A Guide to Engineering a Low-Latency Voice AI Application for the smallest.ai Interview

Introduction: Deconstructing the "Cracked" Full-Stack Voice AI Challenge

This report provides a comprehensive, step-by-step guide to architecting and implementing a high-performance, real-time voice AI demonstration application. The project is strategically designed to serve as a principal portfolio piece for a full-stack engineering candidate interviewing at smallest.ai. It moves beyond a conventional coding exercise to become a direct response to the company's publicly stated hiring mandate, which prioritizes demonstrable, elite skill over traditional credentials.[1] The objective is to construct a system that not only functions flawlessly but also showcases a deep, nuanced understanding of the core principles governing conversational AI—specifically, the management of latency and the orchestration of complex, asynchronous data flows.

The smallest.ai Mandate: Skills Over Credentials

smallest.ai has cultivated a reputation for its unconventional hiring process, famously seeking "cracked" engineers by evaluating their work and tangible skills rather than resumes or academic pedigrees.[1] This philosophy necessitates an application that is more than a simple proof-of-concept; it must be a testament to architectural foresight, engineering rigor, and product awareness. This guide details the creation of such a project, designed from the ground up to be the candidate's "best work."

Simulating a Foundational Model Company

A key differentiator for smallest.ai is its commitment to building proprietary, foundational models from scratch, such as its "Electron" language model and "Lightning" text-to-speech (TTS) engine.[3] While building bespoke models is beyond the scope of a demonstration project, a sophisticated understanding of their purpose can be shown through intelligent simulation. This guide adopts a deliberate strategy of integrating best-in-class third-party APIs to proxy

smallest.ai's in-house stack. The selection of these APIs is not arbitrary; each is chosen to mirror the performance characteristics—ultra-low latency and high fidelity—that smallest.ai itself pursues. This approach demonstrates an understanding of the entire AI pipeline and the critical metrics that define success in the voice domain.

High-Level System Architecture

The system architecture is designed for a low-latency, client-server-client interaction pattern. The data flow is as follows:

1. **Client (React/TypeScript on Vercel):** The user's browser captures microphone audio.
2. **Transport (WebSockets):** Raw audio data is streamed in real-time to the backend over a persistent WebSocket connection.
3. **Backend (Python/FastAPI on Modal):** A serverless function receives the audio stream and orchestrates the AI pipeline.
4. **AI Services (Third-Party APIs):** The backend sequentially streams data through Speech-to-Text (STT), Large Language Model (LLM), and Text-to-Speech (TTS) services.
5. **Return Transport (WebSockets):** The generated audio from the TTS service is streamed back to the client over the same WebSocket connection.
6. **Client Playback:** The browser receives the audio stream and plays it back to the user seamlessly.

This architecture is a simplified yet powerful analogue of a Selective Forwarding Unit (SFU)-based system, where a central server manages media streams, which is a core concept in real-time communication.[6]

Core Technology Justification

The technology stack is selected to align with modern best practices for performance, scalability, and developer experience, reflecting the skills sought in a top-tier full-stack engineer [2]:

- **Frontend (React/TypeScript on Vercel):** React, augmented with TypeScript, provides a robust framework for building a dynamic and type-safe user interface. Vercel offers a seamless, Git-native deployment workflow with optimized performance and effortless environment management.[7]
- **Backend (Python/Modal):** Python is the lingua franca of AI and machine learning. Modal provides a serverless platform that eliminates infrastructure management, allowing for the deployment of scalable, containerized Python applications with simple function decorators. Its native support for asynchronous frameworks like FastAPI and WebSockets makes it ideal for this real-time use case.[9]
- **Transport (WebSockets):** For real-time, bidirectional communication, WebSockets are superior to traditional HTTP. They establish a persistent, full-duplex connection, minimizing the overhead and latency associated with repeated HTTP handshakes, which is critical for streaming audio data.[11]

A crucial element of this project is its alignment with the deeper product philosophy of smallest.ai. Founder Sudarshan Kamath has expressed that the true challenge in conversational AI is not a race to Artificial General Intelligence (AGI) but rather "nailing memory and context" to create an experience that *feels* intelligent.[3] A standard, stateless voice demo would miss this critical nuance. Therefore, this guide incorporates a simple conversational memory mechanism in the backend. By appending a summary of the previous turn to the LLM prompt, the application demonstrates an awareness of this core product challenge, transforming the demo from a purely technical exercise into a thoughtful product prototype. This strategic addition provides a powerful talking point for an interview, signaling that the candidate has researched the company's vision, not just its job description.

The following table summarizes the deliberate selection of third-party APIs chosen to simulate smallest.ai's proprietary stack.

Service	Provider	Selected API/Model	Key Features & Rationale
Speech-to-Text (STT)	Deepgram	Nova-3 Streaming API	**Proxy for smallest.ai's custom STT.** Chosen for its industry-leading low latency, real-time interim_results, and robust endpointing features, which are critical for natural turn-taking in conversation.[13]
Large Language Model (LLM)	Groq	Llama 3 via Groq API	**Proxy for smallest.ai's "Electron" model.** Chosen for its unparalleled inference speed (low time-to-first-token), which is the biggest potential latency bottleneck. Its OpenAI-compatible API simplifies integration.[15]
Text-to-Speech (TTS)	ElevenLabs	Streaming TTS API	**Proxy for smallest.ai's "Lightning" model.** Chosen for its high-quality, natural-sounding voices and its support for low-latency audio streaming via WebSockets or chunked HTTP, allowing playback to begin before the full audio is generated.[17]

Section 1: The Architectural Blueprint for Real-Time Voice

This section establishes the theoretical foundation for the application, defining the critical constraints and justifying the core architectural patterns. A successful real-time system is not built by accident; it is engineered from first principles.

1.1. Latency: The Defining Metric of Conversational AI

In the context of voice communication, latency is the time delay between when a sound is produced and when it is perceived by the listener.[19] Excessive latency is the primary cause of poor quality in voice applications, leading to unnatural pauses and conversational breakdowns. Human interaction is highly sensitive to delay; most listeners will perceive latency above 100-120 milliseconds, and conversations become difficult around 250-300 ms.[20] The International Telecommunication Union (ITU) standard ITU-T G.114 recommends a one-way latency of under 150 ms for most applications and sets an absolute maximum of 400 ms for acceptable quality.[19]

To engineer a system that meets these stringent requirements, it is essential to establish a "latency budget." This involves breaking down the end-to-end (or "glass-to-glass") journey of the audio signal and allocating a maximum permissible delay to each stage of the pipeline. This quantitative approach forces a rigorous, engineering-driven mindset, transforming qualitative goals ("low latency") into measurable targets. It provides a framework for identifying critical bottlenecks and making informed trade-offs—for instance, deciding whether a marginal improvement in TTS voice quality is worth an additional 50 ms of delay. Presenting such an analysis demonstrates a mature, systematic approach to performance engineering that is highly valued in elite engineering teams.

The table below outlines an estimated latency budget for our voice AI pipeline. The goal is to consistently achieve a total latency at the lower end of this range.

Pipeline Stage	**Component**	**Estimated Latency (ms)**	**Notes & Justification**

1. User Input	Audio Capture (Client)	< 20	Using AudioWorklet for direct buffer access minimizes client-side processing delay.
2. Upstream	Network to Backend	20 - 50	Dependent on user's connection. Assumes a decent broadband connection.
3. Processing	STT (Deepgram)	50 - 100	Time to receive the first stable interim_result.
4. Processing	LLM (Groq)	50 - 150	Time-to-first-token (TTFT). This is Groq's main advantage.
5. Processing	TTS (ElevenLabs)	80 - 200	Time-to-first-byte (TTFB) for the first audio chunk. Critical latency point.
6. Downstream	Network to Client	20 - 50	Dependent on user's connection.
7. User Output	Client-side Buffering & Playback	20 - 50	Small buffer to prevent jitter, but kept minimal to reduce latency.
Total (Glass-to-Glass)		**260 - 620**	**Targeting the lower end of this range (<300ms) for a "real-time" feel.**

1.2. Choosing the Right Transport: A Deliberate Simulation of an SFU with WebSockets

Real-time communication (RTC) systems are typically built on one of two architectural models: Peer-to-Peer (P2P) or server-based.[6] In a P2P model, clients connect directly to each other. In a server-based model, clients connect to a central server that manages and routes the media streams. The industry standard for scalable, multi-party video and audio calls is the Selective Forwarding Unit (SFU), a type of media server that receives media streams from each participant and forwards them to the others.[6] The SFU architecture reduces the load on each client, as they only need to maintain one upstream connection to the server, and provides a point of centralized control.

For this project, a full WebRTC implementation with P2P capabilities is unnecessary overhead. WebRTC is a complex framework designed for direct browser-to-browser communication, requiring a "signaling" mechanism (often using WebSockets) to coordinate connections.[22] Our use case is different: we need to stream audio from a single client to a central processing pipeline and stream the resulting audio back.

Therefore, the choice of WebSockets as the primary transport layer is a deliberate and intelligent simplification. A WebSocket server acts as a perfect proxy for the role an SFU would play in this specific one-to-one, client-server-client data flow. It provides a persistent, low-latency, bidirectional channel for streaming media, embodying the principle of centralized routing without the implementation complexity of WebRTC signaling protocols like ICE, STUN, and TURN. This decision demonstrates a clear understanding of RTC architectural principles and the ability to choose the right tool for the job.

1.3. The Audio Pipeline: Codecs, Formats, and Data Flow

The efficiency of the audio pipeline depends on the careful selection of codecs and data formats.

- **Audio Codec:** The Opus codec is the de facto standard for interactive speech and audio over the internet.[24] Standardized by the IETF as RFC 6716, it is designed to be highly versatile, scaling from low-bitrate speech to high-quality music while maintaining very low algorithmic delay (typically 26.5 ms).[24] Its excellent packet loss concealment (PLC) capabilities make it robust for transmission over unreliable networks.[25] All WebRTC-compliant browsers are required to support Opus, making it a universally compatible choice.[27]

- **Audio Data Format:** A common approach for capturing browser audio is the MediaRecorder API. However, for our high-performance use case, this API has a significant drawback: it is designed to create self-contained media *files*. The dataavailable event provides chunks in a container format (e.g., WebM with an Opus audio track).[28] These chunks, apart from the first one containing the header, are not independently playable and require complex parsing on the receiving end. This introduces unnecessary latency and processing overhead.
  A more direct and advanced approach is to intercept the raw audio data before it is encoded. By using the Web Audio API's AudioWorklet, we can gain direct access to the raw Pulse-Code Modulation (PCM) data from the microphone. PCM represents the audio waveform as a sequence of numerical samples (e.g., 16-bit signed integers or 32-bit floats). This raw, uncompressed data can be streamed directly to our backend, where STT services like Deepgram can consume it with minimal processing. This method provides lower latency, greater control, and simplifies the backend pipeline.
- **End-to-End Data Flow:** The journey of a single utterance is as follows:
  1. The user's microphone captures an analog audio signal.
  2. The browser's AudioWorklet digitizes this into raw PCM data chunks.
  3. Each chunk is sent as a binary message over the WebSocket connection.
  4. The Python backend receives the binary data and forwards it to the Deepgram streaming STT service.
  5. Deepgram returns text transcripts (interim and final) in real-time.
  6. The backend forwards the final transcript to the Groq LLM service.
  7. Groq streams back its text response.
  8. The backend streams this text response to the ElevenLabs TTS service.
  9. ElevenLabs streams back audio data (e.g., MP3 or raw PCM bytes).
  10. The backend forwards these audio bytes to the client over the WebSocket.
  11. The client's Web Audio API receives the audio bytes, queues them, and plays them back seamlessly, creating a continuous conversational experience.

Section 2: Backend Implementation on Modal: The Python-Powered Core

The backend is the heart of the voice AI system, responsible for orchestrating the complex, real-time flow of data between the client and the various AI services.

Building this on Modal with Python and FastAPI allows for a focus on application logic while leveraging a powerful, scalable, and serverless infrastructure.

2.1. Setting Up the Modal Environment: Images and Secrets

A robust application begins with a reproducible and secure environment. Modal excels at this through its concepts of Images and Secrets.

- **Modal Project Structure:** A Modal application is defined within a Python script, typically centered around a modal.App object. This object serves as a container for all the functions, images, and secrets that constitute the application.
- **Container Image:** To ensure all dependencies are available in the remote execution environment, we define a modal.Image. This object programmatically constructs a Docker image. By using methods like .pip_install(), we can specify all necessary Python packages, such as fastapi, websockets, deepgram-sdk, groq, and elevenlabs.[9] This declarative approach guarantees that every container running the function is identical, eliminating "it works on my machine" issues.
- **Secrets Management:** Hardcoding API keys and other credentials into source code is a major security vulnerability. Modal provides a secure Secret management system.[29] Secrets can be created via the Modal web dashboard or the command-line interface (e.g.,
  modal secret create my-api-secrets DEEPGRAM_API_KEY=...
  GROQ_API_KEY=...).[30] In our application code, we securely inject these credentials into the function's environment by passing a
  secrets argument, like secrets=.[31] The function can then access these credentials as standard environment variables (e.g.,
  os.environ), without the keys ever being exposed in the source code.

2.2. The WebSocket Ingress: A FastAPI-based Endpoint

Modal provides first-class support for ASGI (Asynchronous Server Gateway Interface) applications, which is the standard for modern asynchronous Python web frameworks like FastAPI.[10] This allows us to build a WebSocket server with minimal boilerplate.

The implementation involves decorating a Python function with @modal.asgi_app().

Inside this function, we instantiate a standard FastAPI application. The core of the server is a WebSocket endpoint defined with the @app.websocket("/ws") decorator.[33] The handler function for this endpoint, defined as

async def websocket_handler(websocket: WebSocket):, will manage the entire lifecycle of a single client connection. The first action within this handler must be await websocket.accept(), which completes the WebSocket handshake and establishes the persistent connection.[33] This endpoint serves as the single entry and exit point for all real-time data flowing between the client and our AI pipeline.

2.3. The Asynchronous AI Pipeline: Chaining Streaming Services with asyncio.Queue

A naive implementation of the AI pipeline would involve a simple, sequential chain of await calls: wait for STT to finish, then call the LLM, wait for it to finish, then call TTS. This approach is inefficient and introduces significant "dead air," completely failing to meet our latency budget. The key to a truly real-time experience is to have all components of the pipeline—STT, LLM, and TTS—operating concurrently. As soon as the first word is transcribed, the LLM should begin processing it. As soon as the LLM generates its first token of response, the TTS engine should begin synthesizing the audio.

This creates a classic producer-consumer problem. The STT service *produces* text chunks. The LLM *consumes* text and *produces* response chunks. The TTS service *consumes* response chunks and *produces* audio chunks. The ideal tool for connecting these asynchronous components in Python is asyncio.Queue.[34] This data structure provides a safe way to pass data between concurrently running coroutines, effectively decoupling the services. Each service can operate at its own pace, pulling data from an upstream queue and pushing results to a downstream queue. This architecture is not only highly performant but also more robust and scalable. It is a hallmark of advanced asynchronous system design.

The implementation proceeds as follows:

1. **Initialize Queues:** Within the main WebSocket handler, three asyncio.Queue instances are created: stt_to_llm_queue, llm_to_tts_queue, and tts_to_client_queue.
2. **Define and Gather Tasks:** Several async functions, each representing a stage in

the pipeline, are defined. These are then launched as concurrent tasks using asyncio.gather().
3. **Audio Receiver & STT Producer:** This coroutine runs in a loop, receiving audio from the client via await websocket.receive_bytes(). It streams this audio to the Deepgram service. When Deepgram returns a transcript (especially an is_final: true transcript), the task places it into the stt_to_llm_queue using await stt_to_llm_queue.put(transcript).[14]
4. **LLM Handler (Consumer/Producer):** This task waits for text to appear in the stt_to_llm_queue with await stt_to_llm_queue.get(). It then sends this text to the Groq API in streaming mode. As response tokens arrive from the LLM, it places them into the llm_to_tts_queue.[16] This task also manages the conversational memory, prepending the history to the prompt sent to Groq.
5. **TTS Handler (Consumer/Producer):** This task waits for text chunks from the LLM in the llm_to_tts_queue. It aggregates these chunks into sentences or phrases and streams them to the ElevenLabs TTS API. As the corresponding audio data is returned from ElevenLabs, it places the audio bytes into the final tts_to_client_queue.[18]
6. **Client-Side Sender (Consumer):** A final, simple coroutine waits for audio data in the tts_to_client_queue. Upon receiving a chunk, it sends it back to the client over the WebSocket using await websocket.send_bytes(audio_chunk).

2.4. Engineering for Conversation: Handling Interruptions and Turn-Taking

A key feature of natural human conversation is the ability to interrupt, or "barge-in." A rigid, sequential pipeline makes this difficult to implement. The decoupled, queue-based architecture, however, makes it remarkably simple.

The logic resides within the Audio Receiver task. When it receives a new audio chunk from the client, it can check a shared state variable, is_ai_speaking. If this flag is true, it signals that the user is speaking over the AI's response—an interruption. To handle this, the task simply needs to clear the downstream queues (llm_to_tts_queue and tts_to_client_queue). This action immediately purges any pending text for the LLM and any synthesized audio waiting to be sent to the client. The AI's response is instantly cut off, and the new user utterance is processed, beginning a fresh conversational turn. This elegant solution demonstrates a practical understanding of conversational agent design that goes far beyond a basic API integration.

Section 3: Frontend Implementation with React & TypeScript: The User Interface

The client-side application is the user's sole point of interaction with the system. Its implementation must be clean, performant, and robust, particularly in its handling of real-time audio input and output, which are notoriously complex in a browser environment.

3.1. Project Setup and Vercel Configuration

The project begins with a standard setup using Vite or create-react-app with the TypeScript template. This provides a modern build pipeline and the benefits of static typing.

Deployment to Vercel requires managing environment variables correctly. The WebSocket URL of the deployed Modal backend must be available to the React application. This is configured in the Vercel project settings under "Environment Variables".[7] A variable, for example

REACT_APP_WEBSOCKET_URL, is created for the Production and Preview environments. For local development, this variable is stored in a .env.local file at the project root. This file can be created manually or populated automatically by linking the local project to Vercel (vercel link) and running vercel env pull.[8]

3.2. The useVoiceAgent Custom Hook: Encapsulating Real-Time Logic

To adhere to modern React best practices, all complex, stateful logic related to real-time communication should be abstracted away from the UI components. A custom hook, useVoiceAgent, is the ideal pattern for this encapsulation.[39]

This hook will be responsible for:

- Establishing and managing the WebSocket connection.
- Handling microphone access and audio capture.
- Managing the audio playback queue.
- Maintaining the application's state, such as connectionStatus ('connecting', 'open', 'closed'), isRecording, isAIResponding, and transcript.
- Exposing a simple API to the UI components, like startConversation() and stopConversation() methods.

By containing this logic within a single hook, UI components remain clean, declarative, and focused solely on presentation.

3.3. Capturing and Streaming Microphone Audio with AudioWorklet

The method of audio capture is a critical decision that significantly impacts latency and performance. While many tutorials and examples utilize the MediaRecorder API [28], a more sophisticated and performant solution is the

AudioWorklet.

The MediaRecorder API is designed for recording media to a file, not for low-latency streaming. It provides data in containerized chunks (e.g., WebM) at variable intervals, which complicates real-time processing. The modern alternative, AudioWorklet, provides direct access to the raw audio stream in a separate, high-priority thread, preventing any interference with the main UI thread and avoiding the performance issues of its deprecated predecessor, ScriptProcessorNode.[42]

Inside an AudioWorkletProcessor, the process method is invoked by the browser's audio engine at regular, predictable intervals. This method receives the raw PCM audio data from the microphone as a Float32Array. This gives us precise control. We can take this raw data, downsample it to the 16kHz sample rate preferred by most STT APIs, convert it to 16-bit integers, and send the resulting ArrayBuffer back to the main thread using this.port.postMessage(). The main thread then forwards this clean, raw binary data over the WebSocket. This approach is a clear differentiator, showcasing a deep understanding of the Web Audio API and a commitment to performance that goes beyond standard implementations.

The implementation steps are:

1. Define the AudioWorkletProcessor in a separate JavaScript file (e.g., audio-processor.js).
2. In the useVoiceAgent hook, load this processor into the AudioContext using audioContext.audioWorklet.addModule('audio-processor.js').
3. Request microphone access using navigator.mediaDevices.getUserMedia().
4. Create an AudioWorkletNode from the loaded processor.
5. Connect the microphone's MediaStreamSource to the AudioWorkletNode.
6. Listen for messages from the worklet's port. When a raw audio chunk arrives, send it through the WebSocket.

3.4. Receiving and Playing Back Streaming Audio with the Web Audio API

Playing back a stream of discrete audio chunks from the server presents a significant challenge. A naive approach, such as creating a new <audio> element for each chunk, will result in audible pops, clicks, and timing glitches between chunks.[43] A robust solution requires a queued playback system managed by the Web Audio API.

This system ensures that each audio chunk plays precisely after the previous one has finished, creating a single, seamless audio stream for the user.

1. **State Management:** The useVoiceAgent hook maintains an audioQueue (an array of ArrayBuffer chunks) and an isPlaying boolean flag.
2. **Receiving Chunks:** The WebSocket's onmessage event handler pushes each incoming audio chunk into the audioQueue and triggers a playback function.
3. **The Playback Loop:** A playNextInQueue function orchestrates the playback. It first checks if !isPlaying and if the audioQueue is not empty. If both are true, it proceeds.
4. **Web Audio API Execution:**
   - It dequeues the next chunk: const chunk = audioQueue.shift().
   - It decodes the chunk into an AudioBuffer using audioContext.decodeAudioData(chunk). This asynchronous method handles various formats, like MP3, that might be sent from the TTS service.[42]
   - It creates an AudioBufferSourceNode, assigns the decoded buffer to it (source.buffer = audioBuffer), and connects it to the audio context's destination (the speakers).
   - It sets isPlaying = true and starts playback immediately with source.start(0).
   - **The critical step:** An onended event handler is attached to the

AudioBufferSourceNode. This event fires the moment the chunk finishes playing. The handler sets isPlaying = false and, crucially, calls playNextInQueue() again. This creates a self-perpetuating, event-driven loop that chains the audio chunks together with sample-accurate timing, a technique that is the hallmark of an experienced web audio developer.[44]

3.5. Building the UI Component

The final step is to create a simple React component that utilizes the useVoiceAgent hook. The component will be minimal, containing:

- A single button to toggle the conversation, calling the startConversation() and stopConversation() methods exposed by the hook.
- A status indicator to display the connectionStatus from the hook's state.
- A text area to display the live transcript as it is updated by the hook.

This separation of concerns ensures the UI is simple and declarative, while the complex real-time logic is neatly managed within the custom hook.

Section 4: Deployment and Interview Preparation

A well-engineered project is only half the battle; its value is fully realized when it is deployed and can be articulated effectively. This section covers the final steps of deployment and provides a strategic guide for discussing the project in a high-stakes interview.

4.1. Deploying the Backend to Modal

Deploying the Python backend to Modal is a remarkably streamlined process. Once the application script (e.g., main.py) is complete and the necessary secrets are configured in the Modal dashboard, deployment is a single command:

```
modal deploy main.py
```

Modal handles the containerization, provisioning, and deployment, providing a persistent, public WebSocket URL (wss://...) in the command-line output. This URL is the endpoint that the frontend application will connect to.

4.2. Deploying the Frontend to Vercel

Deploying the React frontend to Vercel is equally straightforward:

1. Push the project's source code to a GitHub, GitLab, or Bitbucket repository.
2. In the Vercel dashboard, create a new project and link it to the repository.
3. Vercel will typically auto-detect the framework (Vite or Create React App) and configure the build settings appropriately.
4. In the project's settings, navigate to the "Environment Variables" section. Add the REACT_APP_WEBSOCKET_URL variable, pasting the wss:// URL obtained from the Modal deployment.[37]
5. A push to the main branch will automatically trigger a production deployment.

The following table serves as a consolidated checklist for all required environment variables and secrets across both platforms.

Variable Name	**Service**	**Environment(s)**	**Description & Example Value**
DEEPGRAM_API_KEY	Modal	Secret Store	Your API key from the Deepgram console.
GROQ_API_KEY	Modal	Secret Store	Your API key from the Groq console.
ELEVENLABS_API_KEY	Modal	Secret Store	Your API key from the ElevenLabs console.
REACT_APP_WEBSOCKET_URL	Vercel	Production, Preview, Development	The wss:// URL of your deployed Modal app.

4.3. Talking Points for the Interview: Articulating Your Design Decisions

Building the application is not enough. A "cracked" engineer must be able to articulate the *why* behind their technical decisions. This project is designed to generate numerous opportunities to showcase deep technical reasoning. The following are likely questions and strong, well-supported answers.

- **Question: "Why did you choose WebSockets instead of a full WebRTC implementation?"**
  - **Answer:** "While WebRTC is the standard for multi-party, peer-to-peer communication, this application's core requirement is to stream audio from a single client to a centralized processing pipeline and back. For this client-server-client topology, WebRTC's P2P connection setup and signaling complexity would be unnecessary overhead. I chose WebSockets because they provide the ideal transport for this use case: a persistent, low-latency, bidirectional channel. This approach effectively simulates the role of a Selective Forwarding Unit (SFU) in routing media to a central point, demonstrating an understanding of core RTC architectural principles while selecting the most direct and efficient tool for the specific problem."
- **Question: "How did you approach minimizing latency in the system?"**
  - **Answer:** "My approach was systematic, starting with a 'latency budget' that allocated a maximum delay to each stage of the pipeline—from audio capture to STT, LLM, TTS, and final playback. This quantitative framework guided all my technology choices. The most significant potential bottleneck is LLM inference, which is why I selected the Groq API for its state-of-the-art time-to-first-token. Every other component was also chosen for its streaming capabilities: Deepgram for streaming STT, and ElevenLabs for streaming TTS. The entire backend is architected asynchronously to ensure these services run concurrently, not sequentially, so that processing at each stage begins the moment data is available from the previous one."
- **Question: "Your frontend uses an AudioWorklet. Why not the more common MediaRecorder API?"**
  - **Answer:** "I made a deliberate choice to use an AudioWorklet for performance and control. The MediaRecorder API is designed to create media files, so it provides data in containerized formats like WebM at unpredictable intervals. This requires parsing on the backend and introduces latency. The AudioWorklet, on the other hand, operates in a separate, high-priority audio thread, giving me direct, low-level access to the raw PCM audio data from the

microphone at predictable intervals. This allows me to stream clean, raw audio directly to the STT service, minimizing both client-side and server-side processing and reducing overall latency."

- **Question: "Walk me through your asynchronous backend architecture."**
  - **Answer:** "The backend is designed as a set of decoupled producer-consumer services connected by asyncio.Queues. This avoids a slow, sequential process. There are three main concurrent tasks: an STT task that consumes audio from the client and produces text; an LLM task that consumes text and produces a response; and a TTS task that consumes the response and produces audio. Each task communicates with the next via a dedicated queue. This architecture allows the entire pipeline to operate in parallel. For example, the LLM can start generating a response to the beginning of a sentence while the STT is still transcribing the end of it. This design also makes advanced features like interruption handling trivial to implement by simply clearing the downstream queues when new user audio is detected."
- **Question: "What are the limitations of your demo, and how would you evolve it?"**
  - **Answer:** "The primary limitation is its reliance on third-party APIs, which abstracts away the core modeling challenges. This was a strategic choice for a demo, but to evolve this into a production system in the spirit of smallest.ai, the next step would be to replace these external calls with proprietary models. This would provide full control over the performance, accuracy, and data privacy of the entire stack. Furthermore, I would enhance the conversational capabilities. The current implementation has basic interruption handling and a simple turn-based memory. I would evolve this into a more sophisticated state machine to manage conversational context more robustly, directly addressing the company's stated focus on 'nailing memory and context' to create truly intelligent-feeling agents.[3]"

Conclusion: Beyond the Demo - Pathways to a Production-Grade System

This report has detailed the design and construction of a high-performance, full-stack voice AI application. The resulting project serves as a powerful demonstration of engineering capability, showcasing expertise in modern frontend and backend technologies, a deep understanding of real-time communication principles, and an awareness of the nuanced challenges in conversational AI.

The architecture, while complete for its purpose, is also a strong foundation for further evolution into a production-grade system. Several pathways for enhancement exist:

- **Transition to a Full SFU Architecture:** For applications requiring true multi-party conversations, the WebSocket-based transport could be evolved into a full WebRTC implementation. The WebSocket would be retained for its ideal role in signaling, while media transport would be handled by WebRTC data channels, all managed by a central SFU.
- **Advanced Conversational State Management:** The simple interruption and memory mechanisms can be expanded into a sophisticated conversational management system. This would involve implementing a more robust state machine to track conversational context, manage dialogue turns more gracefully, and retain long-term memory, aligning closely with the philosophical goals articulated by smallest.ai's leadership.[3]
- **Custom Foundational Model Integration:** The ultimate evolution of this project, and one that directly mirrors the core mission of smallest.ai, would be the systematic replacement of third-party APIs with custom-trained, proprietary models. Integrating in-house STT, LLM, and TTS models would grant complete control over the end-to-end pipeline, enabling fine-grained performance tuning, cost optimization at scale, and the development of unique, defensible intellectual property—the very essence of a foundational model company.

By building this project and internalizing the design decisions behind it, a candidate is well-equipped to not only demonstrate their technical prowess but also to engage in a deep, substantive conversation about the future of voice AI.

Works cited

1. Indian-origin founder ditches traditional hiring as ₹40 LPA Bengaluru techie job goes viral: 'This is how we hire' | Trending - Hindustan Times, accessed July 21, 2025, https://www.hindustantimes.com/trending/indianorigin-founder-ditches-traditional-hiring-as-rs-40-lpa-bengaluru-techie-job-goes-viral-this-is-how-we-hire-101740726908685.html
2. 'No CV, No College': This AI startup founder's social media post goes viral, offering Rs 1 crore job | Zee Business, accessed July 21, 2025, https://www.zeebiz.com/startups/news-sudarshan-kamath-founder-of-artificial-intelligence-startup-smallest-ai-post-goes-viral-on-no-cv-no-college-rs-1-crore

-job-372848
3. Why Smallest.ai is Contesting the Idea of 'LLMs' - Analytics India Magazine, accessed July 21, 2025, https://analyticsindiamag.com/ai-features/why-smallest-ai-is-contesting-the-idea-of-llms/
4. Smallest.ai: AI Agents for Enterprise Contact Centers, accessed July 21, 2025, https://smallest.ai/
5. Sudarshan Kamath - WAN-IFRA, accessed July 21, 2025, https://wan-ifra.org/event_speakers/sudarshan-kamath/
6. A short introduction to the architecture of a video call - Daily | API documentation, accessed July 21, 2025, https://docs.daily.co/guides/architecture-and-monitoring/intro-to-video-arch
7. Environment variables - Vercel, accessed July 21, 2025, https://vercel.com/docs/environment-variables
8. Environments - Vercel, accessed July 21, 2025, https://vercel.com/docs/deployments/environments
9. Web endpoints | Modal Docs, accessed July 21, 2025, https://modal.com/docs/guide/webhooks
10. Introducing: WebSockets on Modal | Modal Blog, accessed July 21, 2025, https://modal.com/blog/websocket-launch
11. WebRTC VS WebSocket: A Comparison - Digital Samba, accessed July 21, 2025, https://www.digitalsamba.com/blog/webrtc-vs-websocket
12. WebRTC vs WebSocket: Ideal Protocol for Real-Time Communication - VideoSDK, accessed July 21, 2025, https://www.videosdk.live/blog/webrtc-vs-websocket
13. Live Audio - Deepgram's Docs, accessed July 21, 2025, https://developers.deepgram.com/reference/speech-to-text-api/listen-streaming
14. Getting Started | Deepgram's Docs, accessed July 21, 2025, https://developers.deepgram.com/docs/live-streaming-audio
15. GroqDocs - Build Fast - Groq Cloud, accessed July 21, 2025, https://console.groq.com/docs
16. Text Generation - GroqDocs - Groq Cloud, accessed July 21, 2025, https://console.groq.com/docs/text-chat
17. Text to Speech (product guide) | ElevenLabs Documentation, accessed July 21, 2025, https://elevenlabs.io/docs/product-guides/playground/text-to-speech
18. Streaming | ElevenLabs Documentation, accessed July 21, 2025, https://elevenlabs.io/docs/api-reference/streaming
19. What is Latency? | Twilio, accessed July 21, 2025, https://www.twilio.com/docs/glossary/what-is-latency
20. The ultimate Voice AI Stack - My Framer Site - Coval, accessed July 21, 2025, https://www.coval.dev/blog/the-ultimate-voice-ai-stack
21. Real-time communication - Wikipedia, accessed July 21, 2025, https://en.wikipedia.org/wiki/Real-time_communication
22. WebRTC vs. WebSocket: Key differences and which to use - Ably, accessed July 21, 2025, https://ably.com/topic/webrtc-vs-websocket
23. WebRTC vs WebSocket: Key Differences and which to use to enhance Real-Time

Communication? - Dyte, accessed July 21, 2025, https://dyte.io/blog/webrtc-vs-websocket/
24. en.wikipedia.org, accessed July 21, 2025, https://en.wikipedia.org/wiki/Opus_(audio_format)
25. Opus Codec, accessed July 21, 2025, https://opus-codec.org/
26. xiph/opus: Modern audio compression for the internet. - GitHub, accessed July 21, 2025, https://github.com/xiph/opus
27. Fundamental components of RTC architecture - Real-Time Communication on AWS, accessed July 21, 2025, https://docs.aws.amazon.com/whitepapers/latest/real-time-communication-on-aws/fundamental-components-of-rtc-architecture.html
28. Stream audio from client to server to client using WebSocket - Stack Overflow, accessed July 21, 2025, https://stackoverflow.com/questions/63103293/stream-audio-from-client-to-server-to-client-using-websocket
29. Secrets | Modal Docs, accessed July 21, 2025, https://modal.com/docs/guide/secrets
30. modal secret | Modal Docs, accessed July 21, 2025, https://modal.com/docs/reference/cli/secret
31. modal.Secret | Modal Docs, accessed July 21, 2025, https://modal.com/docs/reference/modal.Secret
32. modal-examples/04_secrets/db_to_sheet.py at main - GitHub, accessed July 21, 2025, https://github.com/modal-labs/modal-examples/blob/main/04_secrets/db_to_sheet.py
33. WebSockets - FastAPI, accessed July 21, 2025, https://fastapi.tiangolo.com/advanced/websockets/
34. Queues — Python 3.13.5 documentation, accessed July 21, 2025, https://docs.python.org/3/library/asyncio-queue.html
35. Advanced Python Generators: Techniques and Use Cases - Codedamn, accessed July 21, 2025, https://codedamn.com/news/python/advanced-python-generators-techniques-use-cases
36. How to Build a Streaming Open-Source Whisper WebSocket Service | by David Richards, accessed July 21, 2025, https://medium.com/@david.richards.tech/how-to-build-a-streaming-whisper-websocket-service-1528b96b1235
37. How do I add environment variables to my Vercel project?, accessed July 21, 2025, https://vercel.com/guides/how-to-add-vercel-environment-variables
38. How to add, edit or remove environment variables in Vercel - delasign, accessed July 21, 2025, https://www.delasign.com/blog/how-to-add-edit-or-remove-environment-variables-in-vercel/
39. WebSocket Integration in React: A Custom Hook Approach with react-use-websocket | by Shalabha Mary Sunny | Medium, accessed July 21,

2025,
https://medium.com/@shalabhasunny/websocket-integration-in-react-a-custom-hook-approach-with-react-use-websocket-bf310dad0512
40. jasonericdavis/speech-recognition-using-react: A web app that demonstrates using the Rev.ai speech to text API using ReactJs - GitHub, accessed July 21, 2025, https://github.com/jasonericdavis/speech-recognition-using-react
41. web-microphone-websocket - Codesandbox, accessed July 21, 2025, https://codesandbox.io/s/web-microphone-websocket-2e864
42. Playing Audio Using Web Audio API | by Selçuk Sert - Medium, accessed July 21, 2025, https://medium.com/@selcuk.sert/playing-audio-using-web-audio-api-949558576646
43. Web Audio API: Occasional Skipped Audio Chunks When Playing Real-Time Stream in VueJS App - Reddit, accessed July 21, 2025, https://www.reddit.com/r/vuejs/comments/1dosnk1/web_audio_api_occasional_skipped_audio_chunks/
44. Real-Time Audio Streaming in React.js: Handling and Playing Live ..., accessed July 21, 2025, https://medium.com/@sandeeplakhiwal/real-time-audio-streaming-in-react-js-handling-and-playing-live-audio-buffers-c72ec38c91fa
45. Streaming + PCM data + websocket + web audio API | by Adrien Desbiaux | Medium, accessed July 21, 2025, https://medium.com/@adriendesbiaux/streaming-pcm-data-websocket-web-audio-api-part-1-2-5465e84c36ea
46. How do I set up a staging environment on Vercel?, accessed July 21, 2025, https://vercel.com/guides/set-up-a-staging-environment-on-vercel