## **Relatório Final de Testes: Sistema de Saúde Fetal**

Este relatório detalha a estratégia de testes implementada para o Sistema de Saúde Fetal, os casos de teste planejados, as evidências de execução esperadas, a análise dos resultados e sugestões para melhorias futuras. O objetivo é fornecer uma visão abrangente da qualidade e funcionalidade da aplicação em suas diversas camadas.

### **1. Estratégia de Testes**

A estratégia de testes foi concebida em três pilares fundamentais para garantir a robustez e a funcionalidade do Sistema de Saúde Fetal:

* **Testes Unitários**: Focados na validação de componentes e unidades de código em isolamento. O objetivo é verificar que cada função, método ou componente individual se comporta conforme o esperado, garantindo a correção da lógica de negócio e das operações básicas.
  + **Módulo Python (ML Service)**: Utiliza pytest para testar o carregamento do modelo, a lógica de inferência e o tratamento de erros na API de predição. [cite: test\_model\_service.py]
  + **Backend (Node.js/TypeScript)**: Emprega Jest e Supertest para validar os endpoints da API REST, incluindo validação de dados de entrada, comunicação mockada com o módulo Python e simulação de interações com o banco de dados (atualmente mockado). [cite: server.test.ts]
  + **Frontend (React)**: Usa Jest e React Testing Library para verificar a renderização dos componentes, a manipulação de eventos e a exibição correta dos dados na interface do usuário. [cite: App.test.tsx]
* **Testes End-to-End (E2E)**: Realizados com Cypress, simulam o fluxo completo de um usuário interagindo com o sistema através do frontend. Estes testes validam a integração entre todas as camadas da aplicação (frontend, backend e módulo Python de ML) e garantem que os cenários de uso críticos funcionem do início ao fim. [cite: fetal\_health\_form.cy.ts]
* **Testes de Carga**: Embora não implementados no código-fonte do projeto, a estratégia contempla a utilização de uma ferramenta externa (Apache JMeter) para simular cenários de alto volume de usuários e requisições. O objetivo é avaliar o desempenho do sistema sob estresse, identificando gargalos e limites de escalabilidade. [cite: jmeter\_test\_plan\_existence]

### **2. Casos de Teste Planejados**

#### **2.1. Testes Unitários**

**Módulo Python (ML Service - tests/test\_model\_service.py)**:

* **Carregamento do Modelo**:
  + Sucesso ao carregar o arquivo do modelo (model\_at.sav).
  + Falha ao carregar o modelo (arquivo não encontrado ou erro de leitura).
* **Inferência**:
  + Realização da inferência com dados de entrada válidos, retornando a predição esperada.
  + Verificação de que a inferência falha se o modelo não estiver carregado.
* **Endpoint /predict**:
  + Resposta de sucesso (HTTP 200) para requisições com dados válidos.
  + Resposta de erro (HTTP 400) para JSON vazio ou dados de exame ausentes/nulos.
  + Resposta de erro (HTTP 400) para dados de exame com tipo inválido (não-numérico).
  + Resposta de erro (HTTP 500) em caso de falha interna durante a inferência do modelo.

**Backend (Node.js/TypeScript - Backend/src/\_\_tests\_\_/server.test.ts)**:

* **Endpoint POST /registros**:
  + Sucesso na inserção de um novo registro no "banco de dados" mockado após uma predição bem-sucedida do ML Service.
  + Retorno de erro (HTTP 400) para CPF inválido (formato incorreto ou ausente).
  + Retorno de erro (HTTP 400) para campos de exame (examData) ausentes ou nulos.
  + Retorno de erro (HTTP 400) para campos de exame com tipo de dado inválido (e.g., string em vez de número).
  + Retorno de erro (HTTP 500) em caso de falha na comunicação com o ML Service.
  + Retorno de erro (HTTP 500) em caso de falha simulada ao salvar o registro no "banco de dados".
  + Processamento correto de diferentes valores de fetalhealth (1, 2, 3) retornados pelo ML Service.
* **Endpoint GET /registros**:
  + Retorno de todos os registros existentes quando nenhum parâmetro de CPF é fornecido.
  + Retorno de registros filtrados por um CPF específico.
  + Retorno de um array vazio se nenhum registro for encontrado (com ou sem filtro de CPF).
  + Retorno de erro (HTTP 500) em caso de falha simulada ao buscar registros no "banco de dados".

**Frontend (React - frontend/src/App.test.tsx)**:

* **Renderização Inicial**:
  + Verificar se o título principal da aplicação é renderizado.
  + Verificar se o formulário de "Novo Exame" e todos os seus campos de entrada são renderizados.
* **Interação do Formulário**:
  + Assegurar que os valores dos campos do formulário são atualizados corretamente ao digitar.
  + Simular o envio do formulário com dados válidos e verificar a exibição do resultado da saúde fetal (e.g., "Normal", "Suspeito", "Patológico").
* **Exibição de Dados**:
  + Verificar se a seção de "Histórico de Registros" é renderizada.
  + Confirmar que os registros históricos (mockados) são carregados e exibidos corretamente na tabela.

#### **2.2. Testes End-to-End (Cypress - frontend/cypress/e2e/fetal\_health\_form.cy.ts)**

* **Fluxo de Submissão de Exame Bem-Sucedido**:
  + Limpar o banco de dados (mockado no backend) antes de cada execução para garantir um estado inicial limpo.
  + Visitar a página inicial do frontend.
  + Preencher todos os campos do formulário de exame com dados válidos.
  + Clicar no botão "Enviar Exame".
  + Verificar se o resultado da saúde fetal é exibido corretamente (e.g., "Saúde Fetal: Normal").
  + Confirmar que o registro recém-adicionado aparece na tabela de histórico.
* **Validação de CPF no Frontend**:
  + Tentar submeter o formulário com um CPF em formato inválido.
  + Verificar se uma mensagem de erro apropriada é exibida na interface do usuário (e.g., um window.alert simulado).

#### **2.3. Testes de Carga (Apache JMeter - Arquivo .jmx existente)**

* **Simulação de Usuários Concorrentes**:
  + Configurar grupos de threads para simular múltiplos usuários acessando simultaneamente os endpoints POST /registros e GET /registros (com e sem filtro de CPF).
* **Medição de Desempenho**:
  + Registrar e analisar o tempo médio, mínimo e máximo de resposta para cada tipo de requisição.
  + Observar os percentis de tempo de resposta (e.g., 90%, 95%, 99%) para entender a experiência da maioria dos usuários.
* **Monitoramento de Erros**:
  + Registrar a taxa de erro (percentual de requisições que falharam) sob diferentes níveis de carga.
* **Análise de Comportamento sob Carga**:
  + Observar como o throughput (requisições por segundo) e os tempos de resposta se comportam à medida que o número de usuários simultâneos aumenta.

### **3. Evidências de Execução (Estatísticas, Gráficos)**

As evidências de execução são geradas pelas respectivas ferramentas de teste. Como este relatório é estático, as evidências são descritas como saídas esperadas:

#### **3.1. Testes Unitários**

* **Módulo Python (pytest)**:
  + **Saída no Terminal**: ======================= X passed in Y.YYs ======================= (onde X é o número total de testes unitários Python).
  + **Exemplo**:  
    ============================= test session starts ==============================  
    platform win32 -- Python 3.10.0, pytest-8.4.1, pluggy-1.6.0  
    rootdir: D:\N3TdS\N3TdS  
    collected 12 items  
      
    tests/test\_model\_service.py ............ [100%]  
      
    ======================= 12 passed in 1.41s =======================
* **Backend (Node.js/TypeScript - Jest)**:
  + **Saída no Terminal**: PASS src/\_\_tests\_\_/server.test.ts seguido da lista de testes aprovados.
  + **Exemplo**:  
    PASS src/\_\_tests\_\_/server.test.ts  
     Backend API - Integração com Banco de Dados  
     POST /registros  
     √ deve inserir um novo registro no banco de dados e retornar 200  
     √ deve retornar 400 se o CPF for inválido  
     √ deve retornar 500 em caso de erro no serviço de ML  
     √ deve retornar 500 em caso de erro ao salvar no banco de dados  
     √ deve retornar 400 se um campo obrigatório de examData estiver faltando  
     √ deve retornar 400 se um campo numérico de examData for inválido  
     √ deve processar corretamente um fetalhealth diferente do padrão  
     GET /registros  
     √ deve retornar todos os registros se nenhum CPF for fornecido  
     √ deve retornar registros filtrados por CPF  
     √ deve retornar 500 em caso de erro ao buscar registros  
     √ deve retornar um array vazio se nenhum registro for encontrado  
      
    Test Suites: 1 passed, 1 total  
    Tests: 11 passed, 11 total  
    Snapshots: 0 total  
    Time: X.XXX s
* **Frontend (React - Jest)**:
  + **Saída no Terminal**: PASS src/App.test.tsx seguido da lista de testes aprovados.
  + **Exemplo**:  
    PASS src/App.test.tsx  
     App Component Unit Tests  
     √ deve renderizar o título principal do aplicativo  
     √ deve renderizar o formulário de novo exame com todos os campos necessários  
     √ deve atualizar os valores do formulário ao digitar nos inputs  
     √ deve exibir o resultado da saúde fetal após o envio do formulário com sucesso  
     √ deve carregar e exibir os registros históricos  
      
    Test Suites: 1 passed, 1 total  
    Tests: 5 passed, 5 total  
    Snapshots: 0 total  
    Time: X.XXX s

#### **3.2. Testes End-to-End (Cypress)**

* **Interface do Cypress Test Runner**: Exibição visual da execução dos testes no navegador, com cada passo marcado como sucesso (verde) ou falha (vermelho).
* **Relatórios (Opcional, se configurado)**: Geração de vídeos das execuções, capturas de tela em caso de falha e relatórios HTML detalhados com a duração de cada teste e passo.
* **Exemplo Visual**:

#### **3.3. Testes de Carga (Apache JMeter)**

* **Relatórios Gerados pelo JMeter**:
  + **Summary Report / Aggregate Report**: Tabelas com métricas como:
    - **#Samples**: Total de requisições enviadas.
    - **Average**: Tempo médio de resposta (ms).
    - **Min/Max**: Tempos mínimo e máximo de resposta (ms).
    - **Error %**: Percentual de requisições com erro.
    - **Throughput**: Número de requisições por segundo.
    - **Latência (Percentis)**: Tempos de resposta em que 90%, 95% ou 99% das requisições foram concluídas.
  + **Gráficos**:
    - **Response Times Over Time**: Gráfico de linha mostrando a evolução do tempo de resposta ao longo do teste.
    - **Throughput**: Gráfico de linha mostrando o volume de requisições processadas por segundo.
    - **Active Threads Over Time**: Gráfico mostrando o número de usuários virtuais ativos ao longo do teste.
    - **Errors/sec**: Gráfico mostrando a taxa de erros por segundo.
  + **Exemplo de Gráfico (Response Times Over Time)**:

### **4. Análise dos Resultados**

Com base na estratégia e nos resultados esperados:

* **Robustez da Lógica Interna**: A passagem dos testes unitários em todas as camadas (Python ML, Backend Node.js, Frontend React) indica que a lógica de negócio, validações de dados, e a funcionalidade individual de cada componente estão corretas. Isso fornece uma base sólida para a aplicação.
* **Integração de Componentes**: Os testes E2E com Cypress, ao simular o fluxo completo do usuário, validam a comunicação e a integração entre o frontend, o backend e o módulo de ML. A aprovação desses testes confirma que o sistema funciona como uma unidade coesa.
* **Comportamento do Sistema (Potencial de Carga)**: A execução dos testes de carga com JMeter (utilizando o arquivo .jmx existente) é crucial para entender o desempenho do sistema sob estresse.
  + **Resultados Positivos**: Tempos de resposta baixos e estáveis, juntamente com uma baixa taxa de erro sob carga, indicariam que o sistema é eficiente e escalável para o volume de usuários simulado.
  + **Resultados Negativos**: Aumento significativo nos tempos de resposta, alta taxa de erro, ou throughput decrescente sob carga, apontariam para gargalos de desempenho que precisam ser investigados e otimizados (seja no backend, no ML service, ou na infraestrutura).

### **5. Sugestões de Melhoria**

1. **Implementação de Persistência Real no Backend**:
   * **Prioridade**: Alta.
   * **Detalhes**: Atualmente, o backend utiliza um banco de dados mockado. Para que o sistema seja funcional em um ambiente real, é fundamental configurar e integrar um banco de dados persistente (e.g., PostgreSQL, conforme já indicado em database.ts). Isso envolve:
     + Remover a lógica de mock de activePool e activeInitDatabase em server.ts.
     + Configurar as variáveis de ambiente (DB\_HOST, DB\_PORT, DB\_USER, DB\_PASSWORD, DB\_NAME) para a conexão real.
     + Garantir que a função initDatabase seja executada para criar a estrutura da tabela no banco de dados real.
   * **Impacto**: Essencial para a funcionalidade principal de armazenamento de registros.
2. **Aprimoramento e Expansão dos Testes E2E**:
   * **Prioridade**: Média.
   * **Detalhes**: Adicionar mais cenários de teste Cypress para cobrir:
     + Comportamento da UI em caso de erros retornados pelo backend (e.g., mensagens de erro específicas para o usuário).
     + Testes de acessibilidade (Cypress pode ser integrado com ferramentas como axe-core).
     + Testes de responsividade em diferentes tamanhos de tela.
   * **Impacto**: Aumenta a confiança na experiência do usuário e na robustez da aplicação em diferentes situações.
3. **Integração Contínua (CI/CD) para Testes**:
   * **Prioridade**: Média.
   * **Detalhes**: Configurar um pipeline de CI/CD (e.g., GitHub Actions, GitLab CI, Jenkins) para automatizar a execução de todos os testes (unitários, E2E) a cada commit ou pull request.
   * **Impacto**: Garante que problemas sejam identificados precocemente no ciclo de desenvolvimento, melhora a qualidade do código e acelera o feedback para os desenvolvedores.
4. **Análise e Otimização de Desempenho (Pós-Testes de Carga)**:
   * **Prioridade**: Alta (após a execução dos testes de carga).
   * **Detalhes**: Com base nos relatórios do JMeter, identificar os gargalos de desempenho (seja no backend, no módulo de ML ou na comunicação entre eles). As otimizações podem incluir:
     + **Backend**: Otimização de queries SQL (quando o DB real for implementado), caching, ajuste de configurações do Express.
     + **Módulo ML**: Otimização do modelo (e.g., quantização, poda), uso de hardware acelerado (GPU) se aplicável, ou otimização do código Python para inferência.
     + **Comunicação**: Otimização da rede ou do protocolo de comunicação entre o backend e o serviço de ML, se a latência for um problema.
   * **Impacto**: Garante que o sistema possa lidar com o volume de usuários esperado em produção de forma eficiente.
5. **Documentação de API (OpenAPI/Swagger)**:
   * **Prioridade**: Baixa/Média.
   * **Detalhes**: Documentar a API REST do backend usando padrões como OpenAPI/Swagger. Isso facilitaria a integração com outros sistemas e a manutenção futura.
   * **Impacto**: Melhora a colaboração e a manutenibilidade do projeto.

Este relatório visa fornecer uma base sólida para a compreensão do estado atual dos testes do projeto e as direções para seu aprimoramento contínuo.