

Gemini Voice Assistant (Hybrid Pipeline):

Uma interface de voz conversacional robusta que implementa um pipeline híbrido de processamento de áudio. O projeto combina a privacidade e precisão do processamento local (Whisper) com a velocidade e inteligência da nuvem (Google Gemini 1.5 Flash/Pro), finalizando com síntese de voz nativa (gTTS).

Este projeto foi desenhado para ser resiliente a mudanças de API e compatível com ambientes de rede restritivos (proxies corporativos).

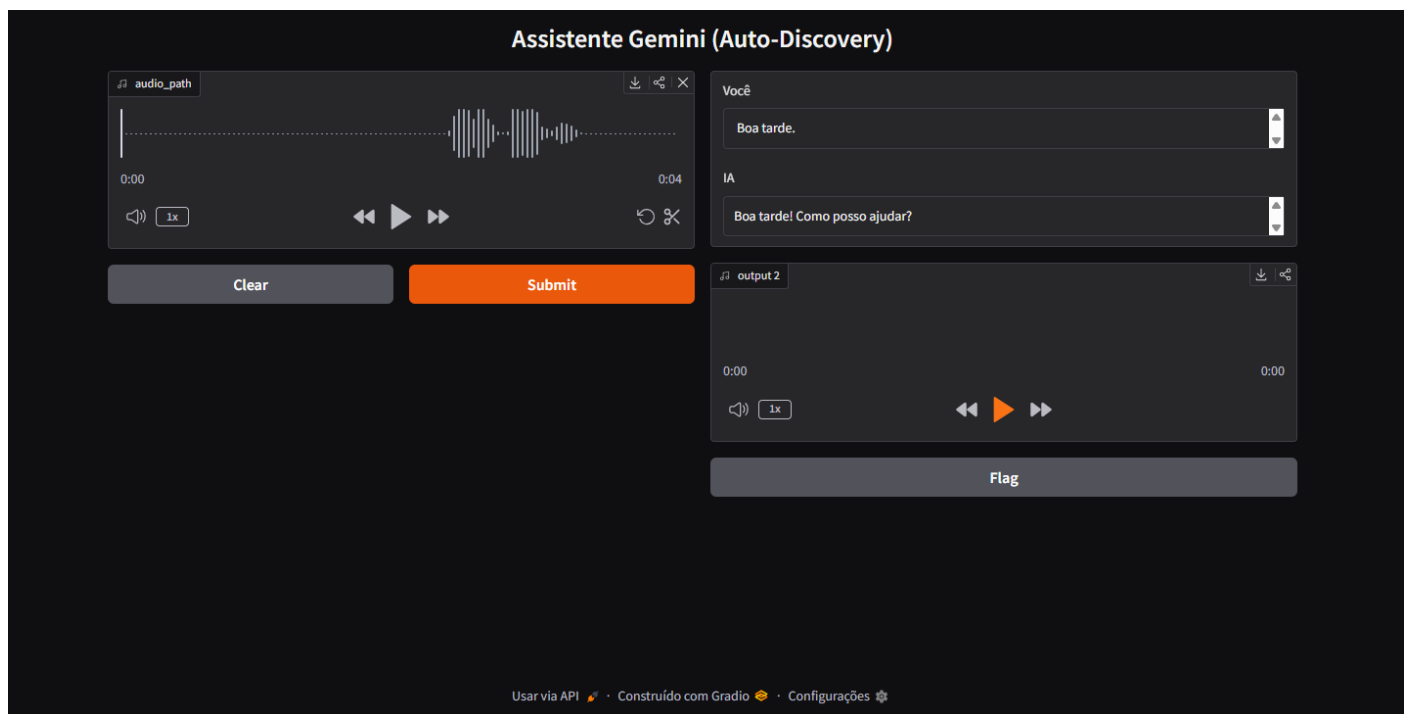
Arquitetura do Pipeline:

O sistema opera em três estágios síncronos:

Ingestão & Transcrição (Local): O áudio é capturado e processado localmente via gemin-whisper (modelo base), garantindo que a transcrição bruta não dependa de latência de rede.

Raciocínio (Cloud): O texto é enviado ao Google Gemini. O script implementa uma lógica de Auto-Discovery que negocia automaticamente o melhor modelo disponível na conta do usuário (priorizando Flash > Pro).

Síntese (Cloud): A resposta textual é limpa (remoção de artefatos Markdown) e convertida em áudio via gTTS.



Pré-requisitos:

Uma conta Google ativa.

<https://colab.research.google.com/notebooks/intro.ipynb>

Chaves de API:

<https://aistudio.google.com/app/api-keys>

Instalação e Configuração:

```
1  !# Instalação correta das dependências
2  !pip install -U google-generativeai gradio gTTS openai-whisper python-dotenv

1  import os
2  import whisper
3  import gradio as gr
4  import google.generativeai as genai
5  from google.api_core import exceptions as google_exceptions
6  from gtts import gTTS
7  import uuid
8  import logging
9  import getpass
10 from dotenv import load_dotenv
11
12 # Isso impede que o Python tente sair pelo proxy para falar com o localhost
13 os.environ['NO_PROXY'] = 'localhost,127.0.0.1,0.0.0.0'
14
```

Construção do script em Python:

```
# --- LÓGICA DE SELEÇÃO DINÂMICA DE MODELO ---
def obter_modelo_dinamico():
    """
    Lista os modelos disponíveis na sua conta e seleciona o melhor candidato
    automaticamente, evitando erros de 484 por nomes hardcoded.
    """
    print("🔍 Consultando API do Google para listar modelos disponíveis...")

    try:
        # Pega todos os modelos que suportam gerar texto ('generateContent')
        todos_modelos = [m for m in genai.list_models() if 'generateContent' in m.supported_generation_methods]

        # Ordena preferência: Flash (rápido) > Pro (inteligente) > Qualquer outro Gemini
        # Isso cria uma lista de prioridade baseada no nome
        def pontuacao_modelo(m):
            nome = m.name.lower()
            if '1.5-flash' in nome: return 3
            if '1.5-pro' in nome: return 2
            if 'gemini' in nome: return 1
            return 0

        # Ordena a lista pela pontuação (do maior para o menor)
        todos_modelos.sort(key=pontuacao_modelo, reverse=True)

        if not todos_modelos:
            raise ValueError("Nenhum modelo compatível encontrado na sua conta Google AI.")

        # Pega o campeão
        modelo_escolhido = todos_modelos[0]
        nome_real = modelo_escolhido.name # Ex: 'models/gemini-1.5-flash-001'

        print(f"✅ Modelo selecionado automaticamente: {nome_real}")
        print(f"    (Descrição: {modelo_escolhido.description})")

        return genai.GenerativeModel(nome_real)

    except Exception as e:
        logging.critical(f"Falha ao listar modelos. Erro: {e}")
        # Último recurso desesperado: tenta o legacy puro se a listagem falhar
        return genai.GenerativeModel('gemini-pro')
```

```
# --- Configuração Inicial ---
load_dotenv()
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Autenticação
api_key = os.getenv("GOOGLE_API_KEY")
if not api_key:
    # Fallback para input manual se não estiver no .env
    api_key = getpass.getpass("Digite sua Google API Key: ")

genai.configure(api_key=api_key)
```

```
# Inicializa
whisper_model = whisper.load_model("base")
gemini_model = obter_modelo_dinamico() # <--- Aqui a mágica acontece

# --- Processamento (O resto segue igual) ---
def processar_audio(audio_path):
    if audio_path is None:
        return "Nenhum áudio.", "Sem input.", None

    request_id = str(uuid.uuid4())[:8]

    # 1. Transcrição
    try:
        audio = whisper_model.transcribe(audio_path, fp16=False)
        texto = audio["text"]
    except Exception as e:
        return f"Erro Whisper: {e}", "", None

    # 2. LLM
    try:
        logging.info(f"[{request_id}] Gerando resposta...")
        response = gemini_model.generate_content(texto)
        resposta = response.text
    except Exception as e:
        logging.error(f"Erro Gemini: {e}")
        resposta = f"Erro na API: {str(e)}"

    # 3. TTS
    path_audio = None
    try:
        if resposta and not "Erro" in resposta:
            clean_text = resposta.replace("*", "")
            tts = gTTS(text=clean_text, lang='pt', slow=False)
            path_audio = f"resp_{request_id}.mp3"
            tts.save(path_audio)
    except Exception:
        pass

    return texto, resposta, path_audio
```

```
# Interface
iface = gr.Interface(
    fn=processar_audio,
    inputs=gr.Audio(sources=["microphone"], type="filepath"),
    outputs=[gr.Textbox(label="Você"), gr.Textbox(label="IA"), "audio"],
    title="Assistente Gemini (Auto-Discovery)"
)

if __name__ == "__main__":
    # server_name="127.0.0.1" força rodar localmente sem tentar abrir para rede
    iface.launch(debug=True, server_name="127.0.0.1", server_port=7860, share=False)
```