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.

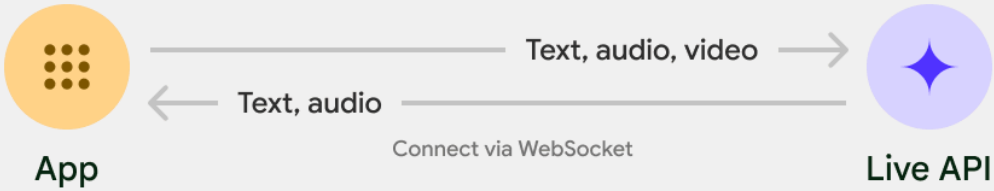
You can try the Live API in Google AI Studio (https://aistudio.google.com/app/live). To use the Live API in Google AI Studio, select **Stream**.

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:

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality } from '@google/genai';

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
const model = 'gemini-2.0-flash-live-001';
const config = { responseModalities: [Modality.TEXT] };

async function main() {

  const session = await ai.live.connect({
    model: model,
    callbacks: {
      onopen: function () {
        console.debug('Opened');
      },
      onmessage: function (message) {
        console.debug(message);
      },
      onerror: function (e) {
        console.debug('Error:', e.message);
```

```
      },
      onclose: function (e) {
        console.debug('Close:', e.reason);
      },
    },
    config: config,
  });

  // Send content...

  session.close();
}

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 (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality } from '@google/genai';

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
```

```
const model = 'gemini-2.0-flash-live-001';
const config = { responseModalities: [Modality.TEXT] };

async function live() {
  const responseQueue = [];

  async function waitMessage() {
    let done = false;
    let message = undefined;
    while (!done) {
      message = responseQueue.shift();
      if (message) {
        done = true;
      } else {
        await new Promise((resolve) => setTimeout(resolve, 100));
      }
    }
    return message;
  }

  async function handleTurn() {
    const turns = [];
    let done = false;
    while (!done) {
      const message = await waitMessage();
      turns.push(message);
      if (message.serverContent && message.serverContent.turnComplete) {
        done = true;
      }
    }
    return turns;
  }
```

```
const session = await ai.live.connect({
  model: model,
  callbacks: {
    onopen: function () {
      console.debug('Opened');
    },
    onmessage: function (message) {
      responseQueue.push(message);
    },
    onerror: function (e) {
      console.debug('Error:', e.message);
    },
    onclose: function (e) {
      console.debug('Close:', e.reason);
    },
  },
  config: config,
});

const inputTurns = 'Hello how are you?';
session.sendClientContent({ turns: inputTurns });

const turns = await handleTurn();
for (const turn of turns) {
  if (turn.text) {
    console.debug('Received text: %s\n', turn.text);
  }
  else if (turn.data) {
    console.debug('Received inline data: %s\n', turn.data);
  }
}

session.close();
```

```
}

async function main() {
  await live().catch((e) => console.error('got error', e));
}

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:

Python (#python) JavaScript (#javascript)

```
// Test file: https://storage.googleapis.com/generativeai-downloads/data/16000.wav
// Install helpers for converting files: npm install wavefile
import { GoogleGenAI, Modality } from '@google/genai';
import * as fs from "node:fs";
import pkg from 'wavefile';
const { WaveFile } = pkg;

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
const model = 'gemini-2.0-flash-live-001';
const config = { responseModalities: [Modality.TEXT] };

async function live() {
  const responseQueue = [];
```

```
async function waitMessage() {
  let done = false;
  let message = undefined;
  while (!done) {
    message = responseQueue.shift();
    if (message) {
      done = true;
    } else {
      await new Promise((resolve) => setTimeout(resolve, 100));
    }
  }
  return message;
}

async function handleTurn() {
  const turns = [];
  let done = false;
  while (!done) {
    const message = await waitMessage();
    turns.push(message);
    if (message.serverContent && message.serverContent.turnComplete) {
      done = true;
    }
  }
  return turns;
}

const session = await ai.live.connect({
  model: model,
  callbacks: {
    onopen: function () {
      console.debug('Opened');
    },
```

```
    onmessage: function (message) {
      responseQueue.push(message);
    },
    onerror: function (e) {
      console.debug('Error:', e.message);
    },
    onclose: function (e) {
      console.debug('Close:', e.reason);
    },
  },
  config: config,
});

// Send Audio Chunk
const fileBuffer = fs.readFileSync("sample.wav");

// Ensure audio conforms to API requirements (16-bit PCM, 16kHz, mono)
const wav = new WaveFile();
wav.fromBuffer(fileBuffer);
wav.toSampleRate(16000);
wav.toBitDepth("16");
const base64Audio = wav.toBase64();

// If already in correct format, you can use this:
// const fileBuffer = fs.readFileSync("sample.pcm");
// const base64Audio = Buffer.from(fileBuffer).toString('base64');

session.sendRealtimeInput(
  {
    audio: {
      data: base64Audio,
      mimeType: "audio/pcm;rate=16000"
    }
```

```
    }

  );

  const turns = await handleTurn();
  for (const turn of turns) {
    if (turn.text) {
      console.debug('Received text: %s\n', turn.text);
    }
    else if (turn.data) {
      console.debug('Received inline data: %s\n', turn.data);
    }
  }

  session.close();
}

async function main() {
  await live().catch((e) => console.error('got error', e));
}

main();
```

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

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality } from '@google/genai';
import * as fs from "node:fs";
import pkg from 'wavefile';
```

```
const { WaveFile } = pkg;

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
const model = 'gemini-2.0-flash-live-001';
const config = { responseModalities: [Modality.AUDIO] };

async function live() {
  const responseQueue = [];

  async function waitMessage() {
    let done = false;
    let message = undefined;
    while (!done) {
      message = responseQueue.shift();
      if (message) {
        done = true;
      } else {
        await new Promise((resolve) => setTimeout(resolve, 100));
      }
    }
    return message;
  }

  async function handleTurn() {
    const turns = [];
    let done = false;
    while (!done) {
      const message = await waitMessage();
      turns.push(message);
      if (message.serverContent && message.serverContent.turnComplete) {
        done = true;
      }
    }
```

```
  return turns;
}

const session = await ai.live.connect({
  model: model,
  callbacks: {
    onopen: function () {
      console.debug('Opened');
    },
    onmessage: function (message) {
      responseQueue.push(message);
    },
    onerror: function (e) {
      console.debug('Error:', e.message);
    },
    onclose: function (e) {
      console.debug('Close:', e.reason);
    },
  },
  config: config,
});

const inputTurns = 'Hello how are you?';
session.sendClientContent({ turns: inputTurns });

const turns = await handleTurn();

// Combine audio data strings and save as wave file
const combinedAudio = turns.reduce((acc, turn) => {
  if (turn.data) {
    const buffer = Buffer.from(turn.data, 'base64');
    const intArray = new Int16Array(buffer.buffer, buffer.byteOffset, buffer.byteLength / Int16Array.
    return acc.concat(Array.from(intArray));
```

```
    }
    return acc;
  }, []);

  const audioBuffer = new Int16Array(combinedAudio);

  const wf = new WaveFile();
  wf.fromScratch(1, 24000, '16', audioBuffer);
  fs.writeFileSync('output.wav', wf.toBuffer());

  session.close();
}

async function main() {
  await live().catch((e) => console.error('got error', e));
}

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.

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality } from '@google/genai';

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
const model = 'gemini-2.0-flash-live-001';

const config = {
  responseModalities: [Modality.AUDIO],
  outputAudioTranscription: {}
};

async function live() {
  const responseQueue = [];

  async function waitMessage() {
    let done = false;
    let message = undefined;
    while (!done) {
      message = responseQueue.shift();
      if (message) {
        done = true;
      } else {
        await new Promise((resolve) => setTimeout(resolve, 100));
      }
    }
    return message;
  }

  async function handleTurn() {
    const turns = [];
```

```
  let done = false;
  while (!done) {
    const message = await waitMessage();
    turns.push(message);
    if (message.serverContent && message.serverContent.turnComplete) {
      done = true;
    }
  }
  return turns;
}

const session = await ai.live.connect({
  model: model,
  callbacks: {
    onopen: function () {
      console.debug('Opened');
    },
    onmessage: function (message) {
      responseQueue.push(message);
    },
    onerror: function (e) {
      console.debug('Error:', e.message);
    },
    onclose: function (e) {
      console.debug('Close:', e.reason);
    },
  },
  config: config,
});

const inputTurns = 'Hello how are you?';
session.sendClientContent({ turns: inputTurns });
```

```
  const turns = await handleTurn();

  for (const turn of turns) {
    if (turn.serverContent && turn.serverContent.outputTranscription) {
      console.debug('Received output transcription: %s\n', turn.serverContent.outputTranscription.text)
    }
  }

  session.close();
}

async function main() {
  await live().catch((e) => console.error('got error', e));
}

main();
```

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

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality } from '@google/genai';
import * as fs from "node:fs";
import pkg from 'wavefile';
const { WaveFile } = pkg;

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
const model = 'gemini-2.0-flash-live-001';

const config = {
```

```
  responseModalities: [Modality.TEXT],
  inputAudioTranscription: {}
};

async function live() {
  const responseQueue = [];

  async function waitMessage() {
    let done = false;
    let message = undefined;
    while (!done) {
      message = responseQueue.shift();
      if (message) {
        done = true;
      } else {
        await new Promise((resolve) => setTimeout(resolve, 100));
      }
    }
    return message;
  }

  async function handleTurn() {
    const turns = [];
    let done = false;
    while (!done) {
      const message = await waitMessage();
      turns.push(message);
      if (message.serverContent && message.serverContent.turnComplete) {
        done = true;
      }
    }
    return turns;
  }
```

```
const session = await ai.live.connect({
  model: model,
  callbacks: {
    onopen: function () {
      console.debug('Opened');
    },
    onmessage: function (message) {
      responseQueue.push(message);
    },
    onerror: function (e) {
      console.debug('Error:', e.message);
    },
    onclose: function (e) {
      console.debug('Close:', e.reason);
    },
  },
  config: config,
});

// Send Audio Chunk
const fileBuffer = fs.readFileSync("16000.wav");

// Ensure audio conforms to API requirements (16-bit PCM, 16kHz, mono)
const wav = new WaveFile();
wav.fromBuffer(fileBuffer);
wav.toSampleRate(16000);
wav.toBitDepth("16");
const base64Audio = wav.toBase64();

// If already in correct format, you can use this:
// const fileBuffer = fs.readFileSync("sample.pcm");
// const base64Audio = Buffer.from(fileBuffer).toString('base64');
```

```
  session.sendRealtimeInput(
    {
      audio: {
        data: base64Audio,
        mimeType: "audio/pcm;rate=16000"
      }
    }
  );

  const turns = await handleTurn();

  for (const turn of turns) {
    if (turn.serverContent && turn.serverContent.outputTranscription) {
      console.log("Transcription")
      console.log(turn.serverContent.outputTranscription.text);
    }
  }
  for (const turn of turns) {
    if (turn.text) {
      console.debug('Received text: %s\n', turn.text);
    }
    else if (turn.data) {
      console.debug('Received inline data: %s\n', turn.data);
    }
    else if (turn.serverContent && turn.serverContent.inputTranscription) {
      console.debug('Received input transcription: %s\n', turn.serverContent.inputTranscription.text);
    }
  }

  session.close();
}
```

```
async function main() {
  await live().catch((e) => console.error('got error', e));
}

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.

Python (#pyhon) JavaScript (#javascript)

```
const config = {
  responseModalities: [Modality.TEXT],
  systemInstruction: "You are a helpful assistant and answer in a friendly tone."
};
```

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:

Python (#python) JavaScript (#javascript)

```
let inputTurns = [
  { "role": "user", "parts": [{ "text": "What is the capital of France?" }] },
  { "role": "model", "parts": [{ "text": "Paris" }] },
]

session.sendClientContent({ turns: inputTurns, turnComplete: false })

inputTurns = [{ "role": "user", "parts": [{ "text": "What is the capital of Germany?" }] }]

session.sendClientContent({ turns: inputTurns, turnComplete: 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:

```
const config = {
  responseModalities: [Modality.AUDIO],
  speechConfig: { voiceConfig: { prebuiltVoiceConfig: { voiceName: "Kore" } } }
};
```

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:

Python (#python) JavaScript (#javascript)

```
const config = {
  responseModalities: [Modality.AUDIO],
  speechConfig: { languageCode: "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 (#python) JavaScript (#javascript)

```
const model = 'gemini-2.5-flash-preview-native-audio-dialog';
const config = { responseModalities: [Modality.AUDIO] };

async function main() {

  const session = await ai.live.connect({
```

```
    model: model,
    config: config,
    callbacks: ...,
  });

  // Send audio input and receive audio

  session.close();
}

main();
```

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:

Python (#python)JavaScript (#javascript)

```
const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY", httpOptions: {"apiVersion": "v1alpha"} });

const config = {
  responseModalities: [Modality.AUDIO],
  enableAffectiveDialog: 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`**:

Python (#python) JavaScript (#javascript)

```
const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY", httpOptions: {"apiVersion": "v1alpha"} });

const config = {
  responseModalities: [Modality.AUDIO],
  proactivity: { proactiveAudio: 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 (#python) JavaScript (#javascript)

```
const model = 'gemini-2.5-flash-exp-native-audio-thinking-dialog';
const config = { responseModalities: [Modality.AUDIO] };

async function main() {

  const session = await ai.live.connect({
    model: model,
    config: config,
    callbacks: ...,
  });

  // Send audio input and receive audio

  session.close();
}

main();
```

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.

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality } from '@google/genai';

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
const model = 'gemini-2.0-flash-live-001';

// Simple function definitions
const turn_on_the_lights = { name: "turn_on_the_lights" } // , description: '...', parameters: { ... }
const turn_off_the_lights = { name: "turn_off_the_lights" }

const tools = [{ functionDeclarations: [turn_on_the_lights, turn_off_the_lights] }]

const config = {
  responseModalities: [Modality.TEXT],
  tools: tools
}

async function live() {
  const responseQueue = [];

  async function waitMessage() {
    let done = false;
    let message = undefined;
    while (!done) {
      message = responseQueue.shift();
```

```
    if (message) {
      done = true;
    } else {
      await new Promise((resolve) => setTimeout(resolve, 100));
    }
  }
  return message;
}

async function handleTurn() {
  const turns = [];
  let done = false;
  while (!done) {
    const message = await waitMessage();
    turns.push(message);
    if (message.serverContent && message.serverContent.turnComplete) {
      done = true;
    } else if (message.toolCall) {
      done = true;
    }
  }
  return turns;
}

const session = await ai.live.connect({
  model: model,
  callbacks: {
    onopen: function () {
      console.debug('Opened');
    },
    onmessage: function (message) {
      responseQueue.push(message);
    },
```

```
    onerror: function (e) {
      console.debug('Error:', e.message);
    },
    onclose: function (e) {
      console.debug('Close:', e.reason);
    },
  },
  config: config,
});

const inputTurns = 'Turn on the lights please';
session.sendClientContent({ turns: inputTurns });

let turns = await handleTurn();

for (const turn of turns) {
  if (turn.serverContent && turn.serverContent.modelTurn && turn.serverContent.modelTurn.parts) {
    for (const part of turn.serverContent.modelTurn.parts) {
      if (part.text) {
        console.debug('Received text: %s\n', part.text);
      }
    }
  }
  else if (turn.toolCall) {
    const functionResponses = [];
    for (const fc of turn.toolCall.functionCalls) {
      functionResponses.push({
        id: fc.id,
        name: fc.name,
        response: { result: "ok" } // simple, hard-coded function response
      });
    }
```

```
      console.debug('Sending tool response...\n');
      session.sendToolResponse({ functionResponses: functionResponses });
    }
  }

  // Check again for new messages
  turns = await handleTurn();

  for (const turn of turns) {
    if (turn.serverContent && turn.serverContent.modelTurn && turn.serverContent.modelTurn.parts) {
      for (const part of turn.serverContent.modelTurn.parts) {
        if (part.text) {
          console.debug('Received text: %s\n', part.text);
        }
      }
    }
  }

  session.close();
}

async function main() {
  await live().catch((e) => console.error('got error', e));
}

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:

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality, Behavior } from '@google/genai';

// Non-blocking function definitions
const turn_on_the_lights = {name: "turn_on_the_lights", behavior: Behavior.NON_BLOCKING}

// Blocking function definitions
const turn_off_the_lights = {name: "turn_off_the_lights"}

const tools = [{ functionDeclarations: [turn_on_the_lights, turn_off_the_lights] }]
```

`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"`)

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality, Behavior, FunctionResponseScheduling } from '@google/genai';

// for a non-blocking function definition, apply scheduling in the function response:
const functionResponse = {
  id: fc.id,
  name: fc.name,
  response: {
    result: "ok",
    scheduling: FunctionResponseScheduling.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.

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality } from '@google/genai';

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
const model = 'gemini-2.0-flash-live-001';
```

```
const tools = [{codeExecution: {}}]
const config = {
  responseModalities: [Modality.TEXT],
  tools: tools
}

async function live() {
  const responseQueue = [];

  async function waitMessage() {
    let done = false;
    let message = undefined;
    while (!done) {
      message = responseQueue.shift();
      if (message) {
        done = true;
      } else {
        await new Promise((resolve) => setTimeout(resolve, 100));
      }
    }
    return message;
  }

  async function handleTurn() {
    const turns = [];
    let done = false;
    while (!done) {
      const message = await waitMessage();
      turns.push(message);
      if (message.serverContent && message.serverContent.turnComplete) {
        done = true;
      } else if (message.toolCall) {
```

```
      done = true;
    }
  }
  return turns;
}

const session = await ai.live.connect({
  model: model,
  callbacks: {
    onopen: function () {
      console.debug('Opened');
    },
    onmessage: function (message) {
      responseQueue.push(message);
    },
    onerror: function (e) {
      console.debug('Error:', e.message);
    },
    onclose: function (e) {
      console.debug('Close:', e.reason);
    },
  },
  config: config,
});

const inputTurns = 'Compute the largest prime palindrome under 100000.';
session.sendClientContent({ turns: inputTurns });

const turns = await handleTurn();

for (const turn of turns) {
  if (turn.serverContent && turn.serverContent.modelTurn && turn.serverContent.modelTurn.parts) {
    for (const part of turn.serverContent.modelTurn.parts) {
```

```
        if (part.text) {
          console.debug('Received text: %s\n', part.text);
        }
        else if (part.executableCode) {
          console.debug('executableCode: %s\n', part.executableCode.code);
        }
        else if (part.codeExecutionResult) {
          console.debug('codeExecutionResult: %s\n', part.codeExecutionResult.output);
        }
      }
    }
  }

  session.close();
}

async function main() {
  await live().catch((e) => console.error('got error', e));
}

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.

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality } from '@google/genai';

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
const model = 'gemini-2.0-flash-live-001';

const tools = [{googleSearch: {}}]
const config = {
  responseModalities: [Modality.TEXT],
  tools: tools
}

async function live() {
  const responseQueue = [];

  async function waitMessage() {
    let done = false;
    let message = undefined;
    while (!done) {
      message = responseQueue.shift();
      if (message) {
        done = true;
      } else {
        await new Promise((resolve) => setTimeout(resolve, 100));
      }
    }
    return message;
  }

  async function handleTurn() {
    const turns = [];
    let done = false;
    while (!done) {
```

```
    const message = await waitMessage();
    turns.push(message);
    if (message.serverContent && message.serverContent.turnComplete) {
      done = true;
    } else if (message.toolCall) {
      done = true;
    }
  }
  return turns;
}

const session = await ai.live.connect({
  model: model,
  callbacks: {
    onopen: function () {
      console.debug('Opened');
    },
    onmessage: function (message) {
      responseQueue.push(message);
    },
    onerror: function (e) {
      console.debug('Error:', e.message);
    },
    onclose: function (e) {
      console.debug('Close:', e.reason);
    },
  },
  config: config,
});

const inputTurns = 'When did the last Brazil vs. Argentina soccer match happen?';
session.sendClientContent({ turns: inputTurns });
```

```
  const turns = await handleTurn();

  for (const turn of turns) {
    if (turn.serverContent && turn.serverContent.modelTurn && turn.serverContent.modelTurn.parts) {
      for (const part of turn.serverContent.modelTurn.parts) {
        if (part.text) {
          console.debug('Received text: %s\n', part.text);
        }
        else if (part.executableCode) {
          console.debug('executableCode: %s\n', part.executableCode.code);
        }
        else if (part.codeExecutionResult) {
          console.debug('codeExecutionResult: %s\n', part.codeExecutionResult.output);
        }
      }
    }
  }

  session.close();
}

async function main() {
  await live().catch((e) => console.error('got error', e));
}

main();
```

Combining multiple tools

You can combine multiple tools within the Live API:

Python (#python) JavaScript (#javascript)

```
const 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 earthquake in California the week of
3. Turn on the lights

Thanks!
`

const tools = [
  { googleSearch: {} },
  { codeExecution: {} },
  { functionDeclarations: [turn_on_the_lights, turn_off_the_lights] }
]

const config = {
  responseModalities: [Modality.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.

Python (#python) JavaScript (#javascript)

```
const turns = await handleTurn();

for (const turn of turns) {
  if (turn.serverContent && turn.serverContent.interrupted) {
    // 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.

Python (#python) JavaScript (#javascript)

```
// example audio file to try:
// URL = "https://storage.googleapis.com/generativeai-downloads/data/hello_are_you_there.pcm"
// !wget -q $URL -O sample.pcm
import { GoogleGenAI, Modality } from '@google/genai';
import * as fs from "node:fs";

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
const model = 'gemini-2.0-flash-live-001';
const config = { responseModalities: [Modality.TEXT] };

async function live() {
  const responseQueue = [];

  async function waitMessage() {
    let done = false;
    let message = undefined;
    while (!done) {
      message = responseQueue.shift();
      if (message) {
        done = true;
      } else {
        await new Promise((resolve) => setTimeout(resolve, 100));
      }
    }
    return message;
```

```
    }

    async function handleTurn() {
      const turns = [];
      let done = false;
      while (!done) {
        const message = await waitMessage();
        turns.push(message);
        if (message.serverContent && message.serverContent.turnComplete) {
          done = true;
        }
      }
      return turns;
    }

    const session = await ai.live.connect({
      model: model,
      callbacks: {
        onopen: function () {
          console.debug('Opened');
        },
        onmessage: function (message) {
          responseQueue.push(message);
        },
        onerror: function (e) {
          console.debug('Error:', e.message);
        },
        onclose: function (e) {
          console.debug('Close:', e.reason);
        },
      },
      config: config,
    });
```

```
  // Send Audio Chunk
  const fileBuffer = fs.readFileSync("sample.pcm");
  const base64Audio = Buffer.from(fileBuffer).toString('base64');

  session.sendRealtimeInput(
    {
      audio: {
        data: base64Audio,
        mimeType: "audio/pcm;rate=16000"
      }
    }

  );

  // if stream gets paused, send:
  // session.sendRealtimeInput({ audioStreamEnd: true })

  const turns = await handleTurn();
  for (const turn of turns) {
    if (turn.text) {
      console.debug('Received text: %s\n', turn.text);
    }
    else if (turn.data) {
      console.debug('Received inline data: %s\n', turn.data);
    }
  }

  session.close();
}

async function main() {
  await live().catch((e) => console.error('got error', e));
```

```
}

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.

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality, StartSensitivity, EndSensitivity } from '@google/genai';

const config = {
  responseModalities: [Modality.TEXT],
  realtimeInputConfig: {
    automaticActivityDetection: {
      disabled: false, // default
      startOfSpeechSensitivity: StartSensitivity.START_SENSITIVITY_LOW,
      endOfSpeechSensitivity: EndSensitivity.END_SENSITIVITY_LOW,
      prefixPaddingMs: 20,
      silenceDurationMs: 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.

Python (#python) JavaScript (#javascript)

```
const config = {
  responseModalities: [Modality.TEXT],
  realtimeInputConfig: {
    automaticActivityDetection: {
      disabled: true,
    }
  }
};

session.sendRealtimeInput({ activityStart: {} })
```

```
session.sendRealtimeInput(
  {
    audio: {
      data: base64Audio,
      mimeType: "audio/pcm;rate=16000"
    }
  }

);

session.sendRealtimeInput({ activityEnd: {} })
```

Token count

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

Python (#python) JavaScript (#javascript)

```
const turns = await handleTurn();

for (const turn of turns) {
  if (turn.usageMetadata) {
    console.debug('Used %s tokens in total. Response token breakdown:\n', turn.usageMetadata.totalToken

    for (const detail of turn.usageMetadata.responseTokensDetails) {
      console.debug('%s\n', detail);
```

```
      }
    }
  }
```

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.

Python (#python) JavaScript (#javascript)

```
const config = {
  responseModalities: [Modality.AUDIO],
  contextWindowCompression: { 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.

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality } from '@google/genai';

const ai = new GoogleGenAI({ apiKey: "GOOGLE_API_KEY" });
const model = 'gemini-2.0-flash-live-001';

async function live() {
  const responseQueue = [];
```

```
async function waitMessage() {
  let done = false;
  let message = undefined;
  while (!done) {
    message = responseQueue.shift();
    if (message) {
      done = true;
    } else {
      await new Promise((resolve) => setTimeout(resolve, 100));
    }
  }
  return message;
}

async function handleTurn() {
  const turns = [];
  let done = false;
  while (!done) {
    const message = await waitMessage();
    turns.push(message);
    if (message.serverContent && message.serverContent.turnComplete) {
      done = true;
    }
  }
  return turns;
}

console.debug('Connecting to the service with handle %s...', previousSessionHandle)
const session = await ai.live.connect({
  model: model,
  callbacks: {
    onopen: function () {
```

```
        console.debug('Opened');
      },
      onmessage: function (message) {
        responseQueue.push(message);
      },
      onerror: function (e) {
        console.debug('Error:', e.message);
      },
      onclose: function (e) {
        console.debug('Close:', e.reason);
      },
    },
    config: {
      responseModalities: [Modality.TEXT],
      sessionResumption: { handle: previousSessionHandle }
      // The handle of the session to resume is passed here, or else null to start a new session.
    }
  });

  const inputTurns = 'Hello how are you?';
  session.sendClientContent({ turns: inputTurns });

  const turns = await handleTurn();
  for (const turn of turns) {
    if (turn.sessionResumptionUpdate) {
      if (turn.sessionResumptionUpdate.resumable && turn.sessionResumptionUpdate.newHandle) {
        let newHandle = turn.sessionResumptionUpdate.newHandle
        // TODO: store newHandle and start new session with this handle
        // ...
      }
    }
  }
```

```
  session.close();
}

async function main() {
  await live().catch((e) => console.error('got error', e));
}

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.

Python (#python) JavaScript (#javascript)

```
const turns = await handleTurn();

for (const turn of turns) {
  if (turn.goAway) {
    console.debug('Time left: %s\n', turn.goAway.timeLeft);
  }
}
```

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.

Python (#python) JavaScript (#javascript)

```
const turns = await handleTurn();

for (const turn of turns) {
  if (turn.serverContent && turn.serverContent.generationComplete) {
    // 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:

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI, Modality, MediaResolution } from '@google/genai';

const config = {
    responseModalities: [Modality.TEXT],
```

```
    mediaResolution: 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

Language	BCP-47 Code
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

Language	BCP-47 Code
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

Language	BCP-47 Code
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 2025-05-29 UTC.