**Voice AI Agent: Technical Documentation**

**Table of Contents**

1. [Introduction](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#introduction)
2. [System Architecture](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#system-architecture)
3. [Components Overview](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#components-overview)
   * [Speech-to-Text Module](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#speech-to-text-module)
   * [Knowledge Base Module](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#knowledge-base-module)
   * [Text-to-Speech Module](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#text-to-speech-module)
   * [Integration Layer](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#integration-layer)
   * [LangGraph Integration](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#langgraph-integration)
4. [Installation and Setup](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#installation-and-setup)
   * [Prerequisites](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#prerequisites)
   * [Environment Setup](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#environment-setup)
   * [Installing Dependencies](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#installing-dependencies)
   * [Knowledge Base Initialization](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#knowledge-base-initialization)
5. [Configuration](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#configuration)
   * [Knowledge Base Configuration](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#knowledge-base-configuration)
   * [LLM Configuration](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#llm-configuration)
   * [Speech-to-Text Configuration](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#speech-to-text-configuration)
   * [Text-to-Speech Configuration](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#text-to-speech-configuration)
   * [LangGraph Configuration](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#langgraph-configuration)
6. [Usage Guide](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#usage-guide)
   * [Basic Text Query Processing](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#basic-text-query-processing)
   * [Audio Processing](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#audio-processing)
   * [End-to-End Pipeline](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#end-to-end-pipeline)
   * [Streaming Capabilities](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#streaming-capabilities)
   * [Using LangGraph Integration](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#using-langgraph-integration)
7. [Code Structure](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#code-structure)
8. [API Reference](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#api-reference)
9. [Testing](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#testing)
   * [Testing Standard Integration](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#testing-standard-integration)
   * [Testing LangGraph Integration](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#testing-langgraph-integration)
   * [Advanced Testing Scenarios](https://claude.ai/chat/975bd514-fe7e-4358-a73a-2b0ec09b2b8b#advanced-testing-scenarios)

**Introduction**

The Voice AI Agent is a comprehensive system that provides end-to-end voice interaction capabilities. It combines speech-to-text (STT), knowledge base retrieval, large language model (LLM) processing, and text-to-speech (TTS) into a unified pipeline. The system can understand spoken questions, search through documents for relevant information, generate contextual responses, and deliver them as natural-sounding speech.

**System Architecture**

The system follows a modular architecture with these key components:

1. **Speech-to-Text (STT)**: Converts audio input to text using Whisper models
2. **Knowledge Base (KB)**: Indexes documents and retrieves information using LlamaIndex
3. **LLM Processing**: Processes queries and generates responses using Ollama models
4. **Text-to-Speech (TTS)**: Converts text responses to speech using Deepgram TTS
5. **Integration Layer**: Connects all components for seamless operation
6. **LangGraph Framework**: Enables sophisticated conversation flows

The data flow generally follows this pattern:

* Audio input → STT → KB Query → LLM Response Generation → TTS → Audio output

Each component can be used individually or as part of the complete pipeline, with multiple integration options available.

**Components Overview**

**Speech-to-Text Module**

The STT module uses Whisper.cpp via the pywhispercpp library for real-time transcription.

**Key Features:**

* Real-time streaming transcription
* Voice activity detection
* Automatic chunking for long audio
* Support for multiple languages
* Configurable accuracy/speed trade-offs

**Main Files:**

* speech\_to\_text/streaming/whisper\_streaming.py: Contains StreamingWhisperASR class
* speech\_to\_text/streaming/chunker.py: Audio chunking utilities
* speech\_to\_text/utils/audio\_utils.py: Audio processing functions

**Knowledge Base Module**

The Knowledge Base module uses LlamaIndex for document indexing, retrieval, and LLM integration.

**Key Features:**

* Document indexing with vector embeddings
* Semantic search for relevant information
* Conversation context management
* Integration with various LLMs
* Support for multiple document formats

**Main Files:**

* knowledge\_base/llama\_index/document\_store.py: Document processing
* knowledge\_base/llama\_index/index\_manager.py: Vector index management
* knowledge\_base/llama\_index/query\_engine.py: Query processing
* knowledge\_base/conversation\_manager.py: Conversation state management

**Text-to-Speech Module**

The TTS module uses Deepgram's API for high-quality speech synthesis.

**Key Features:**

* Natural-sounding voice synthesis
* Streaming capabilities for real-time response
* Multiple voice options
* SSML support for advanced control
* Caching for improved performance

**Main Files:**

* text\_to\_speech/deepgram\_tts.py: Main TTS client
* text\_to\_speech/streaming.py: Streaming TTS functionality
* text\_to\_speech/audio\_utils.py: Audio processing utilities

**Integration Layer**

The integration layer provides a unified interface to all components.

**Key Features:**

* Component abstraction and standardization
* End-to-end pipeline orchestration
* Error handling and recovery
* Performance optimization
* Flexible deployment options

**Main Files:**

* integration/stt\_integration.py: STT integration
* integration/kb\_integration.py: Knowledge base integration
* integration/tts\_integration.py: TTS integration
* integration/pipeline.py: End-to-end pipeline

**LangGraph Integration**

The LangGraph integration provides a graph-based conversation flow.

**Key Features:**

* Node-based processing
* Advanced state management
* Conversation flow control
* Error recovery
* Extensibility

**Main Files:**

* langgraph\_integration/agent.py: Main LangGraph agent
* langgraph\_integration/nodes/: Node implementations
* langgraph\_integration/config.py: LangGraph configuration

**Installation and Setup**

**Prerequisites**

* Python 3.8 or higher
* Ollama for LLM access
* 16GB+ RAM recommended
* CUDA-compatible GPU (optional, for improved performance)

**Environment Setup**

1. **Clone the repository:**
2. git clone <repository-url>
3. cd voice-ai-agent
4. **Set up a virtual environment:**
5. python -m venv venv
6. # On Windows
7. venv\Scripts\activate
8. # On Linux/Mac
9. source venv/bin/activate
10. **Create a .env file for API keys:**
11. DEEPGRAM\_API\_KEY=your\_deepgram\_api\_key
12. OLLAMA\_BASE\_URL=http://localhost:11434

**Installing Dependencies**

1. **Install required packages:**
2. pip install -r requirements.txt
3. **Install Whisper model:**
4. python -c "import pywhispercpp; pywhispercpp.download\_model('base.en')"
5. **Set up Ollama:**
6. # Install Ollama from https://ollama.ai/
7. # Start Ollama server
8. ollama serve
9. # Pull required model
10. ollama pull mistral:7b-instruct-v0.2-q4\_0

**Knowledge Base Initialization**

1. **Create storage directories:**
2. mkdir -p storage
3. **Add documents to the knowledge base:**
4. python -m knowledge\_base.examples.llama\_index\_example --index --directory your\_documents\_folder

Or use the sample documents:

python -m knowledge\_base.examples.llama\_index\_example --index --directory knowledge\_base/knowledge\_docs

**Configuration**

**Knowledge Base Configuration**

Key configuration file: knowledge\_base/config.py

Important settings:

# Vector database settings

VECTOR\_DB\_HOST = "localhost"

VECTOR\_DB\_PORT = 6333

VECTOR\_DB\_COLLECTION = "company\_knowledge"

# Embedding model settings

EMBEDDING\_MODEL = "sentence-transformers/paraphrase-MiniLM-L3-v2"

EMBEDDING\_DEVICE = "cpu" # Set to "cuda" for GPU acceleration

# Document processing settings

CHUNK\_SIZE = 512

CHUNK\_OVERLAP = 50

**LLM Configuration**

Key configuration file: knowledge\_base/llama\_index/llm\_setup.py

Important settings:

# Default model configuration

DEFAULT\_MODEL = "mistral:7b-instruct-v0.2-q4\_0"

OLLAMA\_BASE\_URL = "http://localhost:11434"

# LLM parameters

DEFAULT\_TEMPERATURE = 0.7

DEFAULT\_MAX\_TOKENS = 1024

**Speech-to-Text Configuration**

Configuration is handled in the StreamingWhisperASR constructor:

speech\_recognizer = StreamingWhisperASR(

model\_path="base.en", # Model size: tiny.en, base.en, small.en, medium.en, large-v2

language="en",

n\_threads=4,

chunk\_size\_ms=2000,

vad\_enabled=True,

single\_segment=True,

temperature=0.0

)

**Text-to-Speech Configuration**

Key configuration file: text\_to\_speech/config.py

Important settings:

# Deepgram API settings

deepgram\_api\_key = "your\_api\_key"

# TTS settings

model = "aura-asteria-en"

voice = "nova" # Or other available voices

sample\_rate = 24000

container\_format = "mp3"

# Streaming settings

chunk\_size = 1024

max\_text\_chunk\_size = 100

**LangGraph Configuration**

Key configuration file: langgraph\_integration/config.py

Important settings:

@dataclass

class LangGraphConfig:

# STT configuration

stt\_model: str = "base.en"

stt\_language: str = "en"

# KB configuration

kb\_temperature: float = 0.7

kb\_max\_tokens: int = 1024

kb\_include\_sources: bool = True

# TTS configuration

tts\_voice: Optional[str] = None

# Other settings

debug\_mode: bool = False

save\_state\_history: bool = False

**Usage Guide**

**Basic Text Query Processing**

Using the standard VoiceAIAgent:

import asyncio

from voice\_ai\_agent import VoiceAIAgent

async def process\_text\_query():

# Initialize the agent

agent = VoiceAIAgent(

storage\_dir="./storage",

model\_name="mistral:7b-instruct-v0.2-q4\_0"

)

# Initialize components

await agent.init()

# Process a text query

response = await agent.process\_text("What are your pricing plans?")

print(f"Response: {response['response']}")

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(process\_text\_query())

**Audio Processing**

Processing an audio file:

import asyncio

from voice\_ai\_agent import VoiceAIAgent

async def process\_audio\_file():

# Initialize the agent

agent = VoiceAIAgent(

storage\_dir="./storage",

model\_name="mistral:7b-instruct-v0.2-q4\_0",

whisper\_model\_path="base.en"

)

# Initialize components

await agent.init()

# Process an audio file

result = await agent.process\_full\_audio\_file("path/to/audio.wav")

print(f"Transcription: {result['transcription']}")

print(f"Response: {result['response']}")

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(process\_audio\_file())

**End-to-End Pipeline**

Using the enhanced agent with TTS:

import asyncio

from voice\_ai\_agent\_with\_tts import VoiceAIAgentWithTTS

async def end\_to\_end\_pipeline():

# Initialize the agent with TTS

agent = VoiceAIAgentWithTTS(

storage\_dir="./storage",

model\_name="mistral:7b-instruct-v0.2-q4\_0",

whisper\_model\_path="base.en",

tts\_voice="nova" # Deepgram voice

)

# Initialize components

await agent.init()

# Process audio file with TTS output

result = await agent.end\_to\_end\_pipeline(

audio\_file\_path="path/to/audio.wav",

output\_speech\_file="response.mp3"

)

print(f"Transcription: {result['transcription']}")

print(f"Response text: {result['response']}")

print(f"Response audio saved to: response.mp3")

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(end\_to\_end\_pipeline())

**Streaming Capabilities**

Processing with streaming response:

import asyncio

from voice\_ai\_agent import VoiceAIAgent

async def streaming\_response():

# Initialize the agent

agent = VoiceAIAgent(

storage\_dir="./storage",

model\_name="mistral:7b-instruct-v0.2-q4\_0"

)

# Initialize components

await agent.init()

# Process a text query with streaming response

print("Response: ", end="", flush=True)

async for chunk in agent.process\_text\_streaming("Tell me about your product features"):

if "chunk" in chunk and chunk["chunk"]:

print(chunk["chunk"], end="", flush=True)

print() # New line after response

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(streaming\_response())

**Using LangGraph Integration**

Using the LangGraph-based agent:

import asyncio

from langgraph\_integration import VoiceAILangGraph

from langgraph\_integration.config import LangGraphConfig

from voice\_ai\_agent import VoiceAIAgent

async def use\_langgraph():

# Initialize the base agent for component reuse

base\_agent = VoiceAIAgent(

storage\_dir="./storage",

model\_name="mistral:7b-instruct-v0.2-q4\_0",

whisper\_model\_path="base.en"

)

# Initialize components

await base\_agent.init()

# Create LangGraph config

config = LangGraphConfig(

stt\_model="base.en",

stt\_language="en",

kb\_temperature=0.7,

tts\_voice="nova",

debug\_mode=True

)

# Create LangGraph agent

langgraph\_agent = VoiceAILangGraph(

voice\_ai\_agent=base\_agent,

config=config

)

# Initialize the LangGraph agent

await langgraph\_agent.init()

# Process audio file

result = await langgraph\_agent.process\_audio\_file(

audio\_file\_path="path/to/audio.wav",

speech\_output\_path="response.mp3"

)

print(f"Transcription: {result['transcription']}")

print(f"Response: {result['response']}")

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(use\_langgraph())

**Code Structure**

The project is organized into several main directories:

voice-ai-agent/

├── speech\_to\_text/ # Speech recognition module

│ ├── streaming/ # Real-time transcription components

│ ├── utils/ # Audio utilities

│ └── tests/ # STT tests

├── knowledge\_base/ # Knowledge base and LLM components

│ ├── llama\_index/ # LlamaIndex integration

│ ├── knowledge\_docs/ # Sample documents

│ └── examples/ # Usage examples

├── text\_to\_speech/ # Speech synthesis module

│ ├── deepgram\_tts.py # Deepgram TTS client

│ └── streaming.py # Streaming TTS capabilities

├── integration/ # Integration components

│ ├── stt\_integration.py # STT integration

│ ├── kb\_integration.py # KB integration

│ ├── tts\_integration.py # TTS integration

│ └── pipeline.py # End-to-end pipeline

├── langgraph\_integration/ # LangGraph components

│ ├── agent.py # Main LangGraph agent

│ ├── nodes/ # Node implementations

│ └── config.py # Configuration

├── tests/ # Test scripts

│ ├── test\_integration.py # Integration tests

│ └── test\_langgraph.py # LangGraph tests

├── voice\_ai\_agent.py # Main agent class

└── voice\_ai\_agent\_with\_tts.py # Enhanced agent with TTS

**API Reference**

**VoiceAIAgent**

Main class for voice AI functionality.

**Key Methods:**

* \_\_init\_\_(storage\_dir, model\_name, whisper\_model\_path, language, llm\_temperature, llm\_max\_tokens, use\_langgraph): Initialize the agent
* init(): Initialize all components
* process\_text(text): Process text input
* process\_text\_streaming(text): Process text with streaming response
* process\_audio(audio\_chunk, callback, is\_short\_audio): Process audio input
* process\_full\_audio\_file(audio\_file\_path, chunk\_size\_ms, simulate\_realtime): Process an audio file
* query\_knowledge\_base(query, stream, top\_k): Query the knowledge base directly
* get\_stats(): Get agent statistics

**VoiceAIAgentWithTTS**

Enhanced agent class with TTS capabilities.

**Additional Methods:**

* text\_to\_speech(text): Convert text to speech
* text\_to\_speech\_streaming(text\_generator): Stream text to speech conversion
* end\_to\_end\_pipeline(audio\_file\_path, output\_speech\_file): Run the complete STT → KB → TTS pipeline
* process\_audio\_to\_speech\_stream(audio\_file\_path, tts\_callback, chunk\_size\_ms): Process audio with streaming response

**VoiceAILangGraph**

LangGraph-based agent implementation.

**Key Methods:**

* \_\_init\_\_(voice\_ai\_agent, stt\_integration, kb\_integration, tts\_integration, config): Initialize the LangGraph agent
* init(): Initialize all components
* process\_audio\_file(audio\_file\_path, speech\_output\_path, metadata): Process an audio file
* process\_audio\_data(audio\_data, speech\_output\_path, metadata): Process audio data
* process\_text(text, speech\_output\_path, metadata): Process text input
* cleanup(): Clean up resources

**Testing**

This section covers how to test both approaches of the Voice AI Agent system: the standard integration and the LangGraph integration.

**Testing Standard Integration**

The standard integration uses a straightforward pipeline approach, connecting the STT, Knowledge Base, and TTS components sequentially.

**Step 1: Set Up Test Environment**

First, ensure all dependencies are installed and the knowledge base is properly initialized:

# Install requirements

pip install -r requirements.txt

# Initialize knowledge base with sample documents

python -m knowledge\_base.examples.llama\_index\_example --index --directory knowledge\_base/knowledge\_docs

Make sure Ollama is running for LLM access:

ollama serve

**Step 2: Basic Text Processing Test**

The simplest test is to check text processing functionality using test\_voice\_ai\_agent.py:

python test\_voice\_ai\_agent.py

This will:

* Initialize the VoiceAIAgent
* Test several text queries against your knowledge base
* Display the responses on the console

You should see responses to queries like "What are your pricing plans?" and "Tell me about your product features".

**Step 3: Speech-to-Text Processing Test**

To test the speech recognition component with an audio file:

python test\_stt\_integration.py --audio path/to/your/audio.wav

This script will:

* Initialize the speech recognition component
* Process the audio file
* Transcribe the speech to text
* Display the transcription and any generated response

**Step 4: End-to-End Pipeline Test**

For the complete pipeline test (STT → KB → TTS), use the integration test script:

python tests/test\_integration.py --input-file path/to/your/audio.wav --output-dir ./test\_output

This comprehensive test will:

* Take an audio file with a spoken question
* Transcribe the audio to text
* Query the knowledge base
* Generate a text response
* Convert the response to speech
* Save the speech output to a file

Additional parameters:

* --model-name: Specify a different LLM model (default: "mistral:7b-instruct-v0.2-q4\_0")
* --whisper-model: Specify a different Whisper model (default: "tiny.en")
* --tts-voice: Specify a TTS voice option
* --streaming: Enable streaming mode for more real-time processing

You can examine the console output to see detailed logs of each processing stage, including:

* Transcription results
* Retrieved knowledge base documents
* Generated response
* Timing information for each stage

The generated speech file will be saved in your specified output directory as "response.mp3".

**Testing LangGraph Integration**

The LangGraph integration provides a more sophisticated approach with better state management and a node-based processing flow.

**Step 1: Set Up for LangGraph Testing**

The setup is the same as for standard integration, ensuring all dependencies are installed and the knowledge base is initialized.

**Step 2: Run LangGraph Integration Test**

Use the dedicated LangGraph test script:

python tests/test\_langgraph.py --input-file path/to/your/audio.wav --output-dir ./langgraph\_output

Alternatively, you can test with direct text input:

python tests/test\_langgraph.py --text-input "What are your pricing plans?" --output-dir ./langgraph\_output

This test will:

* Initialize the LangGraph-based Voice AI Agent
* Create a graph with nodes for STT, KB, and TTS processing
* Process the input through the graph
* Save state information and output speech

**Step 3: Examining LangGraph State and Results**

The LangGraph test provides more detailed state information:

* State history is saved to a JSON file in the output directory
* The console output includes node transition information
* You can see how the input flows through different processing stages

The output directory will contain:

* langgraph\_response.mp3: The generated speech response
* state\_history.json: Detailed record of state transitions
* callback\_audio.mp3: Optional callback audio for verification

**Understanding LangGraph vs. Standard Integration**

Key differences in test results:

1. **State Management**: LangGraph tests show explicit state transitions
2. **Error Handling**: LangGraph has more sophisticated error recovery
3. **Performance**: You might notice different timing patterns between the approaches
4. **Extensibility**: LangGraph makes it easier to add new processing nodes

**Step 4: Comparing Transcription and Response Quality**

To compare the quality of both approaches:

1. Run both test scripts with the same input file
2. Compare the transcription accuracy
3. Compare the responses generated
4. Listen to both audio outputs for speech quality

You can run sequential tests with:

python tests/test\_integration.py --input-file sample.wav --output-dir ./standard\_output

python tests/test\_langgraph.py --input-file sample.wav --output-dir ./langgraph\_output

**Advanced Testing Scenarios**

**Testing with Real-Time Audio**

While not directly included in the test scripts, you can modify the code to test with real-time audio input:

1. Capture microphone input using PyAudio or a similar library
2. Stream the audio chunks to the process\_realtime\_audio\_stream method
3. Stream the responses to an audio output device

**Testing Streaming Response Generation**

The system supports streaming response generation, which you can test by modifying the test scripts to:

1. Process the input normally
2. Use the streaming response methods
3. Process each response chunk as it arrives

**Testing With Different Knowledge Bases**

You can test with different knowledge bases by:

1. Creating a new directory with different documents
2. Running the indexing process on that directory
3. Comparing responses with different knowledge sources

# Index a new knowledge base

python -m knowledge\_base.examples.llama\_index\_example --index --directory ./my\_custom\_docs --storage ./custom\_storage

# Test with the new knowledge base

python tests/test\_integration.py --input-file sample.wav --output-dir ./custom\_output --storage-dir ./custom\_storage

**Performance Optimization**

To optimize performance:

1. **GPU Acceleration**:
   * Set EMBEDDING\_DEVICE = "cuda" in knowledge\_base/config.py
   * Configure Ollama for GPU use
2. **Model Size Selection**:
   * Use smaller Whisper models for faster transcription
   * Balance LLM size with response quality needs
3. **Batch Processing**:
   * Adjust EMBEDDING\_BATCH\_SIZE for optimal throughput
   * Configure chunk sizes for document processing
4. **Caching**:
   * Enable TTS caching for repeated phrases
   * Use LlamaIndex caching mechanisms
5. **Parallel Processing**:
   * Increase n\_threads for Whisper processing
   * Use appropriate concurrency settings