REALTIME VOICE AI

CS 224G 2026

Shreeganesh Ramanan

GET THE REPO

Clone or fork the workshop repository:

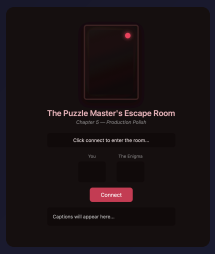
`https://github.com/sramanan/cs224g-2026-EscapeRoom`

- **Clone:** `git clone https://github.com/sramanan/cs224g-2026-EscapeRoom`
- **Fork:** Fork on GitHub, then clone your fork.

Add your OpenAI API key: `cp .env.example .env` and set `OPENAI_API_KEY`.

WELCOME TO THE ENIGMA ESCAPE ROOM

Workshop: Building and Scaling Realtime Voice AI Apps • 45 min



THE CHALLENGE: TALK YOUR WAY OUT

The Tech: Secure WebRTC integration with OpenAI's native audio model.

Today's Goal: Transition from REST text bots to stateful, voice-driven agents.

WHAT IS "NATIVE" REALTIME VOICE AI?

- **Multimodal Input/Output:** Model "hears" audio directly, preserving nuance, tone, and inflection.
- **Ultra-Low Latency:** Targeted responses under 300ms (feels like a real human conversation).
- **Interruptible (Barge-in):** If you speak while the AI is talking, it stops immediately.
- **Powered by WebRTC:** Uses UDP for direct, peer-to-peer audio streaming, avoiding the buffering lag of TCP WebSock

THE ARCHITECTURE BATTLE: OLD VS. NEW

Step	Legacy Chained Pipeline	Native Realtime Pipeline
Orchestration	STT API → Text LLM → TTS API	Microphone → OpenAI WebRTC
Typical Latency	1 - 3+ seconds	< 500ms
Conversational Feel	Robotic, "Walkie-Talkie" style	Natural flow, easy barge-in
Context	Text-only (loses tone)	Preserves emotion and accent

REALTIME PROS & CONS (PRAGMATIC VIEW)

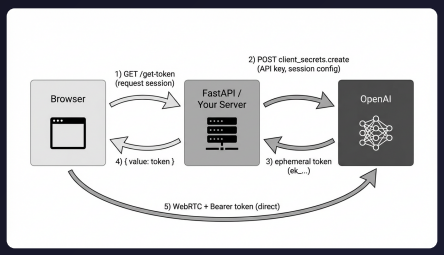
THE PROS: UX IS KING

- **Incredible UX:** Imperceptible latency, true conversational flow.
- **Simple Codebase:** Replaces orchestration logic with a single persistent connection.
- **Built-in Server VAD:** Handles silence detection on the server side.

THE CONS: PRODUCTION COSTS

- **Vendor Lock-in:** Limited to OpenAI's voices; cannot swap T providers.
- **Higher Cost:** Audio tokens are significant. A single convers can burn budget quickly.
- **Ephemeral State:** If the connection drops, short-term conversational context is instantly lost.

THE ARCHITECTURE OF TRUST (SESSION TOKENS)

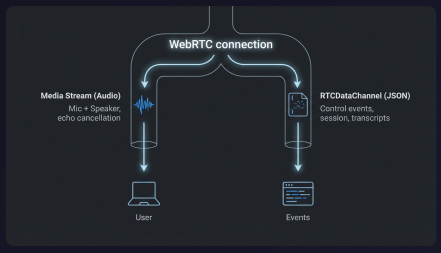


Security in Chapter 1:

1. **Frontend** requests permission to start a session from **Backend**.
2. **Backend** uses master API Key to request a short-lived **Ephemeral Token** from OpenAI via REST.
3. **Backend** sends Ephemeral Token to **Frontend**.
4. **Frontend** uses Ephemeral Token to establish direct **WebRTC Connection** to OpenAI.

Never expose your Master API Key in client-side code.

MODALITIES: TRACKS VS. CHANNELS



The WebRTC Connection is a single "pipe" with two parallel lanes:

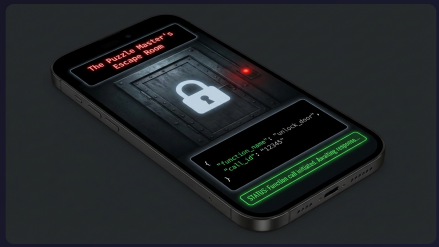
LANE 1: MEDIA STREAM (AUDIO)

- Handles microphone input and speaker output.
- WebRTC automatically manages Echo Cancellation, Noise Suppression, and Jitter.

LANE 2: RTCDATACHANNEL (JSON LOGIC)

- Used for control events.
- **Session is set on the backend** when creating the ephemeral token (Chapters 1–3: instructions, voice, `unlock_door` t
  The client never sends `session.update`.
- **OpenAI sends:** `response.output_audio_transcript.delta` (captions), `response.function_call_arguments.done` (tool execution).

FUNCTION CALLING / THE WIN STATE (CHAPTER 3)



How an LLM opens a digital door:

1. **Define Tool (backend):** When creating the token, the session includes the `unlock_door` tool (accepts `code`) and instructions. No client `session.update`.
2. **Interrogation:** Student talks to Puzzle Master across multiple turns; The Enigma reveals the code gradually.
3. **Tool Trigger:** Student says the code (random each session, e.g. "4829").
4. **Server Event:** LLM emits `response.function_call_arguments.done` with `name: "unlock_door"`.
5. **Client Execution (Ch 3):** React intercepts this on the data channel, verifies the code, sends tool result + `response.create`, and unlocks the door UI.

BACKEND WEBSOCKETS & SERVER-SIDE CONTROL (CHAPTERS 4–5)

The sideband pattern: Your server opens a second connection to the same Realtime session and can see every event, run and control the conversation.

- **Same session, two connections:** The browser keeps WebRTC for audio. After SDP exchange, the `Location` header giv `call_id`. The frontend sends that to your backend; your backend opens `wss://api.openai.com/v1/realtim` `call_id=...` with your API key.
- **Tool calling on the server:** When the model emits `response.function_call_arguments.done` (e.g. `unlock_door`), your backend receives it, validates the code, calls your DB or APIs, then sends the tool result and `response.create` back to OpenAI. The frontend only gets a simple `unlock_result` from your WebSocket.
- **Control the conversation:** Over the sideband you can inject instructions, update tools, or send `response.cancel`. Y can do data lookups (e.g. check inventory, verify a passcode in your DB) without exposing logic to the client.
- **Chapter 4:** Scaffold with a TODO — you implement the sideband. **Chapter 5:** Full implementation plus production polis (visualizers, captions, guessed codes).

Docs: Realtime server controls (sideband)

HANDS-ON WORKSHOP TIME

30 MINUTES

Follow the `# TODO` comments for Chapters 1, 2, and 3.

Raise your hand if you hit an issue.

SCALING CONCERNS (STATE & RECONNECTION)

STATEFULNESS IS THE BOTTLENECK

- Realtime sessions are ephemeral; context drops with the connection.
- **Production Requirement:** Backend must log every `conversation.item` (user and assistant messages).
- **Scaling Strategy:** Backend must provide a mechanism to restore conversation history to a *new* session upon reconne

MONITOR COST

- Audio tokens are significantly more expensive than text tokens.
- Always use `turn_detection` settings carefully. Set limits in OpenAI billing dashboard.

PRODUCT & UX POLISH CONSIDERATIONS

HANDLING "BARGE-IN" (INTERRUPTIBILITY)

- **With WebRTC (our app):** The server manages the output audio buffer. When VAD detects user speech, the server cancel in-progress response and *automatically truncates* unplayed audio. No client `conversation.item.truncate` nee
- **With WebSocket:** The client controls playback, so on `input_audio_buffer.speech_started` the client must st playback and send `conversation.item.truncate` (with `audio_end_ms`) so the model knows where it was cut

USER FEEDBACK (VISUALIZERS)

- Voice AI must "breathe." Static UIs increase anxiety.
- **Requirement:** Implement visualizers mapping local mic and remote AI audio tracks using the Web Audio API AnalyserN (see Chapter 5).

SCALABILITY METRICS THAT MATTER

Metric	Target	Significance
TTFT (Time to First Token/Audio)	< 300ms	Necessary for "human" conversational feel.
Connection Stability	99.9% (Server)	Backend bottleneck in token exchange.
Jitter Buffer	Minimized	Jitter causes audio stutter. Managed by WebRTC browser API.
VAD Accuracy	High	Low false interruptions. Low background noise trigger.

Tune VAD: Set `turn_detection` (e.g. `semantic_vad` or `server_vad`) in the session when your backend creates the

FINAL AMA & CLOSE

RECAP

1. **WebRTC** is the protocol for native low-latency audio.
2. **Architecture of Trust** — backend creates the token (with session: persona, voice, tools); frontend never sees the API
3. **Data Channel** carries events (captions, `function_call_arguments.done`); **Media Tracks** carry audio.
4. **Backend WebSocket (sideband)** lets the server join the same call to run tools, do data lookups, and control the conversation (Chapters 4–5).
5. **Scaling** means managing state and reconnection history.