ChatGPT

Plano de Implementação do Projeto de IA Corporativa Autônoma (RAG + QLoRA)

1. Curadoria e Estruturação do Dataset Corporativo Brasileiro

Nesta primeira etapa, coletaremos um corpus extenso de textos corporativos em português (estimativa de 50–100 GB) a partir de fontes públicas relevantes [1]. O objetivo é montar um conjunto de documentos que reflita o conhecimento empresarial brasileiro e sirva tanto para *treinar* o sistema quanto para *validá-lo*. Em seguida, estruturaremos esse corpus e extrairemos dele um dataset de **instruções** (perguntas e respostas) para especializar o modelo de linguagem.

Coleta automatizada de documentos públicos: Utilize *web scraping* e parsing de arquivos para reunir documentos corporativos relevantes:

- **Relatórios Anuais de empresas (B3)** – baixe PDFs de demonstrações financeiras e relatórios de administração de companhias listadas na B3 [2].
- **Normas Técnicas (ABNT)** – obtenha documentos públicos ou sumários de normas da ABNT que possam ser acessados gratuitamente (ou utilize textos explicativos dessas normas).
- **Resoluções Regulatórias (CVM e outros órgãos)** – colete resoluções da CVM e possivelmente da ANPD, BACEN etc., disponíveis em seus sites oficiais [2].
- **Artigos e notícias do setor (Portal *TI Inside*, etc.)** – extraia artigos sobre adoção de IA e tendências nas PMEs brasileiras para dar contexto contemporâneo [3].

Ferramentas para extração e limpeza: Desenvolva uma *pipeline* automatizada em Python 3.11+ para baixar e extrair texto bruto dos documentos coletados [4]. Dicas de implementação:

- Use **requests** ou **Selenium** (se necessário) para baixar páginas HTML e PDFs. Em seguida, use **BeautifulSoup4** para fazer o *parse* do HTML e extrair texto de páginas web institucionais [4].
- Para PDFs corporativos, utilize **PyMuPDF (fitz)** para ler e extrair o texto mantendo a estrutura de parágrafos e tabelas se possível [4]. Lembre-se de tratar encoding e remover cabeçalhos/rodapés repetitivos nesses documentos.
- Empregue **spaCy 3.7** com modelo de português para limpar e anotar linguisticamente o texto [4]. Por exemplo, use spaCy para segmentar sentenças e detectar entidades nomeadas. Isso ajuda a identificar e possivelmente remover dados pessoais sensíveis (cumprindo a LGPD) durante a ingestão.
- Padronize a formatação: remova caracteres inválidos, espaços excessivos e converta documentos escaneados em texto usando OCR se preciso. Ao final, normalize todos os textos para UTF-8.

Segmentação em chunks preservando contexto: Após reunir o corpus bruto, realize o pré-processamento e **segmentação dos documentos em trechos/colegas** (*chunking*) [5]. Cada chunk será uma unidade indexável no sistema de busca. Boas práticas:

- Divida cada documento em seções lógicas, por exemplo por capítulos, seções ou tópicos, **mantendo cabeçalhos e contexto necessário** para compreensão [5]. Uma técnica é incluir o título da seção ou documento como prefixo em cada chunk, garantindo contexto.

- Controle o tamanho dos chunks para ~500 tokens (cerca de 400–800 palavras) por trecho, com **sobrelapamento** de ~50 tokens entre chunks adjacentes. Isso evita perda de contexto em fronteiras de corte.
- Utilize spaCy ou NLTK para quebrar texto em sentenças e parágrafos. Evite cortar no meio de frases ou tabelas. Se um parágrafo for muito grande, divida-o, mas garanta que partes correlatas (ex: pergunta e resposta em FAQ) fiquem juntas.
- Armazene cada chunk com metadados: por exemplo, nome do documento de origem, data, categoria (financeiro, RH, norma etc.). Esses metadados serão úteis para filtragem ou para exibir a fonte na interface.

Geração do dataset de instruções (Q&A): Em paralelo à preparação do corpus, construa um **dataset de perguntas e respostas** voltado a cenários corporativos para realizar o *fine-tuning* do modelo [6]. O objetivo é produzir de 12k a 18k pares de **instrução -> resposta**, cobrindo tópicos e dúvidas comuns nas empresas brasileiras [7]. Orientações para montar esse dataset:

- Foque em **casos de uso corporativos reais:** por exemplo, perguntas sobre *compliance* regulatório ("Quais são as obrigações da empresa sob a LGPD?"), análise de cláusulas contratuais, síntese de atas de reunião, explicação de indicadores financeiros, dúvidas de RH (férias, contratações) etc. [7]. Garanta variedade: inclua perguntas diretas, situacionais ("O que fazer se...?") e analíticas ("Como interpretar tal indicador?").
- **Geração inicial automatizada:** utilize o próprio corpus para criar perguntas. Por exemplo, para um determinado relatório ou norma, gere uma pergunta cuja resposta esteja contida naquele documento. Você pode usar técnicas como: pegar títulos de seções e transformá-los em perguntas, ou usar uma *LLM* auxiliar (como GPT-4) para ler um trecho e sugerir uma pergunta relevante sobre ele, junto com a resposta baseada no conteúdo.
- **Revisão e refinamento manual:** após a geração automatizada, revise cada par pergunta-resposta para garantir correção factual e linguagem natural. Refine enunciados que ficaram muito artificiais e valide se as respostas realmente respondem às perguntas de forma clara. Idealmente, envolva alguém com conhecimento de negócios para verificar a validade. Elimine ou corrija quaisquer alucinações introduzidas pela geração automática.
- Formate o dataset em um padrão adequado para *fine-tuning* instrucional. Uma opção é JSONL (um JSON por linha) com campos como `"prompt"` (ou `"instruction"`), `"response"` e talvez um campo de categoria. Por exemplo: `{"instruction": "Explique o que é o Demonstrativo de Resultado de Exercício (DRE).", "response": "O DRE é um relatório financeiro que... (explicação baseada no corpus)..."}`.
- **Cuidado com dados sensíveis:** garanta que nas respostas não apareçam nomes reais ou informações pessoais identificáveis que possam estar nos documentos (a não ser que sejam informações públicas, como nome de um ministro em uma portaria pública). Se necessário, anonimize nomes de pessoas ficticiamente (ex: "Funcionário X" em vez de nome real) para evitar problemas com a LGPD.

Meta ao final da etapa: Ter um **dataset robusto de conhecimento corporativo brasileiro** – composto pelo corpus de documentos textuais segmentados e por milhares de pares de Q&A relevantes – pronto para alimentar as próximas fases [7] [8]. Esse material servirá tanto para **indexação no sistema de busca** quanto para **treinamento do modelo de linguagem**, garantindo que o projeto tenha base em conteúdo real e útil do dia a dia empresarial.

2. Implementação do Sistema de Recuperação *HybridRAG*

Com o dataset em mãos, implementaremos o módulo de **retrieval** que fornecerá contexto para geração de respostas. A arquitetura será de RAG *híbrido*, combinando busca vetorial densa com busca

baseada em conhecimento estruturado (grafo/ontologia), além de elementos de busca lexical tradicional [9] [10]. O objetivo é maximizar a **cobertura e precisão** na recuperação de informações relevantes para cada pergunta do usuário.

Ontologia de domínio e grafo de conhecimento: Inicie criando uma ontologia corporativa básica para organizar conceitos-chave e suas relações [11] [12]. Essa ontologia servirá como um **grafo de conhecimento** para apoiar consultas mais complexas. Recomendações:

- **Defina entidades e relações principais:** por exemplo, tipos de entidade podem incluir *Empresa*, *Setor*, *Regulamentação*, *Pessoa-chave*, *Indicador Financeiro*, etc. Defina relacionamentos relevantes, como *Empresa pertence a Setor*, *Empresa é regulada por* (tal órgão ou norma), *Norma refere-se a* (tal assunto), etc.
- **Extração de triplas do corpus:** utilize técnicas de *Relation Extraction* para povoar o grafo automaticamente [12]. Uma abordagem é usar um modelo transformer pré-treinado para *triplet extraction* (por exemplo, um modelo do HuggingFace para *OpenIE* ou *spaCy* com regras de dependência) a fim de extrair relações sujeito-verbo-objeto dos textos. Por exemplo, de uma frase "A empresa X adquiriu a empresa Y em 2022", extrair (X, adquiriu, Y). Concentre-se em relações que caibam na ontologia definida (ignorando relações irrelevantes).
- **Construção do grafo:** utilize uma estrutura de grafo em Python (como **networkx** para um grafo leve em memória, ou uma base RDF pequena, se preferir) para inserir as entidades e relações extraídas. Assegure que os nós tenham identificadores consistentes (ex: mesmo nome de empresa mapeia para um único nó). Inclua também no grafo nós que representem documentos importantes (um nó para cada Norma, Relatório, etc., ligado às entidades que nele aparecem).
- **Uso do grafo na recuperação:** Desenvolva funções de consulta ao grafo que, dado um termo ou entidade presente na pergunta, retornem nós relacionados. Por exemplo, se a pergunta menciona "LGPD", o grafo pode retornar a entidade *LGPD (Lei 13.709)* e seus relacionamentos (como *órgão regulador: ANPD*, ou *tipo: Lei*, *ano:2018*). Essas informações poderão ajudar a contextualizar a busca vetorial (por exemplo, expandindo a query com termos relacionados via grafo, como descrito adiante).

Indexação vetorial com ChromaDB: Em paralelo, configure o mecanismo de busca vetorial para permitir recuperação semântica de trechos textuais relevantes [13] [10]. Escolhemos **ChromaDB** pela facilidade de uso local (na fase de desenvolvimento) e possibilidade de migração para um serviço gerenciado como Pinecone na produção [13]. Passos para implementar:

- **Gerar embeddings densos:** Utilize um modelo de *Sentence Embeddings* multilíngue de alta qualidade, garantindo suporte robusto ao português. O plano propõe o modelo **BGE-m3** [10] – modelo open-source multilíngue do BAAI – que gera representações vetoriais adequadas para captura de similaridade semântica geral. Esse modelo tem a vantagem de suportar múltiplos idiomas e até produzir diferentes tipos de embedding (denso e possivelmente esparso) em uma passada. Como alternativa, você pode considerar modelos do **SentenceTransformers** como `multi-qa-mpnet-base-dot-v1` (treinado para QA) ou o **BERTimbau** adaptado para embeddings, mas o BGE-m3 já está indicado como otimizado para buscas multilíngues.
- **Indexação no ChromaDB:** Crie uma instância do Chroma e insira cada chunk de documento com seu embedding associado. Guarde metadados úteis (título, fonte, etc.). Configure o Chroma para usar uma busca aproximada eficiente (provavelmente ele usará índice HNSW internamente). Certifique-se de persistir o índice em disco para reutilizar sem reprocessar tudo em cada execução (Chroma permite especificar um diretorio persistente).
- **Busca vetorial densa:** Implemente a função de consulta que, dado uma pergunta do usuário, gera seu embedding (mesmo modelo usado nos documentos) e busca no Chroma os *n* vetores mais similares. Isso trará trechos possivelmente relevantes por semelhança semântica. Ajuste *n*

(por exemplo, top 5 ou 10) conforme necessário – você pode começar com top 10 para não perder contexto e depois refinar.

- **Busca lexical esparsa (BM25):** Para complementar a busca densa, incorpore também uma busca por termo-chave usando BM25 [10]. Apesar de embeddings capturarem bem similaridade de significado, às vezes detalhes exatos (códigos de lei, números, nomes específicos) são melhor encontrados por busca lexical. Implementação:
- Concatene todos os chunks em um pequeno índice textual – por exemplo, usando a biblioteca **rank-bm25** (simples de usar em memória) ou **Whoosh** para um índice mais avançado. Como a quantidade de dados pode ser grande, avalie performance; talvez limite a busca lexical a campos específicos (por ex, títulos ou sumários) para não sobrecarregar.
- Ao receber a pergunta, remova stopwords em português (use lista do NLTK ou spaCy) e aplique o algoritmo BM25 para obter, por exemplo, os top 5 documentos/chunks por correspondência exata. Isso identificará resultados que têm termos coincidentes importantes com a pergunta.
- **Combinação de resultados (fusão de evidências):** Desenvolva lógica para **unir os resultados** das buscas densa e esparsa. Por exemplo, obtenha top 10 pela vetorial e top 5 pela BM25 e una em uma lista (removendo duplicatas). Você pode simplesmente concatenar e depois filtrar, ou atribuir um peso a cada método e ordenar por uma pontuação combinada. Uma abordagem simples: normalizar os escores de similaridade (cosine) e BM25 e somar com pesos iguais. Ou, para algo mais elaborado, use um modelo de *learning to rank* futuramente.

Re-ranking por relevância contextual: Aplique uma etapa de *re-ranking* nos resultados unificados para melhorar a precisão [14]. Ideia: utilizar um modelo *cross-encoder* que avalia a relevância de cada trecho em relação à pergunta, produzindo uma pontuação melhor informada. Por exemplo:

- Considere usar um modelo do tipo **MonoT5** treinado em português. O *MonoT5* original (em inglês) é um modelo T5 que reranqueia documentos; existem variações traduzidas ou você pode usar um *mT5* ou XLM-R fine-tunado para QA em português. Se não encontrar um facilmente, uma alternativa é usar **MiniLM CrossEncoder** multilingue do *SentenceTransformers* (que dá uma pontuação de relevância).
- Implementação: para cada candidato recuperado, concatene a pergunta e o texto do trecho (p.ex.: "`Pergunta: ... Trecho: ...`") e alimente no cross-encoder para obter uma pontuação de relevância. Faça isso nos top 10-15 candidatos para não ficar muito lento.
- Ordene os resultados pela pontuação do cross-encoder (descartando aqueles com pontuação muito baixa). Assim, mesmo que a busca vetorial traga um trecho sem relação exata, o re-ranker pode rebaixá-lo se não encaixar bem na pergunta. Isso tende a melhorar a precisão final dos contextos apresentados ao modelo gerativo [14].

Expansão de consultas (Query Expansion): Implemente técnicas para reformular a pergunta do usuário automaticamente antes da busca, aumentando o *recall* [15]. Isso ajuda a evitar casos em que termos diferentes impeçam a correspondência. Abordagens possíveis:

- **Sinônimos e termos relacionados:** Utilize a ontologia/grafo criado – por exemplo, se a pergunta menciona “folha de pagamento”, você pode acrescentar termos como “salário” ou “holerite” na busca. Da mesma forma, “demissão” pode expandir para “rescisão”, etc. Use um dicionário de sinônimos em português (como **NLTK WordNet** para português, ou uma lista customizada de termos corporativos).
- **Expansão via contexto do grafo:** Se a pergunta mencionar uma entidade presente no grafo (ex: uma empresa ou norma), recupere do grafo nós conectados e extraia palavras-chave relevantes. Por exemplo, se pergunta cita “Norma X”, o grafo poderia sugerir “regulamentação Y relacionada” ou “setor Z” e você inclui esses termos na query textual.
- **LLM-based paraphrasing:** Outra ideia é usar um modelo de linguagem (como nosso próprio modelo fine-tunado, quando pronto, ou um modelo menor) para gerar variantes da pergunta.

Ex: “Como calcular a margem EBITDA?” -> expandir para “Como é feita a apuração da margem EBITDA?” ou “Qual é a fórmula da margem EBITDA?”. Combine os termos chave dessas variantes na busca.
- Implemente a expansão de forma **opcional e controlada** – por exemplo, só aplique se a busca inicial retornar poucos resultados ou se detectar termos muito genéricos. E mantenha rastreável o que foi adicionado (para poder explicar ou debugar resultados).

Integração e testes do módulo de recuperação: Desenvolva a interface do *retriever* como uma função ou classe que dada uma pergunta retorna um conjunto de documentos/trechos relevantes e, possivelmente, dados do grafo pertinentes [16]. Garanta que:

- A resposta inclua tanto o texto dos trechos quanto referências (IDs ou títulos) para permitir a exibição das fontes depois. Poderá ser útil retornar, por exemplo, uma lista de objetos `{document_id, score, text, metadata}`.
- Se o grafo foi usado, retorne também as informações encontradas (por ex., “termo X relacionado a Y no grafo”) para eventualmente serem usadas no *prompt* do modelo ou mostradas como notas.
- **Performance:** Faça testes unitários e de integração medindo o tempo de resposta para uma busca típica [17]. Otimize para atingir latência **< 1 segundo** por consulta no banco vetorial local [17]. Dicas de otimização: mantenha o processo de busca vetorial residente em memória (não reler índice do disco a cada chamada), reutilize modelos carregados (embeddings, cross-encoder) evitando carregar a cada vez, e eventualmente paralelize a aplicação do re-ranker nos candidatos usando threads ou AsyncIO.
- **Precisão:** Verifique qualitativamente os resultados para diferentes tipos de perguntas. Ajuste parâmetros como número de documentos retornados, peso da fusão vetorial vs lexical, etc., para equilibrar *recall* e *precisão*. Por exemplo, se notar muitos resultados irrelevantes, diminua *n* ou seja mais restritivo na expansão de consulta; se faltar conteúdo, aumente *n* ou amplie a expansão.

Meta ao final da etapa: Ter um componente *HybridRAG* funcional, capaz de dado uma pergunta do usuário **retornar trechos de documentos relevantes e (opcionalmente) subgrafos pertinentes** do conhecimento corporativo [16]. Esse módulo será a base para o passo de geração, garantindo que o modelo de linguagem opere com conhecimento atualizado e específico das fontes fornecidas.

3. Fine-Tuning do Modelo Mistral 7B Instruct com QLoRA

Com o corpus preparado e o sistema de recuperação implementado, o próximo passo é **especializar o modelo de linguagem** para que ele possa entender perguntas corporativas em português e fornecer respostas corretas, fundamentadas e alinhadas ao tom empresarial. Usaremos o modelo aberto **Mistral 7B Instruct v0.3** como base, realizando *fine-tuning* eficiente via **QLoRA** [18]. Esse método nos permitirá ajustar o modelo em uma GPU de 24GB sem esgotar recursos, adicionando conhecimento do nosso dataset de instruções.

Configuração do ambiente de treinamento: Prepare uma máquina com GPU potente – idealmente uma **GPU com ≥24 GB VRAM** (por exemplo, NVIDIA RTX 4090, A5000 ou A100) [19]. Instale as bibliotecas necessárias:

- **Hugging Face Transformers** (versão atualizada, ex: 4.33+), para carregar o modelo Mistral 7B e gerenciar o treinamento.
- **bitsandbytes** (versão compatível, ex: 0.41) para suportar quantização em 4-bit (NF4) e otimizadores de 8-bit.

- **PEFT (Parameter-Efficient Fine-Tuning)** para aplicar LoRA no modelo facilmente.
- **Unsloth** (opcional) – uma biblioteca que acelera e otimiza fine-tuning de LLMs, especialmente em QLoRA. Pode ser instalada via pip diretamente do GitHub. O *Unsloth* open-source promete **reduzir em ~62% a VRAM** necessária e acelerar em ~2.2x o treino do Mistral 7B em uma A100 [20], graças a técnicas como FlashAttention v2 e otimizações customizadas. Considere usar Unsloth se enfrentar lentidão ou perto do limite de memória.

Preparação dos pesos e quantização: Baixe os pesos do **Mistral 7B Instruct v0.3** (disponível no HuggingFace Hub). Utilize a função `from_pretrained` do Transformers com argumentos para carregar em modo quantizado de 4 bits. Exemplo:

```
model = AutoModelForCausalLM.from_pretrained("mistral-7b-instruct-v0.3",
    device_map="auto", load_in_4bit=True, dtype=torch.float16,
    quantization_config=BitsAndBytesConfig(load_in_4bit=True,
llm_int8_threshold=0))
```

Isso carregará o modelo quantizado em *int4* (formato NF4) economizando memória. Fixe o modelo em modo *inference* (parâmetros congelados) pois só treinaremos os deltas do LoRA.

Configuração do LoRA: Defina os hiperparâmetros do adaptador LoRA para ajuste fino. Com base no projeto, utilizaremos um **ranke relativamente alto (r=64)** para capturar nuances do português técnico e jurídico [21]. Parâmetros sugeridos:

- **Rank (r):** 64 – número de dimensões das matrizes de baixo ranque. É um valor alto que aumenta a capacidade de aprendizado de detalhes, mas consome mais VRAM (se necessário, poderíamos reduzir para 32 para economizar memória, mas tentar manter 64 conforme proposto).
- **Alpha:** 16 – escalamento interno do LoRA (hiperparametro que muitas vezes se coloca igual ao rank, aqui está 16 conforme o projeto, talvez para regularização) [21].
- **Camadas a aplicar LoRA:** normalmente em modelos Transformer aplica-se LoRA nas projeções de atenção e talvez nas MLPs. O default do PEFT para causal LM costuma ser aplicar em `q_proj`, `v_proj` das atenções. Para captar mais, pode-se incluir `k_proj` e `o_proj`, e nas camadas feed-forward `gate_proj`, `down_proj`, `up_proj`. (No Unsloth, por exemplo, eles aplicam LoRA nesses 8 locais [22]).
- **Dropout do LoRA:** 0 – geralmente não se usa dropout adicional em LoRA, a menos que o dataset seja muito pequeno e haja overfitting, então poderíamos introduzir leve dropout. Inicialmente, mantenha 0.

Configure o objeto `LoraConfig` do PEFT com esses valores, e envolva o modelo:

```
lora_config = LoraConfig(r=64, lora_alpha=16,
target_modules=["q_proj","k_proj","v_proj","o_proj","gate_proj","down_proj","up_proj"],
bias="none", task_type="CAUSAL_LM")
model = get_peft_model(model, lora_config)
```

Verifique via `model.print_trainable_parameters()` se apenas ~0.1% dos parâmetros estão treináveis (deve listar algo como 6M de 7B).

Hiperparâmetros de treinamento: Configure o treinamento com hiperparâmetros adequados ao nosso dataset (~15k exemplos) [23] e aos limites de hardware:

- **Tamanho de lote efetivo (batch size):** ~128 exemplos [24]. Como não caberá 128 exemplos de 2048 tokens na GPU de 24GB de uma vez, utilize **gradiente acumulado**. Por exemplo, batch_size_per_step = 8 e `gradient_accumulation_steps = 16` para atingir 128 efetivo. Ajuste conforme a capacidade exata – monitore uso de memória e ajuste para evitar *OOM*.
- **Comprimento de sequência (seq_length):** 2048 tokens. Mistral 7B Instruct v0.3 deve suportar 2048 tokens de contexto. Configure o *trainer* para truncar/expandir as sequências do dataset de instruções para esse comprimento. Se a maioria das nossas perguntas e respostas for curta (<512 tokens combinados), podemos usar um seq_length menor (ex: 1024) e depois estender na inferência se necessário.
- **Épocas:** 2–3 épocas completas sobre o dataset de ~15k pares [23]. O projeto estima ~2–3 horas por época em GPU 24GB [19], então 2–3 épocas seriam ~6–9 horas de treino. Monitorar a perda e as métricas de validação (ver abaixo) a cada época para decidir se para em 2 ou vai para 3.
- **Taxa de aprendizado (LR) e scheduler:** comece com LR inicial de 1e-4 [21]. Utilize um *scheduler* de decaimento cosseno com *warmup*. Por exemplo: 100 passos de *warmup* (ou ~2-3% do total de passos) subindo linearmente de 0 a 1e-4, depois decaimento cosinedecay até ~0 no final. O Transformers Trainer possui `CosineLearningRateSchedule`.
- **Otimizador:** use **AdamW 8-bit** (fornecido pelo bitsandbytes) para eficiência. Isso reduz a memória do otimizador drasticamente sem perda significativa de qualidade. Configure weight_decay=0.01.
- **Gradient Checkpointing:** habilite para economizar VRAM durante o forward (trade-off com tempo). Com `model.gradient_checkpointing_enable()`, o consumo de memória GPU baixa significativamente permitindo batch maior, ao custo de refazer computações na backprop (que é aceitável).
- **Mixed Precision:** ative bfloat16 ou float16 mixed precision (`fp16=True` no Trainer) para acelerar computação na GPU e economizar memória, se não já coberto pela quantização (apesar de quantizado, a backprop nos LoRA pesos ainda pode usar fp16).

Loop de treinamento e validação contínua: Utilize o `Trainer` da HuggingFace para conduzir o treinamento, passando os *callbacks* necessários para avaliação periódica. Monte um **conjunto de validação** separado (algumas centenas de perguntas-respostas não usadas no treino, ou use as próprias perguntas de teste definidas na etapa de validação) e a cada, digamos, 500 passos, avalie o modelo. Duas formas de avaliar:

- **Métricas automatizadas estilo RAGAS:** Para simular o uso real, alimente algumas perguntas de validação ao pipeline completo (retrieval + geração) do modelo atual e avalie *faithfulness*, *relevancy* etc., possivelmente usando um LLM avaliador conforme o framework RAGAS [25]. Isso é complexo de fazer durante treino, mas você pode simplificar: fixe 5 perguntas de validação, recupere contextos com o retriever e gere respostas com o modelo em treinamento (em modo eval, sem gradiente). Em seguida, compare cada resposta com os documentos de contexto fornecidos medindo *faithfulness* (percentual de frases da resposta encontradas nos documentos) e peça a um modelo avaliador (pode ser GPT-4 via API ou um modelo instrucional forte) para dar nota de relevância. Automatizar isso em cada checkpoint pode ser lento, então talvez faça uma avaliação qualitativa manual em dois pontos: após 1 época e ao final.
- **Métricas tradicionais (BLEU/ROUGE-L/BERTScore):** Mais fácil de automatizar durante treino. Para o conjunto de validação, compare as respostas geradas pelo modelo com as respostas esperadas do dataset (já que são Q&A pares). Calcule **BLEU** e **ROUGE-L** para medir similaridade de texto [26]. Use **BERTScore** com modelo multilingue (ou mesmo BERTimbau) para ter uma métrica de similaridade semântica mais robusta para português. Integre essas métricas no

callback de avaliação do Trainer para ter uma noção se estão melhorando ou se o modelo está aprendendo.
- **Monitoramento de overfitting:** Observe se a perda de treino continua caindo enquanto as métricas de validação param de melhorar ou pioram – nesse caso pare o treino (early stopping) ou reduza LR. Com ~15k exemplos, 2-3 épocas provavelmente bastam.

Após o treinamento, você terá obtido um **modelo ajustado (LoRA + quantização)** pronto para inferência. Teste manualmente algumas perguntas do seu dataset passando pelos documentos recuperados e veja se as respostas estão corretas e no tom adequado (profissional e claro). O modelo deve conseguir, dado um *prompt* contendo os documentos relevantes e a pergunta, **gerar respostas corretas, completas e em linguagem apropriada ao ambiente corporativo** [27].

Dica – fusão dos pesos (opcional): Se desejar exportar um único modelo final (por exemplo, para uso mais fácil), você pode *mesclar* os pesos LoRA de volta no modelo base após o fine-tuning. A biblioteca PEFT oferece `merge_and_unload()` para aplicar os deltas no modelo original (cuidado: isso transforma de volta em FP16, consumindo mais VRAM, mas pode ser salvo em formato half). Alternativamente, mantenha o modelo base + LoRA separado para uso, pois continuar em 4-bit + LoRA é mais leve para servir na API.

4. Desenvolvimento da API RESTful com FastAPI

Com o módulo de recuperação e o modelo fine-tunado prontos, encapsularemos tudo em uma **API RESTful** robusta, que permitirá interagir com o sistema via chamadas HTTP. Optamos pelo framework **FastAPI (0.110+)** por sua performance e suporte a async, o que será útil para lidar com requisições sem bloquear o servidor [28]. A arquitetura deve seguir princípios de *clean architecture*: separar claramente a lógica de negócio (RAG + geração) das camadas de interface e de acesso a dados [29]. A API também deve incorporar autenticação, controle de uso e conformidade com LGPD.

Design dos endpoints principais: Planejamos os seguintes endpoints REST na versão 1 da API (prefixo `/api/v1/`), cobrindo funcionalidades de chat, gerenciamento de documentos e monitoramento [30]:

- **POST `/api/v1/chat/completions`:** Endpoint para enviar uma pergunta do usuário e obter uma resposta gerada pelo assistente, fundamentada nos documentos internos. Esse é o coração do sistema (similar em conceito à API do OpenAI ChatCompletion).
- *Request:* incluir um JSON com pelo menos a pergunta (`question` ou `prompt`). Opcionalmente, poderá aceitar um histórico de conversa para contexto (implementação de chat multi-turn) – mas inicialmente podemos começar com perguntas independentes.
- *Processamento:* no handler deste endpoint, implementar a lógica para: (1) autenticar o usuário (ver seção de segurança abaixo); (2) usar o módulo de **retrieval HybridRAG** para obter trechos relevantes nos documentos para a pergunta; (3) formatar um *prompt* para o modelo contendo a pergunta e os contextos (por exemplo: `"Documentos relevantes:\n[1] ...\n[2] ... \nPergunta: ...\nResposta:"`); (4) gerar a resposta usando o modelo de linguagem ajustado (via `model.generate` ou pipeline do HuggingFace). Considere configurar um *timeout* razoável, já que a geração pode levar 1-5 segundos dependendo do tamanho da resposta.
- *Response:* retornar um JSON com a resposta gerada. Inclua também as **fontes** utilizadas – por exemplo, uma lista dos documentos ou trechos que foram fornecidos como contexto (poderíamos retornar identificadores ou pequenos trechos). Isso é fundamental para transparência (e para a interface exibir as referências). Formato sugerido: `{"answer": "...texto da resposta...", "sources": [ {"doc_id": "...", "snippet": "..."} , ... ] }`.

- *Observação:* Considere paginar ou limitar o tamanho da resposta se for muito longa, ou oferecer um parâmetro para comprimento máximo. Para chat multi-turn, você pode manter em memória um dicionário {user_id: chat_history} e incluir as últimas interações no *prompt*, mas tenha cautela com o tamanho total (não exceder 2048 tokens).

- **POST** `/api/v1/documents`**:** Endpoint para upload/inserção de novos documentos na base de conhecimento do sistema [31]. Isso permite atualizar o índice com informações específicas de uma empresa, por exemplo.

- *Request:* pode usar `multipart/form-data` para upload de arquivo (PDF, DOCX) com campo `file`, ou receber JSON com um campo `text` e metadados. Suporte a PDF é prioritário.
- *Processamento:* Ao receber um arquivo, validar o tipo (restringir a formatos suportados). Use **PyMuPDF** para extrair texto se PDF. Se DOCX, usar **python-docx** ou converter para txt (alternativamente exigir PDF para simplificar). Depois, reutilize a pipeline de processamento da etapa 1: limpeza, segmentação em chunks e geração de embeddings para esses novos trechos. Insira-os no **ChromaDB** (ou vetor-store que estiver usando). **Atualize também a ontologia/grafo** se pertinente: por exemplo, se um documento novo menciona entidades novas, passe-o pelo extrator de triplas para incorporar novas relações.
- *Response:* retorne status de sucesso e talvez quantos chunks foram adicionados ou um ID do documento. Ex: `{"status": "ok", "chunks_indexed": 5}`. Em caso de erro (formato não suportado, PDF vazio etc.), retorne código 400 com mensagem adequada.

- *Considerações:* Mantenha atenção à **anonimização** aqui – se um usuário fizer upload de um documento interno que contém dados pessoais (nomes, CPFs), você deve ter uma opção de anonimizar. Poderia ser um parâmetro `anonymize=true` na requisição que, se setado, aciona uma função para encontrar padrões de PII (por ex., usando regex para CPF/CNPJ, ou usando spaCy NER para `PERSON`, `ORG`, `LOC`) e substituir por placeholders antes de indexar. Assim, você evita armazenar dados sensíveis em texto pleno no índice [32]. Documente para o usuário que a anonimização remove informações pessoais e que a LGPD está sendo seguida (privacy by design).

- **GET** `/api/v1/analytics`**:** Endpoint para obter métricas de uso e desempenho da API [33]. Útil tanto para admins quanto para exibirmos no dashboard da interface.

- *Processamento:* Este handler pode agregar informações dos logs ou de um pequeno banco de dados interno. Métricas básicas a incluir: número total de perguntas respondidas, número de documentos inseridos, tempo médio de resposta (talvez o p95 também), taxa de erro ou de perguntas sem resposta, etc. Se houver diferentes usuários, poderia dar breakdown por usuário.
- *Implementação:* Uma solução simples é manter contadores em memória/disk (ex: um Redis incrementando a cada pergunta respondida) e tempos acumulados. Porém, como teremos poucos usuários em testes, até mesmo uma variável global incrementada (com proteção a concorrência) pode servir. Em produção, preferiria armazenar logs estruturados e consultar.
- *Response:* retornar JSON do tipo: `{"total_queries": 123, "avg_response_time_ms": 850, "documents_indexed": 20, "queries_last_24h": 15, ...}`. Defina claramente quais métricas fornecer. Evite expor dados muito sensíveis (por ex., não listar perguntas dos usuários aqui, apenas números agregados, para privacidade).

- **GET** `/api/v1/compliance`**:** Endpoint voltado a fornecer relatórios de conformidade e auditoria [34]. Isso demonstra transparência e adesão à LGPD, permitindo verificar, por exemplo, decisões automatizadas.

- *Processamento:* Pode retornar informações como: registro das últimas N interações do sistema (sem detalhes pessoais, talvez mostrando apenas tipo de consulta e confirmação de que fontes foram apresentadas), políticas de privacidade aplicadas, e estatísticas de quantos dados pessoais foram anonimizados.
- Um possível retorno: `{"privacy_policy": "... resumo ...", "automated_decisions_log": [ {...} ], "personal_data_handling": {...} }`. Por exemplo, *automated_decisions_log* poderia listar as últimas consultas que envolveram decisões automáticas (talvez se o sistema chegou a aconselhar algo crítico) com timestamp e indicativo de que foram registradas.
- Se implementado, esse endpoint seria útil principalmente para administradores ou para demonstração da preocupação com compliance. Na interface, poderia mostrar um texto "Sistema em conformidade com LGPD. Veja /compliance para detalhes."

Segurança e controle de acesso: Garantir segurança é primordial antes de liberar a API externamente [35]. Assegure:

- **Autenticação via OAuth2/JWT:** Implemente um fluxo simples de OAuth2 com **JSON Web Tokens** para proteger os endpoints [35]. Em FastAPI, pode-se usar `OAuth2PasswordBearer` para obter um token mediante usuário/senha numa rota `/api/v1/auth/token`. Para a prova de conceito, armazenar credenciais em um banco simples (ou até um dict em memória para usuários de teste). Ao receber username/password válidos, gerar um JWT assinado (use uma secret key forte) contendo o user id e talvez escopo. Configure os endpoints para exigir *Bearer token* no header Authorization. Assim, somente usuários autenticados poderão usar `/chat`, `/documents`, etc.
- Use bibliotecas como **PyJWT** ou a integração do FastAPI with OAuth2. Defina tempo de expiração razoável (ex: 1 hora) para os tokens e talvez um refresh flow se necessário.

- Para não complicar muito, você pode fornecer um token fixo para os avaliadores usarem na demonstração (por exemplo, gerar manualmente um JWT válido e entregar junto com a documentação, simulando um usuário "admin"). Mas documente e já deixe implementado o mecanismo para futuras expansões.

- **Autorização e escopos:** Se houver múltiplos usuários com diferentes níveis, implemente verificações de escopo/permissão. Por exemplo, somente um usuário admin poderia acessar `/analytics` ou `/compliance`. FastAPI permite verificar escopos embutidos no JWT. No TCC, talvez não haja essa diferenciação, mas é bom mencionar.

- **Rate limiting:** Para evitar abusos (DOS ou uso indevido), implemente *rate limiting*. Por exemplo, limite cada token a, digamos, 5 requisições por segundo ou 100 por hora, etc. Você pode usar um middleware como **SlowAPI** (que integra com FastAPI) que utiliza Redis para contar requests por IP ou por chave de API. Ou implementar manualmente um sistema básico: manter em memória timestamps das últimas requests por usuário e rejeitar com 429 Too Many Requests se ultrapassar certo limiar. Como teremos poucos usuários em teste, talvez não seja acionado, mas demonstre a preocupação.

- **Logging e auditoria:** Configure logging estruturado em toda requisição, para auditoria e depuração [36]. Inclua nos logs: timestamp, usuário, endpoint acessado, duração da requisição, e talvez um hash ou ID único da requisição. FastAPI permite usar middleware para logging. Utilize o **logging** do Python ou biblioteca como **structlog** para imprimir JSON de logs. Armazene logs em arquivo rotacionado.

- Tome **cuidado para não logar conteúdo sensível**: por exemplo, não registre o texto completo das perguntas ou respostas, ou os documentos enviados, pois podem conter dados confidenciais. Se precisar registrar algo para análise de erros, considere truncar ou mascarar dados pessoais (ex: logue só 100 primeiros caracteres da pergunta).
- Esses logs serão úteis inclusive para preencher o endpoint `/compliance`, provendo rastreabilidade de decisões automatizadas (ex: uma entrada de log poderia indicar "Consulta X realizada, fontes A, B usadas, resposta dada").
- **Anonimização e privacy-by-design:** Incorpore mecanismos de proteção de dados pessoais diretamente na API. Já mencionamos a anonimização de documentos enviados. Adicionalmente, se identificar que a pergunta do usuário contém um dado pessoal (ex: "O CPF 123.456.789-10 está na lista de aprovados?"), a API pode interceptar e decidir: ou mascarar o dado antes de processar ou rejeitar conforme políticas. Isso é avançado, mas ao menos documente essas preocupações. Talvez marque no log ou na resposta quando dados pessoais foram detectados e tratados.
- Siga as diretrizes da **LGPD**: mantenha um aviso claro (por exemplo, no `/compliance` ou na documentação da API) de que tipos de dados pessoais o sistema pode processar e como são protegidos (anonimização, criptografia em repouso se aplicável, etc.).

Estrutura interna da aplicação: Mantenha a separação de camadas. Por exemplo, crie módulos Python como:

- `retrieval.py` – com funções `retrieve_documents(question)` encapsulando o passo 2 (talvez carregando globalmente o index para rápida utilização).
- `generation.py` – com função `generate_answer(question, docs)` que formata o prompt e usa o modelo para gerar a resposta. Inclua aqui prompts fixos ou instruções sistêmicas se necessário (como "Responda em português formal e cite as fontes.").
- `api/routers` – módulos definindo os endpoints (chat, docs, etc.), que chamam as funções de negócio acima. Isso ajuda a manter o código organizado e facilitaria trocar o framework web no futuro se preciso (Clean Architecture).

Durante o desenvolvimento, **teste cada endpoint individualmente** com ferramentas como *curl* ou *Postman*, simulando casos de sucesso e erro. Escreva alguns testes unitários/integrados (usando pytest + HTTPX, por exemplo) para verificar que `/chat` retorna resposta contendo alguma das fontes do `/documents` após você inserir um documento, etc.

Ao final desta etapa, teremos uma **API backend funcional**, integrando nosso *retriever* e *gerador*, e pronta para ser consumida pelo front-end. A API estará protegida por autenticação JWT, terá limites básicos de uso e logging para auditoria [35]. Estaremos prontos então para desenvolver a interface web que consumirá esses serviços.

5. Interface em React para Uso e Demonstração

Para tornar a solução acessível e demonstrável a usuários finais e avaliadores, desenvolveremos uma **interface web interativa** usando **React 18 + TypeScript** [37]. Essa interface servirá como um *frontend* simples, simulando uma aplicação corporativa consumindo a API. Os principais componentes da interface serão: um chat com o assistente virtual, uma tela para upload de documentos e uma seção para visualizar métricas e status do sistema.

Setup do projeto React: Inicie um novo projeto React (usando *create-react-app* com template TypeScript, ou Vite + React para mais agilidade). Organize a estrutura em componentes funcionais usando Hooks. Considere utilizar um framework de UI modular para ganhar produtividade, como **Chakra UI** ou **Material-UI**, para facilmente montar elementos de formulário, botões, tabelas etc., com design consistente.

Autenticação (login): Implemente uma tela de login simples se a API exigir JWT. Um formulário que coleta usuário e senha, envia para `/auth/token` da API, e armazena o token JWT recebido (no localStorage ou em um state global). Configure o cliente HTTP (pode usar **axios** ou fetch) para adicionar o header Authorization Bearer <token> em todas requisições subsequentes. Para simplicidade, você pode também habilitar no backend um **CORS** liberal (origem do localhost dev) para permitir essas requisições durante desenvolvimento.

Componente de Chat Interativo: Esta é a parte central da interface:

- Apresente um campo de entrada de texto onde o usuário digita a pergunta, e um botão *Enviar* (ou tecla Enter) para submeter. Opcionalmente, suporte *multi-turn*: mantenha o histórico de conversas em um estado (um array de {role, text}) e exiba em formato de chat (balões de usuário e do assistente). Porém, lembre-se: se implementarmos multi-turn, o frontend precisará enviar também o histórico ao backend ou gerenciar localmente. Para TCC, é aceitável mesmo um comportamento de *single-turn* (pergunta-resposta independente), contanto que isso esteja claro.
- Ao enviar a pergunta, chame o endpoint `/chat/completions` via fetch/axios. Trate o estado de *loading*: exiba um indicador de carregamento enquanto aguarda a resposta. Assim que a resposta chegar, acrescente ao histórico a mensagem do usuário e a resposta do assistente.
- **Exibição da resposta com fontes:** Formate a resposta do assistente em um balão ou card diferenciado (cor diferente, alinhar à esquerda vs. usuário à direita, etc.). Abaixo do texto da resposta, liste as fontes utilizadas. Por exemplo: mostrar algo como "**Fontes consultadas:** Documento X (trecho Y), Relatório Z..." com possibilidade de o usuário clicar para talvez ver mais detalhes. Os dados para isso vêm do campo `"sources"` retornado pela API. Você pode exibir apenas os títulos ou identificadores dos documentos, ou um trecho curto de cada (limitado a 1–2 linhas) para dar contexto. Isso é importante para **transparência**, mostrando que a resposta tem embasamento [38].
- Garanta que a interface seja *scrollable* conforme o histórico cresce. Use CSS flexbox ou similar para o painel de chat ocupar altura disponível.
- (Opcional avançado): implementar resposta *streaming* – FastAPI pode enviar eventos SSE ou *chunks* da resposta. Mas isso é complexidade adicional; pode ser ignorado se a latência estiver aceitável (1-2s).

Componente de Upload de Documentos: Permita que o usuário (avaliador) insira novos documentos para testar a capacidade da API de aprender novos dados em tempo real.

- Faça um formulário com campo de seleção de arquivo (restrinja a PDF ou texto para evitar casos não tratados) e um botão de upload. Ao submeter, chame o endpoint `/documents`. Mostre feedback: por ex., "Arquivo carregado com sucesso, 5 novos documentos indexados." ou erros retornados (arquivo inválido etc.).
- Após um upload bem sucedido, você pode limpar o formulário e possivelmente atualizar algum estado local que indica quantos docs já foram enviados, ou re-carregar as métricas.
- Lembre de incluir instruções para o usuário do tipo "Envie documentos PDF para adicionar ao conhecimento do assistente (dados serão anonimizados automaticamente conforme a LGPD)." Isso reforça a confiança e demonstra as features de compliance (se implementadas) [32].

Componente de Métricas (Analytics Dashboard): Apresente alguns indicadores para avaliar o desempenho do sistema durante os testes.

- Chame o endpoint `/analytics` periodicamente (ou ao abrir a tela) para obter as métricas agregadas [39]. Exiba-as em um pequeno painel: por exemplo, use componentes de cartão ou lista para mostrar *Total de perguntas respondidas*, *Documentos indexados*, *Tempo médio de resposta*, *Taxa de acertos* (se calcularmos algum acerto), etc.
- Poderia incluir um pequeno gráfico de barras ou linha mostrando número de queries por dia (se armazenado), mas dado o escopo, um simples texto/numero já serve.
- Se houver dados qualitativos (ex: % de respostas com fontes, % de consultas que envolveram dados pessoais), destaque-os também.

Considerações de UX e implementação:

- **Responsividade:** garanta que a interface funcione bem em tamanhos diferentes (pelo menos desktop e tablet). Use unidades flexíveis e evite largura fixa exagerada.
- **Estilo e branding:** Dê à interface um visual limpo e corporativo. Talvez use as cores azul ou verde (comuns em dashboards corporativos). Coloque o título do projeto e uma breve descrição no topo da página.
- **Mensagens de erro e bordas:** trate erros das requisições: se `/chat` retorna erro (ex: 500 ou 401 se token expirou), mostre uma notificação ao usuário ("Houve um erro na geração da resposta, tente novamente."). Se `/documents` falhar, exiba "Upload falhou: motivo...". Utilize componentes de alerta ou toast do framework escolhido para isso.
- **Privacidade na UI:** Inclua na interface indicativos de privacidade. Por exemplo, no rodapé ou em uma seção "Sobre/Compliance", mencionar: "Este sistema opera em conformidade com a LGPD. Dados pessoais fornecidos podem ser anonimizados. Nenhuma informação sensível é armazenada sem consentimento." Isso alinha com o *privacy by design* apresentado.
- **Teste de usabilidade:** Antes da demo final, teste a interface com colegas: a interação está clara? O chat e o upload funcionam como esperado? Faça ajustes para torná-la o mais intuitiva possível, já que avaliadores podem não ser técnicos.

Ao finalizar, teremos uma aplicação web onde o avaliador poderá: **fazer perguntas ao assistente virtual, ver as respostas com referências, enviar novos documentos para atualização, e monitorar métricas de performance.** Esta interface servirá como prova concreta do funcionamento da API, facilitando demonstrações e coleta de feedback.

6. Validação Automatizada e Qualitativa

Nesta etapa, validaremos rigorosamente a solução, tanto através de **métricas automatizadas quantitativas** quanto por **avaliação humana qualitativa**, para assegurar que o objetivo do projeto foi atingido. A validação deve medir a **eficácia técnica** (desempenho do RAG e do modelo) e a **utilidade prática** (satisfação de usuários especialistas), bem como verificar conformidade legal e robustez.

Avaliação quantitativa automatizada: Desenvolva um conjunto de **testes e métricas** para medir o desempenho do sistema de forma objetiva. Planeje pelo menos dois cenários de teste: (i) comparação direta com sistemas comerciais e (ii) métricas de qualidade intrínseca das respostas.

- **Conjunto de perguntas de teste:** Prepare ~200 perguntas de **teste** cobrindo diversos domínios corporativos e níveis de complexidade [40]. Por exemplo: 50 perguntas de Recursos Humanos (férias, contratação, direitos trabalhistas), 50 de Finanças (indicadores, relatórios, obrigações fiscais), 50 Jurídicas/Compliance (LGPD, contratos, normas específicas) e 50 de TI/Gestão

(ferramentas, melhores práticas). Inclua tanto perguntas **diretas** (factual, resposta curta) quanto **analíticas** (que exigem síntese de informações de diferentes fontes) [40] . Muitas dessas perguntas podem ser similares às do dataset de instruções, mas não exatamente iguais (para realmente testar generalização).

- **Baseline vs. nosso modelo:** Para estabelecer referência, execute essas perguntas em **modelos/API comerciais conhecidos** sem conhecimento dos documentos internos [41] . Por exemplo, chame a API do **OpenAI GPT-4** (ou GPT-3.5) e **Anthropic Claude 2** com cada pergunta, *sem fornecer nossos documentos*. Isto simula a capacidade de um serviço genérico responder com conhecimento público até 2025. Salve essas respostas. (Atenção aos custos de API, você pode limitar a talvez 50 perguntas para GPT-4 dada a viabilidade).
- **Execução no nosso sistema:** Em paralelo, rode nosso sistema (via API desenvolvida) para responder às mesmas perguntas, garantindo que ele use seu *corpus* interno. Como estamos testando a solução completa, faça isso via a API `/chat/completions` para incluir todo o fluxo RAG. Armazene as respostas obtidas.
- **Métricas RAG (via RAGAS/TruLens):** Utilize frameworks de avaliação de RAG para medir aspectos chaves nas respostas coletadas [42] . O trabalho cita *RAGAS* e *TruLens*, que fornecem métricas como:
- **Faithfulness (Fidedignidade):** Percentual de afirmações na resposta que são suportadas pelas fontes fornecidas [43] . Em TruLens isto é chamado de *groundedness*. Essencialmente mede o quanto a resposta se mantém fiel aos documentos e não alucina. Calcule automatizadamente: o RAGAS pode usar um LLM para verificar cada frase da resposta contra as fontes. Idealmente, esperamos alta pontuação, >90%, indicando poucas alucinações.
- **Answer Relevancy (Relevância da Resposta):** O quanto a resposta realmente responde à pergunta feita [44] . Também mensurável via LLMs avaliadores (ex.: "Dê nota 1-5 se a resposta foi pertinente").
- **Context Precision e Recall:** Métricas que avaliam se o módulo de recuperação trouxe os documentos corretos [45] . *Context Precision* seria quantos dos documentos trazidos eram realmente necessários/úteis, e *Context Recall* se todos os documentos necessários foram trazidos. Essas métricas ajudam a analisar a parte de retrieval independentemente da geração. RAGAS suporta isso fazendo comparação entre contextos e alguma referência.
- **RAGAS score geral:** É possível combinar todos os anteriores em um score global. Utilize a biblioteca **ragas** (disponível via pip) para facilitar – ela integra com LangChain/LlamaIndex para rodar essas avaliações [46] [47] .
- **Métricas tradicionais (texto-para-texto):** Além das acima, calcule métricas clássicas comparando as respostas do nosso sistema com respostas de referência ou com as dos modelos comerciais:
- Se você tiver criado **gabaritos** para as perguntas (respostas ideais curtas), use **BLEU** e **ROUGE-L** para ver similaridade (embora para QA aberta essas métricas nem sempre capturam qualidade, podem indicar grosso modo cobertura de conteúdo) [26] .
- Use **BERTScore** com um modelo de língua portuguesa (ex: `neuralmind/bert-base-portuguese-cased`) para medir a similaridade semântica da resposta do nosso sistema vs. uma resposta esperada ou vs. a do GPT-4.
- **Distorções/Alucinações:** Tente também detectar automaticamente alucinações usando o próprio GPT-4: peça para julgar se a resposta contém algo não suportado. Ou simplesmente use a métrica de faithfulness acima como indicativo (ex: percentagem de frases não encontradas nas fontes).
- **Comparação de resultados:** Analise os scores médios do nosso modelo vs. GPT-4/Claude. Esperamos que para perguntas dependentes dos documentos internos, o nosso sistema supere significativamente os baselines em *relevância* e *fundamentação*, já que GPT-4 sem acesso aos documentos pode dar respostas genéricas ou desatualizadas. Documente casos interessantes:

por exemplo, se GPT-4 respondeu “não sei” ou chutou errado, enquanto nosso assistente respondeu corretamente citando a fonte X – isso evidencia o valor do RAG + fine-tuning.

- **Desempenho não funcional:** Meça também a **latência média** de resposta do nosso sistema e a taxa de erros. No log ou via `/analytics`, confira se a meta de tempo médio **< 2s no p95** foi atingida [48]. Verifique a taxa de consultas em que o sistema não encontrou resposta ou alucinou (>5% seria preocupante segundo a meta) [48]. Se acima do desejado, anote para discussões de melhoria.
- (Opcional) **Teste de carga:** não é foco principal, mas se possível dispare várias requisições concorrentes para ver se a API aguenta (ex: usando `locust` ou `hey` com 10 rps). Verifique se o rate limiting funciona e se a memória se mantém estável.

Avaliação qualitativa (humana): Além dos números, é fundamental verificar a **qualidade percebida** e a adequação das respostas no contexto real de uso. Planeje um pequeno estudo com especialistas de domínio e possivelmente implementações piloto:

- **Feedback de especialistas:** Convide **5 profissionais** de áreas distintas (por exemplo: um advogado, um contador, um gerente de RH, um gestor de TI, um gerente de projetos) [49]. Cada um receberá acesso à interface web e um conjunto de **tarefas simuladas** relacionadas à sua área. Por exemplo:
- Para o advogado: “Use o assistente para verificar se a cláusula X do contrato está em conformidade com a lei Y” ou “Pergunte sobre as penalidades da LGPD para incidentes de vazamento de dados.”
- Para o contador: “Pergunte qual foi o lucro líquido da empresa ABC em 2022 (um dado presente em um relatório anual carregado).”
- Cada especialista deve interagir livremente com o chat, talvez fazendo de 5 a 10 perguntas cada, inclusive seguindo perguntas de clarificação (testando um pouco a continuidade do chat).
- **Questionário Likert:** Após usar o sistema, cada especialista responde a um breve questionário Likert (1–7) avaliando vários critérios [49]:
- **Utilidade das respostas:** As respostas ajudaram a resolver a tarefa? (1 – nada úteis, 7 – extremamente úteis)
- **Clareza e linguagem:** As respostas foram claras, bem escritas e em tom adequado?
- **Confiabilidade percebida:** O especialista sentiu confiança nas respostas? (ex: porque havia fontes, ou porque as info batiam com conhecimento dele)
- **Facilidade de uso da interface:** A interação (chat/upload) foi intuitiva, rápida, sem problemas?
- **Aderência à privacidade:** O usuário percebeu se dados pessoais foram tratados adequadamente? (por ex, se notou alguma anonimização ou aviso; ou se ficou preocupado em algum momento com privacidade).
- **Coleta e análise:** Compile as respostas do questionário e calcule médias. Observe comentários abertos que eles fizerem – muitas vezes eles podem apontar limitações, casos de erro ou sugestões. Por exemplo, se um advogado notar que a resposta citou uma lei desatualizada, isso é insight importante.
- **Estudos de caso em PMEs reais:** Se houver oportunidade, implante a solução experimentalmente em 2–3 pequenas empresas parceiras por umas duas semanas [50]. Pode ser difícil no contexto acadêmico, mas mesmo um teste informal conta. A ideia é observar *anecdotal evidence*: por exemplo, numa PME de contabilidade, ver se o funcionário consegue agilizar alguma consulta usando o sistema. Reúna depoimentos ou exemplos de sucesso/falha. Esses casos enriquecem muito a validação qualitativa.
- **Análise qualitativa:** Consolide os pontos fortes e fracos percebidos. Por exemplo, se todos elogiaram a clareza mas 3 de 5 apontaram que em perguntas mais analíticas o sistema às vezes foge do contexto, isso indica onde melhorar. Avalie também a **concordância entre avaliadores humanos e métricas automáticas** – ex: se o RAGAS apontou faithfulness alto mas algum

especialista achou uma resposta duvidosa, investigar a causa (pode ser um erro no documento, ou interpretação).

Resultados esperados: Espera-se que a **solução tenha desempenho comparável ou superior** aos sistemas baseline nos aspectos críticos de relevância e fundamentação [51]. Idealmente, nosso modelo fine-tunado com RAG deve responder com informações corretas suportadas pelas fontes, enquanto os modelos genéricos podem não responder ou dar respostas vagas. As métricas devem refletir isso (p.ex., faithfulness próximo de 100% para nosso, contra bem menor para GPT-4 sem dados). Em termos de percepção, esperamos **aceitação alta** dos especialistas, indicando que o sistema seria útil no dia a dia. Feedback negativo ou taxas de erro >5% devem ser analisados e relatados como pontos a melhorar.

Além disso, avalie os requisitos não-funcionais medidos: se a latência média ficou dentro da meta (<2s) e se o sistema rodou bem em hardware comum (CPU/GPU de teste), indicando viabilidade para uso em PMEs sem infraestrutura de supercomputação [48]. Essa validação completa dará confiança de que atingimos o objetivo do projeto ou embasará as conclusões sobre onde houve trade-offs.

7. Documentação e Repositório Aberto

Por fim, consolidaremos todo o trabalho em uma documentação abrangente e um repositório de código aberto, para garantir **transparência, replicabilidade e continuidade** do projeto. Esta etapa envolve organizar o código, escrever guias de uso e destacar como outras pessoas podem reproduzir ou estender nossa solução.

Organização do repositório (GitHub): Publique o código em uma plataforma como GitHub, em um repositório público (a não ser que haja restrições institucionais). Estruture o repo de forma limpa: por exemplo:

```
/api/         -> código da API FastAPI (main.py, routers/, models/, etc.)
/frontend/    -> código da interface React (ou link para repo separado se
preferir)
/scripts/     -> scripts utilitários (ex: script de preparação de dados,
avaliação)
/notebooks/   -> notebooks Jupyter ilustrando passos (opcional, mas bom para
TCC)
/models/      -> (se permitido) pesos ou adaptadores LoRA treinados, ou um
README link para baixá-los
/data_samples/-> alguns exemplos de dados (ex: pequenos trechos do corpus e
2-3 Q&A de exemplo)
README.md
```

Inclua um README detalhado na raiz explicando o propósito do projeto, arquitetura e instruções rápidas de como rodar. Também adicione, se possível, diagramas simples (pode ser ASCII or draw.io export) mostrando a arquitetura RAG, para ilustrar. Isso ajuda quem chegar ao repositório a entender o sistema.

Guia de instalação e deploy: Forneça **instruções de deploy** da API para que outros possam reproduzir o ambiente [52] . Por exemplo:

- **Requisitos de hardware:** note a necessidade de GPU com 24GB para treino e idealmente para inferência também (embora inferir possa até em 16GB com 4-bit). Sugira colab ou paperspace para quem quiser testar sem GPU local.
- **Setup de ambiente backend:** listar dependências (talvez fornecer um `requirements.txt` ou melhor, um `poetry` / `pipenv` file). Orientar a criar e ativar um venv, instalar requerimentos. Mencionar instalação de PyTorch compatível com a GPU e cuda.
- **Baixar modelos:** instruir como baixar o modelo base Mistral 7B (link do HF) e colocar na pasta correta ou configurar variável. Se fornecer os LoRA, explicar como aplicá-los (ou fornecer já um modelo mesclado se possível, embora 7B half precision ~14GB – talvez pesado para incluir diretamente, então LoRA diffs são menores, publicar no HF Hub ou no repo).
- **Inicializar o sistema:** passo a passo para subir o backend: talvez comandos `uvicorn api.main:app --reload` para dev. Explicar configurações de ambiente (como definir SECRET_KEY para JWT, etc.).
- **Executar front-end:** instruir instalação de dependências (`npm install`), configuração da URL da API (se a API rodar local em outra porta, ajustar, possivelmente via arquivo `.env` ou config JS), e rodar `npm start` . Se for aplicar build estático, explique `npm run build` e como servir (pode até integrar no FastAPI a distribuição estática).
- **Carregar dados iniciais:** se possível, forneça um pequeno conjunto de documentos de exemplo (já anonimizados) para o usuário testar imediatamente. Talvez 2 documentos dummy e 5 Q&A no dataset. Explique como indexá-los (ex: "execute `scripts/index_initial.py` para popular o Chroma com os exemplos").
- **Uso:** mostrar como fazer uma requisição via curl, ou simplesmente "abra o front-end em http://localhost:3000 e teste o chat".

Documentação técnica: Além do README, é importante documentar a **metodologia e design** em um documento (que pode ser a própria monografia). Como parte do projeto acadêmico, você elaborará a monografia com toda a descrição. Garanta que na monografia (TCC) haja uma seção de *Metodologia* e *Reprodução do Experimento* bem clara [53] , contendo detalhes suficientes para alguém replicar. Isso inclui: descrição do dataset, do fine-tuning (hiperparâmetros usados), da arquitetura RAG, das ferramentas escolhidas e justificativa, e dos procedimentos de avaliação. Inclua também os resultados quantitativos e qualitativos obtidos, interpretando-os.

No repositório, forneça também um **package de replicação** conforme citado [52] : por exemplo, notebooks ou scripts que automaticamente baixam um modelo, carregam um subconjunto de dados e executam uma inferência ou cálculo de métricas. Uma ideia: um notebook `RAG_demo.ipynb` que mostra passo a passo pegar um exemplo de pergunta, executar retrieval e depois geração com o modelo fine-tunado, mostrando a resposta e fontes – para quem não quiser rodar a API completa.

Exemplos e templates: Inclua exemplos de configuração, como um arquivo `.env.template` para o backend com campos (SECRET_KEY, etc.), para facilitar setup. Para a interface, talvez um `config.js` ou instrução para setar `REACT_APP_API_URL` . Esses pequenos detalhes poupam tempo de quem for rodar.

Licença e termos: Adicione um arquivo LICENSE (escolha uma licença permissiva como MIT ou Apache-2.0, se não houver impedimento). Devido a uso de dados públicos e modelo open-source, isso deve ser viável. Na documentação, esclareça que, apesar de open-source, **os usuários devem checar compliance com LGPD** se usarem com dados próprios – ou seja, você oferece a ferramenta, mas o uso em produção tem responsabilidade de quem usar, especialmente quanto a dados pessoais.

Contribuição futura: Mencione que o projeto é aberto para contribuições. Por exemplo, sugestões de features (multi-idioma, integração com outros modelos), ou melhorias (ex: treinar com mais dados, adicionar interface de admin). Isso dá vida além do TCC, mostrando preocupação em manter e evoluir.

Por fim, prepare um **relatório final ou artigo** resumindo tudo (provavelmente exigência do TCC). O plano menciona até possibilidade de submeter a um simpósio [54] . Mesmo que isso seja pós-TCC, tenha em mente coletar todos aprendizados e resultados obtidos e escrevê-los de forma coesa. Isso não é entregue na implementação, mas é consequência dela.

Em resumo, ao concluir esta etapa teremos: **todo o código fonte disponível publicamente, documentação passo-a-passo de como reproduzir e usar a solução, e registros dos resultados** obtidos. Assim, qualquer desenvolvedor full-stack experiente poderá seguir o guia e **replicar o sistema em seu próprio contexto**, ou até mesmo dar continuidade ao projeto implementando melhorias, cumprindo o objetivo de entregar um **guia acionável e replicável** para democratizar a IA corporativa nas PMEs [52] .

Meta final: Um repositório organizado, bem documentado, contendo código, exemplos de dados não sensíveis e instruções claras de deploy, de modo que outro profissional consiga entender e reutilizar a solução facilmente [52] . Isso assegura que o legado do TCC vá além da banca, contribuindo de fato para a comunidade e possivelmente gerando interesse em aplicações de IA corporativa no Brasil.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [21] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [48] [49] [50] [51] [52] [53] [54] Plano de execução TCC (1).pdf
file://file-VP9DmbgXSg4QbayVEaAPdn

[20] [22] Unsloth update: Mistral support + more
https://unsloth.ai/blog/mistral-benchmark

[46] [47] RAG Evaluation Survey: Framework, Metrics, and Methods | EvalScope
https://evalscope.readthedocs.io/en/latest/blog/RAG/RAG_Evaluation.html