

Chatbot do Kronos

Feito por:
Giovanna Pelati
Júlia Penna
Dmitri Kogake
Theo Correia

2ºI

DESCRIÇÃO DO PROJETO

Visão geral

O Chatbot do Kronos foi desenvolvido com o objetivo de criar um assistente virtual inteligente integrado ao aplicativo Kronos, um sistema de gerenciamento de tarefas fabris que organiza a rotina diária dos funcionários, melhora a produtividade e garante a continuidade dos processos industriais.

Atuação do chatbot

O chatbot atua como um guia interativo, auxiliando usuários iniciantes a se familiarizarem com o aplicativo e suas funcionalidades, utilizando linguagem acessível e respostas precisas.

TECNOLOGIAS UTILIZADAS

Tecnologia	Uso no Projeto
gemini-2.0-flash / gemini-2.5-flash	LLMs para geração de respostas e moderação
LangChain	Framework de orquestração dos agentes
gemini-embedding-001	Geração de embeddings para RAG
MongoDB	Armazenamento de histórico e base de conhecimento
FastAPI + Render	API e hospedagem

OBJETIVO DO SISTEMA

O chatbot foi projetado para responder dúvidas sobre o uso do aplicativo Kronos, como:

Como posso desatribuir uma tarefa?

Como envio um arquivo para justificar minha ausência?

Como funciona a análise GUT?

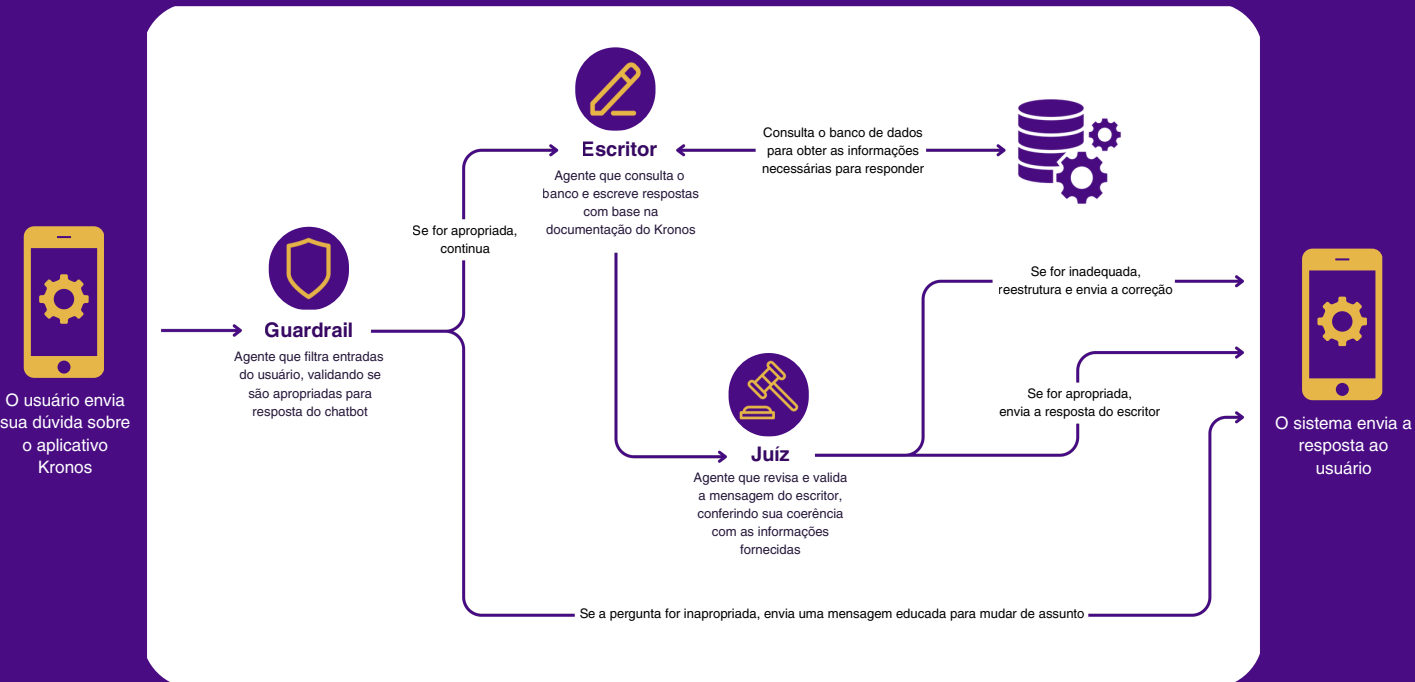
Além disso, ele deve identificar perguntas inadequadas, manter o contexto da conversa e gerar respostas coerentes e seguras.

ARQUITETURA GERAL

O chatbot é composto por três agentes que operam em sequência dentro de uma pipeline de execução:

Agente	Função	Modelo e Configuração	Formato de Saída
Guardrail	Atua como filtro de segurança, validando se a entrada do usuário é apropriada.	gemini-2.0-flash e temperatura 0 (respostas determinísticas e rápidas).	Retorna um JSON estruturado indicando se a entrada é válida ou inválida, junto de uma mensagem educada quando necessário.
Escritor	Utiliza o serviço de Recuperação de Contexto (RAG) para buscar informações relevantes no banco e gerar a resposta final.	gemini-2.5-flash e temperatura 0.3 (respostas mais elaboradas e contextuais).	Retorna a resposta final ao usuário com base no contexto recuperado.
Juiz	Valida e refina a resposta gerada, garantindo coerência, clareza e conformidade com o contexto.	gemini-2.0-flash e temperatura 0 (respostas determinísticas e rápidas).	Retorna um JSON estruturado indicando se a resposta é válida ou inválida, junto de uma correção quando necessário.

Fluxograma de Arquitetura do Chatbot



EXEMPLOS DE RESPOSTA

Input	Output
Oi! Como funciona a redistribuição de tarefas?	<p>Oi! A redistribuição de tarefas no Kronos funciona de duas maneiras principais:</p> <ol style="list-style-type: none">1. Automática: Se um operário não estiver disponível, a tarefa é automaticamente realocada para outro colaborador que seja apto, considerando a disponibilidade, setor, habilidades e prioridades. 🔄2. Por solicitação do operário: Se você precisar desatribuir uma tarefa, deve clicar no botão 'Desatribuir' e justificar o motivo. Um supervisor irá analisar sua solicitação antes que a tarefa seja realocada. ✅ <p>Espero que ajude! 😊</p>
Como faço para colocar um arquivo para justificar minha ausência?	<p>Para anexar um arquivo e justificar sua ausência, você deve ir ao calendário 📅. Lá, ao registrar sua ausência, você poderá justificar o motivo por texto e, se necessário, anexar arquivos como fotos de atestados médicos 📎.</p>
O Kronos também controla as finanças da fábrica?	<p>O sistema Kronos é focado no gerenciamento de tarefas fabris, na organização da rotina dos funcionários, na melhoria da produtividade e na garantia da continuidade do processo. Não há informações nos documentos disponíveis sobre o controle das finanças da fábrica 📁. Para confirmar essa informação, você pode consultar seu supervisor ou gestor.</p>

ESTRUTURA DE PASTAS E ARQUIVOS

- |
- |— agents/ Contém a lógica de cada agente.
 - | |— guardrail_agent.py
 - | |— judge_agent.py
 - | |— rag_agent.py
 - | |— prompts/ Contém os prompts de configuração.
 - | | |— guardrail/
 - | | |— judge/
 - | | |— rag/
 - | | |— fewshots.json Cada agente possui few-shots.
 - | | |— system_prompt.txt Cada agente possui um prompt de sistema.
- |
- |— db_scripts/ Scripts de banco de dados e documentos base.
 - | |— docs.json Base de conhecimento (FAQ).
 - | |— populate_db.py Popula o MongoDB com os documentos.
 - | |— generate_embeddings.py Cria embeddings vetoriais com gemini-embedding-001.
- |
- |— services/ Serviços auxiliares do chatbot.
 - | |— memory_service.py Recupera o histórico de conversas no MongoDB.
 - | |— rag_service.py Fluxo RAG com cosine similarity, retorna os top 3 resultados.
- |
- |— tests/ Conjunto para testes automáticos.
 - | |— questions/
 - | |— answers/
 - | | |— default.json Testes com perguntas padrão do FAQ.
 - | | |— exception.json Testes com entradas inadequadas.

		memory.json	Testes de memória de sessão.
		hallucination.json	Testes de verificação de alucinações.
		tests.py	Script que executa as perguntas e armazena as respostas no answers/.
		docs/	Documentação do chatbot.
		estrutura_de_agentes.png	Imagem do Fluxograma de Arquitetura.
		entrega_do_chatbot.pdf	Este arquivo!
		pipeline.py	Define a execução sequencial dos agentes e o tratamento de histórico.
		main.py	Implementa as rotas da API usando FastAPI.

INFORMAÇÕES ADICIONAIS

Formato de Saída do Guardrail e Juiz

Os dois agentes seguem o mesmo padrão básico: outputs estruturados em JSON com dois campos, "flag" e "message".

No Guardrail:

- "flag": indica se a entrada é válida.
 - 0 → entrada válida
 - 1 → entrada inválida ou ofensiva
- "message": mensagem educada orientando o usuário a mudar de assunto; retorna None quando "flag" = 0.

No Juiz:

- "flag": indica se a resposta do Escritor é adequada.
 - 0 → resposta adequada
 - 1 → resposta inadequada
- "message": resposta reestruturada e corrigida; retorna None quando "flag" = 0.

Dessa forma, os retornos garantem padronização e clareza na comunicação entre os agentes, além de assegurar a condução coerente e controlada do fluxo de respostas.

Histórico de Conversas

Além de armazenar o FAQ, o MongoDB também é responsável por registrar o histórico de conversas do chatbot.

Por meio da classe `MongoDBChatMessageHistory`, cada interação recebe um `session_id` único, e as mensagens do usuário e do chatbot são salvas na coleção `conversations`. Durante a execução do arquivo `pipeline.py`, o histórico é atualizado sempre que o usuário envia uma pergunta ou um agente gera uma resposta, registrando inclusive qual agente produziu a mensagem enviada ao usuário.