Live API

Preview: The Live API is in preview.

The Live API enables low-latency bidirectional voice and video interactions with Gemini, letting you talk to Gemini live while also streaming video input or sharing your screen. Using the Live API, you can provide end users with the experience of natural, human-like voice conversations.

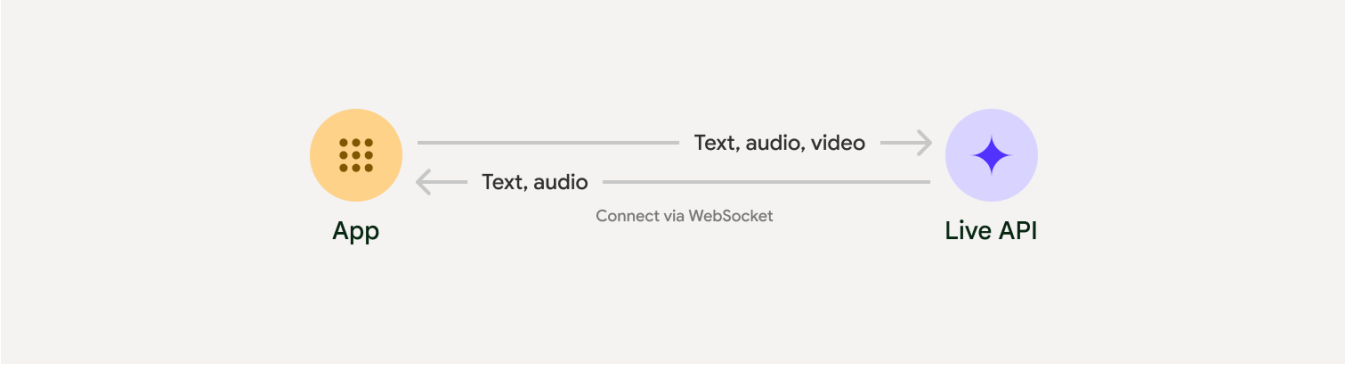
You can try the Live API in Google AI Studio (https://aistudio.google.com/app/live). To use the Live API in Google AI Studio, select **Stream**.

How the Live API works

Streaming

The Live API uses a streaming model over a WebSocket (https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API) connection. When you interact with the API, a persistent connection is created. Your input (audio, video, or text) is streamed continuously to the model, and the model's response (text or audio) is streamed back in real-time over the same connection.

This bidirectional streaming ensures low latency and supports features such as voice activity detection, tool usage, and speech generation.



For more information about the underlying WebSockets API, see the WebSockets API reference (/api/live).

Warning: It is unsafe to insert your API key into client-side JavaScript or TypeScript code. Use server-side deployments for accessing the Live API in production.

Output generation

The Live API processes multimodal input (text, audio, video) to generate text or audio in real-time. It comes with a built-in mechanism to generate audio and depending on the model version you use, it uses one of the two audio generation methods:

- **Half cascade**: The model receives native audio input and uses a specialized model cascade of distinct models to process the input and to generate audio output.
- **Native**: Gemini 2.5 introduces native audio generation (#native-audio-output), which directly generates audio output, providing a more natural sounding audio, more expressive voices, more awareness of additional context, e.g., tone, and more proactive responses.

Building with Live API

Before you begin building with the Live API, choose the audio generation approach (#output-generation) that best fits your needs.

Establishing a connection

The following example shows how to create a connection with an API key:

PythonJavaScript (#javascript)
(#python)

```
import asyncio
from google import genai
```

```
client = genai.Client(api_key="GEMINI_API_KEY")

model = "gemini-2.0-flash-live-001"
config = {"response_modalities": ["TEXT"]}

async def main():
    async with client.aio.live.connect(model=model, config=config) as
        print("Session started")

if __name__ == "__main__":
    asyncio.run(main())
```

Note: You can only set one modality (/gemini-api/docs/live#response-modalities) in the **`response_modalities`** field. This means that you can configure the model to respond with either text or audio, but not both in the same session.

Sending and receiving text

Here's how you can send and receive text:

Python JavaScript (#javascript)
(#python)

```
import asyncio
from google import genai

client = genai.Client(api_key="GEMINI_API_KEY")
model = "gemini-2.0-flash-live-001"

config = {"response_modalities": ["TEXT"]}

async def main():
    async with client.aio.live.connect(model=model, config=config) as
        message = "Hello, how are you?"
        await session.send_client_content(
            turns={"role": "user", "parts": [{"text": message}]}, turn
```

```
        )

        async for response in session.receive():
            if response.text is not None:
                print(response.text, end="")

if __name__ == "__main__":
    asyncio.run(main())
```

Sending and receiving audio

You can send audio by converting it to 16-bit PCM, 16kHz, mono format. This example reads a WAV file and sends it in the correct format:

PythonJavaScript (#javascript)
(#python)

```
# Test file: https://storage.googleapis.com/generativeai-downloads/da
# Install helpers for converting files: pip install librosa soundfile
import asyncio
import io
from pathlib import Path
from google import genai
from google.genai import types
import soundfile as sf
import librosa

client = genai.Client(api_key="GEMINI_API_KEY")
model = "gemini-2.0-flash-live-001"

config = {"response_modalities": ["TEXT"]}

async def main():
    async with client.aio.live.connect(model=model, config=config) as

        buffer = io.BytesIO()
        y, sr = librosa.load("sample.wav", sr=16000)
        sf.write(buffer, y, sr, format='RAW', subtype='PCM_16')
        buffer.seek(0)
```

```
        audio_bytes = buffer.read()

        # If already in correct format, you can use this:
        # audio_bytes = Path("sample.pcm").read_bytes()

        await session.send_realtime_input(
            audio=types.Blob(data=audio_bytes, mime_type="audio/pcm;r
        )

        async for response in session.receive():
            if response.text is not None:
                print(response.text)

if __name__ == "__main__":
    asyncio.run(main())
```

You can receive audio by setting **AUDIO** as response modality. This example saves the received data as WAV file:

PythonJavaScript (#javascript)
(#python)

```
import asyncio
import wave
from google import genai

client = genai.Client(api_key="GEMINI_API_KEY")
model = "gemini-2.0-flash-live-001"

config = {"response_modalities": ["AUDIO"]}

async def main():
    async with client.aio.live.connect(model=model, config=config) as
        wf = wave.open("audio.wav", "wb")
        wf.setnchannels(1)
        wf.setsampwidth(2)
        wf.setframerate(24000)

        message = "Hello how are you?"
        await session.send_client_content(
```

```
            turns={"role": "user", "parts": [{"text": message}]}, turn
        )

        async for response in session.receive():
            if response.data is not None:
                wf.writeframes(response.data)

            # Un-comment this code to print audio data info
            # if response.server_content.model_turn is not None:
            #      print(response.server_content.model_turn.parts[0].

        wf.close()

if __name__ == "__main__":
    asyncio.run(main())
```

Audio formats

Audio data in the Live API is always raw, little-endian, 16-bit PCM. Audio output always uses a sample rate of 24kHz. Input audio is natively 16kHz, but the Live API will resample if needed so any sample rate can be sent. To convey the sample rate of input audio, set the MIME type of each audio-containing Blob (/api/caching#Blob) to a value like **`audio/pcm;rate=16000`**.

Receiving audio transcriptions

You can enable transcription of the model's audio output by sending **`output_audio_transcription`** in the setup config. The transcription language is inferred from the model's response.

```
import asyncio
from google import genai
from google.genai import types

client = genai.Client(api_key="GEMINI_API_KEY")
model = "gemini-2.0-flash-live-001"
```

```
config = {"response_modalities": ["AUDIO"],
          "output_audio_transcription": {}
}

async def main():
    async with client.aio.live.connect(model=model, config=config) as sessi
        message = "Hello? Gemini are you there?"

        await session.send_client_content(
            turns={"role": "user", "parts": [{"text": message}]}, turn_comp
        )

        async for response in session.receive():
            if response.server_content.model_turn:
                print("Model turn:", response.server_content.model_turn)
            if response.server_content.output_transcription:
                print("Transcript:", response.server_content.output_transcr


if __name__ == "__main__":
    asyncio.run(main())
```

You can enable transcription of the audio input by sending **`input_audio_transcription`** in setup config.

```
import asyncio
from google import genai
from google.genai import types

client = genai.Client(api_key="GEMINI_API_KEY")
model = "gemini-2.0-flash-live-001"

config = {"response_modalities": ["TEXT"],
    "realtime_input_config": {
        "automatic_activity_detection": {"disabled": True},
        "activity_handling": "NO_INTERRUPTION",
    },
    "input_audio_transcription": {},
}
```

```
async def main():
    async with client.aio.live.connect(model=model, config=config) as sessi
        audio_data = Path("sample.pcm").read_bytes()

        await session.send_realtime_input(activity_start=types.ActivityStar
        await session.send_realtime_input(
            audio=types.Blob(data=audio_data, mime_type='audio/pcm;rate=160
        )
        await session.send_realtime_input(activity_end=types.ActivityEnd())

        async for msg in session.receive():
            if msg.server_content.input_transcription:
                print('Transcript:', msg.server_content.input_transcription

if __name__ == "__main__":
    asyncio.run(main())
```

Streaming audio and video

To see an example of how to use the Live API in a streaming audio and video format, run the "Live API - Get Started" file in the cookbooks repository:

View on GitHub
(https://github.com/google-gemini/cookbook/blob/main/quickstarts/Get_started_LiveAPI.py)

System instructions

System instructions let you steer the behavior of a model based on your specific needs and use cases. System instructions can be set in the setup configuration and will remain in effect for the entire session.

```
from google.genai import types

config = {
```

```
    "system_instruction": types.Content(
        parts=[
            types.Part(
                text="You are a helpful assistant and answer in a friendly
            )
        ]
    ),
    "response_modalities": ["TEXT"],
}
```

Incremental content updates

Use incremental updates to send text input, establish session context, or restore session context. For short contexts you can send turn-by-turn interactions to represent the exact sequence of events:

PythonJSON (#json)
(#python)

```
turns = [
    {"role": "user", "parts": [{"text": "What is the capital of Franc
    {"role": "model", "parts": [{"text": "Paris"}]},
]

await session.send_client_content(turns=turns, turn_complete=False)

turns = [{"role": "user", "parts": [{"text": "What is the capital of

await session.send_client_content(turns=turns, turn_complete=True)
```

For longer contexts it's recommended to provide a single message summary to free up the context window for subsequent interactions.

Changing voice and language

The Live API supports the following voices: Puck, Charon, Kore, Fenrir, Aoede, Leda, Orus,

and Zephyr.

To specify a voice, set the voice name within the **`speechConfig`** object as part of the session configuration:

PythonJSON (#json)
(#python)

```
from google.genai import types

config = types.LiveConnectConfig(
    response_modalities=["AUDIO"],
    speech_config=types.SpeechConfig(
        voice_config=types.VoiceConfig(
            prebuilt_voice_config=types.PrebuiltVoiceConfig(voice_nam
        )
    )
)
```

Note: If you're using the **`generateContent`** API, the set of available voices is slightly different. See the audio generation guide (/gemini-api/docs/audio-generation#voices) for **`generateContent`** audio generation voices.

The Live API supports multiple languages (#supported-languages).

To change the language, set the language code within the **`speechConfig`** object as part of the session configuration:

```
from google.genai import types

config = types.LiveConnectConfig(
    response_modalities=["AUDIO"],
    speech_config=types.SpeechConfig(
        language_code="de-DE",
    )
)
```

Note: Native audio output (#native-audio-output) models automatically choose the appropriate language and don't support explicitly setting the language code.

Native audio output

Through the Live API, you can also access models that allow for native audio output in addition to native audio input. This allows for higher quality audio outputs with better pacing, voice naturalness, verbosity, and mood.

Native audio output is supported by the following native audio models (/gemini-api/docs/models#gemini-2.5-flash-native-audio):

- `gemini-2.5-flash-preview-native-audio-dialog`
- `gemini-2.5-flash-exp-native-audio-thinking-dialog`

Note: Native audio models currently have limited tool use support. See Overview of supported tools (#tools-overview) for details.

How to use native audio output

To use native audio output, configure one of the native audio models (/gemini-api/docs/models#gemini-2.5-flash-native-audio) and set `response_modalities` to `AUDIO`.

See Sending and receiving audio (#send-receive-audio) for a full example.

Python JavaScript (#javascript)
(#python)

```
model = "gemini-2.5-flash-preview-native-audio-dialog"
config = types.LiveConnectConfig(response_modalities=["AUDIO"])
```

```
async with client.aio.live.connect(model=model, config=config) as ses
    # Send audio input and receive audio
```

Affective dialog

This feature lets Gemini adapt its response style to the input expression and tone.

To use affective dialog, set the api version to **`v1alpha`** and set **`enable_affective_dialog`** to **`true`**in the setup message:

PythonJavaScript (#javascript)
(#python)

```
client = genai.Client(api_key="GOOGLE_API_KEY", http_options={"api_ve

config = types.LiveConnectConfig(
    response_modalities=["AUDIO"],
    enable_affective_dialog=True
)
```

Note that affective dialog is currently only supported by the native audio output models.

Proactive audio

When this feature is enabled, Gemini can proactively decide not to respond if the content is not relevant.

To use it, set the api version to **`v1alpha`** and configure the **`proactivity`** field in the setup message and set **`proactive_audio`** to **`true`**:

PythonJavaScript (#javascript)
(#python)

```
client = genai.Client(api_key="GOOGLE_API_KEY", http_options={"api_ve
```

```
config = types.LiveConnectConfig(
    response_modalities=["AUDIO"],
    proactivity={'proactive_audio': True}
)
```

Note that proactive audio is currently only supported by the native audio output models.

Native audio output with thinking

Native audio output supports thinking capabilities (/gemini-api/docs/thinking), available via a separate model **`gemini-2.5-flash-exp-native-audio-thinking-dialog`**.

See Sending and receiving audio (#send-receive-audio) for a full example.

Python JavaScript (#javascript)
(#python)

```
model = "gemini-2.5-flash-exp-native-audio-thinking-dialog"
config = types.LiveConnectConfig(response_modalities=["AUDIO"])

async with client.aio.live.connect(model=model, config=config) as ses
    # Send audio input and receive audio
```

Tool use with Live API

You can define tools such as Function calling (/gemini-api/docs/function-calling), Code execution (/gemini-api/docs/code-execution), and Google Search (/gemini-api/docs/grounding) with the Live API.

To see examples of all tools in the Live API, run the "Live API Tools" cookbook:

View on GitHub

(https://github.com/google-gemini/cookbook/blob/main/quickstarts/Get_started_LiveAPI_tools.ipynb)

Overview of supported tools

Here's a brief overview of the available tools for each model:

Tool	Cascaded models `gemini-2.0-flash-live-001`	`gemini-2.5-flash-preview-native-audio-dialog`	`gemini-2.5-flash-exp-native-audio-thinking-dialog`
Search	Yes	Yes	Yes
Function calling	Yes	Yes	No
Code execution	Yes	No	No
Url context	Yes	No	No

Function calling

You can define function declarations as part of the session configuration. See the Function calling tutorial (/gemini-api/docs/function-calling) to learn more.

After receiving tool calls, the client should respond with a list of `FunctionResponse` objects using the `session.send_tool_response` method.

Note: Unlike the `generateContent` API, the Live API doesn't support automatic tool response handling. You must handle tool responses manually in your client code.

```
import asyncio
from google import genai
from google.genai import types

client = genai.Client(api_key="GEMINI_API_KEY")
model = "gemini-2.0-flash-live-001"

# Simple function definitions
turn_on_the_lights = {"name": "turn_on_the_lights"}
turn_off_the_lights = {"name": "turn_off_the_lights"}

tools = [{"function_declarations": [turn_on_the_lights, turn_off_the_lights
config = {"response_modalities": ["TEXT"], "tools": tools}

async def main():
    async with client.aio.live.connect(model=model, config=config) as sessi
        prompt = "Turn on the lights please"
        await session.send_client_content(turns={"parts": [{"text": prompt}

        async for chunk in session.receive():
            if chunk.server_content:
                if chunk.text is not None:
                    print(chunk.text)
            elif chunk.tool_call:
                function_responses = []
                for fc in tool_call.function_calls:
                    function_response = types.FunctionResponse(
                        id=fc.id,
                        name=fc.name,
                        response={ "result": "ok" } # simple, hard-coded fu
                    )
                    function_responses.append(function_response)

                await session.send_tool_response(function_responses=functio


if __name__ == "__main__":
    asyncio.run(main())
```

From a single prompt, the model can generate multiple function calls and the code necessary to chain their outputs. This code executes in a sandbox environment,

generating subsequent BidiGenerateContentToolCall (/api/live#bidigeneratecontenttoolcall) messages.

Asynchronous function calling

By default, the execution pauses until the results of each function call are available, which ensures sequential processing. It means you won't be able to continue interacting with the model while the functions are being run.

If you don't want to block the conversation, you can tell the model to run the functions asynchronously.

To do so, you first need to add a `behavior` to the function definitions:

```
  # Non-blocking function definitions
  turn_on_the_lights = {"name": "turn_on_the_lights", "behavior": "NON_BLOC
  turn_off_the_lights = {"name": "turn_off_the_lights"} # turn_off_the_ligh
```

`NON-BLOCKING` will ensure the function will run asynchronously while you can continue interacting with the model.

Then you need to tell the model how to behave when it receives the `FunctionResponse` using the `scheduling` parameter. It can either:

- Interrupt what it's doing and tell you about the response it got right away (`scheduling="INTERRUPT"`),
- Wait until it's finished with what it's currently doing (`scheduling="WHEN_IDLE"`),
- Or do nothing and use that knowledge later on in the discussion (`scheduling="SILENT"`)

```
# Non-blocking function definitions
  function_response = types.FunctionResponse(
      id=fc.id,
      name=fc.name,
```

```
      response={
          "result": "ok",
          "scheduling": "INTERRUPT" # Can also be WHEN_IDLE or SILENT
      }
  )
```

Code execution

You can define code execution as part of the session configuration. See the Code execution tutorial (/gemini-api/docs/code-execution) to learn more.

```
import asyncio
from google import genai
from google.genai import types

client = genai.Client(api_key="GEMINI_API_KEY")
model = "gemini-2.0-flash-live-001"

tools = [{'code_execution': {}}]
config = {"response_modalities": ["TEXT"], "tools": tools}

async def main():
    async with client.aio.live.connect(model=model, config=config) as sessi
        prompt = "Compute the largest prime palindrome under 100000."
        await session.send_client_content(turns={"parts": [{"text": prompt}

        async for chunk in session.receive():
            if chunk.server_content:
                if chunk.text is not None:
                    print(chunk.text)

                model_turn = chunk.server_content.model_turn
                if model_turn:
                    for part in model_turn.parts:
                      if part.executable_code is not None:
                        print(part.executable_code.code)

                      if part.code_execution_result is not None:
                        print(part.code_execution_result.output)
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

Grounding with Google Search

You can enable Grounding with Google Search as part of the session configuration. See the Grounding tutorial (/gemini-api/docs/grounding) to learn more.

```
import asyncio
from google import genai
from google.genai import types

client = genai.Client(api_key="GEMINI_API_KEY")
model = "gemini-2.0-flash-live-001"

tools = [{'google_search': {}}]
config = {"response_modalities": ["TEXT"], "tools": tools}

async def main():
    async with client.aio.live.connect(model=model, config=config) as sessi
        prompt = "When did the last Brazil vs. Argentina soccer match happe
        await session.send_client_content(turns={"parts": [{"text": prompt}

        async for chunk in session.receive():
            if chunk.server_content:
                if chunk.text is not None:
                    print(chunk.text)

                # The model might generate and execute Python code to use S
                model_turn = chunk.server_content.model_turn
                if model_turn:
                    for part in model_turn.parts:
                      if part.executable_code is not None:
                        print(part.executable_code.code)

                      if part.code_execution_result is not None:
                        print(part.code_execution_result.output)
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

Combining multiple tools

You can combine multiple tools within the Live API:

```
prompt = """
Hey, I need you to do three things for me.

1. Compute the largest prime palindrome under 100000.
2. Then use Google Search to look up information about the largest earthqua
3. Turn on the lights

Thanks!
"""

tools = [
    {"google_search": {}},
    {"code_execution": {}},
    {"function_declarations": [turn_on_the_lights, turn_off_the_lights]},
]

config = {"response_modalities": ["TEXT"], "tools": tools}
```

Handling interruptions

Users can interrupt the model's output at any time. When Voice activity detection (#voice-activity-detection) (VAD) 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 BidiGenerateContentServerContent (/api/live#bidigeneratecontentservercontent) message to report the interruption.

In addition, the Gemini server discards any pending function calls and sends a

`BidiGenerateContentServerContent` message with the IDs of the canceled calls.

```
async for response in session.receive():
    if response.server_content.interrupted is True:
        # The generation was interrupted
```

Voice activity detection (VAD)

You can configure or disable voice activity detection (VAD).

Using automatic VAD

By default, the model automatically performs VAD on a continuous audio input stream. VAD can be configured with the `realtimeInputConfig.automaticActivityDetection` (/api/live#RealtimeInputConfig.AutomaticActivityDetection) field of the setup configuration (/api/live#BidiGenerateContentSetup).

When the audio stream is paused for more than a second (for example, because the user switched off the microphone), an `audioStreamEnd` (/api/live#BidiGenerateContentRealtimeInput.FIELDS.bool.BidiGenerateContentRealtimeInput.audio_stream_end) event should be sent to flush any cached audio. The client can resume sending audio data at any time.

```
# example audio file to try:
# URL = "https://storage.googleapis.com/generativeai-downloads/data/hello_a
# !wget -q $URL -O sample.pcm
import asyncio
from pathlib import Path
from google import genai
from google.genai import types

client = genai.Client(api_key="GEMINI_API_KEY")
model = "gemini-2.0-flash-live-001"
```

```
config = {"response_modalities": ["TEXT"]}

async def main():
    async with client.aio.live.connect(model=model, config=config) as sessi
        audio_bytes = Path("sample.pcm").read_bytes()

        await session.send_realtime_input(
            audio=types.Blob(data=audio_bytes, mime_type="audio/pcm;rate=16
        )

        # if stream gets paused, send:
        # await session.send_realtime_input(audio_stream_end=True)

        async for response in session.receive():
            if response.text is not None:
                print(response.text)

if __name__ == "__main__":
    asyncio.run(main())
```

With **`send_realtime_input`**, the API will respond to audio automatically based on VAD. While **`send_client_content`** adds messages to the model context in order, **`send_realtime_input`** is optimized for responsiveness at the expense of deterministic ordering.

Configuring automatic VAD

For more control over the VAD activity, you can configure the following parameters. See API reference (/api/live#automaticactivitydetection) for more info.

```
from google.genai import types

config = {
    "response_modalities": ["TEXT"],
    "realtime_input_config": {
        "automatic_activity_detection": {
            "disabled": False, # default
            "start_of_speech_sensitivity": types.StartSensitivity.START_SEN
```

```
            "end_of_speech_sensitivity": types.EndSensitivity.END_SENSITIVI
            "prefix_padding_ms": 20,
            "silence_duration_ms": 100,
        }
    }
}
```

Disabling automatic VAD

Alternatively, the automatic VAD can be disabled by setting **`realtimeInputConfig.automaticActivityDetection.disabled`** to **`true`** in the setup message. In this configuration the client is responsible for detecting user speech and sending **`activityStart`** (/api/live#BidiGenerateContentRealtimeInput.FIELDS.BidiGenerateContentRealtimeInput.ActivityStart.BidiGenerateContentRealtimeInput.activity_start) and **`activityEnd`** (/api/live#BidiGenerateContentRealtimeInput.FIELDS.BidiGenerateContentRealtimeInput.ActivityEnd.BidiGenerateContentRealtimeInput.activity_end) messages at the appropriate times. An **`audioStreamEnd`** isn't sent in this configuration. Instead, any interruption of the stream is marked by an **`activityEnd`** message.

```
config = {
    "response_modalities": ["TEXT"],
    "realtime_input_config": {"automatic_activity_detection": {"disabled":
}

async with client.aio.live.connect(model=model, config=config) as session:
    # ...
    await session.send_realtime_input(activity_start=types.ActivityStart())
    await session.send_realtime_input(
        audio=types.Blob(data=audio_bytes, mime_type="audio/pcm;rate=16000"
    )
    await session.send_realtime_input(activity_end=types.ActivityEnd())
    # ...
```

Token count

You can find the total number of consumed tokens in the usageMetadata (/api/live#usagemetadata) field of the returned server message.

```
async for message in session.receive():
    # The server will periodically send messages that include UsageMetadata
    if message.usage_metadata:
        usage = message.usage_metadata
        print(
            f"Used {usage.total_token_count} tokens in total. Response toke
        )
        for detail in usage.response_tokens_details:
            match detail:
                case types.ModalityTokenCount(modality=modality, token_coun
                    print(f"{modality}: {count}")
```

Extending the session duration

The maximum session duration (#maximum-session-duration) can be extended to unlimited with two mechanisms:

- Context window compression (#context-window-compression)
- Session resumption (#session-resumption)

Furthermore, you'll receive a GoAway message (#goaway-message) before the session ends, allowing you to take further actions.

Context window compression

To enable longer sessions, and avoid abrupt connection termination, you can enable context window compression by setting the contextWindowCompression (/api/live#BidiGenerateContentSetup.FIELDS.ContextWindowCompressionConfig.BidiGenerateContentSetup.context_window_compression)

field as part of the session configuration.

In the ContextWindowCompressionConfig (/api/live#contextwindowcompressionconfig), you can configure a sliding-window mechanism (/api/live#ContextWindowCompressionConfig.FIELDS.ContextWindowCompressionConfig.SlidingWindow.ContextWindowCompressionConfig.sliding_window) and the number of tokens (/api/live#ContextWindowCompressionConfig.FIELDS.int64.ContextWindowCompressionConfig.trigger_tokens) that triggers compression.

```
from google.genai import types

config = types.LiveConnectConfig(
    response_modalities=["AUDIO"],
    context_window_compression=(
        # Configures compression with default parameters.
        types.ContextWindowCompressionConfig(
            sliding_window=types.SlidingWindow(),
        )
    ),
)
```

Session resumption

To prevent session termination when the server periodically resets the WebSocket connection, configure the sessionResumption (/api/live#BidiGenerateContentSetup.FIELDS.SessionResumptionConfig.BidiGenerateContentSetup.session_resumption) field within the setup configuration (/api/live#BidiGenerateContentSetup).

Passing this configuration causes the server to send SessionResumptionUpdate (/api/live#SessionResumptionUpdate) messages, which can be used to resume the session by passing the last resumption token as the **`SessionResumptionConfig.handle`** (/api/liveSessionResumptionConfig.FIELDS.string.SessionResumptionConfig.handle) of the subsequent connection.

```
import asyncio
from google import genai
from google.genai import types

client = genai.Client(api_key="GEMINI_API_KEY")
model = "gemini-2.0-flash-live-001"

async def main():
    print(f"Connecting to the service with handle {previous_session_handle}
    async with client.aio.live.connect(
        model=model,
        config=types.LiveConnectConfig(
            response_modalities=["AUDIO"],
            session_resumption=types.SessionResumptionConfig(
                # The handle of the session to resume is passed here,
                # or else None to start a new session.
                handle=previous_session_handle
            ),
        ),
    ) as session:
        while True:
            await session.send_client_content(
                turns=types.Content(
                    role="user", parts=[types.Part(text="Hello world!")]
                )
            )
            async for message in session.receive():
                # Periodically, the server will send update messages that m
                # contain a handle for the current state of the session.
                if message.session_resumption_update:
                    update = message.session_resumption_update
                    if update.resumable and update.new_handle:
                        # The handle should be retained and linked to the s
                        return update.new_handle

                # For the purposes of this example, placeholder input is co
                # to the model. In non-sample code, the model inputs would
                # the user.
                if message.server_content and message.server_content.turn_c
                    break
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

Receiving a message before the session disconnects

The server sends a GoAway (/api/live#GoAway) message that signals that the current connection will soon be terminated. This message includes the timeLeft (/api/live#GoAway.FIELDS.google.protobuf.Duration.GoAway.time_left), indicating the remaining time and lets you take further action before the connection will be terminated as ABORTED.

```
async for response in session.receive():
    if response.go_away is not None:
        # The connection will soon be terminated
        print(response.go_away.time_left)
```

Receiving a message when the generation is complete

The server sends a generationComplete (/api/live#BidiGenerateContentServerContent.FIELDS.bool.BidiGenerateContentServerContent.generation_complete) message that signals that the model finished generating the response.

```
async for response in session.receive():
    if response.server_content.generation_complete is True:
        # The generation is complete
```

Media resolution

You can specify the media resolution for the input media by setting the **`mediaResolution`** field as part of the session configuration:

```
from google.genai import types

config = types.LiveConnectConfig(
    response_modalities=["AUDIO"],
    media_resolution=types.MediaResolution.MEDIA_RESOLUTION_LOW,
)
```

Limitations

Consider the following limitations of the Live API when you plan your project.

Response modalities

You can only set one response modality (`TEXT` or `AUDIO`) per session in the session configuration. Setting both results in a config error message. This means that you can configure the model to respond with either text or audio, but not both in the same session.

Client authentication

The Live API only provides server to server authentication and isn't recommended for direct client use. Client input should be routed through an intermediate application server for secure authentication with the Live API.

Session duration

Session duration can be extended to unlimited by enabling session compression (#context-window-compression). Without compression, audio-only sessions are limited to 15 minutes, and audio plus video sessions are limited to 2 minutes. Exceeding these limits without compression will terminate the connection.

Additionally, you can configure session resumption (#session-resumption) to allow the client to resume a session that was terminated.

Context window

A session has a context window limit of:

- 128k tokens for native audio output (#native-audio-output) models
- 32k tokens for other Live API models

Supported languages

Live API supports the following languages.

Note: Native audio output (#native-audio-output) models automatically choose the appropriate language and don't support explicitly setting the language code.

Language	BCP-47 Code
German (Germany)	de-DE
English (Australia)	en-AU
English (United Kingdom)	en-GB
English (India)	en-IN
English (US)	en-US
Spanish (United States)	es-US
French (France)	fr-FR
Hindi (India)	hi-IN

Portuguese (Brazil)	pt-BR
Arabic (Generic)	ar-XA
Spanish (Spain)	es-ES
French (Canada)	fr-CA
Indonesian (Indonesia)	id-ID
Italian (Italy)	it-IT
Japanese (Japan)	ja-JP
Turkish (Turkey)	tr-TR
Vietnamese (Vietnam)	vi-VN
Bengali (India)	bn-IN
Gujarati (India)	gu-IN
Kannada (India)	kn-IN
Malayalam (India)	ml-IN
Marathi (India)	mr-IN
Tamil (India)	ta-IN
Telugu (India)	te-IN

Dutch (Netherlands)	nl-NL
Korean (South Korea)	ko-KR
Mandarin Chinese (China)	cmn-CN
Polish (Poland)	pl-PL
Russian (Russia)	ru-RU
Thai (Thailand)	th-TH

Third-party integrations

For web and mobile app deployments, you can explore options from:

- Daily (https://www.daily.co/products/gemini/multimodal-live-api/)
- Livekit (https://docs.livekit.io/agents/integrations/google/#multimodal-live-api)

What's next

- Try the Live API in Google AI Studio (https://aistudio.google.com/app/live).
- For more info about Gemini 2.0 Flash Live, see the model page (/gemini-api/docs/models#live-api).
- Try more examples in the Live API cookbook (https://github.com/google-gemini/cookbook/blob/main/quickstarts/Get_started_LiveAPI.ipynb), the Live API Tools cookbook

(https://github.com/google-gemini/cookbook/blob/main/quickstarts/Get_started_LiveAPI_tools.ipynb)
, and the Live API Get Started script
(https://github.com/google-gemini/cookbook/blob/main/quickstarts/Get_started_LiveAPI.py).

Last updated 2025-05-24 UTC.