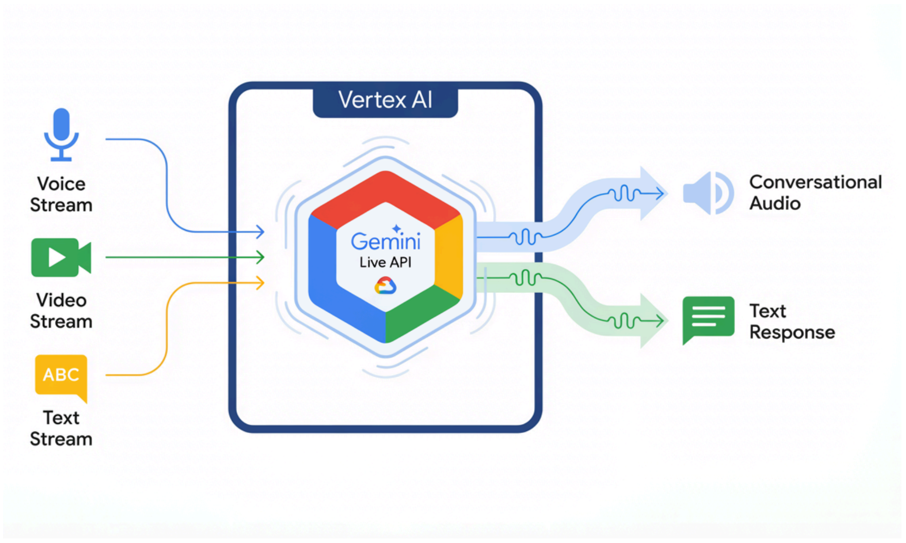
Public

Gemini Live API Cookbook

Dec 12, 2025

Overview

This cookbook outlines the technical specifications and implementation details for the Gemini Live API in Vertex AI. The Live API is designed to enable low-latency, real-time voice and video interactions, processing continuous streams of input to deliver immediate, human-like spoken responses.



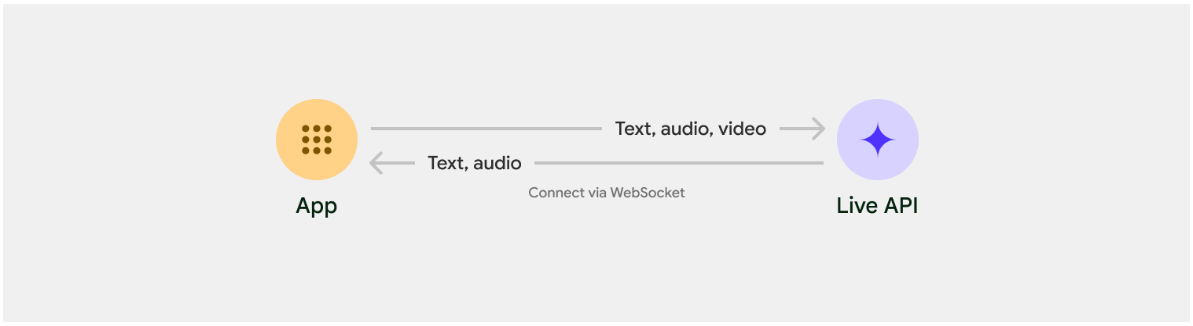
The Live API offers a comprehensive set of features designed for building robust voice agents:

- Native Audio: Provides natural, realistic-sounding speech and improved multilingual performance.
- Multilingual Support: Converse in 24 supported languages.
- Voice Activity Detection (VAD): Automatically handles interruptions and turn-taking naturalness.
- Affective Dialog: Adapts response style and tone to match the user's input expression.
- Proactive Audio: Allows you control when it responds and in what contexts with fewer interruptions.
- Thinking: Uses hidden reasoning tokens to "think" before speaking for complex queries.
- Tool Use: Seamlessly integrates tools like function calling and Google Search for more practical and dynamic interactions.
- Audio Transcriptions: Provides text transcripts of both user input and model output.
- Speech-to-Speech Translation: Optimized for low-latency translation between languages.

Getting Started

This cookbook demonstrates how to get started with the Live API in Vertex AI using WebSocket, a low-level approach to establish a standard WebSocket session and manage raw JSON payloads.

Public



Install Websockets

```
Shell
pip install websockets
```

Set Google Cloud project information

To get started using Vertex AI, you must have an existing Google Cloud project and enable the Vertex AI API. You must authenticate to Vertex AI using an API key or application default credentials (ADC). See authentication methods for more information.

Replace the `[your-project_id]` and `us-central1` with your Google Cloud Project ID and location.

```python
PROJECT_ID = "[your-project-id]"
LOCATION = "us-central1"
```

Choose a Gemini model

Select an appropriate model based on your interaction requirements. See Live API Supported Models.

```python
MODEL_ID = "gemini-live-2.5-flash-native-audio"
model =
f"projects/{PROJECT_ID}/locations/{LOCATION}/publishers/google/models/{MODEL_ID}"
```

Create a Session

Implementation of the Live API requires strict adherence to its WebSocket sub-protocol. The interaction is defined by a sequence of message exchanges: `Handshake`, `Setup`, `Session Loop`, and `Termination`.

Public

Step 1: Handshake

The connection is established via a standard WebSocket handshake. The service URL uses a regional endpoint and OAuth 2.0 bearer tokens for authentication.

```python
api_host = "aiplatform.googleapis.com"
if LOCATION != "global":
    api_host = f"{LOCATION}-aiplatform.googleapis.com"

service_url =
f"wss://{api_host}/ws/google.cloud.aiplatform.v1.LlmBidiService/BidiGenerateContent"
```

You can use `gcloud` command to generate an access token for the current Application Default Credential. The access token is passed in the WebSocket headers (e.g., Authorization: Bearer `<TOKEN>`). Note that the default access token lifetime is `3600` seconds.

```python
token_list = !gcloud auth application-default print-access-token

headers = {"Authorization": f"Bearer {token_list[0]}"}
```

Step 2: Setup

Once the WebSocket connection is established, the client must send a configuration message `BidiGenerateContentSetup` immediately to initialize the session. The setup payload is a JSON object containing the model, `generation_config`, `system_instruction`, and `tools` definitions.

```python
system_instruction = {
    "parts": [{"text": "You are a helpful assistant and answer in a friendly tone."}]
}

config = {
    "response_modalities": ["audio"],
    "speech_config": {
        "language_code": "es-US",
    },
}

setup = {
    "setup": {
        "model": model,
        "system_instruction": system_instruction,
```

Public

```
        "generation_config": config,
    }
}
```

Step 3: Session Loop

After the setup phase, the session enters a bidirectional loop. This is one **example design pattern** for the implementation:

```python
async def main() -> None:
    # Connect to the server
    async with websockets.connect(service_url, additional_headers=headers) as ws:
        print("Connected")

        # 1. Perform Setup (Handshake)
        # This must be a single, awaited call before streaming starts.
        await ws.send(json.dumps(setup))
        await ws.recv()  # Wait for setup completion/response

        # 2. Define the Send Loop
        async def send_loop():
            try:
                while True:
                    # Logic to read microphone/video and send to WS
                    # await ws.send(audio_chunk)
                    await asyncio.sleep(0.02)  # Simulate 20ms audio chunks
            except asyncio.CancelledError:
                pass  # Handle clean exit

        # 3. Define the Receive Loop
        async def receive_loop():
            try:
                async for message in ws:
                    # Logic to play audio or handle "interrupted"
                    print("Received message")
                    # If message.interrupted: stop_playback()
            except websockets.exceptions.ConnectionClosed:
                print("Connection closed")

        # 4. Run both concurrently
        # This allows sending and receiving to happen at the exact same time.
        await asyncio.gather(send_loop(), receive_loop())
```

Public

Send Multimodal Streaming

The API can process text, audio, and video input, and provide text and audio output. The client sends `client_message` payloads, and the server responds with `server_message` payloads.

Client Messages

- `realtime_input`: Used for high-frequency streaming of audio and video chunks. This message type is designed for efficiency and low overhead. It contains media_chunks with Base64-encoded data.
- `client_content`: Used for discrete "turns" or text input. This allows the client to inject text into the conversation (e.g., "The user clicked a button"). It is also used to provide context or conversation history. Sending client_content with "turn_complete": true signals the model to generate a response immediately.
- `tool_response`: Sent by the client after executing a function call requested by the model. It contains the output of the function (e.g., the result of a database query or API call).

Server Messages

- `server_content`: The primary vehicle for the model's output. It contains model_turn data, which includes text parts and inline_data (the audio PCM bytes).4
- `tool_call`: Sent when the model decides to invoke a tool. It contains the function name and arguments.
- `turn_complete`: A boolean flag indicating that the model has finished its current generation turn. This is a signal to the client that the model is now waiting for input.
- `interrupted`: A critical signal indicating that the server has detected user speech (Barge-in) and has ceased generation. This requires immediate handling by the client to stop playback.

Send Text

This is a single turn text to audio conversation session example - send a text message, receive audio output and play the audio. For demonstration purposes, it exits the session loop after a turn completes.

```python
async def send_text(ws, text_input: str):
    """Sends a single text turn."""
    print(f"Input: {text_input}")
    try:
        msg = {
            "client_content": {
                "turns": [{"role": "user", "parts": [{"text": text_input}]}],
                "turn_complete": True,
            }
        }
        await ws.send(json.dumps(msg))
    except Exception as e:
        print(f"Error sending text: {e}")

async def main() -> None:
    async with websockets.connect(service_url, additional_headers=headers) as ws:
        print("Connected")
```

Public

```
        await ws.send(json.dumps(setup))
        await ws.recv()
        async def send_loop():
            await send_text(ws, "Hello? Gemini are you there?")

        audio_data = []
        async def receive_loop():
            async for message in ws:
                response = json.loads(message.decode())
                try:
                    parts = response["serverContent"]["modelTurn"]["parts"]
                    for part in parts:
                        if "inlineData" in part:
                            pcm_data = base64.b64decode(part["inlineData"]["data"])
                            audio_data.append(np.frombuffer(pcm_data,
dtype=np.int16))
                except KeyError:
                    pass
                if response.get("serverContent", {}).get("turnComplete"):
                    print("Turn complete.")
                    display(
                        Audio(np.concatenate(audio_data), rate=24000, autoplay=True)
                    )
                    break  # Exit the loop

        await asyncio.gather(send_loop(), receive_loop())

await main()
```

Send Audio

Implementing real-time audio requires strict adherence to sample rate specifications and careful buffer management to ensure low latency and natural interruptibility.
The Live API supports the following audio formats:

- **Input audio**: Raw 16-bit PCM audio at 16kHz, little-endian
- **Output audio**: Raw 16-bit PCM audio at 24kHz, little-endian

The following is a single turn audio to audio session example - send an audio file as input, receive audio output and play the audio. For demonstration purposes, it exits the session loop after a turn completes.

Public

Python

```python
# Download a sample audio input file
audio_file = "input.wav"
audio_file_url =
"https://storage.googleapis.com/cloud-samples-data/generative-ai/audio/tell-a-story.w
av"

!wget -q audio_file

with wave.open(audio_file, "rb") as wf:
    frames = wf.readframes(wf.getnframes())
    print(f"Read audio: {len(frames)} bytes")
    print(f"Channels: {wf.getnchannels()}")
    print(f"Rate: {wf.getframerate()}Hz")
    print(f"Width: {wf.getsampwidth()} bytes")

display(Audio(filename=audio_file, autoplay=True))

MEDIA_CHUNK_SIZE = 4096  # Chunk size for streaming audio

async def send_audio(ws, audio_file_path: str):
    """Streams an audio file in chunks."""
    print(f"Input Audio File: {audio_file_path}")

    try:
        # Send Input (Simulated from file)
        # In production, this would be a microphone stream
        with open(audio_file_path, "rb") as f:
            while chunk := f.read(MEDIA_CHUNK_SIZE):
                msg = {
                    "realtime_input": {
                        "media_chunks": [
                            {
                                "mime_type": "audio/pcm;rate=16000",
                                "data": base64.b64encode(chunk).decode("utf-8"),
                            }
                        ]
                    }
                }
                await ws.send(json.dumps(msg))
        print("Finished streaming audio.")
    except Exception as e:
        print(f"Error streaming audio: {e}")

async def main() -> None:
    async with websockets.connect(service_url, additional_headers=headers) as ws:
        print("Connected")
```

Public

```
        await ws.send(json.dumps(setup))
        await ws.recv()
        async def send_loop():
            await send_audio(ws, audio_file)

        audio_data = []
        async def receive_loop():
            async for message in ws:
                response = json.loads(message.decode())
                try:
                    parts = response["serverContent"]["modelTurn"]["parts"]
                    for part in parts:
                        if "inlineData" in part:
                            pcm_data = base64.b64decode(part["inlineData"]["data"])
                            audio_data.append(np.frombuffer(pcm_data,
dtype=np.int16))
                except KeyError:
                    pass
                if response.get("serverContent", {}).get("turnComplete"):
                    print("Turn complete.")
                    display(
                        Audio(np.concatenate(audio_data), rate=24000, autoplay=True)
                    )
                    break  # Exit the loop
        await asyncio.gather(send_loop(), receive_loop())

await main()
```

Send Video

Video streaming provides visual context (e.g., "What is this holding?"). Unlike a continuous video file (like.mp4), the Live API expects a sequence of discrete image frames. The Live API supports video frames input at 1FPS. For best results, use native 768x768 resolution at 1FPS.

The following is a single turn video to audio session example - send a video file as input, receive audio output and play the audio. For demonstration purposes, it exits the session loop after a turn completes. The client implementation should capture a frame from the video feed, encode it as a JPEG blob, Base64 encode the blob, and transmit it using the same `realtime_input` message structure as audio.

```python
DEFAULT_IMAGE_ENCODE_OPTIONS = [cv2.IMWRITE_JPEG_QUALITY, 90]

def encode_image(image_data: np.ndarray, encode_options: list[int]) -> bytes:
    """Encodes a numpy array (image) into JPEG bytes."""
```

Public

```python
    success, encoded_image = cv2.imencode(".jpg", image_data, encode_options)
    if not success:
        raise ValueError("Could not encode image to JPEG")
    return encoded_image.tobytes()

async def send_video(ws, video_file_path: str):
    """Streams a video file frame by frame."""
    print(f"Input Video File: {video_file_path}")
    cap = None
    try:
        cap = cv2.VideoCapture(video_file_path)
        if not cap.isOpened():
            raise OSError(f"Cannot open video file: {video_file_path}")

        fps = cap.get(cv2.CAP_PROP_FPS)
        frame_delay = 1 / fps
        print(f"Streaming video with {fps:.2f} FPS (delay: {frame_delay:.4f}s)")

        while cap.isOpened():
            ret, video_data = cap.read()
            if not ret:
                print("End of video stream.")
                break
            processed_jpeg = encode_image(video_data, DEFAULT_IMAGE_ENCODE_OPTIONS)
            b64_data = base64.b64encode(processed_jpeg).decode("utf-8")
            msg = {
                "realtime_input": {
                    "video": {"mime_type": "image/jpeg", "data": b64_data}
                }
            }
            await ws.send(json.dumps(msg))
            await asyncio.sleep(frame_delay)
        print("Signaling end of turn.")
        await ws.send(json.dumps({"realtime_input": {"text": ""}}))
    except Exception as e:
        print(f"Error processing video: {e}")
    finally:
        if cap and cap.isOpened():
            cap.release()

system_instruction = {"parts":[{"text": "You are a helpful assistant watching a 
video. Describe any animals you see."}]}
config = {"response_modalities": ["audio"],}
setup = {
    "setup": {
        "model": model,
```

Public

```
        "system_instruction": system_instruction,
        "generation_config": config,
    }
}

async def main() -> None:
    async with websockets.connect(service_url, additional_headers=headers) as ws:
        print("Connected")
        await ws.send(json.dumps(setup))
        await ws.recv()
        async def send_loop():
            await send_video(ws, video_file)

        audio_data = []
        async def receive_loop():
            async for message in ws:
                response = json.loads(message.decode())
                try:
                    parts = response["serverContent"]["modelTurn"]["parts"]
                    for part in parts:
                        if "inlineData" in part:
                            pcm_data = base64.b64decode(part["inlineData"]["data"])
                            audio_data.append(np.frombuffer(pcm_data,
dtype=np.int16))
                except KeyError:
                    pass
                if response.get("serverContent", {}).get("turnComplete"):
                    print("Turn complete.")
                    display(
                        Audio(np.concatenate(audio_data), rate=24000, autoplay=True)
                    )
                    break  # Exit the loop
        await asyncio.gather(send_loop(), receive_loop())

await main()
```

Tool Use

The Live API seamlessly integrates tools like function calling and Google Search for more practical and dynamic interactions.

Function Calling

Public

You can use function calling to create a description of a function, then pass that description to the model in a request. The response from the model includes the name of a function that matches the description and the arguments to call it with.

Notes: All functions must be declared at the start of the session by sending tool definitions as part of the `setup` message.

Python
```python
# Define function declaration
get_temperature_declaration = {
    "name": "get_temperature",
    "description": "Gets the current temperature for a given location.",
    "parameters": {
        "type": "object",
        "properties": {"location": {"type": "string"}},
        "required": ["location"],
    },
}

# Set tools
tools = {"function_declarations": [get_temperature_declaration]}
setup = {"setup": {"model": model, "generation_config": config, "tools": tools}}

async def main() -> None:
    async with websockets.connect(service_url, additional_headers=headers) as ws:
        print("Connected")
        await ws.send(json.dumps(setup))
        await ws.recv()
        async def send_loop():
            await send_text(ws, "Get the current temperature in New York.")

        responses = []
        async def receive_loop():
            async for message in ws:
                response = json.loads(message.decode())
                if (tool_call := response.get("toolCall")) is not None:
                    for function_call in tool_call["functionCalls"]:
                        responses.append(f"FunctionCall: {function_call!s}\n")
                if (server_content := response.get("serverContent")) is not None:
                    if server_content.get("turnComplete"):
                        print("Turn complete.")
                        print("Response:\n{}".format("\n".join(responses)))
                        break
        await asyncio.gather(send_loop(), receive_loop())

await main()
```

Public

Google Search

The `google_search` tool lets the model conduct Google searches. For example, try asking it about events that are too recent to be in the training data.

Python
```python
# Define google search tool
tools = {"google_search": {}}
setup = {"setup": {"model": model, "generation_config": config, "tools": tools}}

async def main() -> None:
    async with websockets.connect(service_url, additional_headers=headers) as ws:
        print("Connected")
        await ws.send(json.dumps(setup))
        await ws.recv()
        async def send_loop():
            await send_text(ws, "What is the current weather in Toronto, Canada?")

        audio_data = []
        async def receive_loop():
            async for message in ws:
                response = json.loads(message.decode())
                try:
                    parts = response["serverContent"]["modelTurn"]["parts"]
                    for part in parts:
                        if "inlineData" in part:
                            pcm_data = base64.b64decode(part["inlineData"]["data"])
                            audio_data.append(np.frombuffer(pcm_data,
dtype=np.int16))
                except KeyError:
                    pass
                if response.get("serverContent", {}).get("turnComplete"):
                    print("Turn complete.")
                    display(
                        Audio(np.concatenate(audio_data), rate=24000, autoplay=True)
                    )
                    break  # Exit the loop
        await asyncio.gather(send_loop(), receive_loop())

await main()
```

Capabilities

This covers the key capabilities and configurations available with the Live API.

Public

Audio Transcription

In addition to the model response, you can also receive transcriptions of both the audio input and the audio output.

To receive transcriptions, you must update your session configuration with the `input_audio_transcription` and `output_audio_transcription` parameters added.

Python
```
config = {
    "generation_config": {"response_modalities": ["audio"]},
    "input_audio_transcription": {},
    "output_audio_transcription": {},
}
```

Voice Activity Detection (VAD)

Voice Activity Detection (VAD) allows the model to recognize when a person is speaking. This is essential for creating natural conversations, as it allows a user to interrupt the model at any time.

- By default, the model automatically performs voice activity detection on a continuous audio input stream. Voice activity detection can be configured with the `realtime_input_config.automatic_activity_detection` field of the setup message.
- When voice activity detection detects an interruption, the ongoing generation is canceled and discarded. Only the information already sent to the client is retained in the session history. The server then sends a message to report the interruption.
- When the audio stream is paused for more than a second (for example, because the user switched off the microphone), an `audioStreamEnd` event should be sent to flush any cached audio. The client can resume sending audio data at any time.

Python
```
config = {
    "generation_config": {"response_modalities": ["audio"]},
    "realtime_input_config": {
        "automatic_activity_detection": {
            "disabled": False,  # default
            "start_of_speech_sensitivity": "START_SENSITIVITY_HIGH",
            "end_of_speech_sensitivity": "END_SENSITIVITY_HIGH",
            "prefix_padding_ms": 20,
            "silence_duration_ms": 100,
        },
    },
}
```

Public

Native Audio

Native audio provides natural, realistic-sounding speech and improved multilingual performance. It also enables advanced features like, Affective Dialogue and Proactive Audio.

Proactive Audio

When proactive audio is enabled, the model only responds when it's relevant. The model generates text transcripts and audio responses proactively only for queries directed to the device, and does not respond to non-device directed queries.

This example uses a System Instruction and Proactive Audio to test the model's ability to remain silent when the topic is off-subject (French cuisine) and chime in only when the conversation shifts to the instructed topic (Italian cooking).

Python
```python
config = {
    "system_instruction": {
        "parts": [
            {
                "text": "You are an AI assistant in Italian cooking, chime in only when the topic is about Italian cooking."
            }
        ]
    },
    "proactivity": {
        "proactive_audio": True,  # Enable proactive audio
    },
    "generation_config": {"response_modalities": ["audio"]},
}

conversation_turns = [
    # Speaker A speaks, general topic, the model should be silent.
    "Hey, I was just thinking about my dinner plans. I really love cooking.",
    # Speaker B speaks, off-topic (French cuisine). The model should be silent.
    "Oh yes, me too. I love French cuisine, especially making a good coq au vin. I think I'll make that tonight.",
    # Speaker A speaks, shifts to Italian topic. The model should chime in.
    "Hmm, that sounds complicated. I prefer Italian food. Say, do you know how to make a simple Margherita pizza recipe?",
]
```

Public

Affective Dialog

When affective dialog is enabled, the model can understand and respond appropriately to users' emotional expressions for more nuanced conversations.

This scenario enables Affective Dialog (`enable_affective_dialog=True`) and uses a system instruction to create a senior technical advisor persona. The user's input is phrased to convey frustration, prompting an empathetic and helpful response from the model.

```python
affective_config = {
    "system_instruction": {
        "parts": [
            {"text": "You are a senior technical advisor for a complex AI project."}
        ]
    },
    "generation_config": {
        "enable_affective_dialog": True,  # Enable affective dialog
        "response_modalities": ["audio"],
    },
}

affective_dialog_turns = [
    "I have been staring at this API docs for two hours now! It's so confusing and I can't even find where to start the streaming request. I'm completely stuck!",
    # A follow-up turn to see if the model maintains the helpful persona
    "Okay, thanks. I'm using Python. What is the single most important parameter I need to set up for a successful streaming connection?",
]
```

Best Practices

Context Management

Use `ContextWindowCompressionConfig` for long sessions. Native audio tokens accumulate rapidly (approx. 25 tokens/sec of audio).

Client Buffering

Do not buffer input audio significantly (e.g., 1 second) before sending. Send small chunks (20ms - 100ms) to minimize latency.

Resampling

Ensure your client application resamples microphone input (often 44.1kHz or 48kHz) to 16kHz before transmission.

Public

System Instructions

- Define a clear persona in the `system_instruction` field during setup to control the agent's behavior and tone.
- Models perform best when their roles and ideal outputs are well-defined, along with explicit instructions that tell the model how to behave in response to prompts.
- Define personas and roles: Give the model a clear role in the interaction, with skills and knowledge defined, such as telling the model that it is a ship's captain well-versed in seafaring and ship maintenance during the Age of Discovery.

Prompts

- Use clear prompts: Provide examples of what the models should and shouldn't do in the prompts, and try to limit prompts to one prompt per persona or role at a time. Instead of lengthy, multi-page prompts, consider using prompt chaining instead. The model performs best on tasks with single function calls.
- Provide starting commands and information: The Live API expects user input before it responds. To have the Live API initiate the conversation, include a prompt asking it to greet the user or begin the conversation. Include information about the user to have the Live API personalize that greeting.

Language Specification

For optimal performance on Live API, make sure that the API's `language_code` matches the language spoken by the user.

If the expectation is for the model to respond in a non-English language, include the following as part of your system instructions:

```None
RESPOND IN {OUTPUT_LANGUAGE}. YOU MUST RESPOND UNMISTAKABLY IN {OUTPUT_LANGUAGE}.
```

What's Next

- Try the cookbook in a notebook.
- Explore documentation.
- Learn more about demo apps and resources.