Abaixo: 1) visão arquitetural e agentes, 2) funcionalidades mínimas obrigatórias para tornar o agente **genérico para qualquer CSV**, multimodal (gráficos) e com **memória**, 3) endpoints UI/API necessários, 4) prompts / interação com LLM para gerar conclusões, 5) persistência de memória (esquema), 6) roadmap de entregas em sprints, e 7) **trechos de código** (patch conceitual) que você pode aplicar já ao agente.py e ao front.py.

**1) Visão arquitetural (multiagente multimodal — alto nível)**

Componentes principais:

* **Orquestrador (Agent Manager)** — recebe upload, cria/associa um *dataset id* e roteia perguntas aos agentes especializados.
* **Agent de Ingestão / Schema-Inference** — recebe CSVs, detecta encoding, delim, infere tipos (numérico, categórico, datetime), normaliza colunas, gera *metadados* (colunas, nulos, tipos, range).
* **Agent de EDA (Análise)** — executa métricas estatísticas (média, mediana, std, IQR), distribuições, contagens, séries temporais, correlações e clustering (quando aplicável).
* **Agent de Visualização** — gera gráficos (histograma, boxplot, scatter, heatmap) e retorna imagens (URLs ou base64) para UI.
* **Agent de Detecção de Outliers** — IQR / z-score + impacto.
* **Agent de Insights / NLG** — usa LLM (Gemini / LangChain) para transformar estatísticas + gráficos + sample rows em conclusões em PT-BR.
* **Memory Store** — armazena metadados do dataset, histórico de perguntas/respostas, insights gerados, e preferências do usuário (SQLite / Redis).
* **API / Frontend (Streamlit)** — upload de CSV, chat de perguntas, visualização de gráficos e histórico.

Fluxo: Upload → Ingestão (schema + limpeza) → gerar resumo automático (dashboard) → usuário pergunta → orquestrador decide: resposta via pandas direto (rápida) ou via LLM (insights complexos) → visualizar resultado + gráficos → armazenar Q/A + insights em memória.

**2) Funcionalidades mínimas (MVP) — obrigatórias**

* Upload de 1 CSV ou múltiplos CSVs (automático merge se houver chave em comum).
* Detecção automática de delimitador/encoding e casting de tipos.
* Resumo automático: n° linhas, colunas, tipos, missing %, min/max, média/mediana, desvio padrão.
* Visualizações instantâneas: histogramas (numéricos), barras (categóricos), séries temporais (colunas datetime), scatter (duas variáveis), boxplot (outliers).
* Perguntas em linguagem natural (chat) respondidas por:
  + função direta em pandas (respostas objetivas)
  + ou chamada ao LLM para explicações / conclusões (usando payload resumido + links para gráficos)
* Detecção e relatório de outliers (IQR / z-score) + sugestão de tratamento.
* Correlação e matriz de correlação com heatmap.
* Clustering rápido (KMeans, padrão 2-5 clusters quando apropriado) e breve interpretação.
* **Memória persistente**: store dataset metadata, last N insights, last queries/responses.
* **Export**: permitir export das análises (PDF/PNG/CSV resumo).

**3) Endpoints / UI necessários (proposta)**

API (backend Flask — aproveitando sua base):

* POST /api/upload — aceita multipart/form-data (1..N CSV), devolve dataset\_id, metadados iniciais.
* GET /api/summary?dataset\_id=... — retorna resumo estatístico e lista de gráficos gerados (com URLs ou base64).
* POST /api/query — body: {dataset\_id, question} → retorna resposta textual + possíveis gráficos anexados.
* GET /api/plots?dataset\_id=&plot=histogram&col=... — gera/retorna gráficos sob demanda.
* GET /api/memory?dataset\_id= — retorna memória (insights salvos, perguntas anteriores).

Front (Streamlit):

* upload widget (arquivo único ou múltiplo)
* resumo automático ao carregar
* chat/entrada de pergunta
* painel de gráficos com download
* histórico + botão “Salvar insight” / “Marcar como importante”

**4) Prompting / NLG — como gerar conclusões úteis**

**Estratégia**: evite enviar CSV inteiro ao LLM (custoso). Em vez disso:

* envie *resumo estatístico* + *amostra representativa* (p.ex. 20 linhas) + *lista de gráficos* (descrições + links/base64).
* instruções claras: “responda em português, destaque 3 insights principais, 2 riscos/limitações, sugira próximos passos analíticos e apresente 1 visualização recomendada.”

Exemplo de prompt system (para Gemini/LangChain):

Você é um analista de dados. Use APENAS os dados e estatísticas fornecidas.

- Forneça 3 insights acionáveis em português brasileiro.

- Cite números sempre que possível (ex.: 'média R$ 1.234,56').

- Identifique 2 potenciais anomalias e se elas impactam a média.

- Sugira 2 próximos passos analíticos.

Contexto:

<colunas, tipos, nulos, estatísticas resumidas>

<Amostra de 20 linhas: csv-base64 ou textual>

<Lista de gráficos gerados: histogram(colX), boxplot(colY), heatmap(corr\_matrix)>

Pergunta do usuário: {user\_question}

**5) Memória (persistente) — esquema proposto (SQLite)**

Tabela datasets:

* dataset\_id (text PK), name, uploaded\_at, n\_rows, n\_cols, filepath, schema\_json

Tabela insights:

* insight\_id, dataset\_id, created\_at, text, tags, important (bool)

Tabela queries:

* query\_id, dataset\_id, question, response\_summary, raw\_response, created\_at, source (pandas/llm)

Implementação: usar sqlite3 com acesso simples e thread-safe (ou SQLAlchemy se preferir). Permitir memory.get\_recent\_insights(dataset\_id, n=5).

**6) Roadmap incremental (sprints — sem estimativa de tempo)**

Sprint A — **Fundação** (MVP)

* Upload CSV genérico + schema inference
* GET /summary estatísticas básicas
* Histogramas / barras / scatter genéricos
* Perguntas simples via pandas (ex.: "Qual a média de X?")
* Memória básica (salvar dataset + queries)

Sprint B — **Melhorias EDA**

* Outlier detection + boxplot
* Correlação + heatmap
* Cluster rápido (KMeans) com interpretação simples
* UI: seleção de colunas + plot on-demand

Sprint C — **NLG & memória avançada**

* NLG com prompt resumido (usar call\_gemini\_lang\_chain)
* Salvar insights gerados automaticamente
* Permitir salvar/editar insights manualmente na UI

Sprint D — **Polimento**

* Autenticação (se preciso), filtrar por dataset, export para PDF
* Testes automatizados, monitoramento, logs
* Deploy (Docker + CI)

**7) Mudanças de código propostas — trechos aplicáveis agora**

A seguir eu apresento **trechos concretos** (adaptados ao seu agente.py e front.py) para:

1. Endpoint de upload genérico /api/upload
2. Funções auxiliares: infer\_schema, generate\_plots, detect\_outliers, save\_memory.
3. Ajustes no frontend Streamlit para upload de CSV e mostrar resumo/plots.

Obs.: estes são trechos prontos para integrar — você pode aplicar como patch no seu repositório. Ajustei para seguir o estilo do seu código atual.

**7.1 — Backend: snippets a inserir em agente.py (ou criar um novo módulo eda\_agent.py)**

# --- import adicionais ---

import uuid

import sqlite3

import matplotlib

matplotlib.use("Agg")

import matplotlib.pyplot as plt

import io

import json

from sklearn.cluster import KMeans

from scipy import stats

# --------------------------------

# --- simple sqlite memory store (file memory.db) ---

def init\_db(db\_path="memory.db"):

conn = sqlite3.connect(db\_path, check\_same\_thread=False)

cur = conn.cursor()

cur.execute("""

CREATE TABLE IF NOT EXISTS datasets (

dataset\_id TEXT PRIMARY KEY,

name TEXT,

uploaded\_at TEXT,

n\_rows INTEGER,

n\_cols INTEGER,

filepath TEXT,

schema\_json TEXT

)""")

cur.execute("""

CREATE TABLE IF NOT EXISTS queries (

query\_id TEXT PRIMARY KEY,

dataset\_id TEXT,

question TEXT,

response\_summary TEXT,

raw\_response TEXT,

created\_at TEXT,

source TEXT

)""")

cur.execute("""

CREATE TABLE IF NOT EXISTS insights (

insight\_id TEXT PRIMARY KEY,

dataset\_id TEXT,

created\_at TEXT,

text TEXT,

important INTEGER DEFAULT 0

)""")

conn.commit()

return conn

DB\_CONN = init\_db()

def save\_dataset\_metadata(dataset\_id, name, df, filepath):

cur = DB\_CONN.cursor()

schema = infer\_schema(df)

cur.execute("REPLACE INTO datasets (dataset\_id,name,uploaded\_at,n\_rows,n\_cols,filepath,schema\_json) VALUES (?,?,?,?,?,?,?)",

(dataset\_id, name, time.strftime("%Y-%m-%d %H:%M:%S"), df.shape[0], df.shape[1], filepath, json.dumps(schema)))

DB\_CONN.commit()

def save\_query(dataset\_id, question, response, raw, source):

cur = DB\_CONN.cursor()

qid = str(uuid.uuid4())

cur.execute("INSERT INTO queries VALUES (?,?,?,?,?,?,?)",

(qid, dataset\_id, question, response[:2000], raw[:2000], time.strftime("%Y-%m-%d %H:%M:%S"), source))

DB\_CONN.commit()

return qid

# --- schema inference ---

def infer\_schema(df: pd.DataFrame) -> dict:

schema = {}

for col in df.columns:

colseries = df[col]

dtype = str(colseries.dtype)

n\_missing = int(colseries.isna().sum())

n\_unique = int(colseries.nunique(dropna=True))

sample = colseries.dropna().sample(min(3, max(1, colseries.dropna().shape[0]))) if colseries.dropna().shape[0] > 0 else []

sample\_vals = sample.astype(str).tolist() if hasattr(sample, "astype") else []

colinfo = {"dtype": dtype, "missing": n\_missing, "unique": n\_unique, "sample": sample\_vals}

if pd.api.types.is\_numeric\_dtype(colseries):

colinfo.update({"min": None if colseries.dropna().empty else float(colseries.min()),

"max": None if colseries.dropna().empty else float(colseries.max()),

"mean": None if colseries.dropna().empty else float(colseries.mean())})

elif pd.api.types.is\_datetime64\_any\_dtype(colseries):

colinfo.update({"min": None if colseries.dropna().empty else str(colseries.min()),

"max": None if colseries.dropna().empty else str(colseries.max())})

schema[col] = colinfo

return schema

# --- plot generator returns base64 PNG ---

def plot\_to\_base64(fig):

buf = io.BytesIO()

fig.savefig(buf, format="png", bbox\_inches="tight")

buf.seek(0)

b64 = base64.b64encode(buf.read()).decode("utf-8")

plt.close(fig)

return f"data:image/png;base64,{b64}"

def histogram\_plot(df, col):

fig, ax = plt.subplots(figsize=(6,4))

df[col].dropna().plot(kind='hist', bins=30, ax=ax)

ax.set\_title(f"Histograma: {col}")

return plot\_to\_base64(fig)

def boxplot\_plot(df, col):

fig, ax = plt.subplots(figsize=(4,3))

df[col].dropna().plot(kind='box', ax=ax)

ax.set\_title(f"Boxplot: {col}")

return plot\_to\_base64(fig)

def scatter\_plot(df, x, y):

fig, ax = plt.subplots(figsize=(5,4))

ax.scatter(df[x], df[y], alpha=0.6)

ax.set\_xlabel(x)

ax.set\_ylabel(y)

ax.set\_title(f"{y} vs {x}")

return plot\_to\_base64(fig)

def correlation\_heatmap(df, numeric\_cols):

corr = df[numeric\_cols].corr()

fig, ax = plt.subplots(figsize=(6,5))

cax = ax.matshow(corr)

fig.colorbar(cax)

ax.set\_xticks(range(len(numeric\_cols)))

ax.set\_xticklabels(numeric\_cols, rotation=90)

ax.set\_yticks(range(len(numeric\_cols)))

ax.set\_yticklabels(numeric\_cols)

ax.set\_title("Matriz de Correlação")

return plot\_to\_base64(fig), corr.to\_dict()

# --- outlier detection (IQR) ---

def detect\_outliers\_iqr(series):

q1 = series.quantile(0.25)

q3 = series.quantile(0.75)

iqr = q3 - q1

lower = q1 - 1.5 \* iqr

upper = q3 + 1.5 \* iqr

outliers = series[(series < lower) | (series > upper)]

return {"count": int(outliers.shape[0]), "lower": float(lower), "upper": float(upper)}

# --- upload endpoint (adicionar ao Flask app) ---

@app.route("/api/upload", methods=["POST"])

def upload\_csv():

"""

Recebe 1 ou mais CSVs via multipart/form-data.

Retorna dataset\_id e resumo inicial.

"""

if 'files' not in request.files:

return jsonify({"error":"Nenhum arquivo enviado, use 'files' no form-data"}), 400

files = request.files.getlist("files")

# aceitar 1 arquivo ou múltiplos

dfs = []

filenames = []

for f in files:

try:

content = f.read()

# detectar encoding / delim seria um aprimoramento; por enquanto assume utf-8 e sep auto

df = pd.read\_csv(io.BytesIO(content), sep=None, engine='python')

df.columns = df.columns.str.strip()

dfs.append(df)

filenames.append(f.filename)

except Exception as e:

return jsonify({"error": f"Falha ao ler {f.filename}: {str(e)}"}), 400

# Se mais de um, tenta merge heurístico por colunas em comum (CHAVE DE ACESSO)

if len(dfs) == 1:

df\_combined = dfs[0]

else:

# achar coluna em comum

common = set(dfs[0].columns)

for d in dfs[1:]:

common = common.intersection(set(d.columns))

if common:

key = list(common)[0]

df\_combined = dfs[0]

for d in dfs[1:]:

df\_combined = df\_combined.merge(d, on=key, how='outer')

else:

# concat como fallback

df\_combined = pd.concat(dfs, axis=0, ignore\_index=True, sort=False)

dataset\_id = str(uuid.uuid4())

# salvar CSV temporário

path = f"data/{dataset\_id}.csv"

os.makedirs("data", exist\_ok=True)

df\_combined.to\_csv(path, index=False)

save\_dataset\_metadata(dataset\_id, ",".join(filenames), df\_combined, path)

# gerar resumo simples

schema = infer\_schema(df\_combined)

numeric\_cols = [c for c in df\_combined.columns if pd.api.types.is\_numeric\_dtype(df\_combined[c])]

plots = {}

# gerar histograma para até 3 num cols

for col in numeric\_cols[:3]:

try:

plots[f"hist\_{col}"] = histogram\_plot(df\_combined, col)

except Exception:

continue

summary = {

"dataset\_id": dataset\_id,

"n\_rows": df\_combined.shape[0],

"n\_cols": df\_combined.shape[1],

"schema": schema,

"plots": plots

}

return jsonify(summary), 200

Observação: adicionei tratamento simples para merge de múltiplos CSVs e geração de plots como base64 — isso já entrega o comportamento multimodal necessário: o frontend pode exibir imagens base64 diretamente.

**7.2 — Frontend (Streamlit) — trecho para inclusão no seu front.py (substitui parte do bloco “Visualizar Dados Locais”)**

# Adicionar upload widget no topo da página

st.sidebar.header("📁 Upload de CSV")

uploaded = st.sidebar.file\_uploader("Envie 1 ou mais CSVs (selecione múltiplos)", accept\_multiple\_files=True, type=['csv'])

if uploaded:

files = uploaded

with st.spinner("Enviando arquivos para o backend..."):

multipart = []

# usar requests tool 'files'

files\_payload = [('files', (f.name, f.getvalue(), 'text/csv')) for f in files]

try:

resp = requests.post(f"{API\_BASE\_URL}/upload", files=files\_payload, timeout=120)

if resp.status\_code == 200:

payload = resp.json()

st.success("Upload concluído")

st.session\_state.dataset\_id = payload['dataset\_id']

st.write("\*\*Resumo inicial\*\*")

st.write(f"Linhas: {payload['n\_rows']}, Colunas: {payload['n\_cols']}")

st.write("\*\*Esquema (amostra)\*\*")

st.json(payload['schema'])

# render plots

if payload.get('plots'):

st.write("\*\*Gráficos gerados automaticamente\*\*")

for name, b64 in payload['plots'].items():

st.image(b64, use\_column\_width=True)

else:

st.error(f"Erro no upload: HTTP {resp.status\_code} - {resp.text}")

except Exception as e:

st.error(f"Erro de comunicação: {str(e)}")

**8) Como funcionará a “memória” no agente**

* Ao fazer upload, salvamos metadados na tabela datasets.
* Ao cada pergunta, salvamos um registro em queries.
* Insights gerados automaticamente (p.ex. análise inicial com 3 insights) serão salvos em insights e exibidos ao usuário. Haverá botão “Marcar como importante” na UI.
* Implementar endpoint GET /api/memory?dataset\_id=... para recuperar as entradas.

**9) Boas práticas operacionais / limitações**

* **Privacidade**: dados sensíveis devem ser tratados com cuidado — adicione consentimento se for dado pessoal.
* **Tamanho de arquivos**: se > 100MB, usar chunking / processamento em background (ou rejeitar e pedir amostra).
* **Envio ao LLM**: nunca enviar CSV inteiro; enviar resumo + amostra (salvo consentimento).
* **Escalabilidade**: armazenar plots em disco e servir via URL é mais eficiente que base64 em grandes volumes.

**patch do Sprint A** e salvei os arquivos no diretório /mnt/data. Aqui está o que eu gerei:

Arquivos criados

* /mnt/data/eda\_agent.py — helpers EDA (inferência de schema, geração de plots em base64, detecção simples de outliers, sqlite memory).
* /mnt/data/agente\_mvp.py — Flask app com endpoints:
  + POST /api/upload — envia 1..N CSVs e devolve dataset\_id, esquema e plots (base64).
  + GET /api/summary?dataset\_id=... — resumo/plots do dataset salvo.
* /mnt/data/front\_streamlit.py — frontend Streamlit mínimo que consome a API (upload e visualização de resumo/plots).
* /mnt/data/README\_TEST.md — instruções para rodar e testar com curl e Streamlit.

O que fazer agora (passos recomendados)

1. Instalar dependências:
2. pip install flask pandas matplotlib scikit-learn streamlit requests
3. Rodar o backend:
4. python /mnt/data/agente\_mvp.py

API estará em http://localhost:8000.

1. Rodar o frontend (em outro terminal):
2. streamlit run /mnt/data/front\_streamlit.py
3. Testar via curl (exemplo):
4. curl -X POST "http://localhost:8000/api/upload" -F "files=@/caminho/para/seu\_arquivo.csv"

Guarde o dataset\_id do retorno e recupere o resumo:

curl "http://localhost:8000/api/summary?dataset\_id=<dataset\_id>"

Limitações atuais (Sprint A)

* Perguntas em linguagem natural ainda não implementadas — o endpoint /api/query será adicionado na próxima sprint e usará um pipeline híbrido (pandas rápido + LLM para conclusões).
* Heurística simples de detecção de delimitador/encoding — pode ser aprimorada com chardet/cchardet e sniffers.
* Plots retornados como base64 (bom para MVP). Em escala, seria melhor salvá-los em disco e servir via URL.
* Tratamento de arquivos muito grandes ainda não tratado (chunking/streaming).