Tool use with Live API

Tool use allows Live API to go beyond just conversation by enabling it to perform actions in the real-world and pull in external context while maintaining a real time connection. You can define tools such as Function calling (/gemini-api/docs/function-calling) and Google Search (/gemini-api/docs/grounding) with the Live API.

Overview of supported tools

Here's a brief overview of the available tools for Live API models:

Tool	`gemini-2.5-flash-native-audio-preview-12-2025`
Search	Yes
Function calling	Yes
Google Maps	No
Code execution	No
URL context	No

Function calling

Live API supports function calling, just like regular content generation requests. Function calling lets the Live API interact with external data and programs, greatly increasing what your applications can accomplish.

You can define function declarations as part of the session configuration. After receiving tool calls, the client should respond with a list of `FunctionResponse` objects using the `session.send_tool_response` method.

See the Function calling tutorial (/gemini-api/docs/function-calling) to learn more.

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 JavaScript (#javascript)
(#python)

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

client = genai.Client()

model = "gemini-2.5-flash-native-audio-preview-12-2025"

# 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_lig
config = {"response_modalities": ["AUDIO"], "tools": tools}

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

        wf = wave.open("audio.wav", "wb")
        wf.setnchannels(1)
        wf.setsampwidth(2)
        wf.setframerate(24000)  # Output is 24kHz

        async for response in session.receive():
            if response.data is not None:
                wf.writeframes(response.data)
            elif response.tool_call:
                print("The tool was called")
                function_responses = []
                for fc in response.tool_call.function_calls:
                    function_response = types.FunctionResponse(
                        id=fc.id,
                        name=fc.name,
                        response={ "result": "ok" } # simple, hard-codec
                    )
                    function_responses.append(function_response)
```

```
                await session.send_tool_response(function_responses=func

        wf.close()

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

Function calling executes sequentially by default, meaning execution pauses until the results of each function call are available. This ensures sequential processing, which 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:

PythonJavaScript (#javascript)
(#python)

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

`NON-BLOCKING` ensures the function runs 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 JavaScript (#javascript)
(#python)

```
# for a non-blocking function definition, apply scheduling in the functi
  function_response = types.FunctionResponse(
      id=fc.id,
      name=fc.name,
      response={
          "result": "ok",
          "scheduling": "INTERRUPT" # Can also be WHEN_IDLE or SILENT
      }
  )
```

Grounding with Google Search

You can enable Grounding with Google Search as part of the session configuration. This increases the Live API's accuracy and prevents hallucinations. See the Grounding tutorial (/gemini-api/docs/grounding) to learn more.

Python JavaScript (#javascript)
(#python)

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

client = genai.Client()

model = "gemini-2.5-flash-native-audio-preview-12-2025"

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

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

```
        wf = wave.open("audio.wav", "wb")
        wf.setnchannels(1)
        wf.setsampwidth(2)
        wf.setframerate(24000)  # Output is 24kHz

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

                # The model might generate and execute Python code to us
                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)

        wf.close()

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

Combining multiple tools

You can combine multiple tools within the Live API, increasing your application's capabilities even more:

PythonJavaScript (#javascript)
(#python)

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

1. Use Google Search to look up information about the largest earthquake
2. Then turn on the lights

Thanks!
"""

tools = [
```

```
    {"google_search": {}},
    {"function_declarations": [turn_on_the_lights, turn_off_the_lights]}
]

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

# ... remaining model call
```

What's next

- Check out more examples of using tools with the Live API in the Tool use cookbook (https://colab.research.google.com/github/google-gemini/cookbook/blob/main/quickstarts/Get_started_LiveAPI_tools.ipynb) .
- Get the full story on features and configurations from the Live API Capabilities guide (/gemini-api/docs/live-guide).

Last updated 2025-12-18 UTC.