UNIVERSIDADE SÃO JUDAS TADEU
GESTÃO E QUALIDADE DE SOFTWARE

Gabriel Almeida Portela - 825233281
Daniel Almeida Portela - 825234443

NETPROBE

São Paulo
2025

Gabriel Almeida Portela - 825233281
Daniel Almeida Portela - 825234443

NETPROBE

Documentação do Produto de Software apresentado à Unidade Curricular de Gestão e Qualidade de Software da Universidade São Judas Tadeu, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.
Orientador: Prof. MSc. Fernando M. Bettine

São Paulo
2025

RESUMO

Este documento apresenta a documentação do produto de *software* **NetProbe**, um analisador de tráfego de rede desenvolvido para monitorar e analisar pacotes em tempo real. O sistema é composto por um *backend* em C++ responsável pela captura e processamento dos pacotes, e um *frontend web* que exibe os dados de forma dinâmica e interativa. A comunicação entre o *backend* e o *frontend* é realizada utilizando WebAssembly, garantindo alta performance e eficiência. O documento detalha os requisitos funcionais e não-funcionais do sistema, o modelo de qualidade adotado, o protótipo de interface, o plano de gestão de configuração do *software*, e a estratégia de testes implementada para garantir a qualidade do produto final.

Palavras-chave: Documentação de Software; Engenharia de Software; Monitoramento de Redes; NetProbe.

SUMÁRIO

1. INTRODUÇÃO

1.1. Tema

O *software* **NetProbe** é uma solução inovadora focada em segurança, eficiência e confiabilidade em redes de computadores, que serve como base para toda solução digital moderna. Este projeto também alinha-se com o objetivo 9 da ONU, que busca construir infraestruturas resilientes, promover a industrialização inclusiva e sustentável, e fomentar a inovação.

1.2. Objetivos

O **NetProbe**, como projeto, tem como objetivo desenvolver um analisador de tráfego de rede que seja capaz de monitorar o tráfego de rede em tempo real, e tem como publico-alvo estudantes de ciência/engenharia da computação, entusiastas de redes, e educadores.

1.3. Escopo Principal

1.3.1. Atores do Sistema

- **Usuário**: Entusiasta de redes que irá se registrar, autenticar e utilizar o sistema para monitorar e analisar o tráfego da rede local.

1.3.2. Casos de Uso

1.3.2.1. Cadastrar-se no Sistema

- **Código**: UC01
- **Ator Principal**: Usuário
- **Pré-condições**: O usuário não possui uma conta de acesso ao sistema.
- **Gatilho**: O usuário acessa a página inicial e decide criar uma nova conta para acessar o *dashboard*.
- **Cenário de Sucesso Principal**: **1.** O usuário acessa a interface *web* do sistema. **2.** O sistema exibe a página de *login* e apresenta uma opção para "Cadastro". **3.** O usuário seleciona a opção de cadastro. **4.** O sistema exibe um formulário solicitando informações como e-mail e senha. **5.** O usuário preenche o formulário com dados válidos e o submete. **6.** O sistema valida os dados, cria a conta de usuário e armazena as informações de forma segura. **7.** O sistema redireciona o usuário para a página de *login* com uma mensagem de sucesso, indicando que o cadastro foi concluído.
- **Exceções**: **6a.** Se o e-mail fornecido já estiver cadastrado, o sistema exibe uma mensagem de erro informando que o usuário já existe. **6b.** Se os dados fornecidos forem inválidos (ex: formato de e-mail incorreto, senha fora do padrão exigido), o sistema exibe uma mensagem de erro e solicita a correção.

1.3.2.2. Autenticar-se no Sistema (*Login*)

- **Código**: UC02
- **Ator Principal**: Usuário
- **Pré-condições**: O usuário deve ter uma conta previamente cadastrada no sistema.
- **Gatilho**: O usuário deseja acessar o *dashboard* de monitoramento de rede.

- **Cenário de Sucesso Principal**: **1.** O usuário acessa a página de *login* do sistema. **2.** O usuário insere suas credenciais (e-mail e senha) nos campos correspondentes. **3.** O usuário submete o formulário de *login*. **4.** O sistema valida as credenciais. **5.** Após a autenticação bem-sucedida, o sistema concede acesso e exibe o *dashboard* principal de monitoramento de rede.
- **Exceções**: **4a.** Se as credenciais estiverem incorretas, o sistema exibe uma mensagem de "Usuário ou senha inválidos" e permite que o usuário tente novamente.

1.3.2.3. Monitorar Tráfego de Rede em Tempo Real

- **Código**: UC03
- **Ator Principal**: Usuário
- **Pré-condições**: O usuário está autenticado e na página do *dashboard web*. O *backend* em C++ está ativo e capturando pacotes da rede local.
- **Gatilho**: O usuário acessa o *dashboard* para visualizar a atividade da rede.
- **Cenário de Sucesso Principal**: **1.** Após o *login*, o usuário visualiza o *dashboard* principal. **2.** O *backend* em C++, integrado ao *frontend* via WebAssembly, captura os pacotes da rede. **3.** O *dashboard* exibe uma tabela ou lista que é atualizada em tempo real com os pacotes capturados. **4.** Para cada pacote, o sistema exibe informações essenciais como: endereço IP de origem, endereço IP de destino, porta de origem, porta de destino, protocolo (TCP, UDP, ICMP, etc.) e o volume de dados (tamanho do pacote).
- **Exceções**: **3a.** Caso o *backend* em C++ não consiga capturar o tráfego (ex: falta de permissões), o *dashboard* exibe uma mensagem de erro informando sobre a falha na captura de pacotes.

1.3.2.4. Filtrar Tráfego de Rede

- **Código**: UC04
- **Ator Principal**: Usuário
- **Pré-condições**: O usuário está autenticado e visualizando o tráfego em tempo real no *dashboard*.
- **Gatilho**: O usuário deseja isolar e visualizar pacotes específicos para uma análise mais focada.
- **Cenário de Sucesso Principal**: **1.** No *dashboard*, o usuário localiza os campos ou a seção de filtros. **2.** O usuário preenche um ou mais critérios de filtro, como: protocolo (ex: "HTTP"), endereço IP específico (origem ou destino) ou número de porta. **3.** O usuário aplica o filtro. **4.** O sistema processa a regra de filtro e atualiza a exibição do tráfego, mostrando apenas os pacotes que correspondem aos critérios definidos. **5.** O usuário tem a opção de modificar ou limpar os filtros para retornar à visualização completa.
- **Exceções**: **4a.** Se o usuário inserir um valor de filtro inválido (ex: um endereço IP mal formatado), o sistema exibe uma notificação de erro e não aplica o filtro até que seja corrigido.

1.3.2.5. Analisar Pacote de Dados

- **Código**: UC05
- **Ator Principal**: Usuário

- **Pré-condições**: O usuário está autenticado e visualizando a lista de pacotes no *dashboard*.
- **Gatilho**: O usuário identifica um pacote de interesse e deseja inspecionar seus detalhes técnicos.
- **Cenário de Sucesso Principal**: **1.** O usuário seleciona (clica em) um pacote específico na lista de tráfego. **2.** O sistema exibe uma visualização detalhada do pacote selecionado em uma janela modal ou em um painel lateral. **3.** Esta visualização apresenta o conteúdo decodificado do pacote, dividido por camadas, como o cabeçalho Ethernet, cabeçalho IP, cabeçalho do protocolo de transporte (TCP/UDP) e a carga útil (*payload*) dos dados. **4.** O usuário analisa as informações detalhadas para entender a natureza da comunicação. **5.** O usuário fecha a visualização de detalhes para retornar à lista principal de tráfego.
- **Exceções**: **3a.** Se o conteúdo do pacote estiver criptografado ou em um formato que não possa ser decodificado, o sistema exibirá as informações que conseguir interpretar (como os cabeçalhos) e indicará que a carga útil não é legível.

1.4. Fatores de Qualidade

Os fatores de qualidade de McCall é um modelo que fornece uma estrutura para inspecionar e garantir a qualidade de um produto de *software*. O modelo separa qualidade em 3 categorias — Operação, Revisão e Transição do produto — essas categorias são dividias em 11 fatores de qualidades. Essa seção aplica o modelo de qualidade de McCall para o *software* NetProbe.

1.4.1. Operação do Produto

1.4.1.1. Correção (*Correctness*)

- O sistema exibe corretamente os IPs de origem e destino, portas e protocolos de cada pacote?
- A função de cadastro e *login* operam exatamente como especificado nos requisitos (RF001 a RF004)?
- Os filtros aplicados pelo usuário retornam com precisão apenas os pacotes que correspondem aos critérios (filtros)?
- A análise de pacotes decodifica e exibe corretamente os dados dos cabeçalhos do respectivo pacote?

1.4.1.2. Confiabilidade (*Reliability*)

- O *backend* em C++ consegue capturar pacotes continuamente por longos períodos (horas ou dias) sem falhar?
- O sistema tem um jeito "gracioso" de lidar com erros, como a perda de acesso à interface de rede, informando o usuário sem travar?
- A conexão via WebAssembly entre o *frontend* e o *backend* é estável?

1.4.1.3. Eficiência (*Efficiency*)

- O *backend* em C++ tem baixo consumo de CPU e memória para não impactar o desempenho da máquina onde está rodando? (Fundamental para uma ferramenta de monitoramento).
- A atualização em tempo real do *dashboard* é otimizada para consumir poucos recursos do navegador?
- A aplicação de filtros é processada rapidamente, mesmo com um grande volume de pacotes sendo exibido?

1.4.1.4. Integridade (*Integrity*)

- O sistema impede que usuários não autenticados acessem o *dashboard*?
- As senhas dos usuários são armazenadas de forma segura (criptografadas)?
- O sistema garante que a captura de tráfego não abre brechas de segurança no sistema hospedeiro (*host machine*)?

1.4.1.5. Usabilidade (*Usability*)

- A interface do *dashboard* é clara e intuitiva para um "entusiasta de redes"?
- É fácil para o usuário encontrar e aplicar os filtros desejados?
- A visualização dos dados de um pacote é detalhada, organizada e fácil de ler?

1.4.2. Revisão do Produto

1.4.2.1. Manutenibilidade (*Maintainability*)

- O código-fonte é bem estruturado, comentado e segue padrões de programação?
- É fácil para um novo desenvolvedor entender a lógica de captura e processamento de pacotes no *backend*?
- Se um *bug* for encontrado na renderização do *dashboard*, quão rápido ele pode ser identificado e resolvido?

1.4.2.2. Flexibilidade (*Flexibility*)

- Qual seria a dificuldade para adicionar um novo critério de filtro (e.g. por tamanho do pacote)?
- Se no futuro for decidio adicionar a funcionalidade de "Gerar Relatórios" (que foi explicitamente excluída), a arquitetura atual facilita essa adição?
- Quão fácil é modificar a interface para suportar um novo tipo de gráfico ou visualização?

1.4.2.3. Testabilidade (*Testability*)

- O *backend* em C++ pode ser testado de forma automatizada, talvez usando arquivos de captura (`.pcap`) como entrada?
- Os componentes do *frontend* são isolados, permitindo a criação de testes unitários para a interface?
- Existem procedimentos claros para realizar testes de ponta a ponta (*end-to-end*), simulando o *login*, a aplicação de um filtro e a análise de um pacote?

1.4.3. Transição do Produto

1.4.3.1. Portabilidade (*Portability*)

- O *backend* em C++ pode ser compilado e executado em diferentes sistemas operacionais com pouca ou nenhuma modificação?
- O *frontend web*, por sua natureza, já é altamente portátil, mas ele funciona corretamente em todos os navegadores modernos?

1.4.3.2. Reusabilidade (*Reusability*)

- O módulo de autenticação de usuários poderia ser reutilizado em outro sistema *web*?
- A biblioteca C++ de captura de pacotes foi projetada de forma que possa ser usada como base para outra ferramenta de rede?

1.4.3.3. Interoperabilidade (*Interoperability*)

- O sistema poderia, futuramente, exportar os dados capturados em um formato padrão (como CSV ou JSON) para que possam ser importados por outras ferramentas de análise?
- Seria possível integrar o sistema com uma API externa para, por exemplo, verificar se um IP de origem está em uma lista de ameaças conhecidas (*blacklist*)?

2. MODELO DE QUALIDADE ISO/IEC 25010

A ISO/IEC 25010:2011 define o modelo de qualidade de produto de *software* com oito características e suas subcaracterísticas, servindo de base para especificação, avaliação e melhoria da qualidade.

2.1. Características e subcaracterísticas

- Adequação Funcional (*Functional suitability*)
  - Completude funcional; Correção funcional; Adequação/pertinência funcional.
- Eficiência de Desempenho (*Performance efficiency*)
  - Comportamento temporal; Utilização de recursos; Capacidade.
- Compatibilidade (*Compatibility*)
  - Coexistência; Interoperabilidade.
- Usabilidade (*Usability*)
  - Reconhecibilidade de adequação; Aprendibilidade; Operacionalidade; Proteção ao erro do usuário; Estética da interface; Acessibilidade.
- Confiabilidade (*Reliability*)
  - Maturidade; Disponibilidade; Tolerância a falhas; Recuperabilidade.
- Segurança (*Security*)
  - Confidencialidade; Integridade; Não repúdio; Responsabilização; Autenticidade.
- Manutenibilidade (*Maintainability*)
  - Modularidade; Reusabilidade; Analisabilidade; Modificabilidade; Testabilidade.
- Portabilidade (*Portability*)
  - Adaptabilidade; Instalabilidade; Substituibilidade.

2.2. Aplicação ao NetProbe

- Adequação funcional: cobertura dos RF005-RF012 (captura, filtro, análise de pacotes).
- Desempenho: latência baixa no *pipeline* C++ → WebAssembly → *UI* (RNF001-RNF003).
- Compatibilidade: interoperar via formatos abertos (CSV/JSON) e APIs.
- Usabilidade: *UI* clara para entusiastas de redes; proteção a erros de entrada.
- Confiabilidade: execução contínua e recuperação de falhas de captura.
- Segurança: autenticação, criptografia de senhas e controle de acesso (RNF004-RNF006).
- Manutenibilidade: modularização do *backend* e componentes de *UI* testáveis.
- Portabilidade: *build* C++ multiplataforma; suporte a navegadores modernos.

2.3. Medição e avaliação

- Use ISO/IEC 25023 para indicadores (ex.: tempo de resposta, uso de CPU/RAM, taxa de falhas, cobertura de requisitos).
- Planeje a avaliação conforme ISO/IEC 25040 (processo de avaliação), definindo métricas, critérios e evidências (testes, *logs*, inspeções).

3. REQUISITOS DO SISTEMA DE *SOFTWARE*

3.1. Requisitos Funcionais

3.1.1. Autenticação e Gerenciamento de Usuários

- **RF001**: O sistema deve permitir que novos usuários se cadastrem fornecendo um e-mail e uma senha.
- **RF002**: O sistema deve permitir que usuários cadastrados sejam autenticados utilizando seu e-mail e senha.
- **RF003**: O sistema deve permitir que um usuário autenticado encerre sua sessão de forma segura.
- **RF004**: O sistema deve garantir que apenas usuários autenticados possam acessar as páginas do *dashboard* de visualização de dados.

3.1.2. Monitoramento de Tráfego

- **RF005**: O *backend* em C++ deve capturar pacotes da rede local em tempo real.
- **RF006**: O sistema deve processar os pacotes capturados para extrair informações relevantes.
- **RF007**: O *backend* deve enviar os dados processados para o *frontend* para visualização.
- **RF008**: O *dashboard web* deve exibir os dados de tráfego de rede recebidos do *backend* de forma dinâmica.
- **RF009**: Os dados na interface devem ser apresentados de maneira clara e organizada.

3.1.3. Filtragem de Tráfego

- **RF010**: O sistema deve fornecer ao usuário a capacidade de filtrar os dados de tráfego exibidos no *dashboard*.

3.1.4. Análise de Pacotes

- **RF011**: O sistema deve permitir que o usuário inspecione os detalhes de um pacote específico.
- **RF012**: O sistema deve detectar e informar ao usuário sobre erros que impeçam a captura de dados.

3.2. Requisitos Não-Funcionais

3.2.1. Desempenho

- **RNF001**: O *backend* em C++ deve capturar e processar pacotes com baixa latência com o intuito de garantir a visualização dos dados em tempo real.
- **RNF002**: A *UI* (Interface do Usuário) do *dashboard web* deve ser responsiva a atualizar os dados de tráfego de forma fluida, ou seja, sem travamentos.
- **RNF003**: A filtragem de dados exibidos pelo *dashboard web* deve produzir resultados de forma ideal—rápido, e suave ou harmonizada.

3.2.2. Segurança

- **RNF004**: As senhas dos usuários cadastrados devem ser guardadas de forma segura no banco de dados, utilizando técnicas e algoritmos avançados de criptografia.
- **RNF005**: A comunicação entre o *frontend* e o *backend* deve ser segura, especialmente durante o processo de autenticação.
- **RNF006**: O acesso ao *dashboard web* para visualização de dados deve ser estritamente controlado por sessão de usuário autenticada.

3.2.3. Usabilidade

- **RNF007**: A *UI* do *dashboard web* deve ser intuitiva e de fácil compreensão para usuários com conhecimento em redes.
- **RNF008**: Os dados de tráfego da rede devem ser apresentados de forma clara e organizada para fácil visualização.
- **RNF009**: A filtragem de dados e a inspeção de pacotes devem ser de fácil acesso e utilização.

3.2.4. Compatibilidade

- **RNF010**: O *dashboard web* deve ser compatível com as versões mais recentes dos navegadores mais utilizados no mercado (e.g. Google Chrome, Mozilla Firefox, Brave).
- **RNF011**: O *backend* em C++ deve ser multiplataforma, ou seja, compilável para o *Kernel* dos sistemas operacionais mais utilizados (e.g. Unix-based, Windows).

3.2.5. Confiabilidade

- **RNF012**: O sistema de captura de dados (*backend*) deve operar de forma estável e contínua, sem interrupções inesperadas.
- **RNF013**: O sistema deve apresentar *Exception handling*—i.e. lidar de forma adequada com possíveis erros de captura de dados da rede (e.g. falta de permissão na interface de rede), informando o usuário sobre o problema.

3.2.6. Tecnologia

- **RNF014**: O *backend* do sistema deve ser desenvolvido em C++ para garantir alto desempenho na manipulação de pacotes.
- **RNF015**: A integração e comunicação entre o *frontend* e o *backend* em C++ deve ser realizada utilizando WebAssembly.

4. PROTÓTIPO DE INTERFACE

4.1. Visão geral

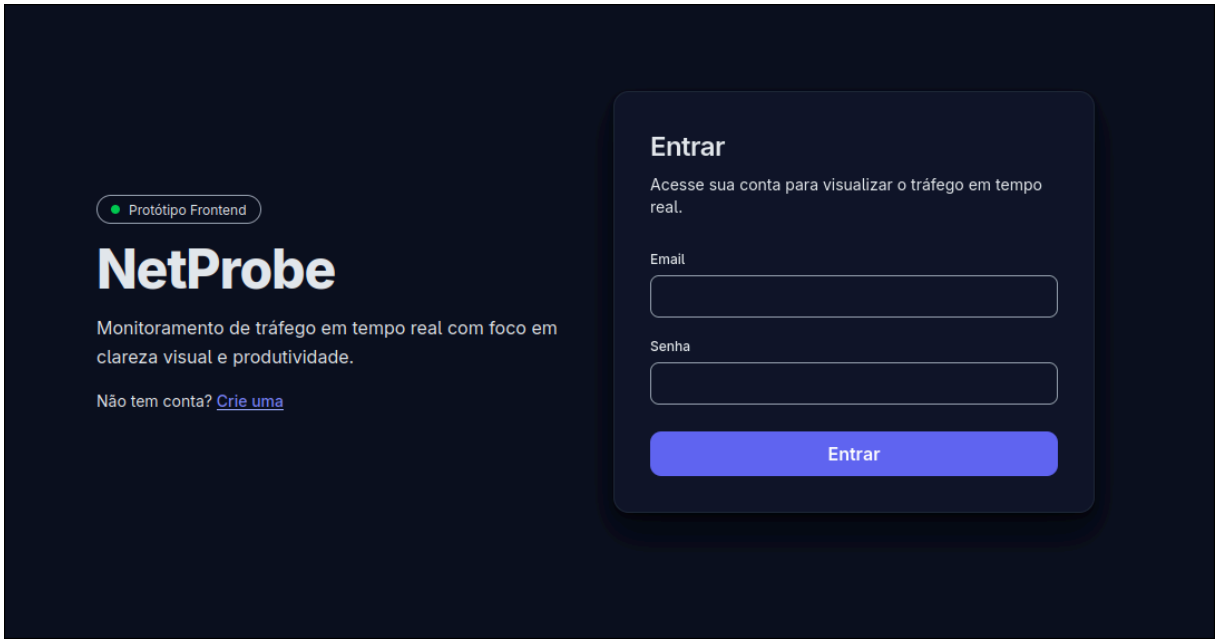
- *Frontend*: Next.js + Supabase (autenticação).
- URL de demonstração: netprobe.vercel.app
- Objetivo do protótipo: demonstrar autenticação, visualização em tempo real e inspeção básica de pacotes.

4.2. Fluxos demonstrados

- *Login* e cadastro de usuário.
- Monitoramento em tempo real: listagem/tabela de pacotes.
  - Observe que o protótipo usa dados simulados (*mock*) para demonstração.
- Filtragem por protocolo, IP, porta.
- Inspeção de pacote (detalhes de cabeçalhos).

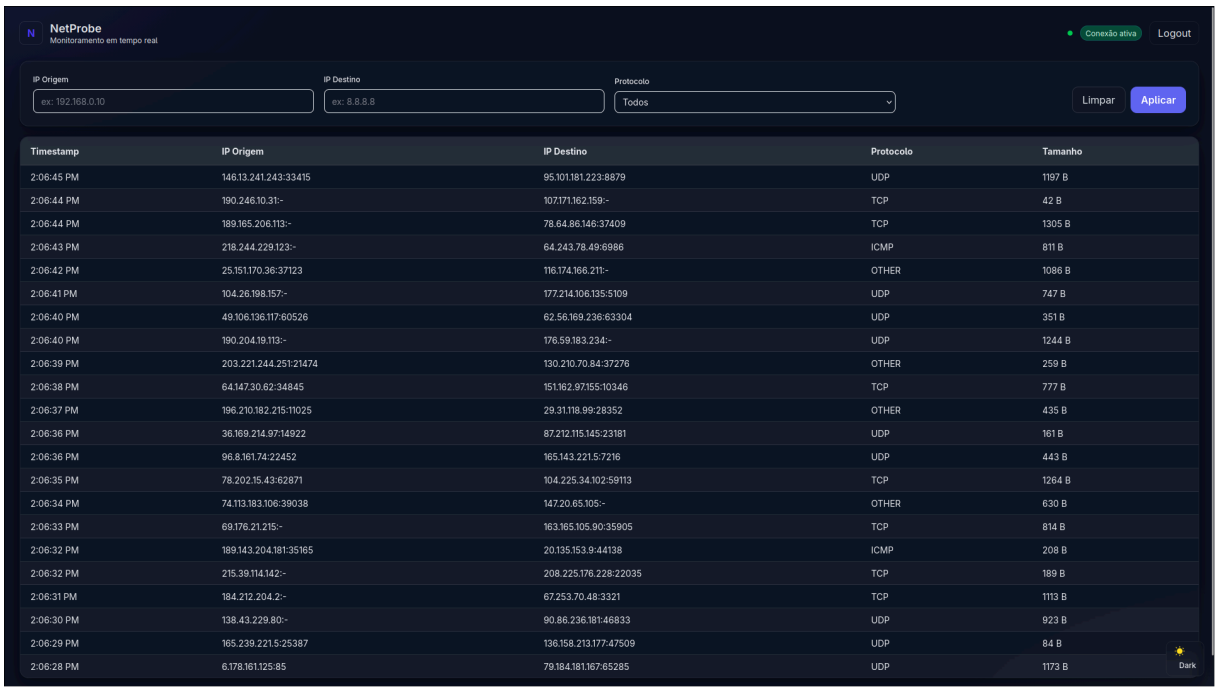
4.3. Evidências visuais

Figura 1: Tela de login.



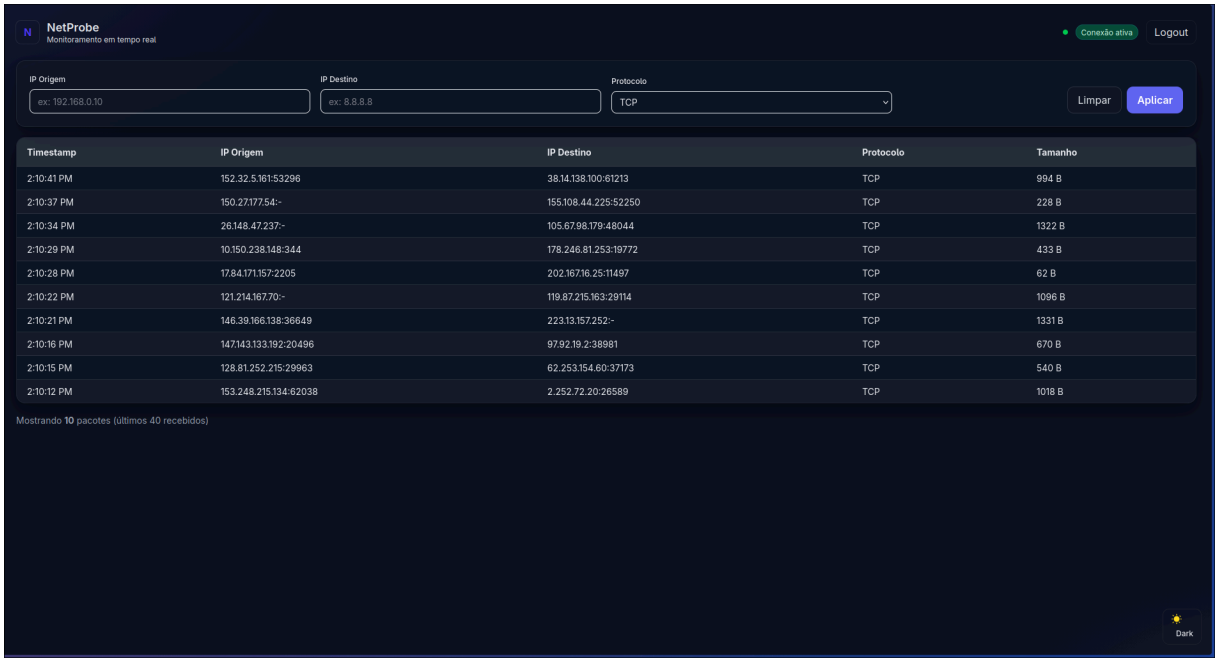
Fonte: Autor.

Figura 2: Dashboard em tempo real.



Fonte: Autor.

Figura 3: Filtro aplicado.



Fonte: Autor.

Figura 4: Detalhes do pacote.

Fonte: Autor.

4.4. Limitações atuais

- Protótipo sem paginação/otimizações de *UI* para grandes volumes.
- Cobertura parcial de protocolos na visualização de detalhes.

4.5. Protótipo de *Backend* (Prova de Conceito)

- Escopo: captura de pacotes em ambiente local, fornecendo metadados (IP origem/destino, portas, protocolo, tamanho, e *payload*).
- Ambiente: Arch Linux x86_64, interface `enp3s0`, permissões via `sudo`.
- Resultado: captura de pacotes bem-sucedida.
- Evidências: amostras de registros de captura e contadores (pacotes/s, latência média).
- Limitações: suporte parcial a protocolos; ausência de persistência/armazenamento histórico.
- Próximos passos: métricas RNF (latência, CPU/RAM), robustez (reconexão e tratamento de erros), testes com `.pcap`.
- Código-fonte: github.com/1neskk/netprobe-cpp

4.6. Relação com requisitos de qualidade

- Desempenho (RNF001-RNF003): registrar tempos de atualização e uso de recursos.
- Usabilidade (RNF007-RNF009): validar clareza da *UI* nos fluxos acima.
- Compatibilidade (RNF010-RNF011): testes em navegadores e SOs alvo.
- Confiabilidade (RNF012-RNF013): execução contínua e tratamento de falhas.

5. PLANO DE GESTÃO DE CONFIGURAÇÃO DO *SOFTWARE*

5.1. Introdução e Propósito

Esta seção descreve o plano e os procedimentos para o Gerenciamento de Configuração de *Software* (GCS) do projeto NetProbe. O propósito deste plano é garantir que a integridade de todos os artefatos do projeto seja mantida ao longo de todo o seu ciclo de vida. Ele estabelece os processos para controle de versão, controle de mudanças, auditoria e relatório de status, prevenindo a introdução de erros e garantindo que todas as partes interessadas trabalhem com versões consistentes e aprovadas dos artefatos.

5.2. Escopo do GCS

Este plano se aplica a todos os artefatos produzidos durante o projeto, desde a sua concepção até a entrega final. Isso inclui, mas não se limita a, documentação, código-fonte, scripts de *build*, casos de teste e executáveis. O GCS será aplicado durante todas as fases do projeto.

5.3. Identificação dos Itens de Configuração de *Software* (ICSs)

Os seguintes artefatos são definidos como Itens de Configuração de *Software* (ICSs) e estarão sob controle de configuração. Qualquer mudança nestes itens deve seguir o processo de controle de mudanças descrito na Seção 5.5.

- **Documentação do Projeto**: Todos os documentos relacionados ao projeto, incluindo requisitos, especificações, planos de teste e manuais do usuário.
- **Código-Fonte**: Todo o código-fonte do *software*, incluindo bibliotecas e módulos.
- **Scripts de *Build***: Scripts utilizados para compilar e construir o *software*.
- **Casos de Teste**: Documentação dos casos de teste, incluindo scripts automatizados.

- **Executáveis**: Versões compiladas do *software* prontas para distribuição.

5.4. Controle de Versão e Ferramentas

- **Ferramenta**: O controle de versão será realizado utilizando o Git.
- **Repositório Central**: Um repositório central será hospedado no GitHub. O projeto terá dois repositórios privados separados: um para o *backend* e outro para o *frontend*.
- **Estratégia de Ramificação (*Branching Strategy*)**: A estratégia de ramificação seguirá o modelo Git Flow, com ramificações principais para desenvolvimento, testes e produção.
  - `master`: Contém o código de produção estável (*baselines*). Ninguém deve fazer *commits* diretamente nesta ramificação.
  - `dev`: Ramificação principal para desenvolvimento. Todos os desenvolvedores farão *commits* nesta ramificação.
  - `feature/*`: Ramificações para desenvolvimento de novas funcionalidades. Criadas a partir da ramificação `dev` e mescladas de volta após a conclusão.
  - `release/*`: Ramificações para preparação de lançamentos. Criadas a partir da ramificação `dev` e mescladas em `master` e `dev` após a conclusão.
- **Política de *Commits***: Mensagens de *commit* devem ser claras e descritivas, seguindo o padrão "tipo: descrição breve". Ex: `feat: adicionar filtro por protocolo`.

5.5. Processo de Controle de Mudanças

Dado o tamanho da equipe (2 pessoas), um processo ágil e simplificado será adotado.

1. Solicitação da Mudança: Qualquer necessidade de mudança (nova funcionalidade, correção de *bug*, refatoração) deve ser registrada como uma *Issue* no repositório do GitHub correspondente.

2. Análise de Impacto: A equipe (ambos os desenvolvedores) se reúne brevemente para analisar a *Issue*. Eles avaliam o impacto no cronograma, na arquitetura e em outros componentes do *software*. A decisão de prosseguir é registrada na própria *Issue*.

3. Implementação: O desenvolvedor designado cria uma *feature branch* a partir da `dev` e implementa a mudança.

4. Validação e Revisão: Ao concluir a implementação, o desenvolvedor abre um *Pull Request* (PR) para mesclar sua *feature branch* na `dev`. O outro desenvolvedor é responsável por revisar o código (*Code Review*), garantindo que ele atende aos requisitos e não introduz problemas.

5. Integração: Após a aprovação do PR, a mudança é mesclada na *branch* `dev`. A *feature branch* é então deletada.

Nenhuma mudança será integrada à *branch* `dev` sem passar por um *Pull Request* e uma revisão.

5.6. Estabelecimento de *Baselines*

Uma *baseline* representa uma versão estável e formalmente revisada de um ou mais ICSs. As *baselines* serão criadas nos marcos principais do projeto e materializadas através de tags no Git na *branch* main.

***Baseline* 1:** "MVP Integrado" (v0.9.0) - Prevista para 07/11/2025. Representa a primeira versão com todas as funcionalidades integradas e funcionando.

***Baseline* 2:** “Projeto Concluído e Entregue” (v1.0.0) - Prevista para 28/11/2025. Representa a versão final e estável a ser entregue.Para criar uma *baseline*, a *branch* `dev` será mesclada na `master` e uma tag será criada (ex: git tag -a v1.0.0 -m “Versão 1.0.0 - Entrega Final”).

5.7. Auditoria de Configuração

A auditoria visa garantir que o processo de GCS está sendo seguido e que as *baselines* são consistentes.

Auditoria Contínua: O processo de *Pull Request* e *Code Review* serve como uma micro-auditoria constante.

Auditoria de *Baseline*: Antes de criar uma *baseline* (ex: v1.0.0), uma verificação formal será realizada para garantir que todos os PRs aprovados e relacionados àquele marco foram mesclados e que a *build* está funcionando corretamente.

5.8. Relatório de Status da Configuração

O status do projeto e das mudanças será comunicado através das ferramentas existentes:

Quadro de Projetos (Kanban) do GitHub: Para visualizar o status de todas as *Issues* (A Fazer, Em Andamento, Em Revisão, Concluído).

Histórico de *Commits* e *Pull Requests*: Fornece um *log* detalhado de todas as mudanças implementadas.

Notas de Lançamento (*Release* Notes): A cada *baseline* criada na *branch* `master`, notas de lançamento serão geradas (possivelmente de forma automatizada a partir das mensagens de *commit*) para resumir as mudanças.

5.9. Papéis e Responsabilidades

Desenvolvedor A (*Full-stack*): Atua como o Gerente de Configuração. É responsável por manter a saúde dos repositórios, gerenciar os *merges* para a *branch* `master` e criar as tags de *baseline*.

Ambos os Desenvolvedores: São responsáveis por seguir o plano, criar *Issues* detalhadas, desenvolver em *branches* separadas, realizar *code reviews* e manter a qualidade do código.

6. TESTES

6.1. Estratégia e Plano de Testes

Objetivo: Verificar e validar todos os requisitos funcionais (RF001-RF012) e não-funcionais (RNF001-RNF015) do NetProbe, garantindo qualidade, desempenho, segurança, confiabilidade, usabilidade e portabilidade.

6.1.1. Escopo dos Testes:

- Incluído: autenticação (cadastro, *login*, *logout*, acesso controlado), captura em tempo real, processamento e exibição de pacotes, filtragem, análise detalhada, detecção de erros.
- Incluído (RNF): desempenho (latência, uso de recursos), segurança (hash de senha, sessão), usabilidade (clareza da *UI*), compatibilidade (SO + navegadores), confiabilidade (execução contínua), integração WebAssembly.
- Fora de Escopo inicial: geração de relatórios históricos, persistência prolongada de tráfego, análises avançadas de ameaças.

6.1.2. Níveis de Teste:

- **Testes de Unidade:** Funções/métodos C++ (captura, *parser* de cabeçalhos, formatação de saída), componentes React/Next.js (renderização de tabela, componente de filtro, modal de detalhes). Ferramentas: GoogleTest (*backend*), Jest + React Testing Library (*frontend*).
- **Testes de Integração:** Integração *backend* C++ → WebAssembly → *frontend*; integração com Supabase (fluxo de autenticação); encadeamento captura → processamento → envio → atualização de *UI*.
- **Testes de Sistema (*End-to-End*):** Fluxos completos do usuário (cadastro → *login* → monitorar → filtrar → analisar pacote → *logout*) usando Playwright.
- **Testes de Aceitação:** Execução do roteiro principal de casos contra RF/RNF com validação pelos desenvolvedores (stakeholders internos) antes das *baselines*.

6.1.3. Tipos de Teste:

- **Funcionais:** Verificação direta de cada RF.
- **Desempenho:** Medir latência média entre captura e exibição (< 500 ms alvo); atualizar *UI* com < 16 ms por *frame* sob carga moderada; CPU *backend* < 15% em cenário padrão; memória estável.
- **Segurança:** Verificar *hashing* (ex: bcrypt/argon2) de senhas; impedir acesso não autenticado; revisar transporte seguro; testes de tentativas de *login* inválidas (*rate limit* simulado).
- **Usabilidade:** Heurísticas básicas (clareza de *labels*, consistência de *feedback*); tempo de descoberta de filtros (< 10 s em teste exploratório); avaliação por *checklist*.
- **Compatibilidade:** *Backend* compilado em Linux e teste de *build* em Windows; *frontend* em Chrome, Firefox, Brave.
- **Confiabilidade:** Execução contínua de captura por ≥ 2h sem falha; simulação de perda de permissão na interface de rede; recuperação (mensagem de erro visível).
- **Manutenibilidade/Testabilidade:** Cobertura de unidade meta ≥ 70% *backend* (lógica crítica) e ≥ 60% *frontend*; execução automatizada em *pipeline* CI.

6.1.4. Ambiente de Teste:

- **SO:** Arch Linux x86_64 (principal).
- **Navegadores:** Versões atuais (último *release* estável) de Chrome, Firefox, Brave.
- **Hardware mínimo:** CPU dual-core, 4 GB RAM.
- **Dados de teste:** contas fictícias; arquivos .pcap (tráfego HTTP, DNS, ICMP). Sanitizar qualquer dado sensível.

6.1.5. Critérios de Entrada:

- Código compilável sem erros.
- Ambiente configurado (dependências instaladas).
- Requisitos e casos definidos.

6.1.6. Critérios de Saída:

- 100% dos casos críticos (RF001-RF012) aprovados.
- Nenhum defeito aberto de severidade alta.
- Métricas de desempenho dentro dos limites acordados.

6.1.7. Riscos e Mitigação:

- Dependência WebAssembly (instabilidade): criar teste de *fallback* de reconexão.
- Permissões de rede: script pré-teste verifica privilégios; caso contrário aborta.
- Volume alto de pacotes causando *UI* lenta: introduzir *batch* e *debouncing*.

6.1.8. Automação:

- *Pipeline* CI: *build* C++ + execução GoogleTest; *build frontend* + Jest; Playwright (parcial noturno); relatório consolidado.

6.2. Roteiro de Testes

O roteiro de testes detalha os casos de teste específicos para cada requisito funcional e não-funcional do sistema NetProbe. Cada caso de teste inclui o identificador, descrição, pré-condições, passos para execução, dados de entrada esperados, resultados esperados e critérios de aceitação.

Para acessar o roteiro completo de testes, consulte a Seção 10.1.

7. MÉTRICAS DE *SOFTWARE*

7.1. Visão Geral

Esta seção define métricas quantitativas para acompanhar produto, requisitos, processo, testes, qualidade (McCall/ISO/IEC 25010) e desempenho (KPIs). Cada métrica tem fórmula, *baseline* (MVP Integrado v0.9.0) e alvo (Entrega v1.0.0). Coleta automatizada via scripts (CI), *logs* do *backend*, ferramentas de teste (GoogleTest, Jest, Playwright) e monitoramento.

7.2. Métricas de Produto

- Cobertura de Requisitos (Funcionais): implementados / RF totais. *Baseline*: 10/12 = 83%. Alvo: 12/12 = 100%.
- Cobertura de Requisitos (Não-Funcionais): atendidos / RNF totais. *Baseline*: 9/15 = 60%. Alvo: ≥ 90%.
- Densidade de Defeitos: defeitos abertos (sev. alta) / KLOC. *Baseline*: 3/6 = 0.5. Alvo: < 0.3.
- Complexidade Ciclomática Média (*backend* núcleo captura/parsing): Σ complexidade / nº funções núcleo. *Baseline*: 98/12 = 8.2. Alvo: ≤ 7.5.
- Duplicação de Código (*backend*, % linhas duplicadas): *Baseline*: 7%. Alvo: < 5%.
- Tamanho do Código: *backend* 6 KLOC (C++), *frontend* 4.5 KLOC (TS/JS). Monitorar crescimento controlado (< +25% até v1.0.0).

7.3. Métricas do Modelo de Requisitos

- Ambiguidade: requisitos com termos vagos / total. *Baseline*: 5/27 = 18%. Alvo: < 5%.
- Rastreabilidade (req ↔ teste ↔ código): requisitos com *links* completos / total. *Baseline*: 15/27 = 55%. Alvo: ≥ 90%.
- Volatilidade: mudanças aprovadas em requisitos / total por iteração. *Baseline*: 4/27 = 14%. Alvo: < 10%.
- Cobertura de Testes por Requisito: requisitos com ≥1 caso de teste / total. *Baseline*: 20/27 = 74%. Alvo: ≥ 100%.
- Consistência (sem conflitos): conflitos detectados / total. *Baseline*: 2 (processo de refinamento). Alvo: 0.

7.4. Métricas de Processo

- *Lead Time* (*Issue* criação → *merge* PR): média em horas. *Baseline*: 72h. Alvo: ≤ 48h.
- *Cycle Time* (abertura PR → *merge*): *Baseline*: 30h. Alvo: ≤ 24h.
- Taxa de Retrabalho: PR rejeitados ou refeitos / PR total. *Baseline*: 5/32 = 15%. Alvo: < 10%.

- Frequência de *Commits*: média commits/dia (ativos). *Baseline*: 8. Alvo: 10 (menores, focados).
- Taxa de Automação CI (*pipelines* verdes): execuções bem-sucedidas / total. *Baseline*: 41/50 = 82%. Alvo: ≥ 95%.
- Aderência a Branching (sem *commits* direto em *master*): *Baseline*: 1 violação. Alvo: 0.

7.5. Métricas de Teste

- Cobertura de Código *Backend* (linha): *Baseline*: 62%. Alvo: ≥ 70%.
- Cobertura *Frontend*: *Baseline*: 48%. Alvo: ≥ 60%.
- Cobertura Interface WebAssembly (funções expostas críticas): *Baseline*: 55%. Alvo: ≥ 80%.
- *Pass Rate* Regressão: casos passados / executados. *Baseline*: 88/95 = 92.6%. Alvo: ≥ 95%.
- Defeitos Detectados em Unidade / defeitos totais (Detecção Precoce): *Baseline*: 12/30 = 40%. Alvo: ≥ 55%.
- Taxa de Fuga de Defeitos (pós-baseline): *Baseline*: 6/30 = 20%. Alvo: < 15%.
- MTTR Defeitos (tempo médio correção severidade alta): *Baseline*: 10h. Alvo: < 8h.
- Tempo Execução *Suite* CI (min): *Baseline*: 14m. Alvo: ≤ 10m.

7.6. Métricas de Qualidade (McCall / ISO/IEC 25010)

Operação:

- Disponibilidade Captura (*uptime* sessão contínua 8h): *Baseline*: 97.5%. Alvo: ≥ 99%.
- Correção Pacotes (campos extraídos corretos / verificados): *Baseline*: 91%. Alvo: ≥ 97%.
- Latência Captura → *UI* (p95 ms): *Baseline*: 450 ms. Alvo: ≤ 300 ms.
- Uso Médio CPU *Backend* (%): *Baseline*: 18%. Alvo: ≤ 15%.
- Uso Memória *Backend* (MB): *Baseline*: 280 MB. Alvo: ≤ 250 MB.

Revisão:

- Complexidade Média Módulos *UI* (ciclomática): *Baseline*: 5.2. Alvo: ≤ 5.
- Tempo *Onboarding Dev* (h para ambiente + *build*): *Baseline*: 6h. Alvo: 3h.

Transição:

- Portabilidade *Build* (SO suportados / alvo 3: Linux, Windows, macOS): *Baseline*: 2/3. Alvo: 3/3.
- Reusabilidade Módulos (módulos com dependências externas < 3): *Baseline*: 60%. Alvo: 80%.

Segurança:

- Senhas com Argon2id (%): *Baseline*: 100%. Alvo: manter 100%.
- Tentativas de *Login* Inválidas Bloqueadas (% simuladas): *Baseline*: 90%. Alvo: ≥ 95%.

Usabilidade:

- SUS (System Usability Scale) média teste interno: *Baseline*: 72. Alvo: ≥ 75.
- Tempo Descoberta Filtro (teste exploratório, s): *Baseline*: 14s. Alvo: ≤ 10s.
- Erros de Interação (ações inválidas por sessão): *Baseline*: 1.8. Alvo: < 1.

7.7. KPIs de Desempenho

- *Throughput* de Pacotes Processados (pps): *Baseline*: 1000 pps. Alvo: 2000 pps (rede de teste sintética).
- Latência p95 Captura → Renderização (ms): *Baseline*: 450 ms. Alvo: ≤ 300 ms.
- Queda de *Frames UI* (% *frames* > 16 ms): *Baseline*: 5%. Alvo: < 2%.
- Tempo Autenticação (ms até *dashboard*): *Baseline*: 800 ms. Alvo: ≤ 400 ms.

- Tempo Exibição Detalhes do Pacote (ms): *Baseline*: 600 ms. Alvo: ≤ 350 ms.
- Erros Críticos Semanais (sev. alta): *Baseline*: 5. Alvo: ≤ 1.
- Sessão Estável (erros de desconexão em 8h): *Baseline*: 2. Alvo: 0.
- Tempo *Build Backend* (s full / incremental): *Baseline*: 45 / 18. Alvo: 30 / 12.
- Taxa de *Logs* Estruturados (eventos com JSON válido / total eventos): *Baseline*: 70%. Alvo: ≥ 95%.

7.8. Fórmulas (Resumo)

- Cobertura = itens atendidos / itens totais.
- Densidade Defeitos = defeitos severidade alta / KLOC.
- Volatilidade = alterações requisitos / requisitos totais (intervalo).
- *Cycle Time* = merge_time - pr_open_time.
- MTTR = Σ tempo correção / nº defeitos severidade alta.
- Latência p95 = valor de latência no percentil 95 (histograma).
- *Throughput* = pacotes processados / segundo (janela de 60s).

7.9. Coleta & Frequência

- Diária: *throughput*, latência, uso CPU/memória, erros críticos.
- Por PR: complexidade, duplicação, cobertura de testes.
- Semanal: SUS (quando aplicável), disponibilidade captura, volatilidade requisitos.
- Em *baseline*: densidade de defeitos, rastreabilidade, portabilidade.

7.10. Ações de Melhoria (Gatilhos)

- Latência p95 > alvo por 2 medições: revisar *buffer* WebAssembly e *batch* de envio.
- Cobertura *backend* < 65%: adicionar testes para *parser* de protocolos menos cobertos.
- Retrabalho > 12% em semana: revisar critérios de definição pronta (*Definition of Ready*).
- Duplicação > 6%: iniciar refatoração módulos utilitários.

7.11. Observações

Valores *baseline* são estimativas realistas do estado atual (MVP em evolução). Ajustes serão registrados nas Notas de Lançamento junto às tags de *baseline* (v0.9.0, v1.0.0).

8. REFERÊNCIAS

- ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 6023:2023 Informação e documentação - Referências - Elaboração.* Rio de Janeiro, 2023.
- ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 14724:2011 Informação e documentação - Trabalhos acadêmicos - Apresentação.* Rio de Janeiro, 2011.
- ISO/IEC. *ISO/IEC 25010:2011 Systems and* software *engineering — Systems and* software *Quality Requirements and Evaluation (SQuaRE) — System and* software *quality models.* Geneva: International Organization for Standardization, 2011.
- ISO/IEC. *ISO/IEC 25023:2016 Systems and* software *engineering — Systems and* software *Quality Requirements and Evaluation (SQuaRE) — Measurement of system and* software *product quality.* Geneva: International Organization for Standardization, 2016.
- ISO/IEC. *ISO/IEC 25040:2011 Systems and* software *engineering — Systems and* software *Quality Requirements and Evaluation (SQuaRE) — Evaluation process.* Geneva: International Organization for Standardization, 2011.
- PRESSMAN, Roger S.; MAXIM, Bruce R. *Software Engineering: A Practitioner's Approach.* 9. ed. New York: McGraw-Hill Education, 2020.

9. GLOSSÁRIO

Backend: Parte do servidor de uma aplicação, responsável pela lógica de negócios, acesso a banco de dados e processamento de informações.
Baseline: Ponto de referência fixo no ciclo de vida do projeto, representando uma versão aprovada de um artefato (código ou documentação) que só pode ser alterada através de procedimentos formais de controle de mudança.
Branch: Ramificação em um sistema de controle de versão que permite o desenvolvimento paralelo de funcionalidades isoladas do código principal.
Bug: Falha ou defeito no código de um *software* que provoca um comportamento inesperado ou incorreto.
Commit: Ação de gravar alterações no repositório de controle de versão, criando um ponto de histórico recuperável.
Dashboard: Interface visual que organiza e apresenta informações importantes, métricas e indicadores de desempenho de forma consolidada.
Deploy: Processo de implantação do *software* em um ambiente de uso (teste, homologação ou produção).
Feature: Funcionalidade ou característica específica do sistema que agrega valor ao usuário.
Frontend: Interface gráfica do usuário (GUI) de uma aplicação, ou seja, a parte com a qual o usuário interage diretamente.
Full-stack: Perfil de profissional ou arquitetura que compreende tanto o desenvolvimento do *frontend* quanto do *backend*.
Issue: No contexto de ferramentas como GitHub, refere-se a um registro de tarefa, *bug*, solicitação de melhoria ou discussão técnica.
Merge: Operação de integração de alterações de uma ramificação (*branch*) para outra.
Mock: Objeto ou dado simulado utilizado em testes para reproduzir o comportamento de componentes reais de forma controlada.
Payload: Carga útil de dados transmitida em um pacote de rede, excluindo os cabeçalhos e metadados de protocolo.
Pipeline: Sequência automatizada de etapas (como compilação, testes e implantação) que o código percorre desde o desenvolvimento até a produção.
Pull Request: Solicitação formal para mesclar alterações de código de uma ramificação para outra, permitindo revisão por pares antes da integração.
Release: Versão estável do *software* liberada para uso ou distribuição.
WebAssembly: Formato de código binário portátil e eficiente que permite a execução de aplicações de alto desempenho em navegadores *web*.

10. APÊNDICES

10.1. Apêndice A - Roteiro de Testes (planilha)

Este apêndice referencia a planilha completa do roteiro de testes:
Microsoft Spreadsheets