

Plano Mestre: Construção do Sistema de Classificação Fiscal Agêntico

Data: 11 de Agosto de 2025

Versão: 1.0 (Consolidada)

1. Sumário Executivo

Este documento descreve o plano completo para a construção de um sistema de Inteligência Artificial de ponta para classificação fiscal (NCM/CEST). O projeto consolida todas as abordagens discutidas:

- **Operação 100% Local:** Usando Ollama para o LLM e FAISS para o banco vetorial.
- **Arquitetura Híbrida:** Combinando um **banco de dados de mapeamento estruturado** para fatos (velocidade e precisão) com um **banco vetorial semântico (RAG)** para contexto e exemplos.
- **Sistema Multiagentes (Agentic RAG):** Utilizando uma equipe de agentes especializados (Expansão, Agregação, NCM, CEST, Reconciliação) que raciocinam, colaboram e auditam o processo.
- **Foco em Eficiência e Consistência:** Implementando uma agregação inteligente para processar lotes de produtos de forma otimizada.
- **Rastreabilidade Total:** Criando um mecanismo para auditar o "raciocínio" de cada agente em cada etapa.

O objetivo final é criar um sistema robusto, auditável, eficiente e que se torna mais inteligente com o tempo.

2. Estrutura Completa do Projeto

A arquitetura de diretórios foi projetada para máxima modularidade e para acomodar futuras expansões de forma organizada.

```
/rag_multiagent_system
|
|-- data/          # 🗂️ Armazenamento de todos os dados
|   |-- raw/        # 📄 Arquivos fonte originais (NCM, CEST, NESH, etc.)
|   |-- processed/  # 📁 Saída final: CSVs com os resultados da classificação
|   |-- knowledge_base/ # 💡 Base de conhecimento processada e pronta para uso
|       |-- ncm_mapping.json # BANCO ESTRUTURADO: Mapeamento NCM ->
|                           Descrição/CESTs
```

```

| | |-- faiss_index.faiss # BANCO VETORIAL: Índice FAISS para busca semântica
| | |-- metadata.db      # BANCO VETORIAL: Metadados dos vetores (SQLite)
|
| |-- feedback/          # ★ [EXPANSÃO FUTURA]
|   |-- golden_set.db
|   |-- verified_expansions.db
|
|-- scripts/             # 🛠 Scripts utilitários para tarefas pontuais
| |-- build_knowledge_base.py # Script para criar o ncm_mapping.json
| |-- test_mapping.py      # Script para testes intermediários do banco de
mapeamento
| |-- test_rag.py         # Script para testes intermediários do banco vettorial
|
|-- src/                 # 📂 Código-fonte principal da aplicação
| |-- agents/             # 🤖 A equipe de especialistas de IA
| |-- ingestion/          # 🚗 Módulos para carregar e preparar os dados brutos
| |-- vectorstore/        # 💬 O coração do sistema RAG
| |-- llm/                # 👤 Comunicação com o Ollama e gestão de prompts
| |-- orchestrator/       # 🎊 O maestro que rege o fluxo de trabalho
| |-- api/                # 🌐 [EXPANSÃO FUTURA] Interface de Revisão Humana
| |-- feedback/            # 💬 [EXPANSÃO FUTURA] Lógica para o Ciclo de Aprendizagem
| |-- config.py           # 🌐 Configurações centralizadas
| |-- main.py              # ⏪ Ponto de entrada para executar a aplicação
|
|-- .env                  # 🔑 Credenciais e configurações de ambiente
|-- requirements.txt       # 📦 Lista de dependências Python
|-- README.md              # 📄 Documentação geral e guias de execução

```

Criação Pastas

```
cd "C:\Users\eniot\OneDrive\Desenvolvimento\Projetos"
```

```
mkdir rag_multiagent_system -Force
```

```
cd rag_multiagent_system
```

```
$folders = @(  
    "data/raw",  
    "data/processed",  
    "data/knowledge_base",  
    "data/feedback",  
    "scripts",  
    "src/agents",  
    "src/ingestion",  
    "src/vectorstore",  
    "src/llm",  
    "src/orchestrator",  
    "src/api",  
    "src/feedback"  
)
```

```
$folders | ForEach-Object { New-Item -Path $_ -ItemType Directory -Force }
```

3. Plano de Implementação Detalhado (Passo a Passo)

Fase 0: Ambiente e Configuração

1. **Ação:** Configurar o ambiente de desenvolvimento.
2. **Passos:**
 - Criar a estrutura de diretórios acima.
 - Criar um ambiente virtual Python (python -m venv venv).
 - Instalar as dependências do requirements.txt.

- Configurar o Ollama e baixar um modelo (ex: ollama pull llama3).
- Copiar .env.example para .env e preencher as variáveis.
- Colocar todos os arquivos fonte (descricoes_ncm.json, Anexos_conv_92_15.csv, nesh-2022.pdf, etc.) na pasta data/raw/.

Fase 1: Fundação dos Dados (O Conhecimento Estruturado)

O objetivo desta fase é unificar as fontes de dados estruturados em um único banco de mapeamento rápido e eficiente.

1. **Ação:** Criar o script scripts/build_knowledge_base.py.

2. **Lógica:**

- **Carregar NCM:** Ler descricoes_ncm.json e criar um dicionário onde a chave é o código NCM de 8 dígitos. A estrutura inicial será: {'ncm_codigo': '...', 'descricao_oficial': '...', 'cests_associados': [], 'gtins_exemplos': []}.
- **Unificar CEST:** Ler Anexos_conv_92_15.csv. Para cada linha, extrair o par NCM-CEST. Localizar o NCM no dicionário e adicionar o CEST à lista cests_associados.
- **Unificar Exemplos:** Ler a tabela de produtos (via extracao_dados.py ou produtos_selecionados.json). Para cada produto, localizar seu NCM no dicionário e adicionar o codigo_barra (GTIN) à lista gtins_exemplos.
- **Salvar:** Salvar o dicionário final como data/knowledge_base/ncm_mapping.json.

3. **Código (scripts/build_knowledge_base.py):**

```
import json
import pandas as pd

def build_mapping_database():
    print("Iniciando construção do banco de mapeamento...")
    # Caminhos (idealmente viriam do config.py)
    ncm_path = 'data/raw/descricoes_ncm.json'
    cest_path = 'data/raw/Anexos_conv_92_15.xlsx - Anexos_conv_92_15.csv'
    # ... carregar outros arquivos de exemplo ...

    # 1. Carregar base NCM
    with open(ncm_path, 'r', encoding='utf-8') as f:
        ncm_data = json.load(f)

    mapping_db = {}
```

```

for item in ncm_data:
    code = item.get("Código", "").replace(".", "")
    if len(code) == 8: # Apenas NCMs completos
        mapping_db[code] = {
            "ncm_codigo": code,
            "descricao_oficial": item.get("Descricao_Completa", ""),
            "cests_associados": [],
            "gtins_exemplos": []
        }

# 2. Mapear CEST
try:
    cest_df = pd.read_csv(cest_path)
    for _, row in cest_df.iterrows():
        ncm = str(row['NCM']).replace(".", "")
        if ncm in mapping_db:
            cest_info = {
                "cest": row['CEST'],
                "descricao_cest": row['DESCRIÇÃO']
            }
            if cest_info not in mapping_db[ncm]['cests_associados']:
                mapping_db[ncm]['cests_associados'].append(cest_info)
except Exception as e:
    print(f"Aviso: Não foi possível processar o arquivo CEST. Erro: {e}")

# 3. Salvar o banco de mapeamento
output_path = 'data/knowledge_base/ncm_mapping.json'
with open(output_path, 'w', encoding='utf-8') as f:
    json.dump(mapping_db, f, indent=2, ensure_ascii=False)

print(f"✅ Banco de mapeamento salvo em {output_path}")

if __name__ == '__main__':
    build_mapping_database()

```

4. Teste Intermediário 1: Validar o Mapeamento

- **Ação:** Criar e executar scripts/test_mapping.py.

- **Lógica:** Carregar o ncm_mapping.json. Fazer buscas por NCMs conhecidos e imprimir os resultados para verificar se as descrições e listas de CESTs estão corretas.

```
import json

def test_mapping(ncm_to_test):
    with open('data/knowledge_base/ncm_mapping.json', 'r') as f:
        db = json.load(f)

    if ncm_to_test in db:
        print(json.dumps(db[ncm_to_test], indent=2, ensure_ascii=False))
    else:
        print(f"NCM {ncm_to_test} não encontrado.")

# test_mapping("22021000")
```

Fase 2: Vetorização do Conhecimento (O Cérebro Semântico)

O objetivo é criar o banco de dados vetorial para busca por similaridade.

1. **Ação:** Implementar a lógica de ingestão no main.py (comando ingest).
2. **Lógica:**
 - **Carregar Fontes:** Usar o data_loader.py para carregar os textos da NESH e as descrições da tabela de produtos.
 - **Chunking:** Usar o chunker.py para dividir a NESH em pedaços. Para os produtos, cada descrição é um "chunk".
 - **Metadados:** É crucial que cada chunk de produto tenha seus metadados associados, especialmente o ncm original. Ex: {'source': 'produtos', 'ncm': '96034090', 'gtin': '...'}.
 - **Vetorização:** Usar o embedder.py para transformar todos os chunks de texto em vetores.
 - **Indexação:** Usar o faiss_store.py para salvar os vetores no índice FAISS e os textos/metadados no banco SQLite.
3. **Teste Intermediário 2: Validar o RAG**
 - **Ação:** Criar e executar scripts/test_rag.py.
 - **Lógica:**
 - **Teste 1 (Busca Semântica Geral):** Fazer uma busca por um termo genérico (ex: "parafusos de metal") e verificar se os resultados retornados são relevantes.
 - **Teste 2 (Busca Híbrida Filtrada):** Fazer uma busca semântica por um

produto (ex: "refrigerante"), mas aplicando um **filtro de metadados** para retornar apenas exemplos que pertencem a um NCM específico (ex: ncm = '22021000'). Isso valida a capacidade de encontrar exemplos relevantes para um NCM.

```
# scripts/test_rag.py (esboço)
# from src.vectorstore.faiss_store import FaissMetadataStore
# from src.vectorstore.embedder import Embedder
# store = FaissMetadataStore(...)
# embedder = Embedder(...)
# query_embedding = embedder.embed_batch(["refrigerante de cola"], 1)
# results = store.search(query_embedding, k=5, metadata_filter={'ncm': '22021000'})
# print(results)
```

Fase 3: Implementação da Arquitetura Agêntica Híbrida

O objetivo é construir os agentes e o orquestrador para que eles utilizem ambos os bancos de conhecimento.

1. **Ação:** Implementar os agentes e o orquestrador (src/agents/, src/orchestrator/).
2. **Lógica do Router (Orquestrador):**
 - o Recebe o lote completo de produtos.
 - o Executa o ExpansionAgent em todos.
 - o Executa o AggregatorAgent para criar os grupos.
 - o Itera apenas sobre os **representantes de cada grupo**.
 - o Para cada representante:
 - **Busca Estruturada:** Consulta o ncm_mapping.json para obter fatos.
 - **Busca Semântica:** Consulta o FaissStore para obter exemplos (com filtro de NCM).
 - **Passa o Contexto:** Envia ambos os contextos (estruturado e semântico) para os agentes NCM e CEST.
 - Executa o ReconcilerAgent.
 - Armazena o resultado e o reasoning_trace em um cache.
 - o Propaga os resultados do cache para todos os produtos do lote com base no grupo_id.
3. **Lógica dos Agentes (NCMAgent, etc.):**
 - o Seus métodos run são atualizados para receber o contexto estruturado e o semântico.
 - o Seus **prompts** são atualizados para incluir seções específicas para cada tipo de contexto, instruindo o LLM a priorizar os fatos estruturados.

- Sua estrutura de retorno é { "result": ..., "trace": ... } para garantir a rastreabilidade.
4. **Ação Final:** Implementar o comando classify no main.py que invoca o Router.

Fase 4: Expansões e Melhorias Futuras

Com a base sólida implementada, o foco se volta para a interação e a inteligência contínua.

1. **Interface de Revisão Humana:**

- **Local:** /src/api/.
- **Objetivo:** Criar endpoints e uma UI para que especialistas possam revisar, corrigir e aprovar as classificações, incluindo a edição das descrições expandidas.

2. **Ciclo de Aprendizagem:**

- **Local:** /src/feedback/ para a lógica e /data/feedback/ para os dados.
- **Objetivo:** Salvar as correções humanas em bancos de conhecimento de "gabaritos" (um para expansões, outro para classificações). Modificar os agentes para que eles consultem estes bancos de alta prioridade antes de qualquer outra fonte.

3. **Otimização de Performance:**

- **Ação:** Quando o volume de dados crescer, modificar o faiss_store.py para usar um índice mais avançado como IndexIVFPQ.

Este plano consolidado fornece um roteiro claro e testável para construir o sistema do início ao fim, garantindo que cada componente seja validado antes de se integrar ao todo, resultando em uma solução final robusta, inteligente e escalável.