UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO (2025)

MERCURIOS: UM SISTEMA PARA AUXÍLIO DE GESTÃO DE ORDENS DE SERVIÇO

Vinicius F. A. do Nascimento[1], Paulo H. L. Silva[2], Angélica F. de Castro[3]

RESUMO

Uma das tarefas da chefia de departamento, para manter o bom funcionamento dos laboratórios e preservar o fluxo das aulas, é fazer o levantamento dos equipamentos que estão ou não em funcionamento no prédio do Laboratório de Ciência da Computação da UFERSA. O presente artigo busca conectar o corpo discente e os responsáveis por solicitarem manutenção, por meio de um sistema *web* que faça uso de agentes inteligentes para simplificar a tarefa de registrar problemas através da linguagem natural, com tecnologias como *spaCy* para extração de entidades das mensagens para compor uma ordem de serviço e os *transformers* do *Hugging Face*, usando uma versão *fine-tuned* do *BERTimbau Base* para filtrar mensagens válidas, incompletas e inválidas. Ao final do desenvolvimento, alcançou-se um produto funcional que atendeu aos objetivos com oportunidades para melhorias futuras.

Palavras-chave: Ordem de serviço, manutenção, agentes inteligentes.

1 INTRODUÇÃO

O funcionamento adequado do maquinário é fundamental para a dinâmica de aprendizado em um laboratório de atividades de ensino, pesquisa e extensão em computação pois, mesmo que a aula possa ser ministrada e as informações passadas aos alunos, há grande valor em os discentes poderem praticar conceitos vistos em aula. Para alcançar tal feito, a UFERSA dispõe de uma equipe de servidores que realizam manutenções periódicas em equipamentos como centrais de ar e computadores, localizados no Laboratório de Ciência da Computação (LCC).

O LCC é um prédio de dois pavimentos com um perímetro de aproximadamente 134 metros. Possui 7 laboratórios para graduação e uma sala multimídia onde aulas e palestras são ministradas, 2 laboratórios híbridos para graduação e pós-graduação, 1 laboratório para ações de extensão e 4 laboratórios de pesquisa. Em cada sala, tem ao menos uma central de ar e, num total, há cerca de 240 computadores (UFERSA, 2017).

Entretanto, dada esta proporção do prédio, é compreensível a dificuldade de registrar todos os problemas em tempo útil e, mesmo quando são registrados, acabam não sendo resolvidos conforme a urgência devido a questões internas de logística, burocracia e alta demanda de manutenção entre os setores, fora a falta de um servidor para o acompanhamento

---

[1] *Graduando em Ciência da Computação pela UFERSA - Campus Mossoró.*
[2] *Doutor em Engenharia Elétrica e de Computação pela UFRN em 2017.*
[3] *Doutora em Geodinâmica pela UFRN em 2007.*

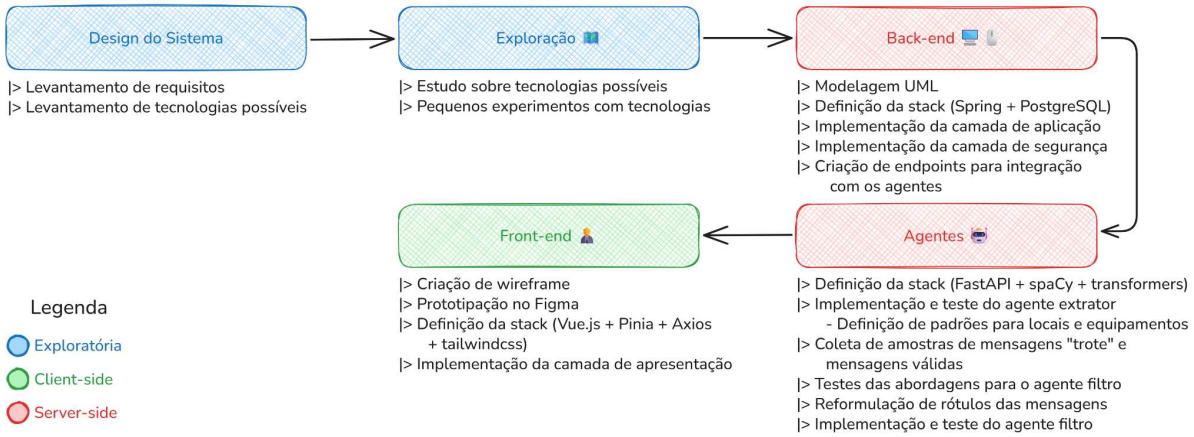
sistemático destas. A questão é que, por muitas vezes, essa situação de *softwares* desatualizados e equipamentos não funcionando atrapalha o fluxo das aulas, dificultando a transmissão de conhecimento.

Dada a dificuldade de solucionar os problemas mais internos referentes à gestão, o presente trabalho propõe a criação de um sistema *web* que possa ser utilizado tanto pelo corpo discente, registrando por meio da linguagem natural - forma como nos comunicamos - os problemas como ordens de serviço conforme encontrados, esperando simplificar a submissão, quanto a chefia de departamento, como uma ferramenta de auxílio para sistemas já existentes a fim de coletar e centralizar dados acerca das problemáticas existentes nos laboratórios.

2 MÉTODO

Nesta seção, será apresentado um passo a passo que foi seguido para a criação do sistema, detalhado posteriormente no desenvolvimento. O objetivo é dar uma visão geral do processo que foi realizado, conforme descrito na Figura 01.

Figura 01 - Descrição do método aplicado para o sistema.



Fonte: Elaborado pelo autor (2025).

Foi realizado um levantamento de requisitos e de possíveis tecnologias para cada etapa a fim de se ter uma base antes de prosseguir, com a experimentação de algumas tecnologias para a aplicação. Em seguida, implementou-se a camada de aplicação do lado do servidor, construindo a lógica para se lidar com o banco de dados e a segurança para proteção das rotas. Consoante a isso, os agentes foram implementados em conjunto com uma coleta de amostras para fins de testes e adaptação dos modelos. Por fim, a camada de apresentação foi elaborada, com o desenho em wireframe e a prototipação do sistema precedendo sua implementação.

As etapas foram definidas conforme ilustrado, sendo seguidas de maneira incremental conforme o desenvolvimento do sistema. Nas seções seguintes, serão detalhadas as etapas descritas.

3 DESENVOLVIMENTO

Nesta seção, abordaremos a arquitetura seguida para o desenvolvimento do sistema, interação entre cada componente, implementação e a escolha das tecnologias.

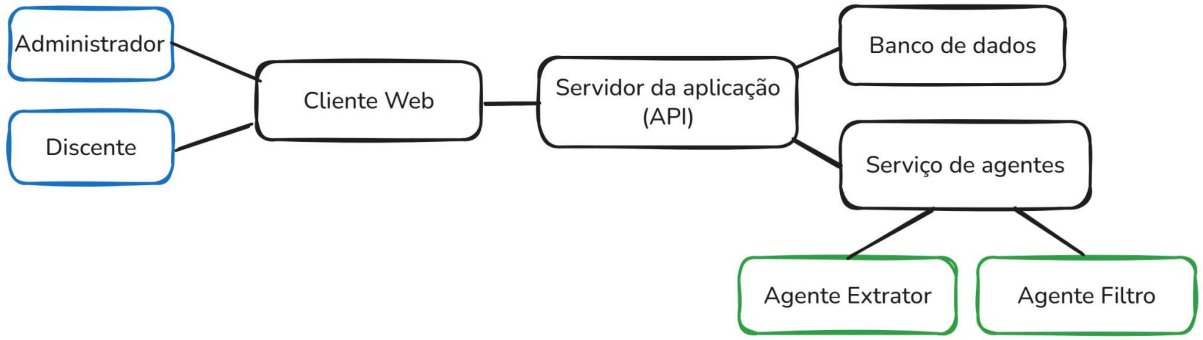
3.1 Arquitetura do Sistema

Nos dias de hoje, a praticidade de se usar serviços de mensageria, como *Telegram* e *Whatsapp*, levantou a possibilidade de se desenvolver um *chatbot* para os usuários interagirem com o sistema.

Porém, o uso dessas tecnologias poderia acabar se categorizando como *Shadow IT* (TI invisível em tradução livre), termo referente ao uso de uma ferramenta de Tecnologia da Informação (TI) - recursos tecnológicos voltados ao processamento de informação - não oficial da organização sendo usada para tal, o que pode implicar em redundância de processos e conflito de recursos (NETO *et al.*, 2019) e, também, na questão da segurança, em pontos como a irretratabilidade - que diz respeito a ações realizadas no sistema serem da responsabilidade irrefutável dos autores - e autenticidade - a garantia que os dados associados a uma entidade são legítimos.

Dito isso, o sistema foi desenhado da seguinte forma, conforme visto na Figura 02.

Figura 02 - Arquitetura simplificada do sistema.



Fonte: Elaborado pelo autor (2025).

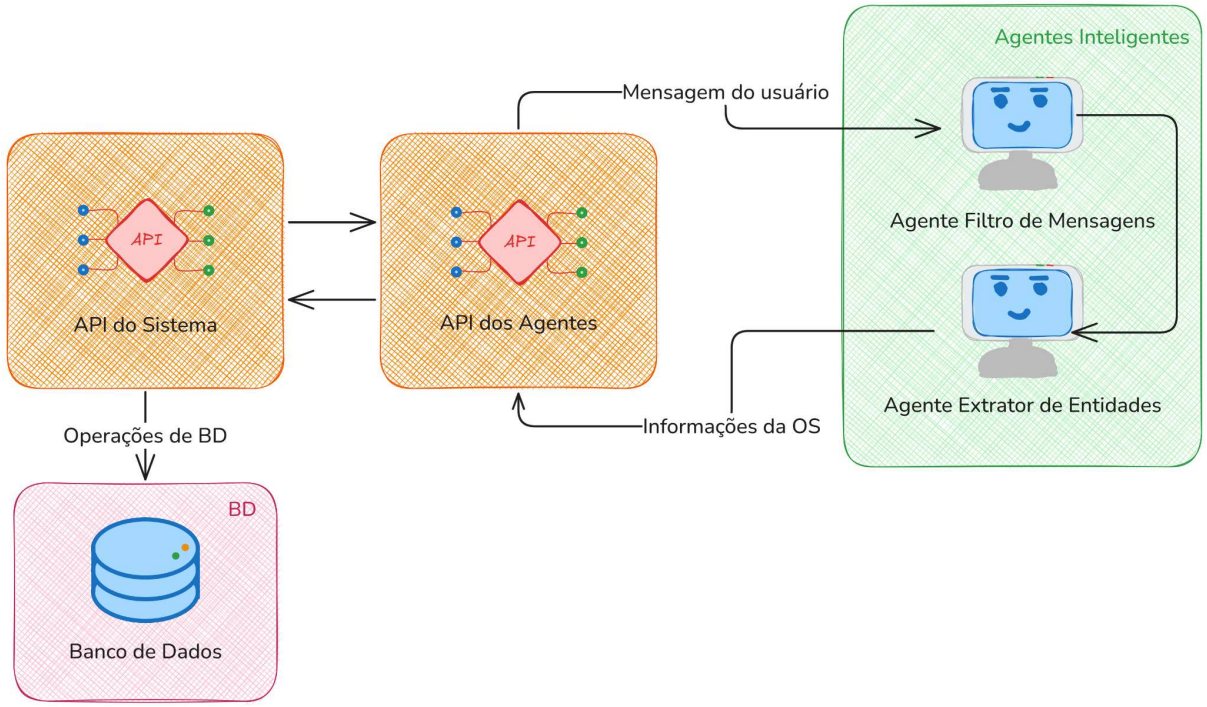
Foi escolhido o modelo de arquitetura cliente-servidor de *tier 3* (OLUWATOSIN, 2014), que consiste na divisão do sistema em camadas de aplicação e banco de dados - *server-side* (lado do servidor, em tradução livre), onde reside a lógica de negócio - e apresentação - *client-side* (lado do cliente, em tradução livre), onde o usuário vai interagir com o sistema via interface - por ser um modelo bem consolidado e adotado mesmo nos dias de hoje. Vale ressaltar que, ainda do lado do servidor, há um componente auxiliar ao componente principal onde estão localizados os agentes inteligentes. Os tópicos, bem como as tecnologias escolhidas e implementação, serão discutidos a seguir.

3.2 *Server-side*

Nesse modelo de arquitetura cliente-servidor, o servidor tem como função lidar com o processamento de dados e aplicação da lógica de negócios, bem como o banco de dados. Para este sistema, temos a camada de aplicação contendo duas APIs (*Application Programming Interfaces*) que se comunicam via HTTP (*Hypertext Transfer Protocol*), um protocolo

genérico definido pela RFC 2616 da camada de aplicação que não armazena estado e serve para comunicação entre sistemas de hipermídia (NETWORK WORKING GROUP, 1999). A arquitetura pode ser constatada na Figura 03.

Figura 03 - Representação do *back-end* da aplicação.



Fonte: Elaborado pelo autor (2025).

As componentes dessa camada serão melhor detalhadas nas seções seguintes.

3.2.1 API do sistema

De uma maneira geral, a API funciona como um mediador entre o *client* e o banco de dados, sendo uma interface entre o usuário e o banco de dados, implicando em melhor segurança e, pelo fato de ser um modelo mais descentralizado, maior escalabilidade e desempenho nesta camada (NYABUTO et al, 2024). Para a implementação desta API, modelou-se o diagrama de classes UML (*Unified Modeling Language*) e nele pode-se verificar que uma Ordem de Serviço é composta por:

- Um identificador numérico único (ID) para representá-la no banco de dados.
- Uma marcação de tempo (criadoEm) que representa a data que a ordem foi registrada.
- Uma descrição sobre o problema do qual a ordem se trata, com até 300 caracteres.
- Um *status* que indica o estado atual da ordem.
    - Dividido em três *status* que indicam que a ordem foi concluída (NAO_RESOLVIDO, PARCIALMENTE_RESOLVIDO e RESOLVIDO) e três *status* que indicam que ainda está em processamento (STANDBY, SOB_ANALISE e ENVIADO).

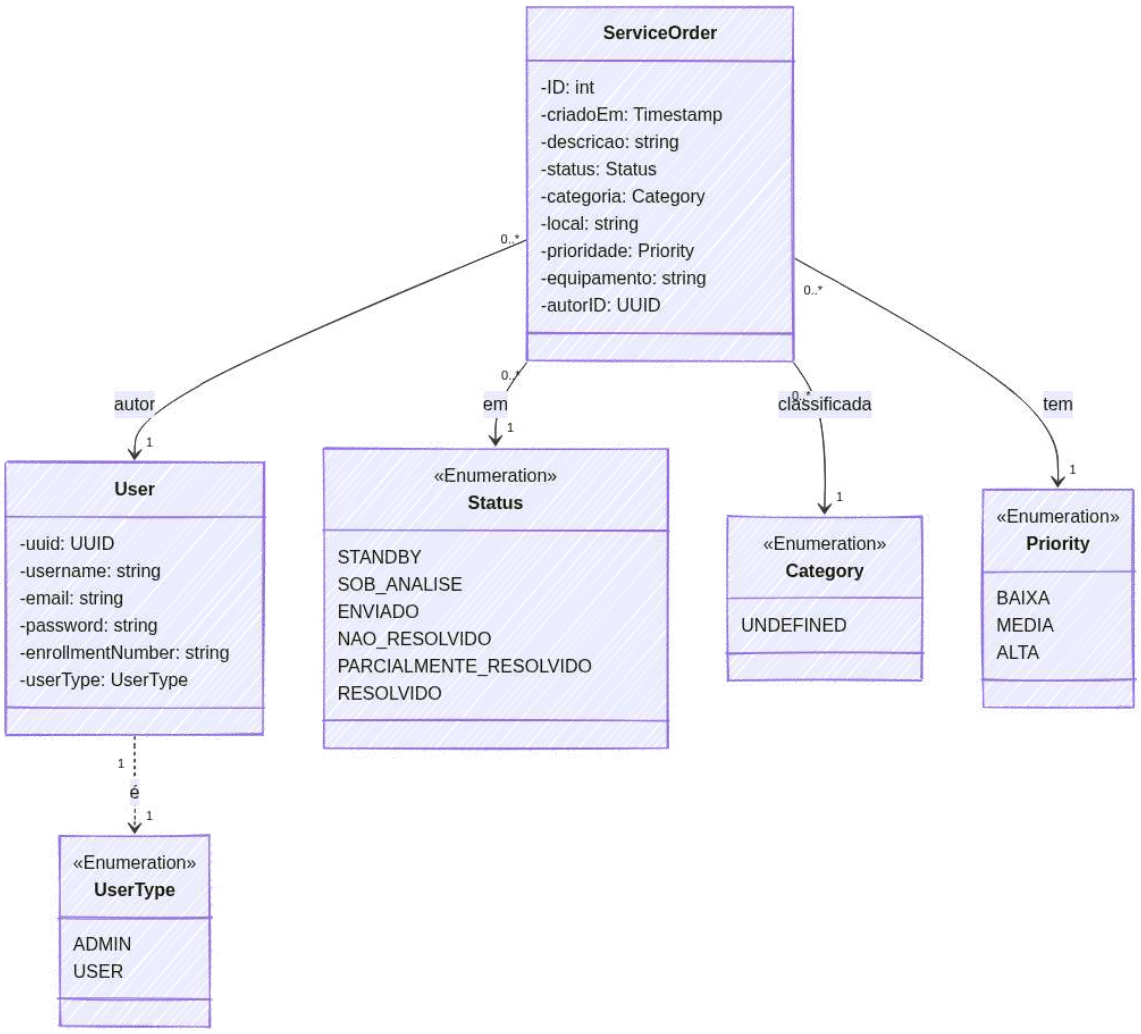
- Uma categoria que indica o grupo ao qual a ordem pertence.
- Um local ao qual a ordem se refere.
- Uma prioridade atribuída, dividida em ALTA, MÉDIA e BAIXA.
- Um equipamento, referente ao alvo do problema.
- Um ID do autor

Já o usuário, também presente no diagrama, é dotado dos seguintes campos:

- ID, que nesse caso é um UUID (*Universally Unique Identifier*).
- *E-mail*.
- Nome de usuário (*username*).
- Senha (*password*).
- Número de matrícula (*enrollment Number*).
- Tipo de Usuário (*userType*), que indica o nível de permissões do usuário.

O diagrama pode ser observado na Figura 04.

Figura 04 - Diagrama de Classes UML.



Fonte: Elaborado pelo autor (2025).

Nesse sistema, todo usuário que for registrado é automaticamente alocado como um usuário comum (*USER*). Um único usuário administrador (*ADMIN*) é criado, caso não exista, no momento que esse serviço é iniciado.

Em termos de tecnologias, existem diversas abordagens hoje em dia para o desenvolvimento de sistemas *web*, usando uma vasta combinação de *frameworks*, definidos por um conjunto de classes que constituem um projeto que pode ser reutilizado em uma maior granularidade sem necessitar conhecer sua implementação (JOHNSON e FOOTE, 1988), e bancos de dados de diferentes tipos, sendo eles SQL (*Structured Query Language*), uma abordagem mais tradicional de bancos de dados relacionais, e NoSQL (*Not Only SQL*), bastante usada em grandes montantes de dados por ser mais flexível e não seguir uma estrutura rigidamente definida.

Para essa API, escolheu-se a linguagem *Java* com o *framework Spring*, uma solução usada em aplicações de grande porte, preparando detalhes enquanto os desenvolvedores podem focar na lógica de negócio da camada de aplicação (BROADCOM, 2025), bem como dependências que já preparam o ambiente, como *Spring Security* para a camada de segurança que exige autenticação do usuário.

Para o banco de dados, optou-se por usar o *PostgreSQ*L, um banco de dados relacional *open-source* (POSTGRESQL, 1996). Para esta versão do sistema, optou-se por se desenvolver usando a *stack Servlet* do *Spring*, bastante consolidada e amplamente usada hoje em dia.

Em questão de segurança, foi adotado o *JSON Web Token* (JWT), padrão descrito pela RFC 7519 que consiste em um conjunto de pedaços de informação criptografados em uma estrutura *JSON Web Signature* (JWS) e/ou *JSON Web Encryption* (JWE), basicamente sendo uma sequência de partes *URL-safe* separados por pontos finais (IETF, 2015).

O projeto foi estruturado seguindo o modelo *Model-View-Controller* (FOWLER, 2006), que divide as responsabilidades e elementos em três camadas diferentes:

- Na *Model*, encontramos informações sobre o domínio na forma de entidades, representando os dados do banco de dados.
- Na *View*, temos a apresentação das informações, que no caso dessa API, está mais referente aos *Data Transfer Objects* (DTOs) que controlam quais os dados retornados ao *client*.
- No *Controller*, temos a lógica de negócio, onde se lida com os dados e operações através de *endpoints*.
- Adicionalmente, temos os *Services* e *Repositories* (Serviços e Repositórios, respectivamente e em tradução livre), responsáveis por realizar operações com o banco de dados.

Por fim, tem-se a conexão com a API auxiliar onde se encontram os agentes usando o *RestTemplate*, um *client* síncrono para requisições HTTP, devido à *stack* escolhida. O sistema, no geral, consegue fazer operações de CRUD - acrônimo para *Create, Read, Update and Delete* (Criar, Ler, Atualizar e Deletar em tradução livre), quatro operações básicas de uma API - para ordens de serviço, bem como o registro e login de usuários no sistema.

3.2.2 API dos agentes

Complementar à API principal, essa serve de conexão com os agentes inteligentes, objetivando possibilitar a aplicação montar uma ordem de serviço com base em uma mensagem escrita em linguagem natural.

Define-se agentes inteligentes como tudo que puder "sentir" o ambiente e "atuar" no mesmo, no caso de agentes de *software*, estes recebem uma entrada e, com base nisso, tomam decisões, geram dados ou gravam informações (RUSSELL e NORVIG, 2022). Determinou-se a criação de dois agentes, descritos como:

- **Agente Filtro de Mensagens Inválidas**: verifica se a mensagem recebida se trata de um "trote", ou seja, uma mensagem inválida. Ex.: "Deveriam instalar uns *puffies* no extensão".
- **Agente Extrator de Entidades**: recebe a mensagem válida e extrai alguns campos a partir dela, sendo esses o local, equipamento - alvo do problema - e a descrição.

Os agentes foram desenvolvidos usando a linguagem *Python*, amplamente usada na área de aprendizado de máquina (ML, do inglês *Machine Learning*), e disponibilizados com o *FastAPI*, um *framework web* que usa *Python Hints* (RAMÍREZ, 2025), uma indicação do tipo de dado se espera para uma variável. Cada um desses será detalhado nas subseções seguintes.

3.2.2.1 Agente Extrator

O primeiro agente a ser desenvolvido. Para isto, foi usada a biblioteca *spaCy* (EXPLOSION, 2025), feita para processamento de linguagem natural (NLP, do inglês *Natural Language Processing*), definido como um conjunto de tarefas como normalização de texto através da "tokenização" - separação de partes dos textos, como palavras ou partes de palavras - e "lematização" - definir se um conjunto de palavras possui um radical em comum - (SOUZA et al, 2020; JURAFSKY, D; MARTIN, J. H, 2025), que tem ênfase em transformar mensagens escritas em linguagem natural em uma mensagem compreendida pelo computador.

O agente usa um modelo de porte médio pré-treinado para língua portuguesa (*pt_core_news_md*), e transforma a entrada em um objeto especial do *spaCy* chamado *Doc*, um conjunto de *tokens* que possuem características como classe gramatical, conteúdo literal, lemas, etc. A partir desse objeto, ele pode checar, usando o *Matcher* ou *PhraseMatcher*, que são classes do *spaCy* que permitem achar uma sequência de *tokens* baseado em um conjunto de regras, por padrões definidos numa estrutura de dicionário, definida por um conjunto chave-valor.

Para o caso dessa aplicação, a escolha foi o *Matcher* cuja correspondência é feita de maneira bem similar à expressões regulares. Ao iterar sobre o *Doc*, cada ocorrência é armazenada em uma lista, uma para locais e outra para equipamentos.

Ademais, para desacoplar o código-fonte do agente, os padrões foram definidos em um arquivo JSONs (*JavaScript Object Notation*) à parte, que é carregado junto com *Matcher*

na memória. Alguns dos padrões que foram definidos no JSON podem ser constatados na Figura 05.

Figura 05 - Alguns *patterns* para equipamentos e locais definidos em arquivo *JSON*.

File: patterns.json

```
[
    {
        "id": "EQUIP_PATTERN",
        "patterns": [
            [
                {
                    "LEMMA": {
                        "IN": [
                            "computador",
                            "pc",
                            "laptop"
                        ]
                    }
                },
                {
                    "IS_DIGIT": true,
                    "OP": "*"
                }
            ],
            [
                {
                    "LOWER": {
                        "IN": [
                            "computador",
                            "pc",
                            "laptop"
                        ]
    },
    {
        "id": "LOCATION_PATTERN",
        "patterns": [
            [
                {
                    "LOWER": "lab"
                },
                {
                    "IS_SPACE": true,
                    "OP": "?"
                },
                {
                    "LOWER": {
                        "REGEX": "(?:[1-6]|i{1,3}|iv|v|vi)"
                    }
                }
            ],
            [
                {
                    "LOWER": "labcomp"
                },
                {
                    "IS_SPACE": true,
                    "OP": "?"
                },
                {
                    "IS_DIGIT": true
                }
            ],
```

Fonte: Elaborado pelo autor (2025).

O resultado, então, é um objeto contendo o local, equipamento e descrição do problema - nesse caso é a mensagem original -, conforme visto na Figura 06.

Figura 06 - *Log* do Agente Extrator em funcionamento no *FastAPI*.

```
=-=-=-=-=- Campos extraídos! -=-=-=-=-=
|> Mensagem: Lá na sala multimídia o projetor tá com a lente suja e tá tudo embaçado.
|> Local encontrado: SALA MULTIMÍDIA
|> Equipamento encontrado: PROJETOR

INFO:     127.0.0.1:53964 - "POST /validate_os HTTP/1.1" 200 OK
```

Fonte: Elaborado pelo autor (2025).

Caso o agente não encontre um local ou um equipamento, ele retornará um erro 422 do HTTP - *Unprocessable Entity*, ou entidade não processável em tradução livre - , indicando que o servidor não pode processar as instruções contidas.

3.2.2.1 Agente Filtro

Nesse agente, optou-se por utilizar os *transformers* do *Hugging Face* (HUGGING FACE, 2025), uma plataforma onde a comunidade de aprendizado de máquina - uma subárea da inteligência artificial - pode compartilhar modelos pré-treinados e *datasets* para as mais diversas aplicações, como geração e classificação de vídeos e imagens e, no caso deste agente, um modelo de classificação de texto.

Inicialmente foram coletadas algumas amostras de mensagens-exemplo para saber o que o sistema podia esperar e para o treinamento dos modelos. Foi aplicado um formulário

com as seguintes questões, direcionado à discentes de ciência da computação de todos os períodos:

- “Descreva em poucas palavras algum problema em relação aos equipamentos do LCC que você tenha percebido em sua caminhada acadêmica.”
- “Agora, pensando em um tom mais descontraído, identifique / invente algum outro problema. (Pode extrapolar o quanto quiser e nem precisa ser relacionado a um equipamento do LCC)”

Houveram 11 respostas ao formulário e, a fim de popular aumentar o volume de dados deste conjunto de amostras, alguns exemplos foram inseridos manualmente e outros foram gerados sinteticamente através de *Large Language Models* (LLMs), grandes modelos de linguagem de uso geral como *Gemini 2.5 Pro* e o *GPT-4o*, usando os exemplos existentes como base.

Para essa primeira tentativa, foram utilizadas, então, 32 mensagens originais de discentes e 99 mensagens sintéticas distribuídas nos rótulos “trote” - representando mensagens de tom mais jocoso ou mal-intencionadas - e “real” - representando reportes de problemas reais do LCC.

Para desenvolver o filtro, foram testadas duas abordagens:

- Modelos *zero-shot*, que não recebem demonstrações prévias e podem inferir sem atualizar seus pesos apenas com a instrução em linguagem natural (BROWN et al, 2020), no caso da classificação de texto, fornecendo os rótulos e a sentença a ser avaliada.
- *Fine-tuning* de um modelo, que adapta um modelo pré-treinado mais generalista a um caso de uso específico, atualizando seus pesos com base num *dataset* supervisionado - dados rotulados (BROWN et al, 2020).

Na primeira tentativa, foram testados os modelos *distilbart-mnli-12-1* e o *bart-large-mnli*, dois modelos de Inferência de Linguagem Natural (NLI, do inglês *Natural Language Inference*) que se baseiam na classificação *zero-shot*. O pseudocódigo do uso dos modelos do *Hugging Face* pode ser visto no Quadro 01.

Quadro 01 - Implementação do teste dos modelos de classificação *zero-shot*.

```
// Cria um modelo a partir de um pipeline passando como argumento o tipo de
modelo (no caso seria o zero-shot-classification) e o nome do modelo
classificador = Pipeline(“tipo_do_modelo”, “nome_do_modelo”)

# Classifica um prompt (mensagem) em alguma das labels (rótulos), retornando
cada label com uma probabilidade atribuída.
# Ex.: { labels: ["sério", "trote"], scores: [.43, .57] }
rotulos_candidatos = [“rotulo_1”, “rotulo_2”]
resposta = classificador(“mensagem_a_ser_classificada”, rotulos_candidatos)
```

Fonte: Elaborado pelo autor (2025).

Iterando sobre o *dataset*, foram testados todos os exemplos, separados em originais e sintéticos, usando como métrica a acurácia, definida pela razão das previsões corretas sobre o total de previsões. Os resultados podem ser constatados na Tabela 01.

Tabela 01 - Acurácia dos modelos de classificação de texto *zero-shot* com rótulos “sério” e “trote”.

distilbart-mnli-12-1		**bart-large-mnli**	
Dados originais	**Dados sintéticos**	**Dados originais**	**Dados sintéticos**
0.5	-	0.625	0.434

Fonte: Elaborado pelo autor (2025).

Conforme visto acima, os resultados foram insatisfatórios, ficando em torno de 50% de acurácia. Com base nessa constatação, houveram duas hipóteses: os rótulos estavam muito genéricos para o modelo inferir ou essa abordagem não é a melhor para este caso. A fim de testar a primeira hipótese os rótulos foram reformulados, agora sendo:

- **Reporte_de_problema**: originalmente o “sério”, referente a uma mensagem que deve ser registrada como ordem de serviço.
- **Solicitação_incompleta**: referente a uma mensagem que, por mais que pareça válida, carece de informações para compor uma ordem, no caso o local ou o equipamento.
- **Solicitação_inválida**: originalmente o “trote”, referente a uma mensagem que não diz respeito ao propósito do sistema de registrar ordens de serviço para equipamentos

Com a mudança de rótulos, cada mensagem foi reavaliada para um destes três. Para essa segunda tentativa, foram usadas 30 mensagens originais e 60 exemplos sintéticos - gerados após esta rotulação dos dados originais. Os resultados podem ser vistos na Tabela 02.

Tabela 02 - Acurácia dos modelos de classificação de texto *zero-shot* com rótulos "Reporte_de_problema", "Solicitação_incompleta" e "Solicitação_inválida".

distilbart-mnli-12-1		**bart-large-mnli**	
Dados originais	**Dados sintéticos**	**Dados originais**	**Dados sintéticos**
0.4	0.567	0.267	0.517

Fonte: Elaborado pelo autor (2025).

Percebe-se uma piora significativa com os dados originais e uma melhora com os dados sintéticos para ambos os modelos e, devido a isso, optou-se pelo o *fine-tuning*.

O modelo escolhido foi o *BERTimbau Base*, baseado no modelo BERT, um modelo bidirecional fundamentado na arquitetura de *Transformer Encoder*, uma rede neural

estruturada com mecanismos de auto-atenção e de *multi-head*[4], onde atenção é, em essência, uma forma de inferir o significado de um *token* com base nos *tokens* relacionados a este dentro de um contexto (SOUZA et al, 2020; JURAFSKY, D; MARTIN, J. H, 2025). O pseudocódigo do procedimento de *fine-tuning* é descrito no Quadro 2.

Quadro 02 - Pseudocódigo do procedimento de *Fine-tuning*.

```
// Carrega o dataset na memória com um determinado conjunto de características
rotulos = [“rotulo_1”, “rotulo_2”, ..., “rotulo_n”]
caracteristicas = Caracteristicas(
    rotulo = rotulos,
    numero_de_rotulos = 3, // Declara explícitamente
    tipo_de_dado = “texto”, // Indica o tipo de dado que tem no dataset
    ...    // Outras características
)
dataset = Dataset(“caminho_para_o_arquivo”, caracteristicas)

// Divide dataset em dados de treino, teste e validação
dados_separados = dataset.dividir_conjunto(
    amostra_teste = 0.1,
    amostra_validacao = 0.1,
    dividir_pela_coluna = “rotulo”
)

// Para esse exemplo, ficam 80% dos dados para treino, 10% para teste e 10% para
validação

// Cria uma relação para guardar os dados separados
separacao_datasets = {
    "treino": dados_separados["treino"],
    "teste": dados_separados["teste"],
    "validacao": dados_separados["validacao"]
})

// Carrega o modelo e o tokenizador de um modelo pré-treinado
tokenizador = AutoTokenizador.do_modelo(“nome_do_modelo”)
modelo = AutoModeloParaAlgo.do_modelo(“nome_do_modelo”, numero_de_rotulos = 3)

// Tokeniza as partes do dataset
parametros = [“parametro_1”, “parametro_2”, ..., “parametro_n”]
datasets_tokenizados = [] // Inicializa com uma lista vazia
para amostra em separacao_datasets faça
    datasets_tokenizados.adiciona(tokenizador(amostra, parametros))

// Você pode fazer um pré-processamento, como remover ou renomear colunas nessa
etapa

// Carrega uma métrica
metrica = Metrica.carrega(“nome_da_metrica”)

// Função para computar a métrica de avaliação
função computar_metrica(predicao_da_avaliacao):
  predicoes, rotulos = predicao_da_avaliacao
```

[4] *head* é um termo comum quando se trata de *transformers* para se referir à camadas estruturadas específicas. No caso do *multi-head*, são várias funções de atenção - *head* - que operam em camadas paralelas no mesmo nível (JURAFSKY, D; MARTIN, J. H; 2025).

```
  predicoes = indice_valor_maximo(predictions)
  retorna metrica.computa(predicoes, rotulos)

// Define hiperparâmetros para o treinamento do modelo
hiperparametros_treinamento = {
    estrategia_avaliacao: “tipo_estrategia”,
    epocas_treinamento: numero_epocas,
    metrica_de_avaliacao: metrica,
    ...    // Outros parâmetros
}

// Inicializando o Trainer
treinador = Treinador(
    modelo,
    hiperparametros_treinamento,
    datasets_tokenizados["treino"],
    datasets_tokenizados["validacao"],
    tokenizador,
    computar_metrica,
    ...    // Outros parâmetros
)

treinador.treinar()
```

Fonte: Elaborado pelo autor (2025).

Para esta tentativa, que exige um volume maior de amostras, foram usadas 257 amostras, sendo as 30 mensagens originais mais 227 mensagens sintéticas. Foram testadas duas métricas de avaliação diferentes:

- *f1-score*, que balanceia casos de falsos positivos e falsos negativos
- *precision*, que tem ênfase em casos de falsos positivos

Os resultados podem ser constatados na Tabela 03.

Tabela 03 - Pontuação do *fine-tuning* do *BERTimbau Base* para cada métrica de avaliação.

F1-score	***Precision***
0.88	0.93

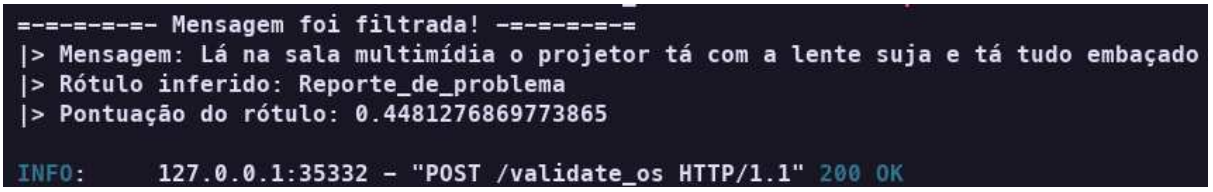
Fonte: Elaborado pelo autor (2025).

A pontuação - o que o modelo conseguiu acertar do conjunto de validação - se mostrou bastante aceitável nas duas métricas, mas considerando o caso deste agente, que está verificando se uma mensagem é inválida, a métrica final escolhida foi a *precision*, pois é mais danoso à aplicação se uma mensagem válida não passar no filtro, ou seja, um falso positivo.

Uma vez adaptado, o modelo foi carregado na aplicação como parte do agente. Seu retorno é uma tupla com o rótulo que teve a maior pontuação e a pontuação, que, na prática, representa a probabilidade de cada rótulo para a mensagem enviada. Como são 3 rótulos, foi determinado que para uma mensagem ser inválida ela deve ser classificada com o rótulo “Mensagem_inválida” e ter um *score* maior que 0.35 - valor empírico indicando que a

pontuação superou, por muito, um terço, que seria 0.33 -, retornando um erro HTTP 422. A Figura 07 exemplifica um caso onde a mensagem é válida.

Figura 07 - *Log* do Agente Filtro em funcionamento no *FastAPI*.



Fonte: Elaborado pelo autor (2025).

Dessa forma, a mensagem pode ser repassada ao agente extrator que formará uma entidade a partir do conteúdo do texto e, por fim, repassará à API principal, onde a Ordem de Serviço será armazenada no banco de dados.

3.3 *Client-side*

Sobre o modelo cliente-servidor, o cliente tem a responsabilidade conectar o usuário ao servidor, realizando requisições a este e apresentando as respostas em uma interface gráfica na qual o usuário pode interagir, o que será discutido nas subseções seguintes.

3.3.1 Interface do usuário

A interface foi projetada visando dispositivos *desktop*. Inicialmente, foi feito um modelo de baixa fidelidade no papel, chamado *wireframe*. Posteriormente, esse foi transposto ao meio digital, contando com duas visualizações diferentes: uma para o usuário discente - com uma página para registro de ordens de serviço através de uma mensagem escrita em linguagem natural e uma para listagem das ordens enviadas pelo usuário - e do usuário administrador - apenas uma página com a listagem de todas as ordens de serviço que foram registradas. A partir do *wireframe*, foi feito um protótipo no *Figma*, um serviço para, entre outras atividades, projetar *designs*.

Para a implementação, foi escolhido o *Vue.js*, um *framework Javascript* progressivo, performático e versátil para construir interfaces *web* (YOU, 2014). Em complemento, foi utilizado o *Axios*, um *client* HTTP baseado em *promises*, que são objetos especiais do *Javascript* que conectam o código de produção - no caso a API da aplicação - ao código de consumo - que no caso é o *Vue* - quando o resultado estiver pronto (ZABRISKIE, 2014), e o Pinia, uma biblioteca para gerenciar *stores*, permitindo compartilhar recursos carregados no *client* entre as páginas/componentes (MOROTE, 2019). A parte referente ao CSS foi feita usando *Tailwind*, um *framework* do CSS que permite a personalização com base em nomes representativos de classes (TAILWIND, 2025), com auxílio de LLMs para o mapeamento de cada classe com base no protótipo, salvo algumas exceções.

Para preservar o fluxo da aplicação do lado do administrador, o usuário comum pode excluir ou editar as ordens por ele registradas no período de um dia após o registro destas. Após isso, apenas o administrador poderá realizar estas operações. Dessa forma, garante-se que a ordem estará definida no momento em que o administrador for realizar a curadoria para, efetivamente, montar ordens de serviço.

4 RESULTADOS OBTIDOS

A partir do sistema em funcionamento, foram feitas validações com base em casos de uso com usuários comuns - discentes do curso de Ciência da Computação sem definição de um período específico.

Em primeiro momento, foi realizada uma verificação e validação da interface com base nas heurísticas de Nielsen. Conforme descrito por (ROGERS; SHARP; PREECE, 2013 apud BARRETO et al., 2018, p. 156), as heurísticas de Nielsen são um meio para avaliar uma interface com base em princípios de usabilidade, sendo estas:

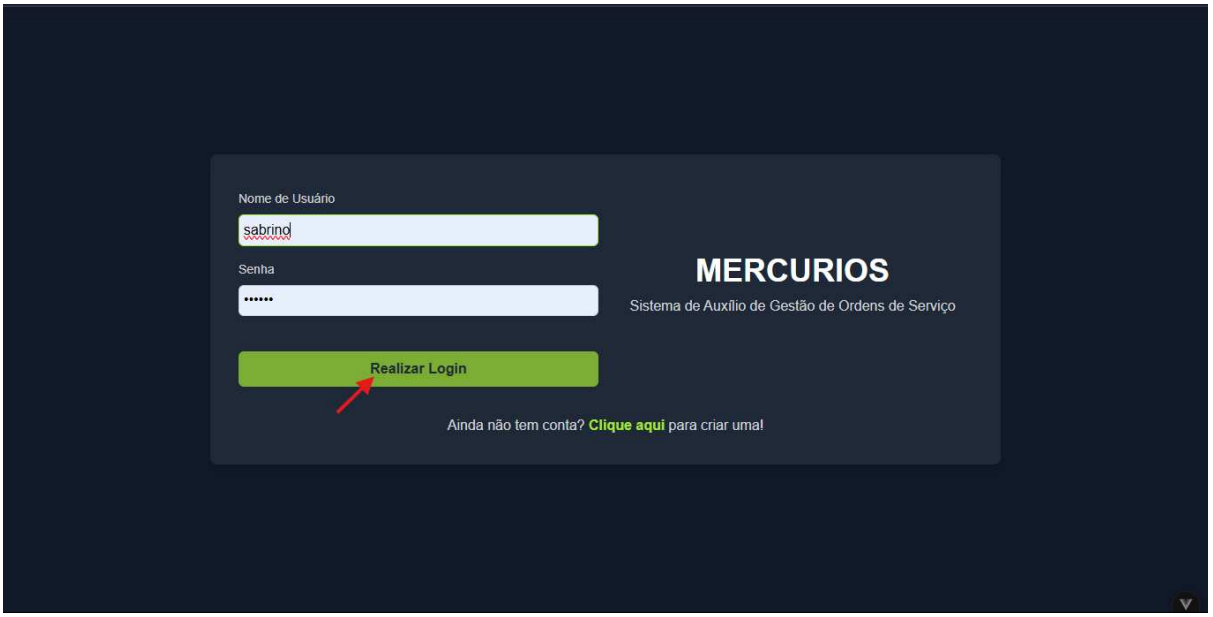
1. O sistema deve explicitar os status das ações por meio de feedback em tempo apropriado (Visibilidade).
2. O sistema deve usar termos que sejam familiares ao usuário (Correspondência ao mundo real).
3. O sistema deve fornecer uma saída caso o usuário cometa algum engano (controle e liberdade).
4. O sistema deve ser consistente e manter padrões ao longo da aplicação para o usuário não ter de inferir que dois termos significam o mesmo (padronização).
5. O sistema deve evitar erros e, se ocorrerem, apresentar mensagens de confirmação ao usuário antes que a ação aconteça (prevenção de erros).
6. O sistema deve ser intuitivo e evitar a memorização do usuário, com objetos e funcionalidades visíveis, espalhando dicas e instruções comuns (reconhecimento > lembrança).
7. O sistema deve conter aceleradores - atalhos e teclas de abreviação - personalizáveis que facilitem o uso do sistema tanto a novatos quanto mais experientes (eficiência).
8. O sistema deve manter apenas elementos relevantes para evitar que o usuário se confunda, assim focando nas funcionalidades pertinentes (minimalismo).
9. O sistema deve ser claro e direto com as mensagens de erro, ajudando o usuário a se recuperar de tais erros, identificando e sugerindo soluções (recuperação de erros).
10. O sistema deve prover documentação para ser consultada em caso de dúvidas, mesmo se o sistema estiver sendo bem claro, trazendo um passo a passo direto e resumido de cada funcionalidade (documentação).

Na análise, foi observado que três heurísticas não foram supridas, sendo estas a prevenção de erros (5) - não há forma do usuário cancelar ou editar a ordem de serviço até ela ser salva no banco de dados - , eficiência (7) - faltam facilitadores no sistema, como o uso da tecla "*Enter*" para enviar a mensagem na tela inicial em vez de clicar no botão - e documentação (10) - não existe uma documentação que possa ser acessada pelo usuário com informações detalhadas sobre o funcionamento do sistema. Com base nisso, foram desenvolvidos casos de uso simples para a aplicação para uma validação do sistema com possíveis usuários, discorridos à seguir.

4.1 Cenário de uso I: Registrar uma Ordem de Serviço

O discente se autentica, realizando login no sistema com uma conta previamente criada, informando seu nome de usuário e senha (Figura 08).

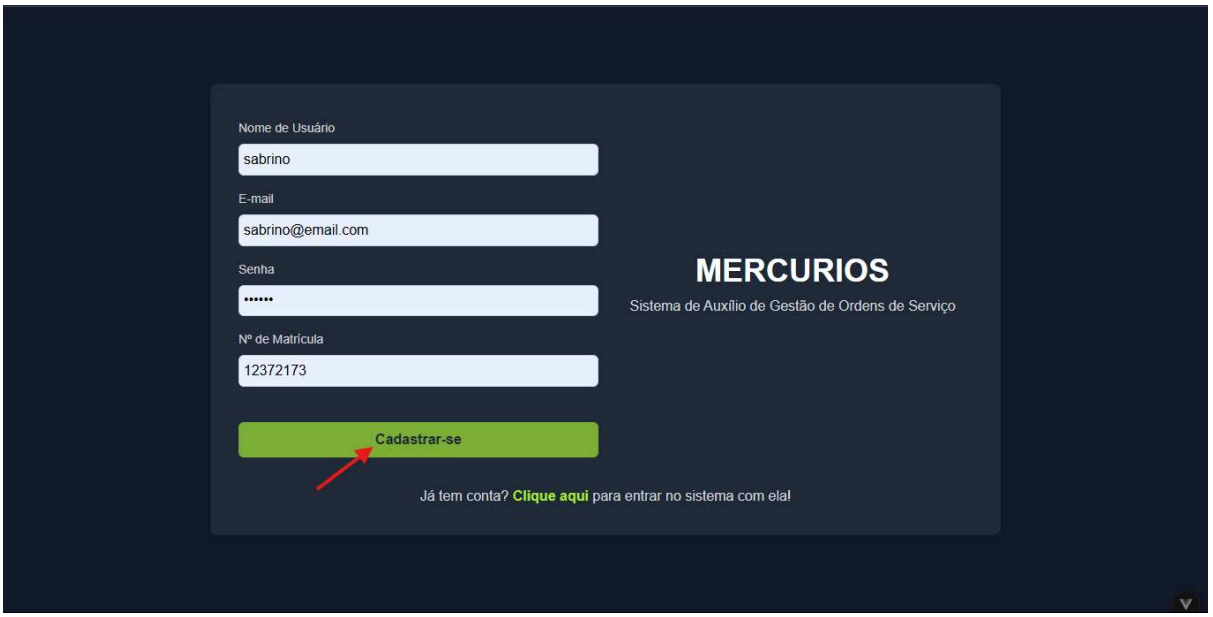
Figura 08 - Usuário se autentica na tela de *login* do sistema.



Fonte: Elaborado pelo autor (2025).

Caso não possuam uma conta, ele clica em “Clique aqui” na mensagem do rodapé e é redirecionado para uma tela de cadastro onde ele irá preencher um formulário com Nome de Usuário, Email, Senha e Número de Matrícula (Figura 09), retornando a tela de *login* ao fim da operação em caso de sucesso.

Figura 09 - Usuário se cadastra na tela de cadastro do sistema.



Fonte: Elaborado pelo autor (2025).

Após se autenticar, é redirecionado para a tela onde pode registrar uma ordem de serviço, com uma descrição de como usar o sistema (Figura 10).

Figura 10 - Tela inicial do usuário comum (discente).

Fonte: Elaborado pelo autor (2025).

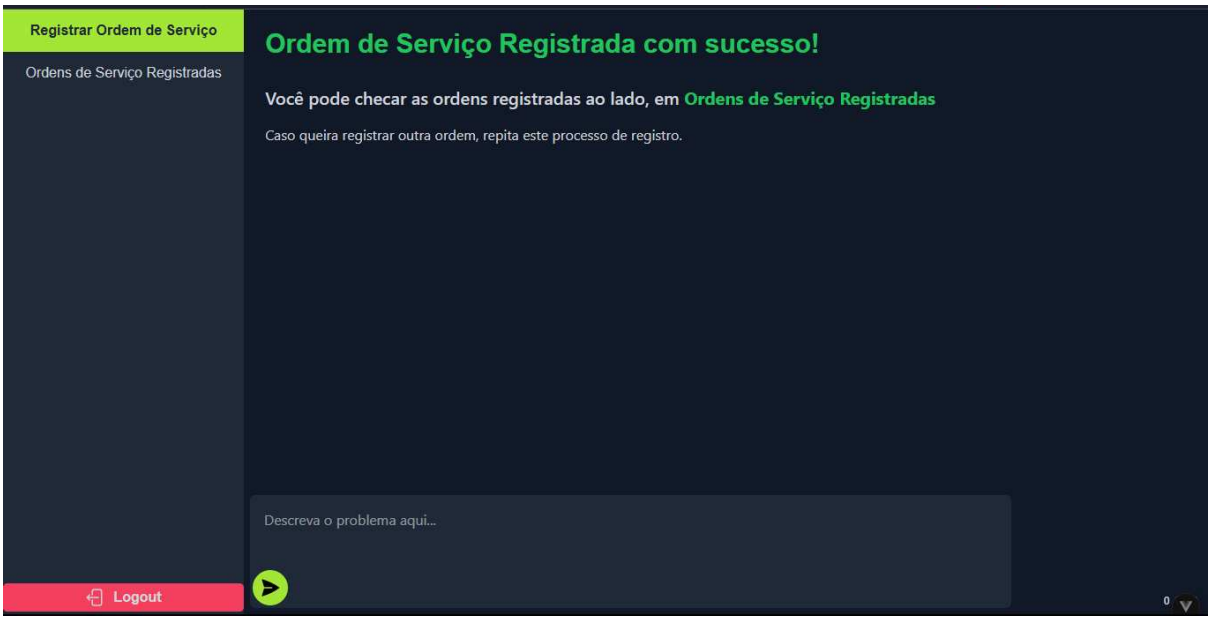
Nesse momento, ele descreve o problema, seguindo as instruções escritas nessa tela inicial, na caixa de texto e clica no botão para enviar sua solicitação. Um indicador de carregamento deve aparecer abaixo do passo a passo enquanto a mensagem é processada pelos agentes e registrada no banco de dados (Figura 11).

Figura 11 - Usuário descrevendo ordem válida, enviando e aguardando processamento pelos agentes.

Fonte: Elaborado pelo autor (2025).

Ao finalizar, a mensagem que antes mostrava um tutorial deve apresentar um indicador de sucesso, em tom de verde, e dar o direcionamento ao usuário para onde ele pode verificar a ordem registrada (Figura 12).

Figura 12 - Mensagem indicando o sucesso da operação.



Fonte: Elaborado pelo autor (2025).

4.2 Cenário de uso II: Tentar adicionar uma Ordem de Serviço inválida

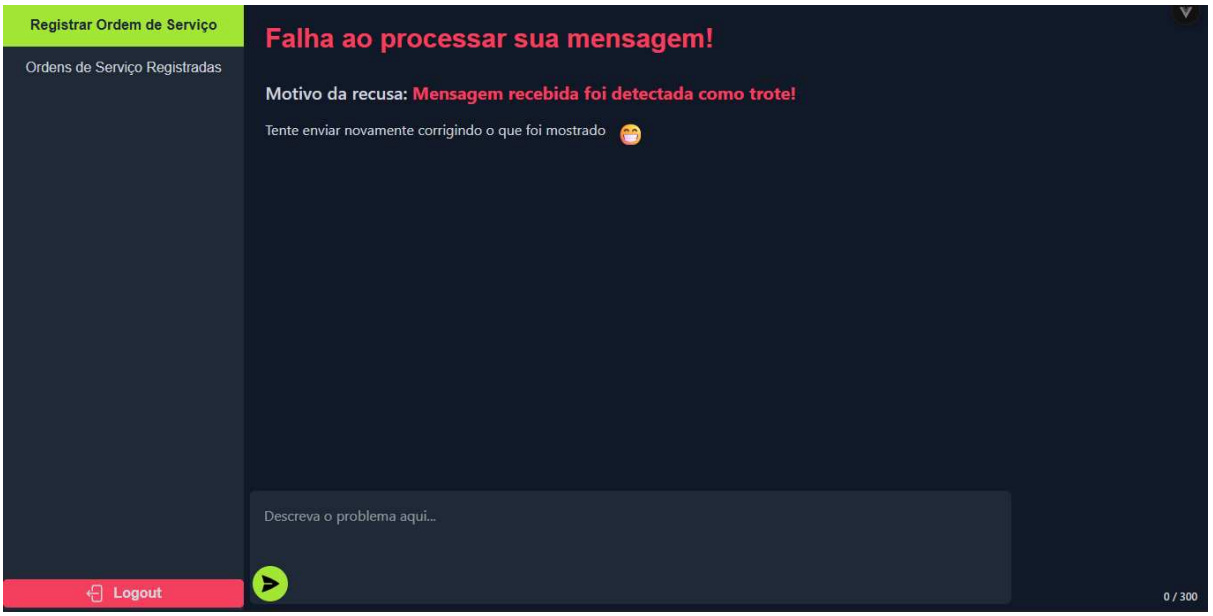
Similar ao anterior, o discente passa pelo processo de autenticação e é redirecionado a tela inicial. Para este caso, ele descreve uma mensagem que deve ser considerada inválida, à título de exemplo por fugir do escopo esperado já que é absurda, e a envia para ser revisada pelos agentes (Figura 13).

Figura 13 - Usuário descrevendo ordem inválida e clicando no botão de envio.

Fonte: Elaborado pelo autor (2025).

Um indicador de carregamento deve aparecer enquanto a mensagem é processada e, ao finalizar, a mensagem que mostrava o tutorial muda para uma em vermelho, informando que o processamento não foi possível junto ao motivo da recusa (Figura 14).

Figura 14 - Mensagem indicando a falha na operação.



Fonte: Elaborado pelo autor (2025).

4.3 Cenário de uso III: Editar uma Ordem de Serviço

O discente autenticado, na página inicial, clica na barra de navegação - menu lateral - no item nomeado como "Ordens de Serviço Registradas", que, como diz o nome, deve redirecionar para as ordens que foram registradas por ele (Figura 15).

Figura 15 - Usuário clicando na barra de navegação para ver as ordens registradas.

Fonte: Elaborado pelo autor (2025).

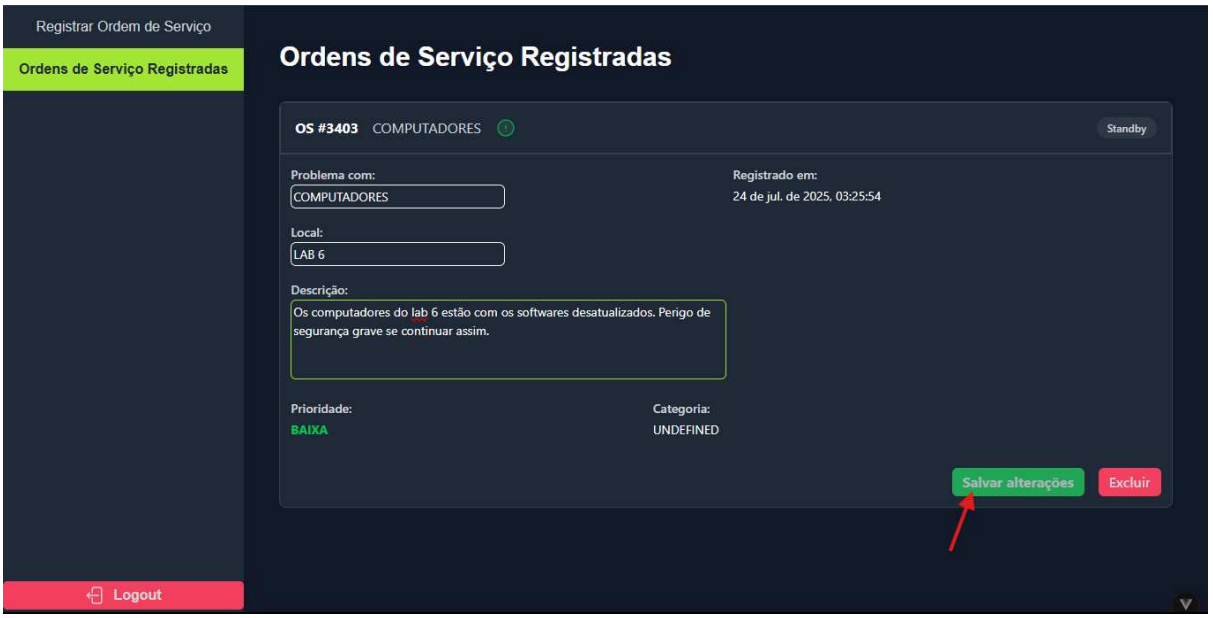
O usuário é redirecionado a uma tela na qual pode verificar as ordens que ele registrou, com uma listagem em que cada item contém como informações principais o ID da Ordem de Serviço registrada, o nome do equipamento, um ícone que representa a prioridade e uma *badge* que representa o *status* atual (Figura 16).

Figura 16 - Tela de ordens registradas pelo usuário.

Fonte: Elaborado pelo autor (2025).

Selecionando alguma das ordens que tenha sido registrada nas últimas 24 horas - condição que foi determinada para edição e remoção pelo usuário comum -, o usuário deve conseguir ver as informações da ordem em campos de um formulário, editando campos à sua escolha e clicando no botão de “Salvar Alterações” (Figura 17).

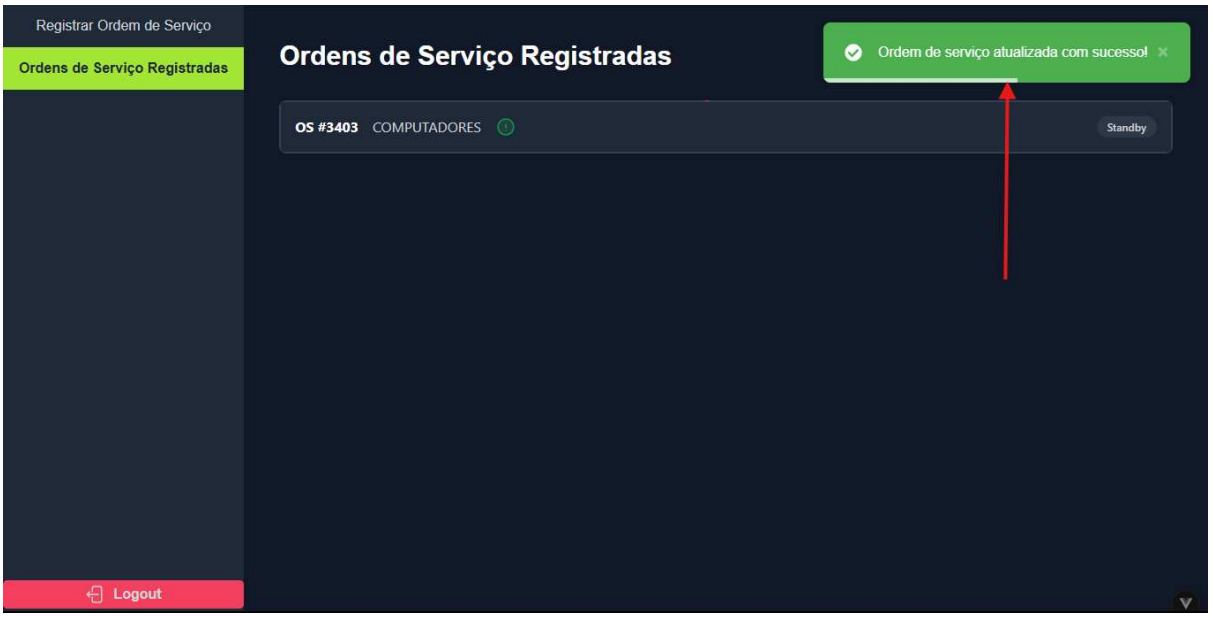
Figura 17 - Usuário edita informações de uma ordem e clica para salvá-las.



Fonte: Elaborado pelo autor (2025).

Após concluir a operação de atualização no banco de dados, uma mensagem deve aparecer no canto superior direito informando que a ordem de serviço foi atualizada com sucesso (Figura 18), podendo ser verificada pelo usuário caso clique novamente na ordem para verificar seu conteúdo.

Figura 18 - Mensagem informando o sucesso da operação de atualização.

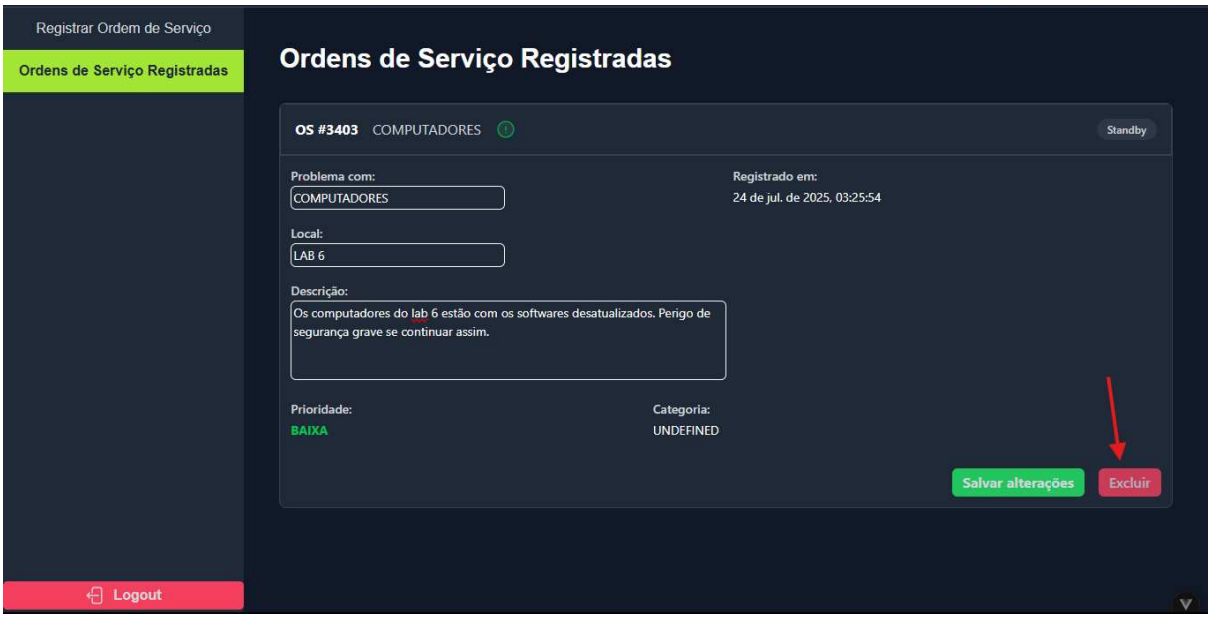


Fonte: Elaborado pelo autor (2025).

4.4 Cenário de uso IV: Excluir uma Ordem de Serviço

O usuário autenticado, na tela de ordens registradas, seleciona alguma das ordens de serviço, que tenham sido criadas no período de um dia, que deseja remover, e, então, clica no botão de “Excluir” (Figura 19).

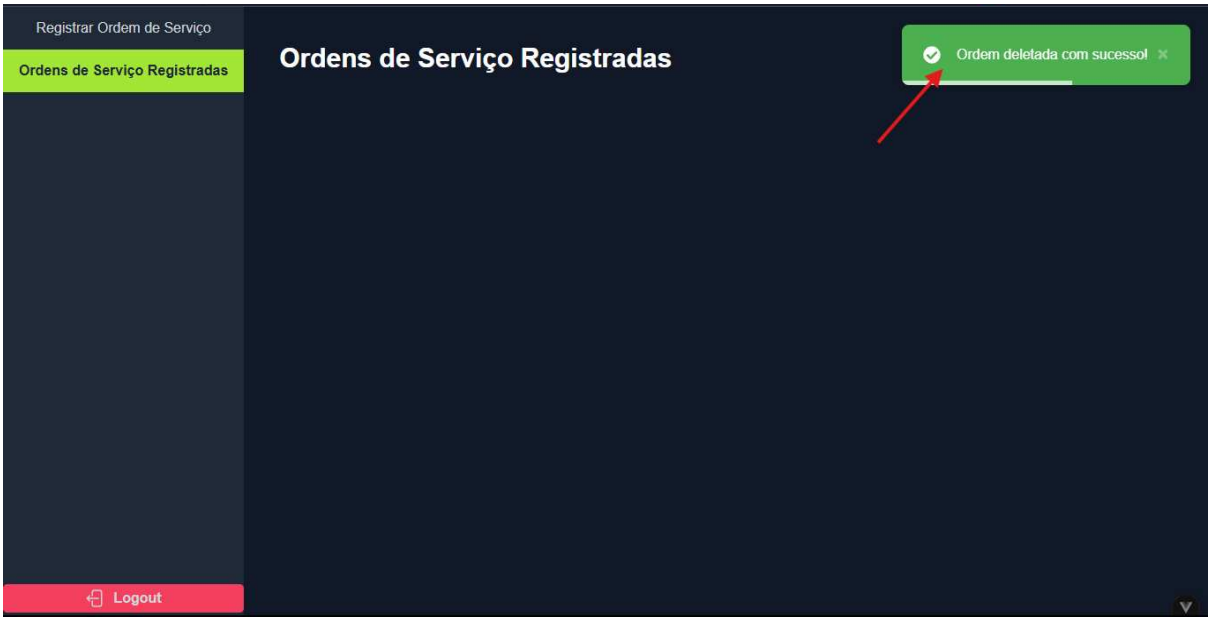
Figura 19 - Usuário clica em botão para excluir uma ordem selecionada.



Fonte: Elaborado pelo autor (2025).

Após alguns segundos, uma mensagem deve aparecer no canto superior direito informando que a mensagem foi removida com sucesso (Figura 20).

Figura 20 - Mensagem informando sucesso na operação de exclusão.



Fonte: Elaborado pelo autor (2025).

Ao apresentar aos usuários em potencial, foi possível comprovar o que havia observado segundo a avaliação das heurísticas, bem como se coletou *feedback* que pode servir para melhorias futuras. A etapa de validação também serviu para identificar a necessidade de ajustes no filtro. O modelo inicial classificava algumas das ordens inválidas como válidas. Após uma recalibragem do critério de classificação - o limiar da pontuação para considerar uma mensagem como inválida - o filtro se mostrou mais rigoroso, o que indicou um desafio de balanceamento entre especificidade e sensibilidade, também evidenciando a revisão dos dados de teste do *dataset*.

Os seguintes pontos foram levantados pelos usuários:

- "[...] Tá considerando bastante uma escrita formal e seria bom que desse pra escrever de uma forma mais informal", "Acho que eu entendi. Ele pega só se escrever no presente [...]" o que indica um possível sobreajuste do modelo a uma linguagem menos coloquial, o que pode dificultar o uso do sistema considerando que este foi pensado para diminuir burocracias e formalidades.
- "[...] Seria legal ter uns atalhos…", conforme já observado na análise.
- "Devia haver a possibilidade de corrigir uma mensagem antes do envio…", algo que também foi observado com base nas heurísticas de Nielsen.
- "Ele podia ser mais claro que se trata só desse bloco", referindo-se a redução do escopo ao LCC nessa versão do sistema.
- "Também podia colocar algum tipo de exemplo, uma frase modelo para indicar o que o sistema espera".

- “É bem comum e seria bom ter uma área do usuário para poder editar e ver as informações, daí dava pra saber quem tá usando o sistema”.
- “Seria interessante se tivesse uma lista dos locais”, considerando que o sistema tem um viés mais do lado técnico, focando em equipamentos, e não contempla todas as salas do LCC.
- “Para evitar uma recusa imediata, seria interessante uma terceira via como uma fila de mensagens [...] que não fosse nem aceitação nem recusa nesse filtro, para uma revisão mais manual”.
- “Até por questões de UX [...] seria bom ter alguma indicação da ordem que foi salva invés de só mudar a *label*, sabe?”.

Este *feedback* coletado reforçou o que foi levantado na avaliação heurística e ajudou a verificar alguns pontos sobre o filtro, evidenciando a necessidade de trabalhar mais nele. As sugestões foram consideradas como sugestão de trabalhos futuros.

5 CONSIDERAÇÕES FINAIS

O projeto que foi proposto para este trabalho visou, desde sua concepção enquanto ideia, diminuir o nível de burocracia para se registrar ordens de serviço e dar poder à comunidade discente para poder reportar eventuais problemas em equipamentos, devido ao maior contato que esses têm com o laboratório no geral. Como o sistema foi projetado com muitos componentes, a complexidade de integração foi refletida durante o desenvolvimento.

A etapa de validação, realizada com usuários discentes, serviu para identificar alguns comportamentos inesperados no agente filtro, possivelmente relacionado com a falta de diversidade dos dados sintéticos que serviram de base para o modelo.

Para além, também destacaram-se possíveis melhorias em termos de experiência de usuário (UX, do inglês *User eXperience*), como um *feedback* mais detalhado do sistema acerca dos processamentos realizados ou atalhos do teclado. Ainda assim, o sistema demonstrou ser funcional para seu propósito e a interface foi avaliada positivamente pelos usuários nos testes de usabilidade.

Como sugestões para trabalhos futuros, cogita-se a refatoração da API feita no *Spring* para a *stack Reactive*, que é mais moderna e pode combinar com a ideia de microsserviços, bem como a implementação de melhorias, o uso de *brokers* de mensageria como o *RabbitMQ* para uma maior assincronicidade e desacoplamento entre o cliente e o servidor, também podendo auxiliar na terceira via citada nos *feedbacks*, que era parte do projeto original, bem como a implementação de mais agentes, como um avaliador de prioridade que as define com base em ordens registradas anteriormente.

Adicionalmente, também pode se considerar a implementação das na visão do administrador, permitindo visualizar todas as ordens registradas, bem como a implementação de *dashboards* que extraiam informações úteis com base nas ordens registradas, auxiliando a chefia de departamento a tomar decisões administrativas.

REFERÊNCIAS

APEXCHARTS. **ApexChart.js: Modern & Interactive Open-source Charts** . Disponível em: https://www.apexcharts.com/. Acesso em: 17 de julho de 2025.

BARRETO, Jeanine dos S. et al. **Interface humano-computador**. Porto Alegre: SAGAH, 2018. E-book. p.156. ISBN 9788595027374. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9788595027374/. Acesso em: 18 de julho de 2025.

BROADCOM. **Spring Boot**. Disponível em: https://spring.io/projects/spring-boot. Acesso em: 16 de julho de 2025.

BROWN, Tom et al. **Language models are few-shot learners**. Advances in neural information processing systems, v. 33, p. 1877-1901, 2020. Disponível em: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf. Acesso em: 09 de agosto de 2025.

EXPLOSION. **spaCy: Industrial-strength Natural Language Processing (NLP) in Python**. Disponível em: https://spacy.io/. Acesso em: 16 de julho de 2025.

FOWLER, Martin. **Padrões de arquitetura de aplicações corporativas**. Porto Alegre: **Bookman**, 2006. E-book. p.309. ISBN 9788577800643. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9788577800643/. Acesso em: 25 de julho de 2025.

HUGGING FACE. **Hugging Face - The AI community building the future.** Disponível em: https://huggingface.co/. Acesso em: 24 de julho de 2025.

IETF. **JSON Web Token (JWT)**. Disponível em: https://datatracker.ietf.org/doc/html/rfc7519. Acesso em: 17 de julho de 2025.

JOHNSON, Ralph E.; FOOTE, Brian. **Designing reusable classes**. **Journal of object-oriented programming, v. 1, n. 2, p. 22-35**, 1988. Disponível em: https://www.academia.edu/50283400/Designing_Reusable_Classes. Acesso em: 09 de agosto de 2025.

JURAFSKY, Daniel; MARTIN, James H. **Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models. 3. ed.** [2025]. Disponível em: https://web.stanford.edu/~jurafsky/slp3. Acesso em: 19 de julho de 2025.

MOROTE, Eduardo S. M.. **Pinia: The intuitive store for Vue.js**. Disponível em: https://pinia.vuejs.org/. Acesso em: 16 de julho de 2025.

NETO, Marcos J. Ferreira et al. **Universidade Digital: Gerenciamento de Ordem de Serviço.** Disponível em: https://central.arapiraca.ufal.br/nti/dist/pdf/Gerenciamento_de_Ordem_de_Servi%C3%A7o.pdf. Acesso em: 09 de agosto de 2025.

NETWORK WORKING GROUP. **RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1**. Disponível em: https://datatracker.ietf.org/doc/html/rfc2616. Acesso em: 17 de julho de 2025.

NYABUTO, Mr Geofrey Mwamba; MONY, V.; MBUGUA, Samuel. **Architectural review of client-server models. International journal of scientific research and engineering trends, v. 10, n. 1, p. 139-143**, 2024. Disponível em: https://www.academia.edu/download/111170439/Architectural_Review_of_Client_Server_Models.pdf. Acesso em: 09 de agosto de 2025.

OLUWATOSIN, Haroon Shakirat. **Client-server model**. **IOSR Journal of Computer Engineering, v. 16, n. 1, p. 67-71**, 2014. Disponível em: https://www.researchgate.net/profile/Shakirat-Sulyman/publication/271295146_Client-Server_Model/links/5864e11308ae8fce490c1b01/Client-Server-Model.pdf. Acesso em: 09 de agosto de 2025.

THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. **PostgreSQL: The World's Most Advanced Open Source Relational Database**. Disponível em: https://www.postgresql.org/. Acesso em: 17 de julho de 2025.

RAMÍREZ, Sebastián. **FastAPI framework: high performance, easy to learn, fast to code, ready for production**. Disponível em: https://fastapi.tiangolo.com/. Acesso em: 16 de julho de 2025.

RUSSELL, Stuart J.; NORVIG, Peter. **Inteligência Artificial: Uma Abordagem Moderna. 4. ed**. Rio de Janeiro: GEN LTC, 2022. E-book. p.i. ISBN 9788595159495. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9788595159495/. Acesso em: 18 de julho de 2025.

SOUZA, Fábio; NOGUEIRA, Rodrigo; LOTUFO, Roberto. **BERTimbau: pretrained BERT models for Brazilian Portuguese**. In: **Brazilian conference on intelligent systems**. Cham: Springer International Publishing, 2020. p. 403-417. Disponível em: https://repositorio.unicamp.br/acervo/detalhe/1161906. Acesso em: 09 de agosto de 2025.

TAILWIND LABS INC.. **Tailwind CSS: Rapidly build modern websites without ever leaving your HTML**. Disponível em: https://tailwindcss.com/. Acesso em: 16 de julho de 2025.

UFERSA. **Ciência da Computação - LCC**. Disponível em: https://cc.ufersa.edu.br/lcc/. Acesso em: 24 de julho de 2025.

YOU, Evan. **Vue.js: The Progressive JavaScript Framework**. Disponível em: https://vuejs.org/. Acesso em: 16 de julho de 2025.

ZABRISKIE, Matt. **Axios: Promise based HTTP client for the browser and node.js**. Disponível em: https://axios-http.com/. Acesso em: 16 de julho de 2025.