Classificação de Tickets usando LLM (gpt)

Visão Geral

Serviço FastAPI com uma rota única — POST `/classificacao` — que:

- lê um CSV de tickets;
- para cada linha, gera resumo (≤ 3 frases) e classificação em uma única categoria usando OpenAI;
- usa, além do texto do chamado (`text_column`), os campos de contexto `canal` e `prioridade` (opcionais), que são injetados no prompt para melhorar a decisão;
- retorna JSON por item e, opcionalmente, salva um CSV com `summary` e `predicted_category`.

Categorias utilizadas (exemplo típico)

O classificador opera sobre uma lista fechada definida pelo usuário. Um conjunto frequente em Service Desk é:

- **Acesso/Senha** — problemas de autenticação, bloqueio, expiração de senha.
- **Falha de Sistema** — indisponibilidade, erro de aplicação, degradação de serviço.
- **Solicitação de Serviço** — pedidos de criação/alteração de acesso, provisionamento, novas funcionalidades.
- **Informação/Dúvida** — pedidos de esclarecimento, orientações de uso.
- **Infraestrutura/Rede** — Wi-Fi, VPN, latência, cabeamento, hardware de rede.

Observação: a lista pode ser alterada livremente no corpo da requisição; o modelo **sempre** escolherá uma única categoria dentre as fornecidas.

Estrutura das pastas

```
.
├── app
│   ├── __init__.py
│   ├── main.py
│   ├── api
│   │   ├── __init__.py
│   │   └── routes.py
│   ├── core
│   │   ├── __init__.py
│   │   └── prompt.py
│   ├── schemas
```

```
│   │   ├── __init__.py
│   │   ├── requests.py
│   │   └── responses.py
│   ├── services
│   │   ├── __init__.py
│   │   ├── openai_client.py
│   │   └── pipeline.py
│   └── utils
│       ├── __init__.py
│       └── io.py
├── requirements.txt
└── .env.example
```

Explicação de Cada Arquivo/Pasta

`app/`

Módulo raiz da aplicação. Contém inicialização do **FastAPI**, definição e inclusão de rotas, schemas (Pydantic), serviços de integração (OpenAI) e utilitários.

- **`app/main.py`**
  - Instancia o `FastAPI` com **título**, **versão** e **descrição**.
  - Carrega variáveis de ambiente com `dotenv`.
  - **Inclui** o roteador principal (`app.api.routes.router`).
  - Expõe **`GET /health`** para verificação de disponibilidade.
    Responsabilidade: ponto de entrada do ASGI (ex.: `uvicorn app.main:app`).

`app/api/`

Camada HTTP (controladores/rotas).

- **`app/api/routes.py`**
  - Define o `APIRouter` e a **rota principal `POST /classificacao`**.
  - Valida a existência do arquivo CSV (`dataset_path`).
  - Carrega o CSV via `app.utils.io.load_dataframe` com autodetecção opcional de separador/encoding.
  - Valida colunas (`text_column`, `id_column` quando fornecida).

- Seleciona as linhas (`max_rows`) e invoca o **pipeline** (`app.services.pipeline.run_pipeline`).
- Mede tempo total de processamento e retorna `ClassificacaoResponse`.
  Responsabilidade: orquestração por requisição HTTP (sem lógica de LLM/IO de baixo nível).

---

app/core/

Recursos “core” que não dependem de camada HTTP.

- **app/core/prompt.py**

  - Define `SYSTEM_TEMPLATE` (instrução sistêmica para o modelo) e `USER_TEMPLATE` (prompt de usuário com placeholders `{text}`, `{locale}`, `{categories}`).
    Responsabilidade: padronizar o prompt, isolando-o do pipeline e facilitando manutenção.

---

app/schemas/

Contratos de entrada/saída da API (Pydantic).

- **app/schemas/requests.py**

  - **`ClassificacaoRequest`**:
    Campos principais:

    - **`dataset_path`**: caminho do CSV de entrada.
    - **`text_column`**: coluna do texto dos tickets.
    - **`id_column`** (opcional): coluna identificadora.
    - **`categories`**: lista de classes possíveis (strings exatas).
    - **`max_rows`** (opcional): limite de linhas.
    - **`temperature`** (opcional): temperatura do modelo.
    - **`resume_locale`** (opcional): localidade do resumo (ex.: `pt-BR`).
    - **`output_csv_path`** (opcional): caminho para salvar CSV com saídas.
    - **`csv_sep`**, **`csv_encoding`** (opcionais): forçam separador e encoding; se ausentes, são autodetectados.

        - **`openai_model`**: nome do modelo OpenAI (ex.: `gpt-4o-mini`).

- **`app/schemas/responses.py`**

    - **`ItemResult`**: resultado por linha (id, `summary`, `category`).
      `ClassificacaoResponse`: metadados (`provider`, `model`, `n_rows`, `seconds_total`, `output_csv_path`) e `results` (lista de `ItemResult`).

`app/services/`

Integrações e lógica de aplicação.

- **`app/services/__init__.py`**
  Pacote de serviços.
- **`app/services/openai_client.py`**

    - Função `_post_with_retries(...)`: POST com **retries** e backoff para lidar com 429/5xx e erros transitórios do `httpx`.
    - Classe `OpenAIClient`:

        - Lê `OPENAI_API_KEY` e `OPENAI_BASE_URL` do ambiente.
        - Método `complete(system, user)`: chama **`/chat/completions`** com mensagens **system** e **user**, retorna `LLMReply`.

- **`app/services/pipeline.py`**

    - **`_safe_json_parse`**`(s)`: tenta extrair JSON válido do texto do LLM; em falha, devolve rascunho com `category: "UNPARSEABLE"`.
    - **`_inference_row`**`(...)`: formata o prompt com `prompt.py`, aciona `OpenAIClient.complete`, normaliza categoria por caixa (case-insensitive).
    - **`run_pipeline`**`(...)`:

        - Instancia `OpenAIClient`; controla concorrência via `asyncio.Semaphore` com `MAX_CONCURRENCY` (ambiente, padrão 4).

        - Dispara tarefas assíncronas por linha do dataframe (`asyncio.gather`).

        - Se `output_csv_path` for fornecido, salva um CSV contendo colunas originais + `summary` + `predicted_category`.
        - Retorna `List[ItemResult]` e o nome do modelo.

---

app/utils/

Funções utilitárias transversais.

- **app/utils/io.py**
    - `load_dataframe(path, sep=None, encoding=None)`:
        - Se `sep` for fornecido, tenta ler com o `encoding` (ou lista de encodings comuns).
        - Se `sep` for ausente, tenta autodetectar (motor `python`, e fallback para `;`/`,` caso a leitura indique uma única coluna).
        - Em falha, lança `HTTPException` 400 com orientação.
          Responsabilidade: leitura robusta de CSV, com autodetecção opcional.

Arquivos na raiz

- **`requirements.txt`**
  Lista de dependências com versões fixadas:
  `fastapi`, `uvicorn[standard]`, `pydantic`, `pandas`, `python-dotenv`, `httpx`.

- **`.env.example`**
  Modelo de variáveis de ambiente:
    - **`OPENAI_API_KEY`** (**obrigatória**).
    - `OPENAI_BASE_URL` (opcional; padrão oficial da OpenAI).
    - `MAX_CONCURRENCY` (opcional; padrão 4).

Fluxo de Execução (Visão de Alto Nível)

1. **Recepção HTTP** (`POST /classificacao`): validação básica do corpo (Pydantic).
2. **IO CSV** (`utils.io.load_dataframe`): leitura robusta do dataset.
3. **Validação de colunas**: checagem de `text_column` e `id_column`.
4. **Seleção de linhas**: `head(max_rows)`, se fornecido.
5. **Pipeline**:
    - Construção de prompts (`core.prompt`).
    - Chamada OpenAI com **retries** (`services.openai_client`).
    - **Concorrência controlada** (`MAX_CONCURRENCY`) para acelerar sem exceder limites.
    - Normalização de categoria.
    - (Opcional) Escrita de CSV de saída.
6. **Resposta** (`schemas.responses`): metadados + lista de itens (`summary`, `category`, `id` quando houver).

Contrato da API

Endpoint

- **POST `/classificacao`**
  Entrada: `ClassificacaoRequest`
  Saída: `ClassificacaoResponse`

Execução e Testes Locais

Instalação

```
python -m venv .venv
source .venv/bin/activate   # Windows: .venv\Scripts\activate
pip install -r requirements.txt
```

OBS: Para executar a api é necessário criar na raiz do projeto o arquivo .env contendo a var de ambiente da openai. Para fins de teste estou disponibilizando uma chave da minha api particular

OPENAI_API_KEY=sk-proj-XtHKg05NdrliOlrtb3wUMxyCLbFYL5rld6WJUMxqteoRiyIQAEW
E9QmVsTP0V-AaAfzzKTG4KHT3BlbkFJZLMYkG8Hsn5QhgOy6UwALKf2oD75ldlwfGgBsIg
P4TG2vXkRQcOFapWGYX1b4NMXcZ-aBKf3kA

Rodar o servidor

uvicorn app.main:app --reload --host 127.0.0.1 --port 8000

Swagger UI

- Acesse: `http://127.0.0.1:8000/docs`
- Teste `POST /classificacao` com o corpo de exemplo.

Exemplo de Requisição

```
{
  "dataset_path": "tickets_atendimento.csv",
  "text_column": "tipo_solicitacao",
  "id_column": "ticket_id",
  "canal_column": "canal",
  "prioridade_column": "prioridade",
  "categories": [
    "Acesso/Senha",
    "Falha de Sistema",
    "Solicitação de Serviço",
    "Informação/Dúvida",
    "Infraestrutura/Rede"
  ],
  "max_rows": 20,
  "temperature": 0.0,
  "resume_locale": "pt-BR",
  "output_csv_path": "saida_classificacao.csv",
  "csv_sep": ";",
  "csv_encoding": "utf-8-sig",
  "openai_model": "gpt-4o-mini"
}
```

Exemplo de Resposta

{

```
 "provider": "openai",
 "model": "gpt-4o-mini",
 "n_rows": 2,
 "seconds_total": 5.81,
 "results": [
  {
   "id": "123",
   "summary": "Usuário sem acesso ao sistema desde ontem.",
   "category": "Falha de Sistema"
  },
  {
   "id": "124",
   "summary": "Problema de senha relatado via chat.",
   "category": "Acesso/Senha"
  }
 ],
 "output_csv_path": "saida_classificacao.csv"
}
```


Como colocar em produção

1) Variáveis de ambiente (arquivo `.env` ou configuradas no orquestrador)

- `OPENAI_API_KEY=sk-...` (**obrigatória**)
- `OPENAI_BASE_URL=https://api.openai.com/v1` (opcional, mantenha padrão)
- `MAX_CONCURRENCY=4` (controle de paralelismo de chamadas ao LLM)

2) Build e execução local com Docker

```
docker build -t tickets-api:latest .
docker run --rm -p 8000:8000 --env-file .env tickets-api:latest
# Teste
curl http://localhost:8000/health
# Swagger
# http://localhost:8000/docs
```

3) Deploy em Databricks (resumo)

- Publique a imagem em um registry (Docker Hub/GHCR/ECR).
- Em **Compute → Create → Use your own Docker image**, informe a imagem e defina `OPENAI_API_KEY` e demais variáveis.
- Opcional: exponha a porta 8000 via **init script**/Job com `docker run` (em clusters que suportem containers) ou utilize **Model Serving** chamando o serviço HTTP interno.
  (A configuração exata depende do modo de contêiner habilitado no workspace.)