**AOVS Sistemas de Informática S.A.**  
NICOLAS GUILHERME BATISTA MALYSZ

**AGENTE IA COM GOOGLE & ALURA**

DESENVOLVIDO COM LINGUAGEM PYTHON

VIDEIRA - SC

2025

**RESUMO**

A inteligência artificial (IA) está revolucionando a forma como lidamos com tarefas cotidianas, especialmente em ambientes corporativos. Imagine um assistente virtual capaz de triar consultas de service desk, consultar políticas internas e decidir ações automáticas, tudo de forma eficiente e baseada em dados reais. Foi exatamente isso que aprendi em uma imersão prática oferecida pela Alura em parceria com o Google. Utilizando ferramentas como Google Gemini, Retrieval-Augmented Generation (RAG) e LangGraph, construí um sistema de agentes inteligentes para gerenciar políticas internas de uma empresa fictícia chamada Carraro Desenvolvimento.

Neste artigo, compartilho os passos dessa jornada, destacando os conceitos chave, o código implementado e os insights ganhos. Se você é um entusiasta de IA, desenvolvedor ou profissional de TI/RH, essa experiência pode inspirar você a criar soluções semelhantes. Vamos mergulhar nos detalhes!

**INTRODUÇÃO: *O Que São Agentes Inteligentes e Por Que Importam?***

Agentes inteligentes são sistemas de IA que não apenas respondem perguntas, mas também tomam decisões autônomas baseadas em regras e dados. No contexto de um service desk, eles podem automatizar respostas a consultas sobre políticas internas, como reembolsos, home office ou segurança da informação, reduzindo a carga de trabalho humana e aumentando a eficiência.

Essa prática foi parte de um curso da Alura, integrado com tecnologias do Google Cloud, como o Gemini (um modelo de linguagem avançado). O foco foi em desenvolver um agente que:

- Triasse consultas (decidindo se resolver automaticamente, pedir mais info ou abrir chamado).

- Usasse RAG para consultar uma base de conhecimento (políticas em PDF).

- Orquestrasse fluxos com LangGraph para decisões condicionais.

O notebook Jupyter usado como base demonstra o processo passo a passo, desde a configuração até testes reais. Vamos explorar as aulas principais.

**AULA 01: *Configurando o Agente com Google Gemini***

O primeiro passo foi integrar o Google Gemini para criar um sistema de triagem de consultas. Usamos a API do Gemini para processar mensagens de usuários e decidir ações baseadas em regras pré-definidas.

**Instalação e Configuração**

Instalamos as bibliotecas necessárias:

*!pip install -q --upgrade langchain langchain-google-genai google-generativeai*

**Em seguida, configuramos o modelo:**

*from google.colab import userdata*

*from langchain\_google\_genai import ChatGoogleGenerativeAI*

*GOOGLE\_API\_KEY = userdata.get('GEMINI\_API\_KEY')*

*llm = ChatGoogleGenerativeAI(*

*model="gemini-1.5-flash",*

*temperature=0.0,*

*api\_key=GOOGLE\_API\_KEY*

*)*

**Testamos com uma pergunta simples:**

r*esp\_test = llm.invoke("Que ano foi a independência do Brasil")*

*print(resp\_test.content) # Saída: A independência do Brasil foi proclamada em 1822.*

**Sistema de Triagem**

Criamos um prompt para triar mensagens em categorias: "**AUTO\_RESOLVER**" (perguntas claras sobre políticas), "**PEDIR\_INFO**" (mensagens vagas) ou "**ABRIR\_CHAMADO**" (pedidos de exceções). Usamos Pydantic para estruturar a saída:

*from pydantic import BaseModel, Field*

*from typing import Literal, List*

*class TriagemOut(BaseModel):*

*decisao: Literal["AUTO\_RESOLVER", "PEDIR\_INFO", "ABRIR\_CHAMADO"]*

*urgencia: Literal["BAIXA", "MEDIA", "ALTA"]*

*campos\_faltantes: List[str] = Field(default\_factory=list)*

**Testamos com exemplos:**

- "Posso reembolsar a internet?" → **AUTO\_RESOLVER**, BAIXA

- "Quero mais 5 dias de trabalho remoto." → **ABRIR\_CHAMADO**, MEDIA

Essa etapa ensinou a importância de prompts bem estruturados para guiar modelos de IA, evitando respostas aleatórias.

**AULA 02: Construindo a Base de Conhecimento com RAG**

RAG (Retrieval-Augmented Generation) permite que o agente consulte documentos reais antes de responder, garantindo precisão. Carregamos políticas internas de PDFs e criamos um vetor de busca.

**Carregando e Processando Documentos:**

Usamos PyMuPDF para ler PDFs:

*from langchain\_community.document\_loaders import PyMuPDFLoader*

*from langchain\_text\_splitters import RecursiveCharacterTextSplitter*

**Carregar PDFs de políticas (ex: Home Office, Reembolsos, E-mail)**

*docs = []* ***# Código para carregar múltiplos PDF****s*

*splitter = RecursiveCharacterTextSplitter(chunk\_size=300, chunk\_overlap=30)*

*chunks = splitter.split\_documents(docs)*

**Embeddings e Vetor de Busca**

Integramos embeddings do Google:

*from langchain\_google\_genai import GoogleGenerativeAIEmbeddings*

*from langchain\_community.vectorstores import FAISS*

*embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001", google\_api\_key=GOOGLE\_API\_KEY)*

*vectorstore = FAISS.from\_documents(chunks, embeddings)*

*retriever = vectorstore.as\_retriever(search\_type="similarity\_score\_threshold", search\_kwargs={"score\_threshold": 0.3, "k": 4})*

**Consulta com RAG**

Criamos uma chain para responder apenas com base no contexto:

*from langchain\_core.prompts import ChatPromptTemplate*

*from langchain.chains.combine\_documents import create\_stuff\_documents\_chain*

*prompt\_rag = ChatPromptTemplate.from\_messages([...]) # Prompt restritivo*

*document\_chain = create\_stuff\_documents\_chain(llm, prompt\_rag)*

**Função para perguntar:**

*def perguntar\_politica\_RAG(pergunta: str) -> Dict:*

***# Recupera docs relevantes e responde***

Testes mostraram respostas precisas, como "Sim, até R$100/mês" para reembolso de internet, com citações de fontes.

RAG foi um destaque: ele previne "**alucinações**" da IA, ancorando respostas em dados reais, o que é crucial para compliance em empresas.

**AULA 03: *Orquestrando o Agente com LangGraph***

LangGraph permite criar fluxos de decisão como grafos, tornando o agente "raciocinante".

**Definindo o Estado e Nós**

**Usamos TypedDict para o estado:**

f*rom typing import TypedDict, Optional*

*class AgentState(TypedDict, total=False):*

*pergunta: str*

*triagem: dict*

*# Outros campos…*

Nós para triagem, auto-resolver, pedir info e abrir chamado.

**Construindo o Grafo**

*from langgraph.graph import StateGraph, START, END*

*grafo = StateGraph(state\_schema=AgentState)*

*# Adiciona nós e arestas condicionais*

*app = grafo.compile()*

Testes com o grafo decidiram ações corretamente, como abrir chamado para exceções.

LangGraph trouxe flexibilidade: o agente "pensa" em etapas, simulando raciocínio humano.

**Testes e Resultados**

Testamos com várias perguntas:

- Reembolso de internet: Resolvido automaticamente com citações.

- Mais dias remotos: Chamado aberto.

- Perguntas vagas: Pedido de mais info.

Os resultados foram consistentes, destacando como a IA pode otimizar processos internos.

**CONCLUSÃO: *Lições Aprendidas com Alura e Google***

A imersão com **Alura** e **Google** foi uma experiência incrível que me levou a construir um agente inteligente do zero. Com ***Google Gemini***, ***RAG*** e ***LangGraph***, criei um sistema capaz de triar consultas, consultar políticas internas e tomar decisões com precisão, tudo aplicado ao contexto da fictícia Carraro Desenvolvimento. Cada etapa trouxe aprendizados práticos: configurar o Gemini para triagem, usar RAG para respostas baseadas em PDFs e orquestrar fluxos com LangGraph, dando ao agente um raciocínio quase humano. Ver o sistema responder perguntas como “Posso reembolsar a internet?” com citações claras ou decidir abrir chamados foi gratificante. Mais do que código, aprendi a usar IA para resolver problemas reais com eficiência e compliance, unindo tecnologia e prática de forma fluida e envolvente.