






# ✨ Desvendando os Segredos dos Agentes de IA: Sua Aventura com PydanticAI, Streamlit, A2A e MCP! ✨

E aí, futuro mestre dos bots!  Prepare-se para uma jornada épica pelo universo da Inteligência Artificial. Este guia é o seu mapa do tesouro para desvendar quatro tecnologias incríveis que vão turbinar seus projetos de IA: **PydanticAI**, **Streamlit**, o protocolo **A2A** e o conceito de **MCP**.


## O que você vai descobrir nesta aventura?

1.  **PydanticAI**: O cérebro turbinado para seus agentes de IA.
2.  **Streamlit**: Sua varinha mágica para criar interfaces web interativas sem dor de cabeça.
3.  **A2A (Agent-to-Agent)**: O aperto de mão secreto para seus agentes conversarem e colaborarem.
4.  **MCP (Model Component Protocol)**: O tradutor universal para agentes compartilharem suas ferramentas.

Ao final desta jornada, você não apenas entenderá o que são essas ferramentas, mas também construirá um **mini-chatbot** com dois agentes que trabalham em equipe! Um Agente Mestre que coordena as tarefas e um Agente Aprendiz que executa uma função específica. E o melhor: tudo isso com uma interface super amigável feita em Streamlit.

## Por que isso é tão legal?

Imagine criar assistentes virtuais que não só entendem o que você diz, mas também podem usar ferramentas, acessar informações e até mesmo pedir ajuda a outros assistentes para completar tarefas complexas. É como ter sua própria equipe de super-heróis digitais! Com as tecnologias que vamos explorar, construir esses sistemas se torna muito mais simples e divertido.

Este guia foi feito pensando em você, jovem padawan da IA: com linguagem descontraída, exemplos práticos e sem enrolação. O objetivo é que você aprenda, se divirta e, o mais importante, se sinta inspirado a criar seus próprios projetos incríveis. Então, pegue seu café (ou seu suco preferido), ajuste sua cadeira de piloto e vamos decolar! 

---

## Capítulo 1: PydanticAI - O Cérebro Turbinado dos Seus Bots

Se você quer construir chatbots ou agentes de IA que realmente entendem e agem, o PydanticAI é seu novo melhor amigo! Pense nele como um "cérebro" programável que você pode moldar para entender perguntas e executar tarefas específicas.<sup>1</sup>

## Por que o PydanticAI é tão demais?

- 🤖 **Simples de Aprender:** Poucos conceitos para você dominar e já sair criando.
- 🛡️ **Seguro como um Cofre:** Ele usa a "checagem de tipos" do Python para pegar erros antes mesmo que eles aconteçam em produção. É como ter um corretor ortográfico para o seu código, que te avisa na hora se algo está esquisito.<sup>1</sup> Outros frameworks muitas vezes só descobrem o erro quando o programa já está rodando, o que pode ser uma baita dor de cabeça!
- ✂️ **Flexível que só ele:** Funciona com os modelos de linguagem mais famosos, como OpenAI (ChatGPT), Google (Gemini), Anthropic (Claude) e outros.<sup>1</sup>
- 💪 **Poderoso Nível Hard:** Seus bots podem executar código Python de verdade, o que abre um universo de possibilidades!

## Os Blocos de Construção do PydanticAI (Modo Fácil):

Para entender o PydanticAI, você precisa conhecer alguns dos seus "blocos de montar" principais:

### 1. Agent (O Agente):

- **O que é?** É o coração e o cérebro do seu chatbot. Ele processa as mensagens que recebe e decide o que fazer em seguida.<sup>1</sup>
- **Como funciona?** Você diz a ele qual modelo de IA usar (tipo, o "motor" do cérebro, como o gpt-4o-mini) e pode dar a ele uma "personalidade" ou instruções iniciais através de um system\_prompt. Por exemplo: "Você é um assistente super amigável que adora contar piadas sobre dinossauros."

```
Python
# Exemplo de como criar um Agente
from pydantic_ai import Agent

meu_agente_engracado = Agent(
    "openai:gpt-4o-mini", # Modelo de IA econômico e esperto
    system_prompt="Você é um comediante espacial contando fatos sobre planetas."
)
```

### 2. Tools (As Ferramentas Mágicas):

- **O que são?** São superpoderes que você dá ao seu agente! Basicamente, são funções Python normais que o agente pode decidir usar sozinho quando achar necessário para responder a uma pergunta ou executar uma tarefa.<sup>1</sup>

- **Como funciona?** Você define uma função Python e a "decora" com `@agent.tool`. A descrição da função (a docstring) é super importante, pois ajuda o agente a entender o que a ferramenta faz e quando usá-la.

```
Python
# Exemplo de uma Tool
@meu_agente_engracado.tool
def calcular_piada_cosmica(planeta: str) -> str:
    """Cria uma piada curta sobre o planeta fornecido."""
    return f"Por que {planeta} terminou o namoro com a Lua? Porque achou que ela estava passando por muitas fases! Ba dum tss!"
```

Quando o usuário perguntar algo como "Me conte uma piada sobre Marte", o `meu_agente_engracado` pode pensar: "Opa, tenho uma ferramenta `calcular_piada_cosmica` que parece perfeita para isso!" e usá-la.

### 3. Dependencies (Os Ajudantes Necessários):

- **O que são?** São recursos que suas ferramentas podem precisar para funcionar. Pense em coisas como uma conexão com a internet para buscar dados, acesso a um banco de dados, ou até mesmo um arquivo de configuração.<sup>1</sup>
- **Como funciona?** Você define uma classe especial (usando `dataclass`) que lista tudo o que suas ferramentas podem precisar. O `PydanticAI` tem um sistema de "injeção de dependências" nativo, que é como ter um garçom inteligente que traz automaticamente os ingredientes que sua ferramenta precisa, sem que você tenha que buscá-los manualmente toda vez.<sup>1</sup> Isso deixa seu código mais limpo e fácil de testar.

```
Python
# Exemplo de Dependências
from dataclasses import dataclass
import httpx # Para fazer chamadas na web

@dataclass
class DependenciasDoMeuAgente:
    cliente_http: httpx.AsyncClient # Para buscar coisas na internet
    nome_do_arquivo_secreto: str = "segredos.txt"
```

### 4. RunContext (A Mochila de Ferramentas):

- **O que é?** É a forma como suas ferramentas acessam as Dependencies que você definiu. Pense no `RunContext` (ou `ctx` para os íntimos) como uma mochila mágica que cada ferramenta carrega, e dentro dela estão todos os "ajudantes" (dependências) que ela pode precisar.<sup>1</sup>
- **Como funciona?** Quando você define uma tool, você pode dizer que ela recebe um `ctx`. Através de `ctx.deps`, a ferramenta pode pegar qualquer dependência que foi configurada para o agente.

```

Python
# Exemplo de Tool usando RunContext e Dependências
from pydantic_ai import RunContext

@meu_agente_engracado.tool
async def buscar_fato_especial(ctx: RunContext, topico: str) -> str:
    """Busca um fato interessante sobre um tópico espacial usando uma API externa."""
    try:
        resposta = await
ctx.deps.cliente_http.get(f"https://api.fatos-espaciais.com/fato?topico={topico}")
        resposta.raise_for_status() # Verifica se deu erro na chamada
        return resposta.json()["fato"]
    except Exception as e:
        return f"Puxa, não consegui buscar fatos sobre {topico}. Erro: {e}"

```

Com esses blocos, você já pode começar a montar agentes bem espertos! O PydanticAI cuida de muita coisa por baixo dos panos, como a validação dos dados que entram e saem das suas ferramentas, garantindo que tudo funcione direitinho. Uma das suas grandes vantagens é a "Type Safety REAL": ele te ajuda a pegar erros de tipo *antes* de rodar o código, direto no seu editor, o que economiza um tempo precioso de debugging.<sup>1</sup>

Ele também se destaca pela "Validação Durante Streaming". Isso significa que, enquanto o agente está gerando uma resposta em pedacinhos (streaming), o PydanticAI já vai validando cada pedaço. É como ter um corretor automático que funciona em tempo real enquanto você digita, e não só no final do texto.<sup>1</sup> Isso é ótimo para dar feedback mais rápido ao usuário e evitar desperdício de processamento.

E para fechar com chave de ouro: o PydanticAI é "Python Puro". Você não precisa aprender uma linguagem nova ou um "dialeto" específico de um framework. Se você já sabe Python, já está pronto para usar o PydanticAI, o que acelera muito o aprendizado e aumenta a produtividade da equipe.<sup>1</sup>

---

## Capítulo 2: Streamlit - Sua Varinha Mágica para Interfaces Web



Agora que você já sabe como criar o "cérebro" do seu bot com PydanticAI, que tal dar a ele um "rosto" amigável para que as pessoas possam interagir? É aí que entra o **Streamlit**!

## O que é Streamlit?

Streamlit é uma biblioteca Python que transforma scripts de dados em aplicativos web interativos e bonitos, com pouquíssimas linhas de código. É sério, é quase mágica! Se você sabe Python, pode criar uma interface web sem precisar ser um expert em HTML, CSS ou JavaScript.

### Por que o Streamlit é perfeito para jovens desenvolvedores (e para testar nossos agentes)?

- 🚀 **Rápido como um Foguete:** Você vê as mudanças na sua interface instantaneamente enquanto codifica.
- 😊 **Fácil de Usar e Aprender:** A sintaxe é super intuitiva. Se você pode escrever um script Python, pode construir um app Streamlit.
- 🎨 **Bonito por Padrão:** Os componentes já vêm com um design legal, mas você também pode customizar se quiser.
- 🧩 **Cheio de Componentes Prontos:** Botões, caixas de texto, sliders, gráficos, upload de arquivos e muito mais, tudo pronto para usar.<sup>2</sup>
- 🤖 **Amigo da IA:** Perfeito para criar demos rápidas e interfaces para seus modelos de Machine Learning e agentes de IA.

### Criando uma Interface de Chat Simples com Streamlit:

Para o nosso projeto de testar os agentes A2A, vamos precisar de uma interface de chat básica. Com o Streamlit, isso é moleza!

1. **Entrada de Texto:** Para o usuário digitar a mensagem. Usaremos `st.text_input` ou, melhor ainda para chats, `st.chat_input`.
2. **Botão de Envio:** Para o usuário enviar a mensagem. O `st.button` é perfeito para isso.<sup>2</sup>
3. **Área de Exibição da Conversa:** Para mostrar as mensagens do usuário e as respostas do bot. Podemos usar `st.write`, `st.markdown` ou os mais novos `st.chat_message` para um visual de chat moderno.
4. **Manter o Histórico:** Para que a conversa não suma a cada interação, vamos usar o `st.session_state` do Streamlit. Ele é como uma memória de curto prazo para o seu app.<sup>3</sup>

Veja um exemplo básico de como esses componentes funcionam juntos:

```

import streamlit as st

# Título do nosso app
st.title("Meu Primeiro Chatbot com Streamlit! 💬")

# Inicializar o histórico de mensagens no session_state se ainda não existir
if "historico_mensagens" not in st.session_state:
    st.session_state.historico_mensagens = []

# Mostrar mensagens antigas
for mensagem in st.session_state.historico_mensagens:
    with st.chat_message(mensagem["role"]): # "user" ou "assistant"
        st.markdown(mensagem["content"])

# Pegar input do usuário
prompt_usuario = st.chat_input("Diga algo para o bot...")

if prompt_usuario:
    # Adicionar mensagem do usuário ao histórico e mostrar na tela
    st.session_state.historico_mensagens.append({"role": "user", "content":
prompt_usuario})
    with st.chat_message("user"):
        st.markdown(prompt_usuario)

    # Simular resposta do bot
    resposta_bot = f"O bot diz: Recebi sua mensagem - '{prompt_usuario}'! Em breve, serei mais
inteligente."

    # Adicionar resposta do bot ao histórico e mostrar na tela
    st.session_state.historico_mensagens.append({"role": "assistant", "content":
resposta_bot})
    with st.chat_message("assistant"):
        st.markdown(resposta_bot)

```

Com esse esqueleto, já temos uma base para a interface do nosso chatbot de agentes. No próximo capítulo, vamos conectar essa interface aos nossos agentes PydanticAI e fazê-los conversar de verdade!

O Streamlit também é ótimo para visualização de dados. Se seus agentes trabalhassem com números ou precisassem mostrar informações de forma gráfica, você poderia facilmente adicionar gráficos de linha, barra, área, mapas e muito mais, usando bibliotecas como Altair ou Plotly Express diretamente com o Streamlit.<sup>4</sup> Mas para o nosso chat, manteremos as coisas simples e focadas na conversa.

---

## Capítulo 3: A Mágica da Colaboração - A2A, MCP e Seus Agentes em Ação! 🤝 ✨

Chegamos à parte mais emocionante: fazer nossos agentes de IA trabalharem juntos! Para isso, vamos entender dois conceitos importantes: **A2A (Agent-to-Agent)** e **MCP (Model Component Protocol)**. E depois, partiremos para a prática, construindo nossos agentes Mestre e Aprendiz e conectando-os através de uma interface Streamlit.

### Desvendando os Protocolos Secretos:

- **MCP (Model Component Protocol) - O Cardápio Universal de Ferramentas**



- **O que é?** Pense no MCP como uma forma padronizada para um agente (ou modelo de IA) "anunciar" quais ferramentas, recursos (dados) e prompts ele pode oferecer ou usar.<sup>6</sup> É como um cardápio universal que qualquer outro agente pode ler para saber o que está disponível. Ele ajuda a padronizar como os LLMs interagem com ferramentas externas.<sup>7</sup>
- **Por que é útil?** Imagine que você tem um agente que é ótimo em buscar a previsão do tempo. Com o MCP, ele pode descrever essa habilidade ("ferramenta de previsão do tempo que aceita uma cidade como entrada e retorna o clima") de uma forma que outros agentes entendam facilmente, sem precisar conhecer os detalhes internos de como ele faz isso. A ideia é facilitar a integração e reduzir a dependência de um único fornecedor de LLM.<sup>7</sup>
- **No nosso projeto:** Não vamos implementar um servidor MCP completo (como o FastMCP<sup>8</sup>), mas vamos usar o *conceito*. A descrição da tool do nosso Agente Aprendiz (o que ela faz, quais parâmetros espera) é a nossa "entrada no cardápio MCP".

- **A2A (Agent-to-Agent) - O Canal de Comunicação dos Agentes** 🧠💬

- **O que é?** A2A é um protocolo que define como diferentes agentes de IA podem se comunicar, trocar informações de forma segura e coordenar ações, mesmo que tenham sido construídos com tecnologias ou por equipes diferentes.<sup>10</sup> É como um "tradutor universal" ou uma "linguagem de rede" para

agentes.

- **Por que é útil?** No mundo real, tarefas complexas muitas vezes exigem a colaboração de vários especialistas. Com A2A, podemos construir sistemas onde um agente "gerente" pode delegar subtarefas para agentes "especialistas". Por exemplo, um agente de planejamento de viagens pode pedir a um agente especialista em voos para encontrar passagens e a outro especialista em hotéis para reservar acomodações. O A2A usa tecnologias web comuns como JSON-RPC sobre HTTP(S) para essa comunicação.<sup>10</sup>
- **No nosso projeto:** Vamos *simular* uma comunicação A2A. O Agente Mestre vai "enviar uma mensagem" (chamar uma função) para o Agente Aprendiz pedindo para ele executar sua tarefa especializada.

**A relação entre MCP e A2A é complementar.** O MCP ajuda a definir o *quê* um agente pode fazer (suas ferramentas e capacidades). O A2A fornece o *canal* e as regras para que um agente peça a outro para fazer algo e receba a resposta.<sup>6</sup> Pense assim: MCP é o "cardápio" que o Agente Aprendiz mostra, detalhando os "pratos" (ferramentas) que ele sabe cozinhar e os ingredientes necessários. A2A é o "garçom" que leva o pedido do Agente Mestre para o Aprendiz e traz o prato pronto de volta.

## Mãos à Obra: Construindo Nossos Agentes Colaborativos!

Vamos criar dois agentes:

1. **Agente Aprendiz:** Especialista em criar nomes criativos para pets.
2. **Agente Mestre:** Coordena a tarefa e pede ajuda ao Aprendiz.

### Passo 1: Criando o Agente "Aprendiz" Criativo 🎨🐾

Este agente terá uma única tool: gerar um nome divertido para um pet.

- **Arquivo: agente\_aprendiz.py**

Python

```
from pydantic_ai import Agent, RunContext # RunContext não será usado aqui, mas é bom ter
from dotenv import load_dotenv
import os
```

```
load_dotenv() # Carrega variáveis do .env, como OPENAI_API_KEY
```

```
# Verificação de segurança para a chave da API
```

```
if not os.getenv("OPENAI_API_KEY"):
```

```
    raise ValueError("Chave OPENAI_API_KEY não configurada no arquivo .env! Crie o arquivo e
```



```
adicione sua chave.")
```

```
# Criando o agente aprendiz
```

```
# Usaremos um modelo mais simples e rápido para o aprendiz, como o gpt-3.5-turbo ou gpt-4o-mini
```

```
aprendiz_agent = Agent(
```

```
    "openai:gpt-4o-mini", # Ou "openai:gpt-3.5-turbo"
```

```
    system_prompt="Sou um aprendiz criativo e minha especialidade é criar nomes para pets!"
```

```
)
```

```
@aprendiz_agent.tool
```

```
def gerar_nome_pet(animal: str, caracteristica: str) -> str:
```

```
    """
```

```
    Gera um nome divertido para um animal de estimação baseado em sua espécie e uma característica.
```

```
    Por exemplo, para um 'cachorro' 'brincalhão', poderia ser 'Sir Doguinho Brincalhão'.
```

```
    """
```

```
    # Lógica simples para gerar um nome.
```

```
    # Poderia até chamar um LLM aqui para ser mais criativo, mas vamos manter simples.
```

```
    # A LLM do aprendiz_agent vai usar esta tool e formatar a resposta.
```

```
    # A tool em si retorna o nome, e a LLM do aprendiz pode adicionar um toque.
```

```
    # Para garantir que a tool seja chamada, o system_prompt do aprendiz_agent já o direciona.
```

```
    # E a LLM do mestre_agent vai instruí-lo a usar a tool.
```

```
    nome_sugerido = f"{caracteristica.capitalize()} {animal.capitalize()}son da Silva"
```

```
    print(f"Agente Aprendiz (tool gerar_nome_pet): Sugerindo nome '{nome_sugerido}' para {animal} {caracteristica}")
```

```
    return f"Que tal... {nome_sugerido}? Achei genial!" # A tool retorna, e o LLM do aprendiz pode reformatar
```

- Como o Aprendiz "anuncia" sua habilidade (Conceito MCP):  
A descrição da tool (gerar\_nome\_pet) e seus parâmetros (animal: str, caracteristica: str) são como a "documentação" que o MCP ajudaria a padronizar. Se estivessemos usando uma biblioteca como FastMCP, o decorador @mcp.tool() faria esse trabalho de expor a ferramenta para outros agentes.<sup>8</sup> No nosso exemplo, o Agente Mestre "saberá" dessa ferramenta através da nossa simulação de A2A.

Para que o Agente Mestre peça ao Agente Aprendiz para executar gerar\_nome\_pet, ele precisa saber:

1. Que essa ferramenta existe.
2. Quais informações ela precisa (um animal e uma característica).
3. O que ela devolve (uma string com o nome). O MCP é o protocolo que

formalizaria essa "descrição da ferramenta". O A2A é o protocolo que o Mestre usaria para enviar a *solicitação* e o Aprendiz para enviar a *resposta*.

## Passo 2: Preparando o Agente "Mestre" Coordenador 🏰

O Agente Mestre receberá o pedido do usuário, entenderá que precisa de um nome de pet e "chamará" o Agente Aprendiz para fazer o trabalho sujo (e divertido!).

- **Arquivo: agente\_mestre.py**

Python

```
from pydantic_ai import Agent
from dotenv import load_dotenv
import os
```

```
# Importar o agente aprendiz para simular a chamada A2A
```

```
from agente_aprendiz import aprendiz_agent # Importa a instância do agente aprendiz
```

```
load_dotenv()
```

```
if not os.getenv("OPENAI_API_KEY"):
```

```
    raise ValueError("Chave OPENAI_API_KEY não configurada no arquivo.env!")
```

```
# Criando o agente mestre
```

```
mestre_agent = Agent(
```

```
    "openai:gpt-4o-mini", # Pode ser o mesmo modelo ou um diferente
```

```
    system_prompt="""
```

```
    Você é um Agente Mestre que coordena tarefas. Seu objetivo principal é ajudar usuários a obter  
    nomes criativos para seus pets.
```

```
    Quando um usuário pedir um nome para um pet, você DEVE usar a ferramenta  
'coordenar_geracao_nome_pet'.
```

```
    Extraia o tipo de animal e uma característica do pedido do usuário para passar para a  
    ferramenta.
```

```
    Seja amigável e direto na resposta final, apresentando o nome gerado.
```

```
    """
```

```
)
```

```
# Esta função simula a chamada A2A para o Agente Aprendiz
```

```
async def pedir_nome_pet_ao_aprendiz(animal: str, caracteristica: str) -> str:
```

```
    """
```

```
    Simula o envio de uma tarefa A2A para o Agente Aprendiz  
    para gerar um nome de pet e retorna a resposta.
```

```
    """
```

```
    print(f"Agente Mestre: Contatando Agente Aprendiz para gerar nome para {animal}")
```

```
{caracteristica}.")
```

```
# SIMULAÇÃO DA CHAMADA A2A:  
# O Agente Mestre envia uma instrução para o Agente Aprendiz.  
# O Agente Aprendiz (seu LLM interno) interpreta e decide usar sua tool 'gerar_nome_pet'.  
instrucao_para_aprendiz = f"Gere um nome para um {animal} que é {caracteristica}. Use  
sua ferramenta especializada."
```

```
# O aprendiz_agent já tem sua tool 'gerar_nome_pet' registrada.  
# Ao receber a instrução, seu LLM deve identificar que precisa usar essa tool.  
resposta_aprendiz_obj = await aprendiz_agent.run(instrucao_para_aprendiz)
```

```
# A resposta_aprendiz_obj.output conteria a resposta formatada pela LLM do aprendiz,  
# que idealmente seria o nome gerado pela tool, possivelmente com algum floreio.  
resultado_nome = str(resposta_aprendiz_obj.output)
```

```
print(f"Agente Mestre: Resposta recebida do Aprendiz: '{resultado_nome}'")  
return resultado_nome
```

```
# O Agente Mestre tem uma tool para coordenar com o aprendiz  
@mestre_agent.tool  
async def coordenar_geracao_nome_pet(animal: str, caracteristica: str) -> str:  
    """  
    Pede ao Agente Aprendiz para gerar um nome de pet e retorna o resultado.  
    Esta ferramenta DEVE ser usada quando o usuário solicitar um nome para um pet.  
    """  
    print(f"Agente Mestre (tool coordenar_geracao_nome_pet): Iniciando coordenação para  
{animal} {caracteristica}.")  
    nome_gerado_pelo_aprendiz = await pedir_nome_pet_ao_aprendiz(animal,  
caracteristica)  
    # A tool do mestre retorna diretamente o que o aprendiz respondeu.  
    # A LLM do mestre_agent então usará essa saída para formular a resposta final ao usuário.  
    return nome_gerado_pelo_aprendiz
```

- **A Simulação A2A:** A função `pedir_nome_pet_ao_aprendiz` é nossa simulação de uma chamada A2A. Em um sistema real com um SDK A2A <sup>11</sup>, essa função faria uma chamada de rede (provavelmente JSON-RPC sobre HTTP/S <sup>10</sup>) para o Agente Aprendiz. O SDK A2A esconderia toda essa complexidade de rede, fazendo parecer que você está apenas chamando uma função Python normal. Isso é fundamental para que desenvolvedores possam construir sistemas multi-agentes

complexos sem se afogar nos detalhes de comunicação.

### Passo 3: A Interface Mágica com Streamlit ✨

Agora, vamos criar a interface de chat para que o usuário possa interagir com o Agente Mestre.

- **Arquivo: chatbot\_app.py**

```
Python
import streamlit as st
import asyncio
import nest_asyncio # Necessário para rodar asyncio dentro do loop do Streamlit

# Importar o agente mestre (que já sabe como contatar o aprendiz)
from agente_mestre import mestre_agent

# Aplicar nest_asyncio para permitir loop de eventos aninhado
# Isso é uma forma de contornar a questão do asyncio no Streamlit para este exemplo.
# Em produção, podem existir abordagens mais robustas.
nest_asyncio.apply()

st.set_page_config(page_title="Chat de Nomes para Pets 🚀", layout="centered")

st.title("🚀 Chatbot Intergaláctico de Nomes para Pets 🚀")
st.subheader("Fale com o Agente Mestre e veja a mágica acontecer!")
st.markdown("""
Peça ao Agente Mestre para gerar um nome para seu pet!
Por exemplo:
- "Me dê um nome para um **gato** **preguiçoso**"
- "Preciso de um nome para um **cachorro** **aventureiro**"
- "Sugira um nome para um **papagaio** **falante**"
""")

# Inicializar histórico no session_state
if "messages" not in st.session_state:
    st.session_state.messages = [{"role": "assistant", "content": "Olá! Como posso ajudar a nomear seu pet hoje?"}]

# Mostrar mensagens do histórico
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])
```

```

# Input do usuário
prompt_usuario = st.chat_input("Peça ao Mestre (ex: 'gere um nome para um cão aventureiro')")

if prompt_usuario:
    # Adicionar mensagem do usuário ao histórico e mostrar
    st.session_state.messages.append({"role": "user", "content": prompt_usuario})
    with st.chat_message("user"):
        st.markdown(prompt_usuario)

    # Agente Mestre processa o input
    with st.chat_message("assistant"):
        message_placeholder = st.empty()
        message_placeholder.markdown("Agente Mestre está consultando seus ajudantes... 🤔✨")

    try:
        # O mestre_agent.run() é o ponto de entrada.
        # Sua LLM interna, guiada pelo system_prompt e pela descrição da tool
        'coordenar_geracao_nome_pet',
        # deve decidir usar essa tool, extraindo 'animal' e 'caracteristica' do prompt_usuario.
        response_mestre = asyncio.run(mestre_agent.run(prompt_usuario))
        bot_response = str(response_mestre.output)

        message_placeholder.markdown(bot_response) # Mostra a resposta final do
Mestre
        st.session_state.messages.append({"role": "assistant", "content":
bot_response})

    except Exception as e:
        error_message = f"🚨 Oops! Algo deu errado nos bastidores: {e}"
        st.error(error_message) # Mostra o erro de forma mais destacada
        st.session_state.messages.append({"role": "assistant", "content":
error_message})
        # Opcional: logar o erro completo para debugging
        # st.exception(e)

```

- Nota sobre asyncio em Streamlit:

Rodar código assíncrono (como o `agent.run()` do PydanticAI) dentro do Streamlit, que já tem seu próprio loop de eventos, pode ser um desafio. No código acima, usamos `nest_asyncio.apply()` como uma solução simples para este exemplo didático. Em aplicações mais complexas ou de produção, outras abordagens como executar a lógica assíncrona em uma thread separada (`asyncio.to_thread`) ou usar gerenciadores de loop de evento mais sofisticados podem ser necessárias. O importante aqui é entender o fluxo da conversa entre os agentes!

- **Como Rodar e Testar:**

1. **Crie os arquivos:** `agente_aprendiz.py`, `agente_mestre.py`, `chatbot_app.py`.
2. **Crie o arquivo .env:** Na mesma pasta dos scripts, crie um arquivo chamado `.env` e adicione sua chave da API da OpenAI:

Snippet de código

```
OPENAI_API_KEY=sua_chave_secreta_da_openai_aqui
```

3. **Instale as bibliotecas:** Abra seu terminal ou prompt de comando, navegue até a pasta do projeto e instale o que precisamos (você pode criar um arquivo `requirements.txt` com estas linhas e rodar `pip install -r requirements.txt`):  
`pydantic-ai[openai]` `streamlit` `python-dotenv` `nest-asyncio` (Note que `pydantic-ai[openai]` já inclui `openai` e `httpx` como dependências <sup>1</sup>).
4. **Execute o App Streamlit:** No terminal, na pasta do projeto, rode o comando:

Bash

```
streamlit run chatbot_app.py
```

Uma nova aba deve abrir no seu navegador com a interface do chatbot! Experimente pedir nomes como: "Me dê um nome para um gato dorminhoco" ou "Qual nome combina com um hamster aventureiro?".

## Por Baixo dos Panos (Modo Fácil): Como Nossos Agentes Conversam!

Para entender o fluxo todo, imagine este cenário:

Snippet de código

graph TD

```
A["VOCÊ (Jovem Mestre Jedi) 🧐"] -- "Ex: 'Nome para cão corajoso'" --> B;
B -- "Prompt do usuário" --> C["Agente Mestre (Coordenador Esperto) 🏰🧠"];
C -- "Tool: coordenar_geracao_nome_pet('cão', 'corajoso')\n(Simulação de
```

```

Chamada A2A 🧠)" --> D["Agente Aprendiz (Especialista Criativo) 🎨🐾"];
D -- "Tool: gerar_nome_pet('cão', 'corajoso')\n(Conceito MCP: ferramenta
anunciada 📜)" --> E["LLM do Aprendiz decide usar a tool"];
E -- "'Corajoso Cãozão da Silva!'" --> D;
D -- "Resposta do Aprendiz (Simulação de Resposta A2A)" --> C;
C -- "LLM do Mestre formata resposta final" --> F;
F -- "'Que tal Corajoso Cãozão da Silva?'" --> B;
B -- "Mostra na tela" --> A;

style C fill:#f9f,stroke:#333,stroke-width:2px;
style D fill:#ccf,stroke:#333,stroke-width:2px;

```

### Explicação do Diagrama (Mapa do Tesouro):

1. **Você (Jovem Mestre Jedi)** digita seu pedido na **Interface Streamlit** (seu Painel de Controle Mágico).
2. A interface envia essa mensagem para o **Agente Mestre** (o Coordenador Esperto).
3. O Agente Mestre, usando sua inteligência PydanticAI, entende que você quer um nome de pet. Ele então decide usar sua ferramenta interna `coordenar_geracao_nome_pet`.
4. Essa ferramenta do Mestre faz uma "chamada A2A simulada" para o **Agente Aprendiz** (o Especialista Criativo), passando o animal e a característica.
5. O Agente Aprendiz recebe a instrução. Seu LLM interno, sabendo que ele tem uma tool chamada `gerar_nome_pet` (que foi "anunciada" no estilo MCP, com sua descrição e parâmetros), decide usá-la.
6. A tool `gerar_nome_pet` faz sua magia e cria um nome.
7. O Agente Aprendiz envia o nome de volta para o Agente Mestre (simulando uma resposta A2A).
8. O Agente Mestre recebe o nome, seu LLM pode dar um toque final na formatação da resposta, e a envia de volta para a Interface Streamlit.
9. A interface mostra a sugestão de nome para você. Missão cumprida!

Este fluxo, embora simplificado, demonstra como múltiplos agentes podem colaborar. A beleza de usar padrões como A2A e MCP (mesmo que conceitualmente aqui) é que, à medida que seus sistemas crescem, você pode adicionar mais agentes, cada um com suas especialidades, e eles podem se descobrir e interagir de maneira padronizada. A visualização desse fluxo ajuda a desmistificar o que parece ser uma arquitetura complexa, tornando o funcionamento interno muito mais acessível,

especialmente quando se está começando.

---

## Capítulo 4: Missão Cumprida! Próximos Níveis na Sua Jornada Jedi dos Bots! 🌟

Uau! Você chegou até aqui e, se seguiu os passos, construiu um sistema com dois agentes de IA que conversam, colaboram e ainda por cima têm uma interface de chat super legal feita por você! Isso é simplesmente SENSACIONAL! 🎉🥳

### Recapitulando os superpoderes que você desbloqueou:

- Você aprendeu sobre **PydanticAI** e como ele serve de "cérebro" para seus bots, permitindo que eles entendam linguagem, usem ferramentas e tomem decisões.<sup>1</sup>
- Descobriu o **Streamlit**, sua "varinha mágica" para criar interfaces web interativas de forma rápida e fácil, sem precisar ser um expert em front-end.<sup>2</sup>
- Entendeu o conceito de **A2A (Agent-to-Agent)**, o protocolo que permite que seus agentes "fofoquem produtivamente" e trabalhem em equipe, mesmo que sejam diferentes.<sup>10</sup>
- E pegou a ideia do **MCP (Model Component Protocol)**, que ajuda os agentes a "anunciarem" suas ferramentas e capacidades de forma padronizada, como um cardápio universal.<sup>6</sup>

Juntos, esses componentes formam a base para criar sistemas de IA muito mais sofisticados e modulares. O Agente Mestre coordenando o Agente Aprendiz é só um pequeno exemplo do que é possível!

### Ideias para turbinar seus próximos projetos (Modo Aventura ON!):

Agora que você tem esses novos superpoderes, que tal explorar um pouco mais?

- 🧙 **Mais Ferramentas para o Aprendiz:** Que tal dar ao seu Agente Aprendiz novas habilidades? Ele poderia aprender a:
  - Consultar a previsão do tempo de uma cidade.
  - Contar uma piada (talvez buscando em uma API de piadas!).
  - Buscar a definição de uma palavra.
  - (Inspirado nas ideias de "Próximos Passos" do PydanticAI <sup>1</sup>).
- 🧑 **Um Time de Aprendizes:** E se o Agente Mestre pudesse conversar com *vários* Agentes Aprendizes, cada um especialista em uma coisa diferente? Um para nomes, outro para piadas, um terceiro para fatos curiosos... Isso mostra o poder de escalabilidade da comunicação A2A!
- 🎨 **Interface Streamlit Nível PRO:** Explore mais os recursos do Streamlit para



deixar sua interface ainda mais irada!

- Adicione imagens ou GIFs divertidos.
- Se o bot trabalhar com dados, mostre gráficos interativos.<sup>4</sup>
- Mude as cores, fontes e o layout para deixar com a sua cara!
- 🧠 **Experimente Outros Cérebros (LLMs):** O PydanticAI facilita a troca do modelo de linguagem. Tente usar outros modelos da OpenAI (como gpt-4) ou explore modelos do Google (Gemini) ou Anthropic (Claude) para ver como seus agentes se comportam de maneira diferente.<sup>1</sup>
- 📁 **Memória de Longo Prazo:** Faça seu chatbot lembrar de conversas anteriores salvando o histórico em arquivos JSON ou até mesmo em um banco de dados, como ensinado no guia do PydanticAI.<sup>1</sup>

### Uma mensagem final para te inspirar a continuar criando!

O universo dos agentes de IA é vasto, dinâmico e cheio de possibilidades incríveis esperando para serem descobertas. O que você aprendeu e construiu aqui é uma base sólida, um trampolim para voos muito mais altos!

Lembre-se: a curiosidade é seu superpoder mais valioso. Continue perguntando "e se...", continue experimentando novas ideias (mesmo as mais malucas!) e, acima de tudo, continue construindo coisas que te empolguem. Cada linha de código, cada novo agente, cada interface que você cria é um passo adiante na sua jornada como desenvolvedor de IA.

**Para se aprofundar ainda mais, aqui estão alguns links úteis (como os tesouros no final do mapa!):**

- **PydanticAI:**(<https://docs.pydantic.dev/pydantic-ai/>){:target="\_blank"} (Veja também os exemplos em <sup>1</sup>)
- **Streamlit:**(<https://docs.streamlit.io/>){:target="\_blank"}
- **FastMCP (para entender mais sobre servidores MCP):**(<https://www.firecrawl.dev/blog/fastmcp-tutorial-building-mcp-servers-python>){:target="\_blank"} <sup>8</sup>
- **OpenAI (para explorar modelos):** [Plataforma OpenAI](#){:target="\_blank"}

Que a Força (dos Bots) esteja com você, sempre! Continue aprendendo, criando e se divertindo. O futuro da IA está sendo construído agora, e você faz parte disso!



### Referências citadas

1. Exclusividades.pdf

2. st.button - Streamlit Docs, acessado em junho 11, 2025,  
<https://docs.streamlit.io/develop/api-reference/widgets/st.button>
3. Button behavior and examples - Streamlit Docs, acessado em junho 11, 2025,  
<https://docs.streamlit.io/develop/concepts/design/buttons>
4. Building a dashboard in Python using Streamlit, acessado em junho 11, 2025,  
<https://blog.streamlit.io/crafting-a-dashboard-app-in-python-using-streamlit/>
5. Streamlit Part 5: Mastering Data Visualization and Chart Types - DEV Community, acessado em junho 11, 2025,  
<https://dev.to/jamesbmour/streamlit-part-6-mastering-data-visualization-and-chart-types-kip>
6. MCP and ACP: Decoding the language of models and agents - Outshift | Cisco, acessado em junho 11, 2025,  
<https://outshift.cisco.com/blog/mcp-acp-decoding-language-of-models-and-agents>
7. MCP, A2A, ACP: What does it all mean? - Akka, acessado em junho 11, 2025,  
<https://akka.io/blog/mcp-a2a-acp-what-does-it-all-mean>
8. FastMCP Tutorial: Building MCP Servers in Python From Scratch - Firecrawl, acessado em junho 11, 2025,  
<https://www.firecrawl.dev/blog/fastmcp-tutorial-building-mcp-servers-python>
9. A Beginner's Guide to Use FastMCP - Apidog, acessado em junho 11, 2025,  
<https://apidog.com/blog/fastmcp/>
10. Inside Google's Agent2Agent (A2A) Protocol: Teaching AI Agents to Talk to Each Other, acessado em junho 11, 2025,  
<https://towardsdatascience.com/inside-googles-agent2agent-a2a-protocol-teaching-ai-agents-to-talk-to-each-other/>
11. Understanding A2A — The Protocol for Agent Collaboration - Google Cloud Community, acessado em junho 11, 2025,  
<https://www.googlecloudcommunity.com/gc/Community-Blogs/Understanding-A2A-The-Protocol-for-Agent-Collaboration/ba-p/906323>