GeeksforGeeks – Differences Between Git and GitHub].

O GitHub é uma plataforma online que hospeda repositórios Git na nuvem, oferecendo armazenamento centralizado e recursos de colaboração, como pull requests, issues, controle de permissões e integração com automação via *GitHub Actions*. Ele reúne uma comunidade global de desenvolvedores, permitindo que equipes de qualquer tamanho trabalhem de forma organizada e colaborativa. Apesar de depender de conexão à internet e de uma conta na plataforma, o GitHub não substitui o Git, que continua responsável pelo controle de versões local e distribuído, podendo ser usado também com outras plataformas como GitLab, Bitbucket ou servidores privados [GitHub Docs – How do Git and GitHub work together?; DataCamp – Git vs GitHub: Key differences explained; HubSpot – Git vs GitHub].

5.5. Desenvolvimento

Quanto ao desenvolvimento, este será realizado de forma isolada em alguns momentos. Inicialmente, no item 1.5.1, serão apresentados os conceitos básicos da instalação da ferramenta GitHub Desktop. Em seguida, no item 1.5.2, será introduzida a criação e a sincronização de repositórios na mesma ferramenta, além da inclusão de outros elementos, como o *.gitignore* e o *README.md*. Posteriormente, haverá ainda uma subseção,

denominada 1.5.3, na qual será apresentada a colaboração por meio de *branches* com o GitHub Desktop [GitHub 2025].

5.5.1. Instalação e Configuração inicial do GitHub Desktop

Quanto à instalação da aplicação GitHub Desktop, esta pode ser realizada atualmente no macOS 11.0 ou posterior e no Windows 10 (64 bits) ou versão posterior [GitHub 2025]. Para efetuar a instalação da aplicação, o usuário deve seguir os seguintes procedimentos:

1. Acesse a página de *download* do GitHub Desktop).
2. Clique em Baixar para Windows, ou baixar para MacOS.

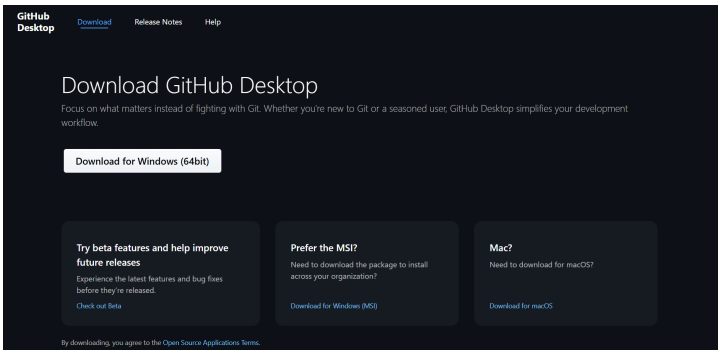


Figura 5.1. Página de descarregamento da ferramenta GitHub Desktop. Fonte: Próprio Autor.

3. Na pasta *Downloads* do computador, o usuário deve clicar duas vezes no arquivo de instalação do GitHub Desktop.

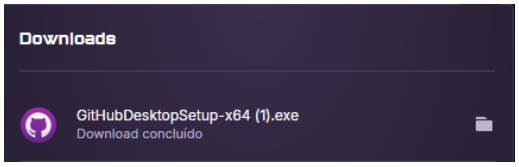


Figura 5.2. Exemplo de criação de *commit* no GitHub Desktop com mensagem clara e objetiva. Fonte: Próprio Autor.

4. GitHub Desktop será executado após a instalação ser concluída.

Após a instalação da aplicação em sua máquina, será necessária a criação de uma conta no GitHub ou no GitHub Enterprise, a fim de que o usuário possa trocar dados entre seus repositórios locais e remotos [GitHub 2025]. Para que o usuário comum se inscreva em uma conta pessoal, devem ser seguidos os seguintes passos:

1. Navegue até a página https://github.com/.

2. Clique em Cadastra-se no GitHub.

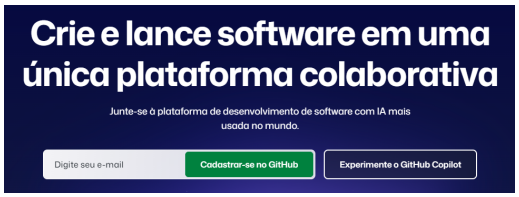


Figura 5.3. Exemplo de criação de commit no GitHub Desktop com mensagem clara e objetiva. Fonte: Próprio Autor.

3. Preencha as informações de cadastro da página, ou como alternativa, clique em *Continue with Google* para se inscrever usando uma conta do Google.

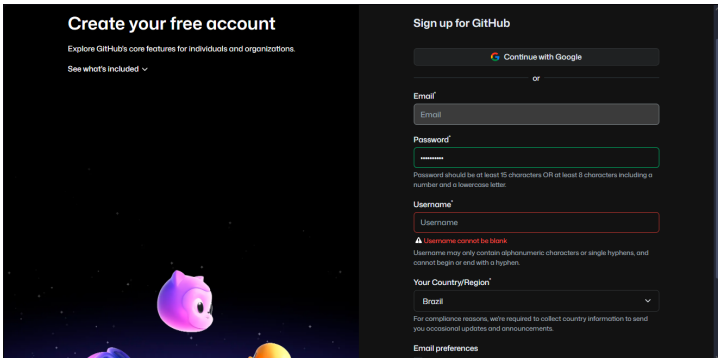


Figura 5.4. Exemplo de criação de commit no GitHub Desktop com mensagem clara e objetiva. Fonte: Próprio Autor.

4. Continue seguindo os prompts indicados pela plataforma para finalizar a criação de sua conta pessoal. Vale ressaltar que durante a inscrição será solicitado ao usuário a verificação de e-mail para fins de segurança.

Com relação à interface da aplicação proposta, que será devidamente apresentada no decorrer do texto, temos a seguinte captura de tela ilustrando a tela inicial dessa ferramenta. Ao visualizá-la, nota-se um evidente minimalismo, com uma interface bastante enxuta, além da ausência de tradução para o português brasileiro, onde toda a interface é exibida em língua inglesa.

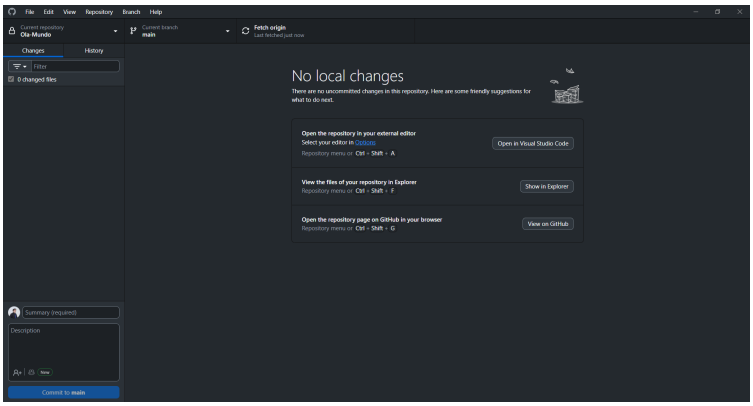


Figura 5.5. Tela inicial do GitHub Desktop. Fonte: Próprio autor.

5.5.2. Criando e Sincronizando Repositórios com GitHub Desktop

Embora o Git possa ser utilizado por meio da linha de comando, muitas equipes e iniciantes preferem ferramentas visuais que simplifiquem sua utilização. O GitHub Desktop é uma dessas soluções, oferecendo uma interface gráfica intuitiva para a criação, sincronização e gerenciamento de repositórios, sem abrir mão das funcionalidades essenciais do versionamento.

5.5.2.1. Criação do repositório local

No GitHub Desktop, o usuário pode criar um novo repositório local diretamente pelo menu *File - New Repository*. Nesse momento, são definidos alguns parâmetros essenciais:

1. **Name:** corresponde ao nome do repositório. É recomendável utilizar nomes significativos e descritivos que facilitem a identificação do projeto.

2. **Description:** campo opcional que permite resumir o objetivo do projeto. Essa descrição auxilia colaboradores a compreenderem rapidamente a finalidade do repositório.

3. **Local path:** define o diretório do computador onde o repositório será armazenado. Manter uma estrutura organizada facilita o gerenciamento de múltiplos projetos.

4. **Initialize this repository with a README:** ao marcar essa opção, o repositório é criado já contendo um arquivo `README.md`. Esse documento funciona como a apresentação inicial do projeto, trazendo informações como objetivos, instruções de instalação e exemplos de uso.

5. **Git ignore:** permite escolher um modelo pré-definido de arquivos a serem ignorados pelo Git, evitando que itens desnecessários sejam versionados. Por exemplo, ao selecionar `Node`, pastas como `node_modules/` não serão incluídas no controle de versão.

6. **License:** possibilita definir a licença do projeto, indicando como o código pode ser utilizado por terceiros. A escolha da *MIT License* é comum em projetos de código aberto por permitir ampla reutilização e modificação com poucas restrições.

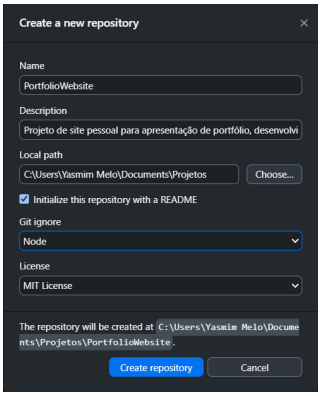


Figura 5.6. Tela criação de um novo repositório. Fonte: Próprio autor.

Após o preenchimento desses campos, basta selecionar **Create repository** para gerar o repositório local já configurado, pronto para receber *commits* e posteriormente ser publicado no GitHub.

5.5.2.2. Adição de arquivos ao repositório

Após a criação do repositório, o GitHub Desktop direciona o usuário para a tela principal do projeto. Nela, inicialmente, não há alterações registradas (*No local changes*), mas o sistema já oferece duas opções fundamentais de interação com o repositório recém-criado:

1. **Abrir o repositório em um editor externo:** O GitHub Desktop possibilita abrir o repositório diretamente em um editor de código, como o Visual Studio Code. Essa funcionalidade agiliza a edição dos arquivos do projeto, dispensando a necessidade de navegar manualmente até a pasta no sistema. O acesso pode ser feito por meio do botão **Open in Visual Studio Code**, disponível na interface principal.

2. **Visualizar os arquivos do repositório no explorador de arquivos:** Caso o usuário deseje acessar a pasta do projeto diretamente no sistema operacional, pode utilizar a

opção **Show in Explorer**. Essa funcionalidade abre a estrutura de diretórios onde o repositório foi criado, possibilitando a inclusão de novos arquivos ou a organização manual do projeto.

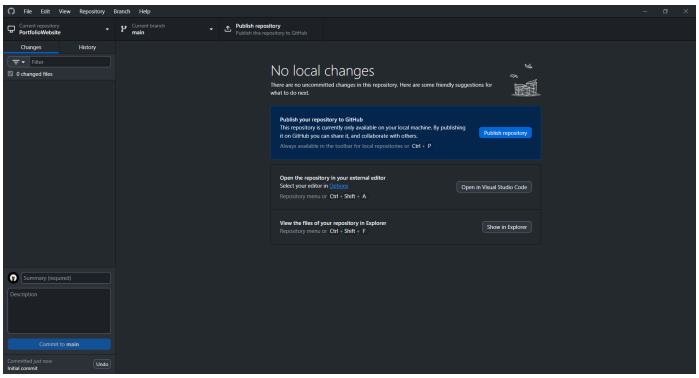


Figura 5.7. Tela inicial após a criação do repositório. Fonte: Próprio autor.

Uma vez que arquivos novos sejam adicionados à pasta do repositório ou que arquivos existentes sejam modificados, o GitHub Desktop detecta automaticamente essas mudanças. O usuário, então, poderá selecionar quais arquivos deseja incluir no próximo *commit*, garantindo que apenas as alterações relevantes sejam registradas no histórico do projeto.

5.5.2.3. Commits com mensagens claras

No GitHub Desktop, cada alteração registrada no repositório precisa ser acompanhada de uma mensagem de *commit*. Esse procedimento é essencial para manter o histórico do projeto organizado e facilitar a colaboração.

1. **Área de mudanças detectadas (Changes):** Sempre que um arquivo é adicionado, modificado ou removido no repositório, o GitHub Desktop exibe essas alterações na aba **Changes**. O usuário pode revisar cada modificação antes de confirmá-la.

2. **Seleção dos arquivos para o commit:** Apenas os arquivos marcados na lista de mudanças serão incluídos no *commit*. Isso permite que o desenvolvedor escolha apenas o que é relevante para registrar no histórico, evitando alterações desnecessárias.

3. **Campo Summary (mensagem obrigatória):** É o título do commit, que deve ser curto e direto. Resume a finalidade da alteração em uma única frase, como por exemplo:

```
feat: adicionar seção de contato
fix: corrigir erro no formulário de login
```

4. **Campo Description (mensagem opcional):** Permite adicionar uma explicação mais detalhada sobre o *commit*. É útil para descrever o contexto da mudança, o motivo da implementação ou observações para outros colaboradores.

5. **Realizar o commit:** Após preencher os campos, o usuário deve clicar no botão **Commit to branch**. A alteração será registrada no histórico local do repositório e ficará disponível para ser enviada ao GitHub posteriormente.

Mensagens claras e objetivas garantem que o histórico do projeto seja compreensível, facilitando futuras consultas, revisões e colaborações entre desenvolvedores.

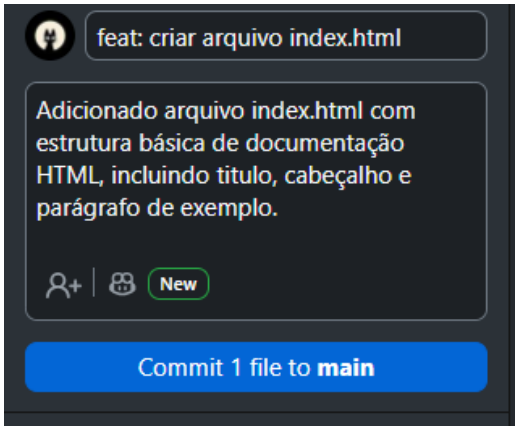


Figura 5.8. Exemplo de criação de commit no GitHub Desktop com mensagem clara e objetiva. Fonte: Próprio Autor.

5.5.3. Sincronização com o GitHub

Após realizar *commits* no repositório local, é necessário garantir que as alterações fiquem disponíveis também no repositório remoto no GitHub. Da mesma forma, ao trabalhar em equipe, é importante manter o repositório local sempre atualizado com as contribuições de outros colaboradores. O GitHub Desktop oferece recursos que facilitam esse processo de envio e recebimento de alterações.

1. **Publicação inicial do repositório:** Clique no botão **Publish repository** para criar automaticamente um repositório remoto no GitHub, vinculado ao seu repositório local.

- Antes de publicar, verifique se já existem *commits* locais, pois, caso contrário, o botão não terá alterações para enviar.
- É possível configurar a visibilidade do repositório como pública (visível a todos) ou privada (restrita).
- Após a publicação, o repositório já estará disponível online e pronto para ser compartilhado.

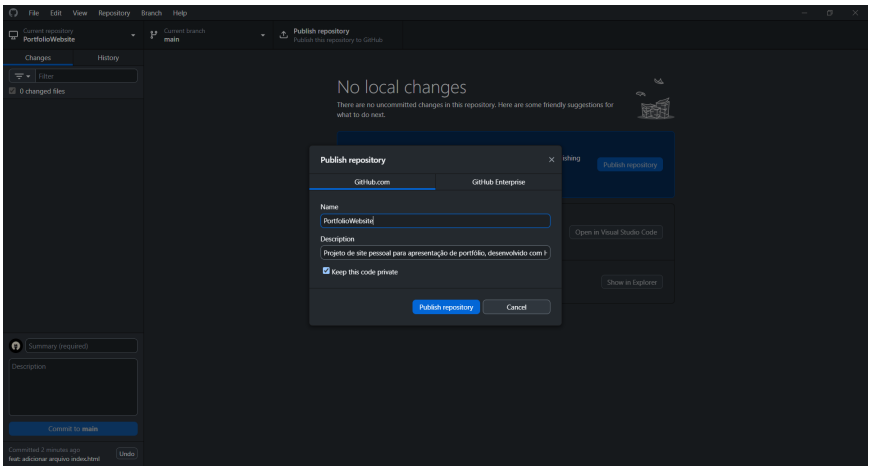


Figura 5.9. Tela para publicação do repositório. Fonte: Próprio autor.

2. **Envio de alterações (Push):** Sempre que novos commits forem criados no repositório local, utilize o botão **Push origin** para enviar essas mudanças ao GitHub, mantendo o repositório remoto atualizado.

3. **Verificação de atualizações (Fetch):** Para checar se há novas alterações feitas por outros colaboradores no repositório remoto, clique em **Fetch origin**.

4. **Baixar alterações remotas (Pull):** Caso sejam encontradas novidades, o botão mudará para **Pull origin**. Clique nele para baixar e aplicar as alterações no repositório local.

5. **Histórico e conflitos:** Após o *pull*, os novos commits aparecerão no histórico do GitHub Desktop. Caso haja conflitos entre alterações locais e remotas, será necessário resolvê-los antes de concluir a sincronização.

5.5.3.1. Clonagem de repositórios existentes

Projetos já hospedados no GitHub podem ser clonados facilmente pelo GitHub Desktop, utilizando a opção **File - Clone Repository**. Esse recurso cria uma cópia completa do

repositório remoto no computador, incluindo todos os arquivos, histórico de *commits e branches*, permitindo que o usuário contribua diretamente no projeto.

1. **Acessar a opção de clonagem:** No menu superior do GitHub Desktop, clique em *File - Clone Repository*. Uma nova janela será exibida para inserir os dados do repositório.

2. **Escolher a origem do repositório:** É possível selecionar um repositório disponível em sua conta do GitHub, em uma organização da qual participa ou inserir manualmente a *Uniform Resource Locator* (URL) de um repositório público.

3. **Definir o diretório local:** Escolha a pasta em seu computador onde deseja armazenar o repositório clonado. Essa será a versão local, totalmente vinculada ao repositório remoto.

4. **Concluir a clonagem:** Após confirmar, o GitHub Desktop fará o download de todos os arquivos e histórico. O projeto será aberto automaticamente na tela principal, pronto para receber *commits*, *pulls* e *pushes*.

5. **Começar a contribuir:** A partir desse momento, o usuário já pode editar arquivos, criar novas *branches* e enviar contribuições, com o GitHub Desktop cuidando da sincronização com o repositório remoto.

5.5.3.2. Inclusão de .gitignore e README.md

O GitHub Desktop permite que arquivos auxiliares, como *.gitignore* e *README.md*, sejam incluídos já no momento da criação do repositório ou adicionados posteriormente. Esses arquivos cumprem funções importantes: o *.gitignore* define quais arquivos e pastas devem ser ignorados pelo versionamento (como *logs*, dependências ou arquivos temporários), enquanto o *README.md* serve como documentação inicial do projeto, apresentando sua finalidade, instruções de uso e informações básicas para colaboradores.

1. **Durante a criação do repositório:** Ao selecionar *File - New Repository*, é possível marcar as opções para gerar automaticamente um arquivo *README.md* e um *.gitignore*. O *.gitignore* pode ser configurado a partir de modelos prontos, de acordo com a linguagem ou tecnologia utilizada no projeto (por exemplo, Node.js, Python, Java).

2. **Adição posterior dos arquivos:** Caso não sejam criados no início, esses arquivos podem ser adicionados manualmente no diretório local do projeto. O GitHub Desktop detectará as novas inclusões, permitindo que sejam versionadas em um *commit*.

3. **Função de cada arquivo:** O *.gitignore* garante que apenas arquivos relevantes sejam controlados pelo Git, evitando poluição do repositório com dados temporários ou específicos do ambiente do desenvolvedor. O *README.md* é exibido automaticamente na página inicial do repositório no GitHub, funcionando como cartão de visita do projeto e facilitando a compreensão por novos colaboradores.

4. **Benefícios:** A inclusão desses arquivos contribui para a organização do projeto, melhora a colaboração entre desenvolvedores e reduz problemas de versionamento.

5.5.3.3. Compreensão do histórico de mudanças

O GitHub Desktop apresenta um histórico visual de todos os *commits* realizados no repositório, permitindo que o usuário acompanhe a evolução do projeto de maneira clara e organizada. Esse recurso mostra, em ordem cronológica, cada alteração registrada, incluindo o autor, a data, a mensagem do *commit* e os arquivos modificados. Dessa forma, torna-se mais fácil realizar auditorias, revisões de código e identificar eventuais regressões que possam ter sido introduzidas.

1. **Acessar a aba de histórico:** Na tela principal do GitHub Desktop, ao lado da lista de alterações pendentes, encontra-se a aba *History*. Ali são exibidos todos os commits já registrados no repositório atual.

2. **Visualizar detalhes de cada commit:** Ao selecionar um *commit* no histórico, a interface mostra os arquivos alterados e, em alguns casos, até o conteúdo exato das modificações realizadas (adições e remoções). Isso facilita a análise de mudanças específicas, sem a necessidade de recorrer à linha de comando.

3. **Identificar a autoria e a data das mudanças:** Cada *commit* apresenta informações sobre quem o realizou e em que momento. Essa funcionalidade é fundamental em projetos colaborativos, pois permite rastrear responsabilidades e compreender o contexto das alterações.

4. **Facilitar auditorias e revisões:** Com a visualização cronológica, é possível auditar o progresso do projeto, revisar decisões tomadas em *commits* passados e até encontrar o ponto em que um problema foi introduzido no código.

5.5.4. Colaboração de Branches com o Github Desktop

Uma *branch* é como uma "linha paralela" do projeto. Ela permite que você trabalhe em algo novo (uma funcionalidade, correção de *bug* ou teste) sem mexer no código principal, que normalmente está na *branch main*. Pense nela como uma cópia do projeto em que você pode fazer alterações à vontade, sem risco de quebrar o que já funciona.

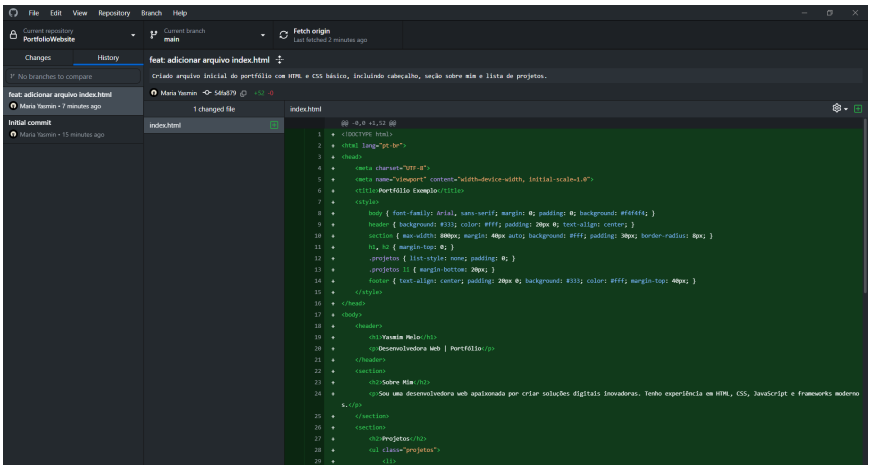


Figura 5.10. Exemplo da tela de históricos. Fonte: Próprio autor.

5.5.4.1. Criando uma branch no GitHub Desktop

1. Abra o GitHub Desktop e selecione o repositório em que quer trabalhar.

2. No topo, clique no botão "*Current Branch*" (ou "Branch atual").

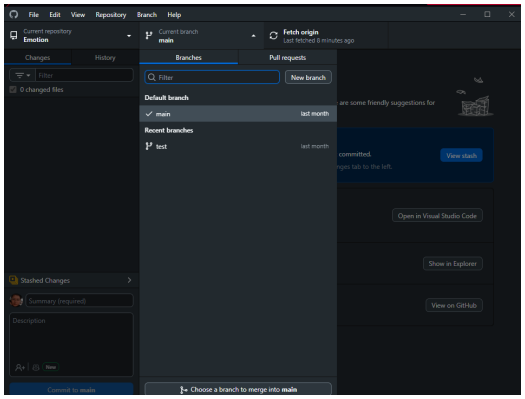


Figura 5.11. Exemplo da tela de Branch atual. Fonte: Próprio autor.

3. Clique em *"New Branch"* (ou "Nova branch").

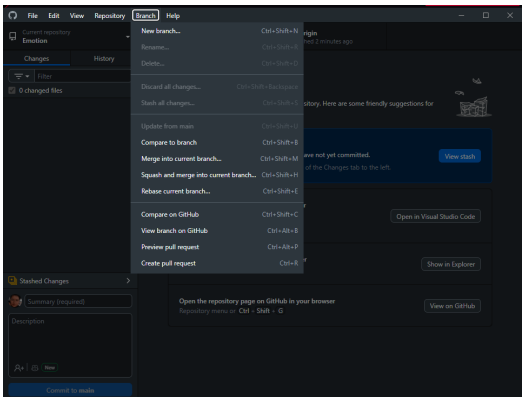


Figura 5.12. Exemplo da tela de nova *Branch*. Fonte: Próprio autor.

4. Dê um nome significativo para a *branch*, como correcao-bug-login ou nova-funcionalidade.

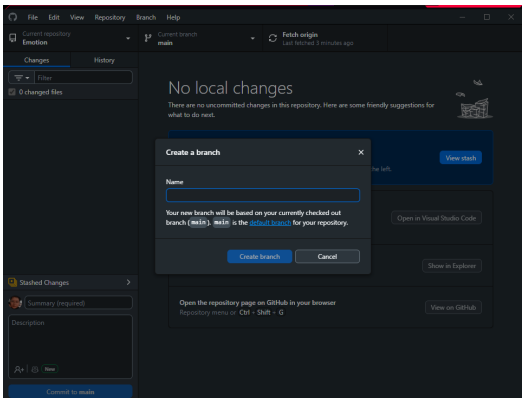


Figura 5.13. Exemplo da tela de colocar nome na *branch*. Fonte: Próprio autor.

5. Clique em *"Create Branch"* (Criar Branch).

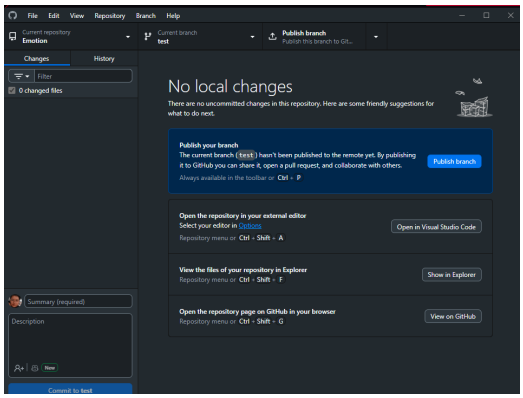


Figura 5.14. Exemplo da tela de *branch* nomeada. Fonte: Próprio autor.

6. O GitHub Desktop troca automaticamente para a nova *branch*. Tudo o que você alterar agora será registrado nela, e não na *branch* principal.

7. Clique em *"Current Branch"*.

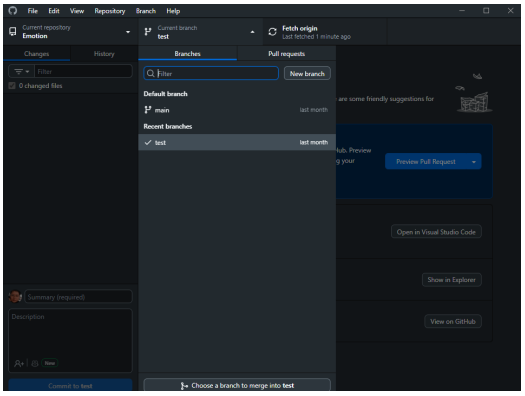


Figura 5.15. Exemplo da tela de lista das atuais *branches*. Fonte: Próprio autor.

8. Você verá a lista de todas as *branches* do repositório.

9. Clique na *branch* que deseja usar e o GitHub Desktop vai automaticamente mudar o foco para ela.

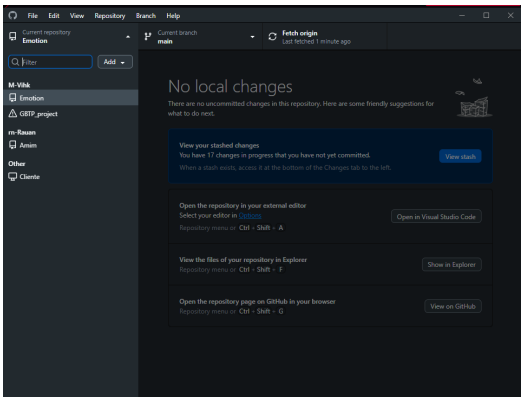


Figura 5.16. Exemplo da tela de seleção de repositório. Fonte: Próprio autor.

10. Agora, qualquer alteração será feita na *branch* selecionada.

5.5.4.2. O que é e como fazer o *merge*?

O *merge* é o processo de unir as alterações feitas em uma *branch* secundária (por exemplo, nova-funcionalidade) de volta para a *branch* principal (*main*). Isso garante que tudo o que você desenvolveu separadamente passe a fazer parte do código principal. Para realizarmos um *merge* no GitHub Desktop, serão desenvolvidos os seguintes passos:

1. No GitHub Desktop, clique em *"Current Branch"* e selecione a *branch* principal (*main*).

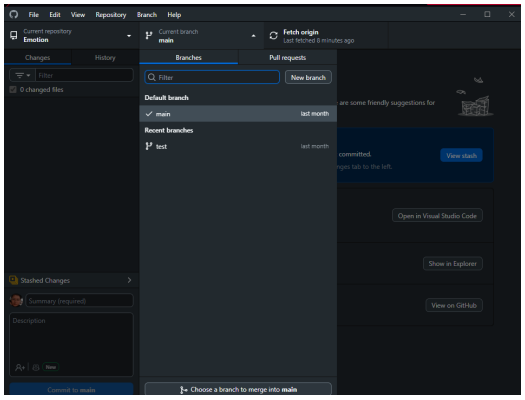


Figura 5.17. Exemplo da tela de *Branch* principal. Fonte: Próprio autor.

2. Vá no menu *Branch* (na parte superior) e clique em "Merge into Current Branch".

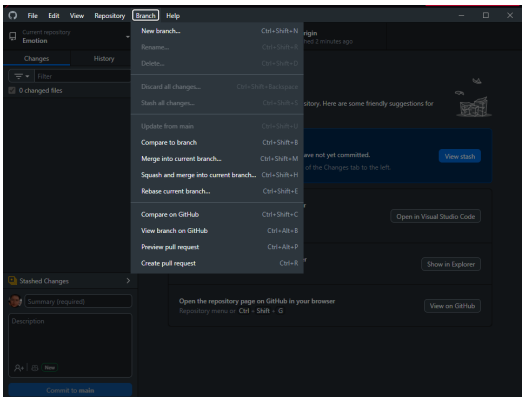


Figura 5.18. Exemplo da tela de *Branch Merge*. Fonte: Próprio autor.

3. Selecione a *branch* que contém as alterações (exemplo: nova-funcionalidade).

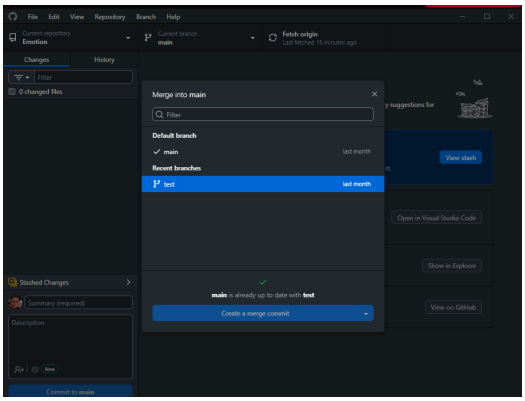


Figura 5.19. Exemplo da tela de *Branch* alterações adicionada a brach principal. Fonte: Próprio autor.

4. O GitHub Desktop vai tentar unir automaticamente as mudanças.

5.5.5. Como resolver conflitos simples e como o GitHub Desktop mostra o conflito

Um conflito de merge acontece quando duas *branches* alteram a mesma linha de um arquivo e o Git não consegue escolher qual versão manter. Nesses casos, o GitHub Desktop mostra uma mensagem de conflito e lista os arquivos afetados para que o usuário resolva manualmente. Já para sincronizar alterações, o processo envolve usar *Fetch origin* para verificar atualizações no repositório remoto, *Push origin* para baixá-las e *Push origin* para enviar seus *commits* locais, garantindo que todos os colaboradores trabalhem na versão mais atualizada [GitHub Docs; Coderefinery, 2025].

Um conflito acontece quando duas *branches* modificam a mesma linha de um arquivo ou mexem em partes que o Git não consegue decidir sozinho qual versão manter. Quando há conflito, o GitHub Desktop exibe uma mensagem como *"This branch has conflicts that must be resolved"*. Ele vai listar os arquivos problemáticos e você precisa abrir esses arquivos para resolver.

5.5.6. Sincronizar alterações com o GitHub

No GitHub Desktop, para manter seu repositório local alinhado com o remoto, você normalmente começa clicando em *"Fetch origin"*: isso verifica se há mudanças no GitHub que ainda não foram baixadas para sua máquina. Se existir algo novo, você pode então usar *"Pull origin"*, que baixa essas alterações e as incorpora ao seu repositório local [GitHub Docs; coderefinery.github.io].

Quando você faz modificações no seu computador, primeiro salva essas mudanças com um *commit*, depois usa *"Push origin"* para enviar aquelas alterações ao repositório no GitHub para que outros colaboradores também possam vê-las. Se existirem *commits* no repositório remoto que você ainda não possui localmente, o GitHub Desktop normalmente pede que você faça um *fetch* antes de permitir o *push*, para evitar conflitos ou divergências de histórico [GitHub Docs].

5.6. Vantagens e Desvantagens

Quanto às vantagens e desvantagens da utilização de ferramentas como o GitHub Desktop, apresentam-se a seguir as formas pelas quais podem ser benéficas ao usuário, bem como alguns pontos negativos observados em seu uso.

5.6.1. Vantagens

No tocante às vantagens em relação à interface de linha de comando, especialmente para o público-alvo menos familiarizado com o Git, destacam-se os seguintes benefícios:

1. Trabalho colaborativo.
2. Controle de conflitos.
3. Plataforma gratuita (até certo nível).

5.6.2. Desvantagens

Quanto às desvantagens encontradas nesta ferramenta, embora poucas, podem ser citadas algumas:

1. Curva de aprendizado inicial.
2. Dependência de internet (para uso do GitHub).
3. Conflitos podem ser complexos de resolver.

5.7. Boas Práticas no Uso do Git e GitHub

O uso de sistemas de controle de versão, como o Git, e de plataformas de hospedagem e colaboração, como o GitHub, é essencial no desenvolvimento de software moderno. Além de permitir o rastreamento das alterações no código, essas ferramentas fomentam a colaboração eficiente entre equipes e garantem maior qualidade no produto final. No entanto, para que se obtenha o máximo de benefícios, é necessário adotar boas práticas que padronizem o fluxo de trabalho e reduzam problemas futuros. A seguir, destacam-se algumas recomendações fundamentais.

5.7.1. Commits pequenos e frequentes

Um erro comum em projetos de software é acumular diversas modificações antes de registrar um *commits*. Essa prática dificulta a rastreabilidade e aumenta a probabilidade de conflitos. O ideal é realizar commits pequenos, que representem uma alteração coesa e independente, como a correção de um bug específico ou a implementação de uma funcionalidade pontual. *Commits* frequentes facilitam a revisão, permitem a reversão de mudanças sem comprometer grandes blocos de código e tornam o histórico do projeto mais legível.

5.7.2. Mensagens de commit claras

A mensagem associada a um commit deve ser descritiva e direta, informando o que foi alterado e, preferencialmente, o motivo. Mensagens vagas como “ajustes” ou “mudanças finais” não auxiliam na compreensão do histórico do projeto. Uma boa prática é utilizar um padrão, como:

- `fix:` para correções de erros.
- `feat:` para implementação de novas funcionalidades.
- `docs:` para alterações na documentação.

Esse tipo de padronização contribui para a manutenção futura do projeto e para a colaboração eficiente em equipe.

5.7.3. Organização do repositório

A estrutura do repositório reflete a maturidade e a profissionalização de um projeto. Pastas mal organizadas ou a ausência de documentação dificultam a contribuição de novos desenvolvedores e prejudicam a escalabilidade do software. Recomenda-se manter um arquivo `README.md` bem escrito, que apresente objetivos, instruções de instalação, dependências e exemplos de uso. Além disso, separar arquivos de código, testes, documentação e recursos auxiliares em diretórios distintos favorece a clareza e a manutenção do projeto.

5.7.4. Nomeação significativa de branches

Branches (ramificações) são fundamentais para o desenvolvimento paralelo de funcionalidades, correções e experimentos. No entanto, nomes genéricos como ”teste”ou ”nova”dificultam a compreensão de seu propósito. Recomenda-se adotar nomenclaturas padronizadas, como:

- `feature/login` para o desenvolvimento de uma nova funcionalidade.
- `bugfix/navbar` para a correção de um problema específico.
- `hotfix/security` para correções críticas e imediatas.

Essa prática organiza o fluxo de trabalho e facilita o gerenciamento do ciclo de vida das alterações.

5.7.5. Revisão de código antes do merge

O processo de *code review* consiste na revisão de código por outros membros da equipe antes da integração na *branch* principal, sendo um dos pilares da qualidade em projetos colaborativos. A revisão permite identificar erros, melhorar a legibilidade e compartilhar conhecimento entre os desenvolvedores. No GitHub, esse processo é facilitado por *pull requests*, que centralizam a discussão sobre uma alteração antes de sua incorporação definitiva. O resultado é um código mais robusto, padronizado e confiável.

5.8. Considerações finais

A adoção dessas boas práticas no uso do Git e do GitHub não apenas melhora a qualidade técnica do código, mas também fortalece a colaboração e a eficiência dentro das equipes de desenvolvimento. Em projetos científicos, acadêmicos ou corporativos, o rigor na aplicação dessas práticas contribui diretamente para a reprodutibilidade, a transparência e a longevidade das soluções desenvolvidas.

Referências

Atlassian. (n.d.). Comparing workflows. *Atlassian Git Tutorials*. https://www.atlassian.com/git/tutorials/comparing-workflows.

Coursera. (2023). What is Git?. *Coursera*. https://www.coursera.org/articles/what-is-git.

Chacon, S., & Straub, B. (2014). *Pro Git* (p. 456). Springer Nature.

Earth Data Science. (n.d.). Basic Git commands. *Earth Data Science Workshops*. https://earthdatascience.org/workshops/intro-version-control-git/basic-git-commands.

Encyclopaedia Britannica. (n.d.). GitHub. *In Britannica*. https://www.britannica.com/technology/GitHub.

Everhour. (n.d.). What is GitHub?. *Everhour Blog*. https://everhour.com/blog/what-is-github.

GitHub, Inc. (s.d.). *Documentação do GitHub Desktop*. Recuperado em 16 de julho de 2025, de https://docs.github.com/pt/desktop.

GitHub. (2025). Instalar o GitHub Desktop. *GitHub Docs*. Recuperado em 8 de setembro de 2025, de https://docs.github.com/pt/desktop/installing-and-authenticating-to-github-desktop/installing-github-desktop.

GitHub. (2025). Criar uma conta no GitHub. *GitHub Docs*. Recuperado em 8 de setembro de 2025, de https://docs.github.com/pt/get-started/start-your-journey/creating-an-account-on-github.

Git SCM. (n.d.). *Getting started: A short history of Git*. Git. https://git-scm.com/book/ms/v2/Getting-Started-A-Short-History-of-Git.

GitLab. (2025). Journey through Git's 20-year history. *GitLab Blog*. https://about.gitlab.com/blog/journey-through-gits-20-year-history.

GitHub. (2025). Git turns 20: A Q&A with Linus Torvalds. *GitLab Blog*. https://github.blog/open-source/git/git-turns-20-a-qa-with-linus-torvalds.

GitHub Docs. (n.d.). About pull requests. *GitHub*. https://docs.github.com/pt/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests.

GitHub Docs. (n.d.). About GitHub and Git. *GitHub*. https://docs.github.com/en/get-started/start-your-journey/about-github-and-git.

I Programmer. (2025). Linus on Git. *I Programmer*. https://www.i-programmer.info/news/82-heritage/17977-linus-on-git.html.

Mats, S. (2020). The story of GitHub: How a weekend hack became the world's code playground. *Medium*. https://stevemats.medium.com/the-story-of-github-how-a-weekend-hack-became-the-worlds-code-playground-928010893bb1.

PSL Models. (n.d.). History of GitHub. *Git Tutorial*. https://pslmodels.github.io/Git-Tutorial/content/background/GitHubHistory.html.