Building Voice Agents

Audio handling

Some transport layers like the default `OpenAIRealtimeWebRTC` will handle audio input and output automatically for you. For other transport mechanisms like `OpenAIRealtimeWebSocket` you will have to handle session audio yourself:

```
import {
  RealtimeAgent,
  RealtimeSession,
  TransportLayerAudio,
} from '@openai/agents/realtime';

const agent = new RealtimeAgent({ name: 'My agent' });
const session = new RealtimeSession(agent);
const newlyRecordedAudio = new ArrayBuffer(0);

session.on('audio', (event: TransportLayerAudio) => {
  // play your audio
});

// send new audio to the agent
session.sendAudio(newlyRecordedAudio);
```

Session configuration

You can configure your session by passing additional options to either the `RealtimeSession` during construction or when you call `connect(...)`.

```
import { RealtimeAgent, RealtimeSession } from
'@openai/agents/realtime';

const agent = new RealtimeAgent({
  name: 'Greeter',
  instructions: 'Greet the user with cheer and answer questions.',
});

const session = new RealtimeSession(agent, {
  model: 'gpt-4o-realtime-preview-2025-06-03',
  config: {
    inputAudioFormat: 'pcm16',
    outputAudioFormat: 'pcm16',
    inputAudioTranscription: {
      model: 'gpt-4o-mini-transcribe',
    },
  },
});
```

These transport layers allow you to pass any parameter that matches session.

For parameters that are new and don't have a matching parameter in the RealtimeSessionConfig you can use `providerData`. Anything passed in `providerData` will be passed directly as part of the `session` object.

Handoffs

Similarly to regular agents, you can use handoffs to break your agent into multiple agents and orchestrate between them to improve the performance of your agents and better scope the problem.

```
import { RealtimeAgent } from '@openai/agents/realtime';

const mathTutorAgent = new RealtimeAgent({
  name: 'Math Tutor',
  handoffDescription: 'Specialist agent for math questions',
  instructions:
    'You provide help with math problems. Explain your reasoning at each
step and include examples',
});

const agent = new RealtimeAgent({
  name: 'Greeter',
  instructions: 'Greet the user with cheer and answer questions.',
  handoffs: [mathTutorAgent],
});
```

Unlike regular agents, handoffs behave slightly differently for Realtime Agents. When a handoff is performed, the ongoing session will be updated with the new agent configuration. Because of this, the agent automatically has access to the ongoing conversation history and input filters are currently not applied.

Additionally, this means that the `voice` or `model` cannot be changed as part of the handoff. You can also only connect to other Realtime Agents. If you need to use a different model, for example a reasoning model like `o4-mini`, you can use delegation through tools.

Tools

Just like regular agents, Realtime Agents can call tools to perform actions. You can define a tool using the same `tool()` function that you would use for a regular agent.

```
import { tool, RealtimeAgent } from '@openai/agents/realtime';
import { z } from 'zod';

const getWeather = tool({
  name: 'get_weather',
  description: 'Return the weather for a city.',
  parameters: z.object({ city: z.string() }),
  async execute({ city }) {
    return `The weather in ${city} is sunny.`;
  },
});

const weatherAgent = new RealtimeAgent({
  name: 'Weather assistant',
  instructions: 'Answer weather questions.',
  tools: [getWeather],
});
```

You can only use function tools with Realtime Agents and these tools will be executed in the same place as your Realtime Session. This means if you are running your Realtime Session in the browser, your tool will be executed in the browser. If you need to perform more sensitive actions, you can make an HTTP request within your tool to your backend server.

While the tool is executing the agent will not be able to process new requests from the user. One way to improve the experience is by telling your agent to announce when it is about to execute a tool or say specific phrases to buy the agent some time to execute the tool.

Accessing the conversation history

Additionally to the arguments that the agent called a particular tool with, you can also access a snapshot of the current conversation history that is tracked by the Realtime Session. This can be useful if you need to perform a more complex action based on the current state of the conversation or are planning to use tools for delegation.

```
import {
  tool,
  RealtimeContextData,
  RealtimeItem,
} from '@openai/agents/realtime';
import { z } from 'zod';

const parameters = z.object({
  request: z.string(),
});

const refundTool = tool<typeof parameters, RealtimeContextData>({
  name: 'Refund Expert',
  description: 'Evaluate a refund',
  parameters,
  execute: async ({ request }, details) => {
    // The history might not be available
    const history: RealtimeItem[] = details?.context?.history ?? [];
    // making your call to process the refund request
  },
});
```

Note

The history passed in is a snapshot of the history at the time of the tool call. The transcription of the last thing the user said might not be available yet.

Approval before tool execution

If you define your tool with `needsApproval: true` the agent will emit a `tool_approval_requested` event before executing the tool.

By listening to this event you can show a UI to the user to approve or reject the tool call.

```
import { session } from './agent';

session.on('tool_approval_requested', (_context, _agent, request) => {
  // show a UI to the user to approve or reject the tool call
  // you can use the `session.approve(...)` or `session.reject(...)`
methods to approve or reject the tool call

  session.approve(request.approvalItem); // or
session.reject(request.rawItem);
});
```

Note

While the voice agent is waiting for approval for the tool call, the agent won't be able to process new requests from the user.

Guardrails

Guardrails offer a way to monitor whether what the agent has said violated a set of rules and immediately cut off the response. These guardrail checks will be performed based on the transcript of the agent's response and therefore requires that the text output of your model is enabled (it is enabled by default).

The guardrails that you provide will run asynchronously as a model response is returned, allowing you to cut off the response based a predefined classification trigger, for example "mentions a specific banned word".

When a guardrail trips the session emits a `guardrail_tripped` event. The event also provides a `details` object containing the `itemId` that triggered the guardrail.

```
import { RealtimeOutputGuardrail, RealtimeAgent, RealtimeSession }
from '@openai/agents/realtime';

const agent = new RealtimeAgent({
  name: 'Greeter',
  instructions: 'Greet the user with cheer and answer questions.',
});

const guardrails: RealtimeOutputGuardrail[] = [
  {
    name: 'No mention of Dom',
    async execute({ agentOutput }) {
      const domInOutput = agentOutput.includes('Dom');
      return {
        tripwireTriggered: domInOutput,
        outputInfo: { domInOutput },
      };
    },
  },
];

const guardedSession = new RealtimeSession(agent, {
  outputGuardrails: guardrails,
});
```

By default guardrails are run every 100 characters or at the end of the response text has been generated. Since speaking out the text normally takes longer it means that in most cases the guardrail should catch the violation before the user can hear it.

If you want to modify this behavior you can pass a `outputGuardrailSettings` object to the session.

```
import { RealtimeAgent, RealtimeSession } from
'@openai/agents/realtime';

const agent = new RealtimeAgent({
  name: 'Greeter',
  instructions: 'Greet the user with cheer and answer questions.',
});

const guardedSession = new RealtimeSession(agent, {
  outputGuardrails: [
    /*...*/
  ],
  outputGuardrailSettings: {
    debounceTextLength: 500, // run guardrail every 500 characters or set
it to -1 to run it only at the end
  },
});
```

Turn detection / voice activity detection

The Realtime Session will automatically detect when the user is speaking and trigger new turns using the built-in voice activity detection modes of the Realtime API.

You can change the voice activity detection mode by passing a `turnDetection` object to the session.

```
import { RealtimeSession } from '@openai/agents/realtime';
import { agent } from './agent';

const session = new RealtimeSession(agent, {
  model: 'gpt-4o-realtime-preview-2025-06-03',
  config: {
    turnDetection: {
      type: 'semantic_vad',
      eagerness: 'medium',
      createResponse: true,
      interruptResponse: true,
    },
  },
});
```

Modifying the turn detection settings can help calibrate unwanted interruptions and dealing with silence. Check out the Realtime API documentation for more details on the different settings

Interruptions

When using the built-in voice activity detection, speaking over the agent automatically triggers the agent to detect and update its context based on what was said. It will also emit an `audio_interrupted` event. This can be used to immediately stop all audio playback (only applicable to WebSocket connections).

```
import { session } from './agent';

session.on('audio_interrupted', () => {
  // handle local playback interruption
});
```

If you want to perform a manual interruption, for example if you want to offer a “stop” button in your UI, you can call `interrupt()` manually:

```
import { session } from './agent';

session.interrupt();
// this will still trigger the `audio_interrupted` event for you
// to cut off the audio playback when using WebSockets
```

In either way, the Realtime Session will handle both interrupting the generation of the agent, truncate its knowledge of what was said to the user, and update the history.

If you are using WebRTC to connect to your agent, it will also clear the audio output. If you are using WebSocket, you will need to handle this yourself by stopping audio playack of whatever has been queued up to be played.

Text input

If you want to send text input to your agent, you can use the `sendMessage` method on the `RealtimeSession`.

This can be useful if you want to enable your user to interface in both modalities with the agent, or to provide additional context to the conversation.

```
import { RealtimeSession, RealtimeAgent } from
'@openai/agents/realtime';

const agent = new RealtimeAgent({
  name: 'Assistant',
});

const session = new RealtimeSession(agent, {
  model: 'gpt-4o-realtime-preview-2025-06-03',
});

session.sendMessage('Hello, how are you?');
```

Conversation history management

The `RealtimeSession` automatically manages the conversation history in a `history` property:

You can use this to render the history to the customer or perform additional actions on it. As this history will constantly change during the course of the conversation you can listen for the `history_updated` event.

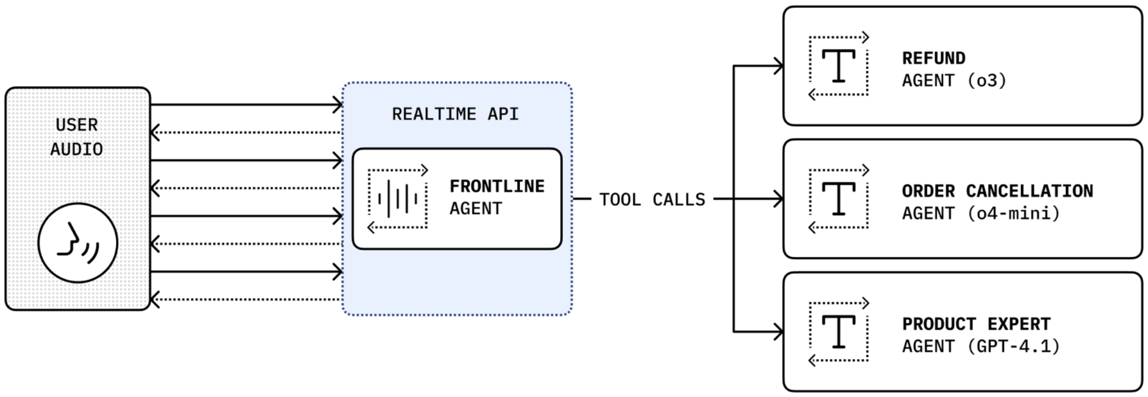
If you want to modify the history, like removing a message entirely or updating its transcript, you can use the `updateHistory` method.

```
import { RealtimeSession, RealtimeAgent } from
'@openai/agents/realtime';

const agent = new RealtimeAgent({
  name: 'Assistant',
});

const session = new RealtimeSession(agent, {
  model: 'gpt-4o-realtime-preview-2025-06-03',
});

await session.connect({ apiKey: '<client-api-key>' });

// listening to the history_updated event
session.on('history_updated', (history) => {
  // returns the full history of the session
  console.log(history);
});

// Option 1: explicit setting
session.updateHistory([
  /* specific history */
]);

// Option 2: override based on current state like removing all agent
messages
session.updateHistory((currentHistory) => {
  return currentHistory.filter(
    (item) => !(item.type === 'message' && item.role === 'assistant'),
  );
});
```

Limitations

1. You can currently not update/change function tool calls after the fact
2. Text output in the history requires transcripts and text modalities to be enabled
3. Responses that were truncated due to an interruption do not have a transcript

Delegation through tools


USER AUDIO
REALTIME API
FRONTLINE AGENT
TOOL CALLS
REFUND AGENT (o3)
ORDER CANCELLATION AGENT (o4-mini)
PRODUCT EXPERT AGENT (GPT-4.1)

By combining the conversation history with a tool call, you can delegate the conversation to another backend agent to perform a more complex action and then pass it back as the result to the user.

```
import {
  RealtimeAgent,
  RealtimeContextData,
  tool,
} from '@openai/agents/realtime';
import { handleRefundRequest } from './serverAgent';
import z from 'zod';

const refundSupervisorParameters = z.object({
  request: z.string(),
});

const refundSupervisor = tool<
  typeof refundSupervisorParameters,
  RealtimeContextData
>({
  name: 'escalateToRefundSupervisor',
  description: 'Escalate a refund request to the refund supervisor',
  parameters: refundSupervisorParameters,
  execute: async ({ request }, details) => {
    // This will execute on the server
    return handleRefundRequest(request, details?.context?.history ?? []);
  },
});

const agent = new RealtimeAgent({
  name: 'Customer Support',
  instructions:
    'You are a customer support agent. If you receive any requests for
refunds, you need to delegate to your supervisor.',
  tools: [refundSupervisor],
});
```

The code below will then be executed on the server. In this example through a server actions in Next.js.

```
// This runs on the server
import 'server-only';

import { Agent, run } from '@openai/agents';
import type { RealtimeItem } from '@openai/agents/realtime';
import z from 'zod';

const agent = new Agent({
  name: 'Refund Expert',
  instructions:
    'You are a refund expert. You are given a request to process a refund
and you need to determine if the request is valid.',
  model: 'o4-mini',
  outputType: z.object({
    reasong: z.string(),
    refundApproved: z.boolean(),
  }),
});

export async function handleRefundRequest(
  request: string,
  history: RealtimeItem[],
) {
  const input = `
The user has requested a refund.

The request is: ${request}

Current conversation history:
${JSON.stringify(history, null, 2)}
`.trim();

  const result = await run(agent, input);

  return JSON.stringify(result.finalOutput, null, 2);
}
```