Grok Voice API

Build interactive voice conversations with Grok models using WebSocket. The Grok Voice API accepts audio and text inputs and creates text and audio responses in real-time.

WebSocket Endpoint: `wss://api.x.ai/v1/realtime`

Authentication

You can authenticate **WebSocket** connections using the xAI API key or an ephemeral token.

⚠

IMPORTANT: It is **recommended to use an ephemeral token** when authenticating from the client side (e.g. browser). If you use the xAI API key to authenticate from the client side, **the client may see the API key and make unauthorized API requests with it.**

Fetching Ephemeral Tokens

You need to set up another server or endpoint to fetch the ephemeral token from xAI. The ephemeral token will give the holder a scoped access to resources.

Endpoint: `POST https://api.x.ai/v1/realtime/client_secrets`

Javascript

```javascript
// Example ephemeral token endpoint with Express

import express from 'express';

const app = express();
const SESSION_REQUEST_URL = "https://api.x.ai/v1/realtime/client_secrets";

app.use(express.json());

app.post("/session", async (req, res) => {
  const r = await fetch(SESSION_REQUEST_URL, {
    method: "POST",
    headers: {
      Authorization: `Bearer ${process.env.XAI_API_KEY}`,
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      expires_after: { seconds: 300 }
    }),
  });

  const data = await r.json();
  res.json(data);
});

app.listen(8081);
```

Using API Key Directly

For server-side applications where the API key is not exposed to clients, you can authenticate directly with your xAI API key.

Server-side only: Only use API key authentication from secure server environments. Never expose your API key in client-side code.

Javascript

```
import WebSocket from "ws";

const baseUrl = "wss://api.x.ai/v1/realtime";

// Connect with API key in Authorization header
const ws = new WebSocket(baseUrl, {
  headers: {
    Authorization: "Bearer " + process.env.XAI_API_KEY,
    "Content-Type": "application/json",
  },
});

ws.on("open", () => {
  console.log("Connected with API key authentication");
});
```

Voice Options

The Grok Voice API supports 6 different voice options, each with distinct characteristics. Select the voice that best fits your application's personality and use case.

Available Voices

Voice	Type	Tone	Description	Sample
Ara	Female	Warm, friendly	Default voice, balanced and conversational	▶
Rex	Male	Confident, clear	Professional and articulate, ideal for business applications	▶
Sal	Neutral	Smooth, balanced	Versatile voice suitable for various contexts	▶
Eve	Female	Energetic, upbeat	Engaging and enthusiastic, great for interactive experiences	▶
Una	Female	Calm, measured	Soothing and deliberate, perfect for guided experiences	▶
Leo	Male	Authoritative, strong	Decisive and commanding, suitable for instructional content	▶

Selecting a Voice

Specify the voice in your session configuration using the `voice` parameter:

Javascript

```javascript
// Configure session with a specific voice
const sessionConfig = {
  type: "session.update",
  session: {
    voice: "Ara", // Choose from: Ara, Rex, Sal, Eve, Una, Leo
    instructions: "You are a helpful assistant.",
    // Audio format settings (these are the defaults if not specified)
    audio: {
      input: { format: { type: "audio/pcm", rate: 24000 } },
      output: { format: { type: "audio/pcm", rate: 24000 } }
    }
  }
};

ws.send(JSON.stringify(sessionConfig));
```

Audio Format

The Grok Voice API supports multiple audio formats for real-time audio streaming. Audio data must be encoded as base64 strings when sent over WebSocket.

Supported Audio Formats

The API supports three audio format types:

Format	Encoding	Container Types	Sample Rate
`audio/pcm`	Linear16, Little-endian	Raw, WAV, AIFF	Configurable (see below)
`audio/pcmu`	G.711 μ-law (Mulaw)	Raw	8000 Hz
`audio/pcma`	G.711 A-law	Raw	8000 Hz

Supported Sample Rates

When using `audio/pcm` format, you can configure the sample rate to one of the following supported values:

Sample Rate	Quality	Description
8000 Hz	Telephone	Narrowband, suitable for voice calls
16000 Hz	Wideband	Good for speech recognition
21050 Hz	Standard	Balanced quality and bandwidth
24000 Hz	High (Default)	Recommended for most use cases

32000 Hz	Very High	Enhanced audio clarity
44100 Hz	CD Quality	Standard for music / media
48000 Hz	Professional	Studio-grade audio / Web Browser

Note: Sample rate configuration is only applicable for `audio/pcm` format. The `audio/pcmu` and `audio/pcma` formats use their standard encoding specifications.

Audio Specifications

Property	Value	Description
Sample Rate	Configurable (PCM only)	Sample rate in Hz (see supported rates above)
Default Sample Rate	24kHz	24,000 samples per second (for PCM)
Channels	Mono	Single channel audio
Encoding	Base64	Audio bytes encoded as base64 string
Byte Order	Little-endian	16-bit samples in little-endian format (for PCM)

Configuring Audio Format

You can configure the audio format and sample rate for both input and output in the session configuration:

Javascript

```javascript
// Configure audio format with custom sample rate for input and output
const sessionConfig = {
  type: "session.update",
  session: {
    audio: {
      input: {
        format: {
          type: "audio/pcm", // or "audio/pcmu" or "audio/pcma"
          rate: 16000 // Only applicable for audio/pcm
        }
      },
      output: {
        format: {
          type: "audio/pcm", // or "audio/pcmu" or "audio/pcma"
          rate: 16000 // Only applicable for audio/pcm
        }
      }
    },
    instructions: "You are a helpful assistant.",
  }
};

ws.send(JSON.stringify(sessionConfig));
```

Connect via WebSocket

You can connect to the realtime model via WebSocket. The audio data needs to be serialized into base64-encoded strings.

The examples below show connecting to the WebSocket endpoint from the server environment.

Javascript

```javascript
import WebSocket from "ws";

const baseUrl = "wss://api.x.ai/v1/realtime";
const ws = new WebSocket(baseUrl, {
  headers: {
    Authorization: "Bearer " + process.env.XAI_API_KEY,
    "Content-Type": "application/json",
  },
});

ws.on("open", function open() {
  console.log("Connected to server.");

  // Configure the session with voice, audio format, and instructions
  const sessionConfig = {
    type: "session.update",
    session: {
      voice: "Ara",
      instructions: "You are a helpful assistant.",
      turn_detection: { type: "server_vad" },
      audio: {
        input: { format: { type: "audio/pcm", rate: 24000 } },
        output: { format: { type: "audio/pcm", rate: 24000 } }
      }
    }
  };
  ws.send(JSON.stringify(sessionConfig));

  // Create a new conversation message and send to server
  let event = {
    type: "conversation.item.create",
    item: {
      type: "message",
      role: "user",
      content: [{ type: "input_text", text: "hello" }],
    },
  };
  ws.send(JSON.stringify(event));

  // Send an event to request a response, so Grok will start processing on our previous message
  event = {
    type: "response.create",
  };
  ws.send(JSON.stringify(event));
});

ws.on("message", function incoming(message) {
  const serverEvent = JSON.parse(message);
  console.log(serverEvent);
});
```

Message types

There are a few message types used in interacting with the models. **Client events** are sent by user to the server, and **Server events** are sent by server to client.

Client Event(s)	Server Event(s)
`"session.update"`	`"session.updated"`
	`"conversation.created"`

| `"input_audio_buffer.append"` | `"input_audio_buffer.speech_started"`
`"input_audio_buffer.speech_stopped"`
`"conversation.item.input_audio_transcription.completed"` || `"conversation.item.commit"`
(Audio)
`"conversation.item.create"` (Text) | `"conversation.item.added"` || `"response.create"` |
`"response.created"`
`"response.output_item.added"`
`"response.done"` ||| `"response.output_audio_transcript.delta"`
`"response.output_audio_transcript.done"`
`"response.output_audio.delta"`
`"response.output_audio.done"` |

Session Messages

Client Events

- `"session.update"` - Update session configuration such as system prompt, voice, audio format and search settings

JSON

```
{
    "type": "session.update",
    "session": {
        "instructions": "pass a system prompt here",
        "voice": "Ara",
        "turn_detection": {
            "type": "server_vad" or null,
        },
        "audio": {
            "input": {
                "format": {
                    "type": "audio/pcm",
                    "rate": 24000
                }
            },
            "output": {
```

```
                "format": {
                    "type": "audio/pcm",
                    "rate": 24000
                }
            }
        }
    }
}
```

Session Parameters:

Parameter	Type	Description
`instructions`	string	System prompt
`voice`	string	Voice selection: `Ara`, `Rex`, `Sal`, `Eve`, `Una`, `Leo` (see Voice Options)
`turn_detection.type`	string | null	`"server_vad"` for automatic detection, `null` for manual text turns
`audio.input.format.type`	string	Input format: `"audio/pcm"`, `"audio/pcmu"`, or `"audio/pcma"`
`audio.input.format.rate`	number	Input sample rate (PCM only): 8000, 16000, 21050, 24000, 32000, 44100, 48000
`audio.output.format.type`	string	Output format: `"audio/pcm"`, `"audio/pcmu"`, or `"audio/pcma"`
`audio.output.format.rate`	number	Output sample rate (PCM only): 8000, 16000, 21050, 24000, 32000, 44100, 48000

Receiving and Playing Audio

Decode and play base64 PCM16 audio received from the API. Use the same sample rate as configured:

Javascript

```javascript
// Configure session with 16kHz sample rate for lower bandwidth (input and output)
const sessionConfig = {
  type: "session.update",
  session: {
    instructions: "You are a helpful assistant.",
    voice: "Ara",
    turn_detection: { type: "server_vad" },
    audio: {
      input: {
        format: {
          type: "audio/pcm",
          rate: 16000 // 16kHz for lower bandwidth usage
        }
      },
      output: {
        format: {
          type: "audio/pcm",
          rate: 16000 // 16kHz for lower bandwidth usage
        }
      }
    }
  }
```

```
};
ws.send(JSON.stringify(sessionConfig));

// When processing audio, use the same sample rate
const SAMPLE_RATE = 16000;

// Create AudioContext with matching sample rate
const audioContext = new AudioContext({ sampleRate: SAMPLE_RATE });

// Helper function to convert Float32Array to base64 PCM16
function float32ToBase64PCM16(float32Array) {
  const pcm16 = new Int16Array(float32Array.length);
  for (let i = 0; i < float32Array.length; i++) {
    const s = Math.max(-1, Math.min(1, float32Array[i]));
    pcm16[i] = s < 0 ? s * 0x8000 : s * 0x7FFF;
  }
  const bytes = new Uint8Array(pcm16.buffer);
  return btoa(String.fromCharCode(...bytes));
}

// Helper function to convert base64 PCM16 to Float32Array
function base64PCM16ToFloat32(base64String) {
  const binaryString = atob(base64String);
  const bytes = new Uint8Array(binaryString.length);
  for (let i = 0; i < binaryString.length; i++) {
    bytes[i] = binaryString.charCodeAt(i);
  }
  const pcm16 = new Int16Array(bytes.buffer);
  const float32 = new Float32Array(pcm16.length);
  for (let i = 0; i < pcm16.length; i++) {
    float32[i] = pcm16[i] / 32768.0;
  }
  return float32;
}
```

Server Events

- `"session.updated"` - Acknowledge the client's `"session.update"` message that the session has been updated

JSON

```
{
    "event_id": "event_123",
    "type": "session.updated",
    "session": {
        "instructions": "You are a helpful assistant.",
        "voice": "Ara",
        "turn_detection": {
            "type": "server_vad"
        }
    }
}
```

Using Tools with Grok Voice API

The Grok Voice API supports various tools that can be configured in your session to enhance the capabilities of your voice agent. Tools can be configured in the `session.update` message.

Available Tool Types

- **Collections Search (`file_search`)** - Search through your uploaded document collections
- **Web Search (`web_search`)** - Search the web for current information
- **X Search (`x_search`)** - Search X (Twitter) for posts and information
- **Custom Functions** - Define your own function tools with JSON schemas

Configuring Tools in Session

Tools are configured in the `tools` array of the session configuration. Here are examples showing how to configure different tool types:

Collections Search with `file_search`

Use the `file_search` tool to enable your voice agent to search through document collections. You'll need to create a collection first using the Collections API.

Javascript

```
const COLLECTION_ID = "your-collection-id"; // Replace with your collection ID

const sessionConfig = {
    type: "session.update",
    session: {
        ...
        tools: [
            {
                type: "file_search",
                vector_store_ids: [COLLECTION_ID],
                max_num_results: 10,
            },
        ],
    },
};
```

Web Search and X Search

Configure web search and X search tools to give your voice agent access to current information from the web and X (Twitter).

Javascript

```
const sessionConfig = {
    type: "session.update",
    session: {
        ...
        tools: [
            {
                type: "web_search",
            },
```

```
            {
                type: "x_search",
                allowed_x_handles: ["elonmusk", "xai"],
            },
        ],
    },
};
```

Custom Function Tools

You can define custom function tools with JSON schemas to extend your voice agent's capabilities.

Javascript

```
const sessionConfig = {
    type: "session.update",
    session: {
        ...
        tools: [
            {
                type: "function",
                name: "generate_random_number",
                description: "Generate a random number between min and max values",
                parameters: {
                    type: "object",
                    properties: {
                        min: {
                            type: "number",
                            description: "Minimum value (inclusive)",
                        },
                        max: {
                            type: "number",
                            description: "Maximum value (inclusive)",
                        },
                    },
                    required: ["min", "max"],
                },
            },
        ],
    },
};
```

Combining Multiple Tools

You can combine multiple tool types in a single session configuration:

Javascript

```
const sessionConfig = {
    type: "session.update",
    session: {
        ...
        tools: [
            {
                type: "file_search",
                vector_store_ids: ["your-collection-id"],
                max_num_results: 10,
```

```
            },
            {
                type: "web_search",
            },
            {
                type: "x_search",
            },
            {
                type: "function",
                name: "generate_random_number",
                description: "Generate a random number",
                parameters: {
                    type: "object",
                    properties: {
                        min: { type: "number" },
                        max: { type: "number" },
                    },
                    required: ["min", "max"],
                },
            },
        ],
    },
};
```

For more details on Collections, see the Collections API documentation. For search tool parameters and options, see the Search Tools guide.

Handling Function Call Responses

When you define custom function tools, the voice agent will call these functions during conversation. You need to handle these function calls, execute them, and return the results to continue the conversation.

Function Call Flow

1. **Agent decides to call a function** → sends `response.function_call_arguments.done` event
2. **Your code executes the function** → processes the arguments and generates a result
3. **Send result back to agent** → sends `conversation.item.create` with the function output
4. **Request continuation** → sends `response.create` to let the agent continue

Complete Example

Javascript

```
// Define your function implementations
const functionHandlers = {
  get_weather: async (args) => {
    // In production, call a real weather API
    return {
      location: args.location,
      temperature: 22,
      units: args.units || "celsius",
      condition: "Sunny",
      humidity: 45
    };
```

```
  },

  book_appointment: async (args) => {
    // In production, interact with your booking system
    const confirmation = `CONF${Math.floor(Math.random() * 9000) + 1000}`;
    return {
      status: "confirmed",
      confirmation_code: confirmation,
      date: args.date,
      time: args.time,
      service: args.service
    };
  }
};

// Handle function calls from the voice agent
async function handleFunctionCall(ws, event) {
  const functionName = event.name;
  const callId = event.call_id;
  const args = JSON.parse(event.arguments);

  console.log(`Function called: ${functionName} with args:`, args);

  // Execute the function
  const handler = functionHandlers[functionName];
  if (handler) {
    const result = await handler(args);

    // Send result back to agent
    ws.send(JSON.stringify({
      type: "conversation.item.create",
      item: {
        type: "function_call_output",
        call_id: callId,
        output: JSON.stringify(result)
      }
    }));

    // Request agent to continue with the result
    ws.send(JSON.stringify({
      type: "response.create"
    }));
  } else {
    console.error(`Unknown function: ${functionName}`);
  }
}

// In your WebSocket message handler
ws.on("message", (message) => {
  const event = JSON.parse(message);

  // Listen for function calls
  if (event.type === "response.function_call_arguments.done") {
    handleFunctionCall(ws, event);
  } else if (event.type === "response.output_audio.delta") {
    // Handle audio response
  }
});
```

Function Call Events

Event	Direction	Description
`response.function_call_arguments.done`	Server → Client	Function call triggered with complete arguments
`conversation.item.create` (function_call_output)	Client → Server	Send function execution result back
`response.create`	Client → Server	Request agent to continue processing

Real-World Example: Weather Query

When a user asks "What's the weather in San Francisco?", here's the complete flow:

Step	Direction	Event	Description
1	→ Server	`input_audio_buffer.append`	User speaks: *"What's the weather in San Francisco?"*
2	← Client	`response.function_call_arguments.done`	Agent decides to call `get_weather` with `location: "San Francisco"`
3	→ Server	`conversation.item.create`	Your code executes `get_weather()` and sends result: `{temperature: 68, condition: "Sunny"}`
4	→ Server	`response.create`	Request agent to continue with function result
5	← Client	`response.output_audio.delta`	Agent responds: *"The weather in San Francisco is currently 68°F and sunny."*

Function calls happen automatically during conversation flow. The agent decides when to call functions based on the function descriptions and conversation context.

Conversation messages

Server Events

- `"conversation.created"` - The first message at connection. Notifies the client that a conversation session has been created

JSON

```
{
    "event_id": "event_9101",
    "type": "conversation.created",
    "conversation": {
        "id": "conv_001",
        "object": "realtime.conversation"
    }
}
```

Conversation item messages

Client

- `"conversation.item.create"` : Create a new user message with text.

JSON

```
{
    "type": "conversation.item.create",
    "previous_item_id": "", // Optional, used to insert turn into history
    "item": {
        "type": "message",
        "role": "user",
        "content": [
            {
                "type": "input_text",
                "text": "Hello, how are you?"
            }
        ]
    }
}
```

Server

- `"conversation.item.added"` : Responding to the client that a new user message has been added to conversation history, or if an assistance response has been added to conversation history.

JSON

```
{
  "event_id": "event_1920",
  "type": "conversation.item.added",
  "previous_item_id": "msg_002",
  "item": {
    "id": "msg_003",
    "object": "realtime.item",
    "type": "message",
    "status": "completed",
    "role": "user",
    "content": [
      {
        "type": "input_audio",
        "transcript": "hello how are you"
      }
    ]
  }
}
```

- `"conversation.item.input_audio_transcription.completed"` : Notify the client the audio transcription for input has been completed.

JSON

```
{
    "event_id": "event_2122",
    "type": "conversation.item.input_audio_transcription.completed",
    "item_id": "msg_003",
    "transcript": "Hello, how are you?"
}
```

Input audio buffer messages

Client

- `"input_audio_buffer.append"`: Append chunks of audio data to the buffer. The audio needs to be base64-encoded. The server does not send back corresponding message.

  JSON

  ```
  {
      "type": "input_audio_buffer.append",
      "audio": "<Base64EncodedAudioData>"
  }
  ```

- `"input_audio_buffer.clear"`: Clear input audio buffer. Server sends back `"input_audio_buffer.cleared"` message.

  JSON

  ```
  {
    "type": "input_audio_buffer.clear"
  }
  ```

- `"input_audio_buffer.commit"`: Create a new user message by committing the audio buffer created by previous `"input_audio_buffer.append"` messages. Confirmed by `"input_audio_buffer.committed"` from server.

  Only available when `"turn_detection"` setting in session is `"type": null`. Otherwise the conversation turn will be automatically committed by VAD on the server.

  JSON

  ```
  {
      "type": "input_audio_buffer.commit"
  }
  ```

Server

- `"input_audio_buffer.speech_started"`: Notify the client the server's VAD has detected the start of

a speech.

Only available when `"turn_detection"` setting in session is `"type": "server_vad"`.

JSON

```
{
  "event_id": "event_1516",
  "type": "input_audio_buffer.speech_started",
  "item_id": "msg_003"
}
```

- `"input_audio_buffer.speech_stopped"`: Notify the client the server's VAD has detected the end of a speech.

Only available when `"turn_detection"` setting in session is `"type": "server_vad"`.

JSON

```
{
  "event_id": "event_1516",
  "type": "input_audio_buffer.speech_stopped",
  "item_id": "msg_003"
}
```

- `"input_audio_buffer.cleared"`: Input audio buffer has been cleared.

JSON

```
{
  "event_id": "event_1516",
  "type": "input_audio_buffer.cleared"
}
```

- `"input_audio_buffer.committed"`: Input audio buffer has been committed.

JSON

```
{
  "event_id": "event_1121",
  "type": "input_audio_buffer.committed",
  "previous_item_id": "msg_001",
  "item_id": "msg_002"
}
```

Response messages

Client

- `"response.create"` : Request the server to create a new assistant response when using client side vad. (This is handled automatically when using server side vad.)

JSON

```
{
    "type": "response.create"
}
```

Server

- `"response.created"` : A new assistant response turn is in progress. Audio delta created from this assistant turn will have the same response id. Followed by `"response.output_item.added"` .

JSON

```
{
  "event_id": "event_2930",
  "type": "response.created",
  "response": {
    "id": "resp_001",
    "object": "realtime.response",
    "status": "in_progress",
    "output": []
  }
}
```

- `"response.output_item.added"` : A new assistant response is added to message history.

JSON

```
{
  "event_id": "event_3334",
  "type": "response.output_item.added",
  "response_id": "resp_001",
  "output_index": 0,
  "item": {
    "id": "msg_007",
    "object": "realtime.item",
    "type": "message",
    "status": "in_progress",
    "role": "assistant",
    "content": []
  }
}
```

- `"response.done"` : The assistant's response is completed. Sent after all the `"response.output_audio_transcript.done"` and `"response.output_audio.done"` messages. Ready for the client to add a new conversation item.

JSON

```
{
```

```
    "event_id": "event_3132",
    "type": "response.done",
    "response": {
        "id": "resp_001",
        "object": "realtime.response",
        "status": "completed",
    }
}
```

Response audio and transcription messages

Client

The client does not need to send messages to get these audio and transcription responses. They would be automatically created following `"response.create"` message.

Server

- `"response.output_audio_transcript.delta"`: Audio transcript delta of the assistant response.

  JSON

  ```
  {
    "event_id": "event_4950",
    "type": "response.output_audio_transcript.delta",
    "response_id": "resp_001",
    "item_id": "msg_008",
    "delta": "Text response..."
  }
  ```

- `"response.output_audio_transcript.done"`: The audio transcript delta of the assistant response has finished generating.

  JSON

  ```
  {
    "event_id": "event_5152",
    "type": "response.output_audio_transcript.done",
    "response_id": "resp_001",
    "item_id": "msg_008"
  }
  ```

- `"response.output_audio.delta"`: The audio stream delta of the assistant response.

  JSON

  ```
  {
    "event_id": "event_4950",
    "type": "response.output_audio.delta",
    "response_id": "resp_001",
    "item_id": "msg_008",
    "output_index": 0,
  ```

```
  "content_index": 0,
  "delta": "<Base64EncodedAudioDelta>"
}
```

- `"response.output_audio.done"` : Notifies client that the audio for this turn has finished generating.

JSON

```
{
    "event_id": "event_5152",
    "type": "response.output_audio.done",
    "response_id": "resp_001",
    "item_id": "msg_008",
}
```