CENTRO UNIVERSITÁRIO INTERNACIONAL UNINTER

ESCOLA SUPERIOR POLITÉCNICA

CST ANÁLISE E DESENVOLVIMENTO DE SISTEMAS - DISTÂNCIA

PROJETO MULTIDISCIPLINAR

ATIVIDADE PRÁTICA

RONALDO OTTONI BORGES DE CARVALHO, 4385395

SUMÁRIO

1 INTRODUÇÃO

O presente estudo de caso analisa a necessidade da instituição VidaPlus, uma rede que administra hospitais, clínicas de bairro, laboratórios e equipes de home care, de implementar um Sistema de Gestão Hospitalar e de Serviços de Saúde (SGHSS). Este sistema surge como uma solução tecnológica para centralizar e otimizar a gestão de suas operações, integrando desde o atendimento de pacientes até a administração hospitalar. O objetivo principal do projeto é unificar processos como cadastro e atendimento de pacientes, gestão de profissionais de saúde, administração de recursos, suporte à telemedicina e garantia de segurança e conformidade, promovendo eficiência e qualidade no atendimento.

Os principais usuários do SGHSS incluem pacientes, que buscam autonomia para agendar consultas e acessar seus históricos clínicos; profissionais de saúde, como médicos e enfermeiros, que necessitam de ferramentas para gerenciar agendas e prontuários; e administradores, responsáveis pelo controle operacional e financeiro das unidades. A relevância do sistema está na sua capacidade de atender às demandas de uma rede de saúde em expansão, assegurando escalabilidade, conformidade com a Lei Geral de Proteção de Dados (LGPD) e acessibilidade, além de oferecer suporte à telemedicina, uma tendência crescente no setor. Assim, o SGHSS não apenas moderniza a gestão da VidaPlus, mas também fortalece sua missão de proporcionar serviços de saúde eficazes e seguros. A API REST do SGHSS fornece endpoints para gerenciar pacientes, consultas e administração.

2 ANÁLISE E REQUISITOS

O objetivo é especificar as funcionalidades esperadas e as características de qualidade do sistema. Este documento apresenta os Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF) do Sistema de Gestão Hospitalar.

ID	Descrição	Detalhamento/Exemplo
RF01	Cadastro de Instituições	Classe designada à registrar as várias unidades administradas pelo grupo, podendo contemplar Hospitais, Clínicas, Laboratórios, Consultórios, etc.
RF02	Cadastro de Planos de Saúde	O Sistema permite cadastrar todo e qualquer Plano de Saúde, com campo de “Descrição” onde deverá ser informado o nome do Plano de Saúde (ex. SUS, Amil, Unimed etc.).
RF03	Cadastro de Pessoas	Cadastro de Pessoas em Geral, este requisito contemplará Pacientes, Profissionais da Saúde e Funcionários em Geral da Instituição.
RF04	Cadastro de Usuários	O Sistema permitirá cadastrar um Usuário desde que esteja vinculado a uma Pessoa previamente cadastrada no Sistema, podendo inclusive ser um Paciente, com acesso restrito unicamente ao seu próprio Prontuário.
RF05	Cadastro de Modalidades	Este requisito contemplará as possibilidades para Registros como Consultas (Online, Presencial, Telefônica, etc)
RF06	Cadastro de Acomodações	O Sistema permitirá uma espécie de mapeamento das acomodações disponíveis na Instituição, considerando 4 níveis como Hospital, Ala, Quarto e Leito. (*exemplo: Hospital Central, UTI, Quarto 1, Leito 5*).
RF07	Cadastro de Registros	O Sistema contará com um cadastro de Registros que consiste em um Prontuário Geral identificado pelo código da Pessoa

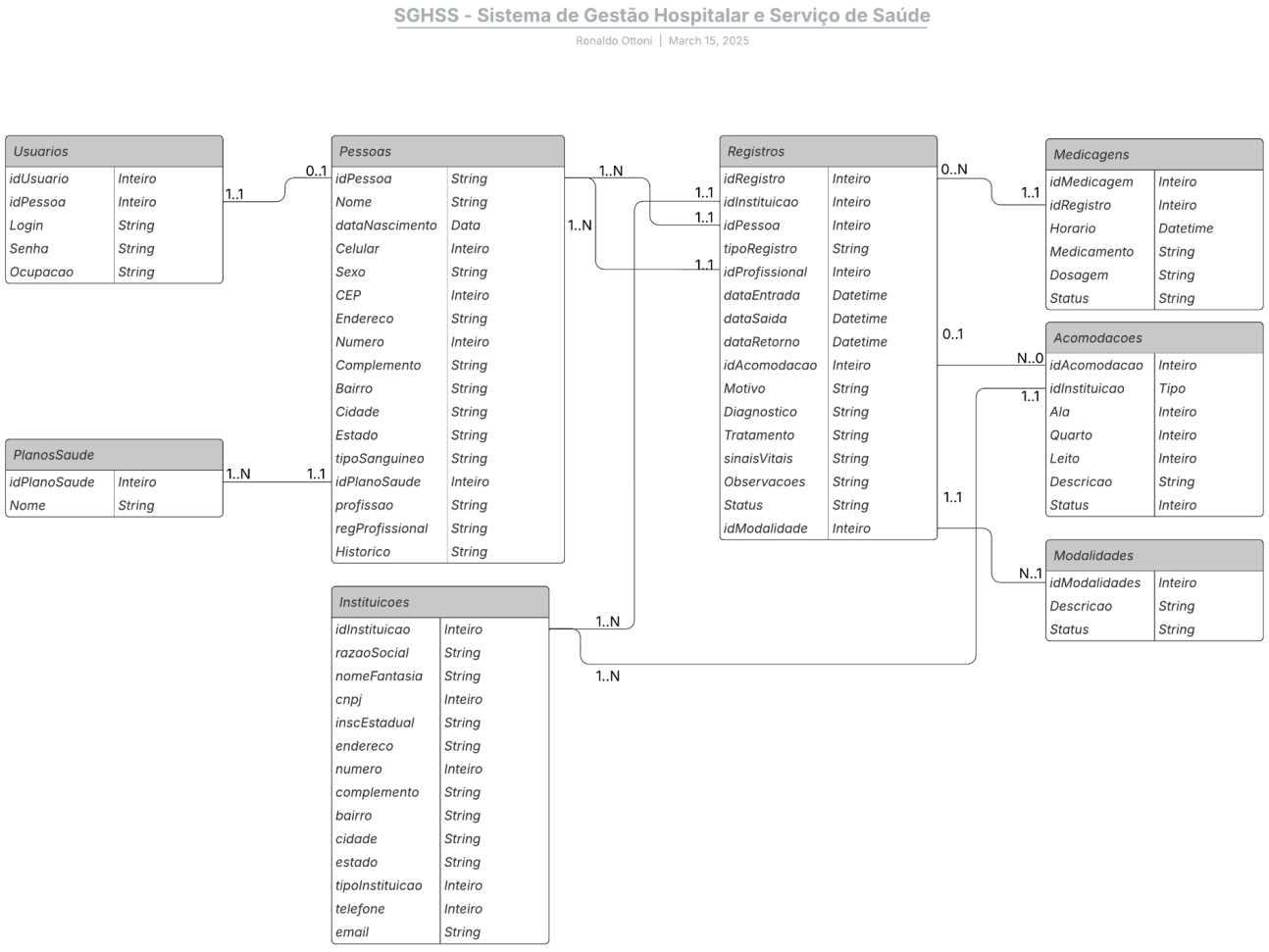
		cadastrada, pode ser Consultas, Exames, Internações entre outros, preenchendo campos como id da pessoa, tipo de registro (Consulta, Exame, Internação, etc), id do profissional (Pessoa cadastrada como Profissional), Data Entrada (Data Entrada para Internação ou Data de Agendamento para Consultas, Exames), data de saída (Data da alta do paciente), sinais Vitais, Sintomas, tratamento, etc.
RF08	Cadastro de Medicagens	Este requisito será responsável por Registrar em forma de agendamentos, quando em caso de internação e a mesma contiver preenchida a seção Tratamento, sendo registrado, Data, Hora, Medicamento, Dosagem e Status da medicação.

ID	Descrição	Detalhamento/Explicação
RNF01	Segurança e Compliance	● Controle de acesso baseado em perfis (ex: paciente, enfermeiro, administrador); ● Garantir conformidade com a Lei Geral de Proteção de Dados (LGPD), como anonimização de dados quando necessário. ● Manter registros de log e auditoria para rastrear ações no sistema. ● Utilizar criptografia de ponta a ponta para proteger dados sensíveis quando necessário (ex: AES-256)
RNF02	Escalabilidade	● Suportar o funcionamento em múltiplas unidades (hospitais, clínicas, laboratórios); ● Permitir a expansão para novos usuários e maior volume de dados sem perda de desempenho.
RNF03	Desempenho	● Garantir tempo de resposta inferior a 2 segundos em consultas críticas (ex: prontuários com alocação em UTI); ● Processar eficientemente grandes volumes de transações simultâneas.
RNF04	Acessibilidade	● Oferecer interface responsiva adaptável tanto para dispositivos móveis quanto para desktops; ● Seguir padrões W3C/WCAG, como

		suporte a leitores de tela e contraste adequado.
RNF05	Disponibilidade	• Manter uptime mínimo de 99,5%, com sistema online 24/7; • Implementar backups automáticos e logs robustos para recuperação de dados em caso de falhas.

3 MODELAGEM E ARQUITETURA

3.1 Diagrama Entidade-Relacionamento



3.2 Descrição dos principais endpoints da API

3.2.1 Cadastro de Pessoas

- Rota: /api/pessoa
- Método: POST
- Descrição: Cadastrar uma nova Pessoa no sistema, como Paciente, Profissional, etc
- Autenticação: Exige login com usuário com nível de Administrador
- Parâmetro:

```
{
"nome": "string, obrigatório",
"dataNascimento": "datetime, obrigatório",
"sexo": "string, obrigatório",
 "celular": "integer, obrigatório",
 "cep": "integer, obrigatório",
 "pais": "string, obrigatório",
 "estado": "string, obrigatório",
 "cidade": "string, obrigatório",
 "bairro": "string, obrigatório",
 "endereco": "string, obrigatório",
 "numero": "integer, opcional",
 "complemento": "string, opcional",
 "tipoSanguineo": "string, opcional",
 "idPlanoSaude": "integer, obrigatório",
 "profissao": "string, opcional",
 "regProfissional": "string, opcional",
 "historico": "string, opcional",
}
```

- Resposta:
  - Sucesso:

```
{
  "id": “P00001”,
  "mensagem": "Pessoa cadastrado com sucesso!"
}
```

  - Falha:

```
{
 "mensagem": "Pessoa não cadastrada!"
}
```

  - Erro:

```
{
  "erro": "Erro ao gravar Pessoa!"
}
```

3.2.2 Consulta de Pessoa

- Rota: /api/pessoa
- Método: GET
- Descrição: Consultar uma pessoa cadastrada no sistema
- Autenticação: Exige login com usuário com nível de Administrador
- Parâmetro:

```
{
 "id": "P00001",
}
```

- Resposta:
  - Sucesso:

```
{
 "nome": "Ronaldo Ottoni Borges de Carvalho",
 "dataNascimento": "1983-03-09",
 "sexo": "M",
 "celular": "41-991471108",
 "cep": "81000-000",
 …
}
```

  - Falha:

```
{
 "mensagem": "Pessoa não localizada!"
}
```

  - Erro:

```
{
 "mensagem": "Erro ao consultar a Pessoa!"
}
```

3.2.3 Consulta de Prontuário

- Rota: /api/prontuario
- Método: GET
- Descrição: Consultar o prontuário da Pessoa (paciente), retornando todos os Registros relacionados à pessoa
- Autenticação: Exige login com usuário de nível médico, enfermeiro, doutor ou paciente
- Parâmetro:

```
{
  "id": "P00001",
}
```

ou

```
{
  "cpf": "70090020050",
}
```

- Resposta:

- Sucesso:

```
{
 "idRegistro": 000071,
 "idInstituicao": 0000021,
 "tipoRegistro": “CON”,
 "idProfissional": 0000011,
 "dataEntrada": "2025-02-03",
 …
},
{
 "idRegistro": 000121,
 "idInstituicao": 0000011,
 "tipoRegistro": “EXA”,
 "idProfissional": 0000029,
 "dataEntrada": "2025-02-05",
 …
}
```

- Falha:

```
{
 "mensagem": "Pessoa ainda não contém Registros!"
}
```

- Erro:

```
{
 "mensagem": "Erro ao consultar os Registros da Pessoa!"
}
```

3.3 Tecnologias utilizadas

A linguagem escolhida foi o Python pois oferece simplicidade e legibilidade, acelerando o desenvolvimento de endpoints como /api/pacientes e /api/registros.

O Flask é minimalista e flexível, permitindo criar uma API robusta com rotas específicas para receber chamadas de um frontend React/Node.js, como requisições POST para pessoas ou GET para prontuários, o Flask também suporta operações assíncronas (com extensões como Flask-Async) para lidar com múltiplos usuários, atendendo ao requisito de desempenho em consultas críticas.

O Banco de Dados escolhido foi o SQLite, leve, embutido e baseado em arquivos. Ideal para protótipos ou sistemas de menor escala, como o SGHSS em fase inicial, devido à sua simplicidade e ausência de servidor dedicado. Armazena dados como cadastros de Pessoas e Registros de forma estruturada. Embora limitado em escalabilidade extrema, é suficiente para o escopo inicial e pode ser substituído por MySQL no futuro, mantendo a compatibilidade com o backend Flask.

O backend foi projeto com uma API RESTful que expõe rotas claras e padronizadas, prontas para receber chamadas de um frontend desenvolvido em React/Node.js. Por exemplo:

- Rota /api/pessoas (POST): Recebe dados JSON de um formulário React para cadastrar pessoas.
- Rota /api/pessoas/{id}/prontuario (GET): Fornece dados em JSON para exibição no frontend. Essa integração é facilitada pelo uso de respostas em formato JSON, códigos HTTP padronizados(ex. 200 para sucesso, 400 para erro) e suporte a cabeçalhos como

Content-Type: application/json, compatíveis com bibliotecas como Axios ou Fetch usadas no React.

4 IMPLEMENTAÇÃO (PROTOTIPAGEM)

A imagem abaixo apresenta trecho do código Python do programa database.py, responsável pela criação do Banco de Dados, criando as tabelas caso não existam e para algumas tabelas incluindo um registro inicial.

```
import sqlite3


def db_connect():
    conn = sqlite3.connect("hospital.db")
    conn.row_factory = sqlite3.Row
    return conn


def init_db():
    conn = db_connect()
    c = conn.cursor()

    # Criar a tabela Instituição caso não exista
    c.execute(
        """
        CREATE TABLE IF NOR EXISTS instituicao (
            idInstituicao INTEGER PRIMARY KEY AUTOINCREMENT,
            razaoSocial VARCHAR(50) NOT NULL,
            nomeFantasia VARCHAR(50) NOT NULL,
            cnpj INTEGER NOT NULL,
            inscEstadual VARCHAR(20),
            endereco VARCHAR(100) NOT NULL,
            numero INTERGER,
            complemento VARCHAR(100),
            bairro VARCHAR(50) NOT NULL,
            cidade VARCHAR(100) NOT NULL,
            estado VARCHAR(2) NOT NULL,
            tipoInstituicao INTEGER NOT NULL
            telefone INTEGER NOT NULL,
            email VARCHAR(100) NOT NULL,
        )
        """
    )
```

```
# Criar a tabela de Planos de Saúde caso não exista
c.execute(
    """
    CREATE TABLE IF NOT EXISTS planosSaude (
        idPlanoSaude INTEGER PRIMARY KEY AUTOINCREMENT,
        descricao VARCHR(050) NOT NULL
    )
    """
)
# Incluir os primeiros Planos de Saúde caso tabela esteja vazia
c.execute("SELECT COUNT(*) FROM planosSaude")
if c.fetchone()[0] == 0:
    c.executemany(
        "INSERT INTO planosSaude (descricao) VALUES (?)",
        [("SUS",), ("Particular",), ("Amil",), ("Unimed",)],
    )

# Criar a tabela de Pessoas se não existir
c.execute(
    """
    CREATE TABLE IF NOT EXISTS pessoas (
        idPessoa INTEGER PRIMARY KEY,
        nome VARCHAR(60) NOT NULL,
        dataNascimento DATETIME NOT NULL,
        sexo VARCHAR(001) NOT NULL,
        celular INTEGER NOT NULL,
        cep INTEGER NOT NULL,
        pais VARCHAR(050) NOT NULL,
        estado VCARCHAR(002) NOT NULL,
        cidade VARCHAR(050) NOT NULL,
        bairro VARCHAR(025) NOT NULL,
        endereco VARCHAR(050) NOT NULL,
        numero INTEGER,
        complemento VARCHAR(030),
        tipoSanguineo VARCHAR(005),
        idPlanoSaude INTEGER NOT NULL,
        profissao VARCHAR(030),
        regProfissional VARCHAR(030),
        historico VARCHAR(800)
    )
    """
```

```
    )

    # Incluir Pessoa padrão para caso tabela esteja vazia
    c.execute("SELECT COUNT(*) FROM pessoas")
    if c.fetchone()[0] == 0:
        c.executemany(
            """INSERT INTO pessoas (idPessoa, nome, dataNascimento, sexo, celular, cep,
pais, estado, cidade, bairro, endereco, idPlanoSaude)
                                    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)""",
            [
                (
                    70090020030,
                    "RONALDO OTTONI BORGES",
                    "1983-03-09 15:15:15",
                    "M",
                    41991471108,
                    83420000,
                    "BRAZIL",
                    "PR",
                    "QUATRO BARRAS",
                    "JD MENINO DEUS",
                    "RUA PAPA JOAO XXIII",
                    1,
                )
            ],
        )
```

O código abaixo apresenta trecho da API em Python com Flask gerenciando as rotas que receberão as chamadas do React/Node.js.

```
    from flask import Flask, jsonify, request
    from datetime import datetime, timedelta, date, time
    import database as db

    app = Flask(__name__)

    # Inicialização do banco de dados na chamada do App
    db.init_db()


    ##########################################################
```

```python
#                Rotas para camada Web                #
#######################################################


#######################################################
#                  Pessoas                            #
#######################################################
@app.route("/api/pessoas", methods=["GET"])
def listar_pessoas():
    pessoas = db.listar_pessoas()
    return jsonify(pessoas)


@app.route("/api/pessoa/{id}/prontuario", methods=["GET"])
def consultar_pessoa():
    data = request.json
    resultado = db.consultar_pessoa(data["idPessoa"])
    return resultado


@app.route("/api/pessoa", methods=["POST"])
def gravar_pessoa():
    data = request.json
    id = data.get("idPessoa", " ")
    numero = data.get("numero", "")
    complemento = data.get("complemento", "")
    tipoSanguineo = data.get("tipoSanguineo", "")
    profissao = data.get("profissao", "")
    regProfissional = data.get("regProfissional", "")
    historico = data.get("historico", "")
    resultado = db.gravar_pessoa(
        id,
        data["nome"],
        data["dataNascimento"],
        data["sexo"],
        data["celular"],
        data["cep"],
        data["pais"],
        data["estado"],
        data["cidade"],
        data["bairro"],
        data["endereco"],
```

```
            numero,
            complemento,
            tipoSanguineo,
            data["idPlanoSaude"],
            profissao,
            regProfissional,
            historico,
        )
        return resultado


    @app.route("/api/pessoa", methods=["DELETE"])
    def deletar_pessoa():
        data = request.json
        resultado = db.deletar_pessoa(data["idPessoa"])
        return resultado
```

5 PLANO DE TESTES

5.1 Critérios de aceitação

Os critérios de aceitação definem quando um teste é considerado bem-sucedido, alinhando-se aos requisitos do SGHSS.

- Funcionalidade:
  - Cadastro de Pessoa: Todos os campos obrigatórios salvos corretamente;
  - Consulta Virtual: Videochamada iniciada com sucesso e prontuário registrado;
- Desempenho:
  - Tempo de Resposta: Máximo de 2 segundos para consultas críticas (ex.: acesso a prontuários);
  - Escalabilidade: Sistema suporta 1.000 usuários simultâneos sem falhas
- Segurança:
  - Acesso: Apenas usuários autorizados acessem dados conforme perfil (paciente, médico, administrador);
  - LGPD: Logs de auditoria registram todas as ações sensíveis.
- Usabilidade:
  - Interface: 90% dos usuários concluem tarefas (ex.: Pacientes, Medicamentos) sem ajuda em menos de 5 minutos;
  - Acessibilidade: Interface atende WCAG 2.1 (ex.: contraste mínimo de 4.5:1).

5.2 Resumo

- Casos de Teste: Cobrem as principais funcionalidades (cadastro, agendamentos) e aspectos críticos (segurança, desempenho);
- Critérios de Aceitação: Alinhados aos requisitos funcionais (ex.: Consulta Virtual) e não funcionais (.: LGPD, escalabilidade);
- Roteiros: Práticos e mensuráveis, com ferramentas sugeridas para execução.

6 RESUMO

O desenvolvimento do Sistema de Gestão Hospitalar e de Serviços de Saúde (SGHSS) da VidaPlus revelou lições valiosas e desafios significativos, ao mesmo tempo em que apontou caminhos para sua evolução futura. Entre as principais lições aprendidas, destaca-se a eficácia de utilizar Python com Flask para criar um backend ágil e funcional, capaz de atender aos requisitos de cadastro de pacientes, agendamento de medicações, Consultas virtuais e administração hospitalar. A simplicidade do Flask permitiu a rápida implementação de uma API RESTful, enquanto sua integração com o SQLite demonstrou ser uma solução prática para persistência de dados em um escopo inicial. Além disso, a preparação das rotas para receber chamadas de um frontend React/Node.js reforçou a importância de projetar sistemas modulares e interoperáveis, garantindo flexibilidade para expansão.

Os desafios enfrentados incluíram a limitação de escalabilidade do SQLite, que, embora eficiente para protótipos, pode não suportar o volume de dados e usuários de uma rede como a VidaPlus em plena operação. Outro ponto crítico foi equilibrar desempenho e usabilidade, especialmente em consultas críticas, onde o tempo de resposta deve ser mínimo. A configuração da API para atender a múltiplos perfis de usuários (pacientes, médicos e administradores) também exigiu atenção cuidadosa ao design das rotas, para assegurar clareza e consistência nas respostas. Esses obstáculos destacaram a necessidade de planejamento antecipado para cenários de alta demanda e a importância de testes rigorosos de carga e usabilidade.

Para evoluções futuras, alguns pontos de atenção são essenciais. Primeiramente, recomenda-se migrar o banco de dados para uma solução mais robusta, como MySQL ou PostrgreSQL, para suportar múltiplas unidades hospitalares e garantir escalabilidade. Outro aspecto é a otimização do backend Flask, possivelmente com a adoção de extensões assíncronas ou a transação para frameworks mais escaláveis, como FastAPI, caso o volume de requisições aumente significativamente. Além disso, a integração de telemedicina com tecnologias como WebRTC deve ser priorizada para atender à crescente demanda por atendimentos remotos, enquanto a interface com React/Node.js pode ser expandida para incluir notificações em tempo real. Por fim, a evolução do SGHSS deve considerar a incorporação de padrões de segurança adicionais e conformidade contínua com regulamentações como a LGPD, assegurando a proteção e rastreabilidade dos dados.

Em suma, o projeto do SGHSS demonstrou o potencial de uma solução tecnológica para modernizar a gestão de saúde, ao mesmo tempo em que evidenciou a importância de adaptar as tecnologias às necessidades de crescimento e desempenho. Esses aprendizados servirão como base para transformar o sistema em uma ferramenta ainda mais eficiente e abrangente no futuro.

7 REFERÊNCIAS

Python

- PYTHON SOFTWARE FOUNDATION. *Python 3.12.2 documentation*. Disponível em: https://docs.python.org/3/. Acesso em: 05 mar. 2025.

Flask

- PALLETS PROJECTS. *Flask documentation (3.0.x)*. Disponível em: https://flask.palletsprojects.com/en/3.0.x/. Acesso em: 10 mar. 2025.

SQLite

- SQLITE CONSORTIUM. *SQLite documentation*. Disponível em: https://www.sqlite.org/docs.html. Acesso em: 08 mar. 2025.

Grok (xAI)

- XAI. *Grok: an AI assistant*. Disponível em: https://x.ai/grok. Acesso em: 15 mar. 2025.
- **Nota**: Como Grok é uma IA da xAI, referenciei o site oficial da xAI. Não há uma "documentação" específica.

Github

- Repositório do protótipo com acesso aberto. https://github.com/ronaldoottoni/SGHSS.